PhD Thesis

FOR THE DEGREE OF Doctor of Philosophy

University of Blida1
INSTITUTE OF AERONAUTICS AND SPACE STUDIES

Speciality: Avionics

By:

Khames Walid

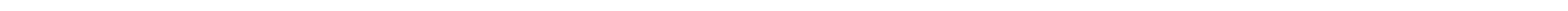
Advanced Processing of Sensing Big Data: A Multicore Architecture-Based Skyline Model

Thesis Directors: **Allel HADJALI & Mohand Lagha**

Defended on 30 January 2025 in front of a jury committee

-JURY-

BOUKRAA	Salah	Professor	President of the jury	University Blida1
HADJALI	Allel	Professor	Thesis Director	ENSMA Poitiers France
LAGHA	Mohand	Professor	Thesis Director	University Blida1
BENBLIDIA	Nadjia	Professor	Examiner	University Blida1
FADLOUN	Samiha	MCA	Examiner	ESI Alger
MEZIANE	Abdelkrim	Research Director	Examiner	CERIST Alger



This thesis is dedicated to all those who have supported and guided me throughout my academic journey. …

Abstract

Rising communication speeds, increased processing power, and the widespread use of hardware and software sensors have all contributed to our capacity for information creation rapidly expanding in recent years. This data is frequently available in the form of continuous streaming data, and the capacity to collect and analyze it in order to obtain insights and find trends represents tremendous potential for many enterprises and scientific applications. Recently, there has been an increase in activity in the academic world about **Q**uery **P**rocessing **O**ver **D**ata **S**tream (QPODS). From the development of computationally efficient algorithms to the design of programming and real-time systems to enable their execution, there are many difficulties to overcome while creating QPODS applications. Two major problems will be addressed in this thesis:

Continuous **S**kyline **P**rocessing **O**ver **D**ata **S**tream: SPODS applications are long-running (24 hr/7 d), so they are exposed to changes in arrival rates and workload properties. The ability of applications to process incoming data in real time is crucial so that they can effectively manage the dynamic workload and provide a cost-effective overall performance.

The necessity of high-performance skyline queries: When dealing with SPODS challenges, having both high throughput and low latency is essential. Software needs to efficiently utilize parallel hardware like multi-core processors.

There is a lack of dynamic strategies with well-known properties of real-time processing, no delay response, and energy efficient performance in the current approaches to the development of SPODS applications, and there is inefficient exploitation of the parallelism of existing multicore architectures. This study makes an attempt to address these gaps by applying established methods like parallel programming and QPoDS.

When it comes to data streams, sliding windowed queries are among the most common types. The most recent data that has been received is processed. The content and size of windows can change over time because they are dynamic data structures. Since various windowing methods (time- or count-based) have specific requirements in terms of data distribution and management policies, the SPODS domain necessitates particular expertise and enhanced features relative to conventional parallel techniques. The time and effort required for parallel programming can be minimized with the help of a well-organized strategy (ex: openMP API). It also makes it easier to understand the relationship between a parallel solution's throughput, latency, and other performance metrics.

While numerous articles have been published regarding the parallelization of regular skyline queries, there is a limited amount of research dedicated specifically to the parallel processing of continuous skyline queries. This study introduces a **P**arallel **R**ange **S**earch **S**kyline (PRSS), a continuous skyline technique for multicore processors specifically designed for sliding window-based high dimensional data streams. The efficacy of the proposed parallel implementation is demonstrated through tests conducted on both real-world and synthetic datasets, encompassing various point distributions, arrival rates, and window widths. The experimental results for a dataset characterized by a large number of dimensions and cardinality demonstrate significant acceleration.

Keywords: Data Management, Big Data, Sensors, Multicore Architecture, continuous skyline, Data Stream , High-dimensionality, Parallel processing.

الملخص

لقد ساهمت سرعات الاتصال المتزايدة، وقوة المعالجة المتزايدة، والاستخدام الواسع النطاق لأجهزة الاستشعار والبرامج في زيادة قدرتنا على إنشاء المعلومات بسرعة في السنوات الأخيرة. تتوفر هذه البيانات غالباً في شكل بيانات متدفقة مستمرة، وتمثل القدرة على جمعها وتحليلها من أجل الحصول على رؤى والعثور على الاتجاهات إمكانات هائلة للعديد من الشركات والتطبيقات العلمية. في الآونة الأخيرة، كان هناك زيادة في النشاط في العالم الأكاديمي حول معالجة الاستعلامات عبر تدفق البيانات. من تطوير الخوارزميات الفعالة حسابياً إلى تصميم البرمجة والأنظمة في الوقت الفعلي لتمكين تنفيذها، هناك العديد من الصعوبات التي يجب التغلب عليها أثناء إنشاء تطبيقات تدفق البيانات.

سيتم تناول مشكلتين رئيسيتين في هذه الأطروحة:

أفق مستمر فوق تدفق البيانات: التطبيقات طويلة الأمد (24 ساعة / 7 أيام)، لذلك فهي معرضة للتغيرات في معدلات الوصول وخصائص عبء العمل. إن قدرة التطبيقات على معالجة البيانات الواردة في الوقت الفعلي أمر بالغ الأهمية حتى تتمكن من إدارة عبء العمل الديناميكي بشكل فعال وتوفير أداء عام فعال من حيث التكلفة.

ضرورة استعلامات الأفق عالية الأداء: عند التعامل مع تحديات تدفق البيانات، فإن الحصول على معدل إنتاج مرتفع وزمن وصول منخفض أمر ضروري. يحتاج البرنامج إلى الاستفادة بكفاءة من الأجهزة المتوازية مثل المعالجات متعددة النواة.

هناك نقص في الاستراتيجيات الديناميكية ذات الخصائص المعروفة للمعالجة في الوقت الفعلي، وعدم وجود استجابة تأخير، والأداء الموفر للطاقة في الأساليب الحالية لتطوير معالجة الأفق عبر تطبيقات تدفق البيانات، وهناك استغلال غير فعال للتوازي بين هياكل متعددة النواة الحالية.

عندما يتعلق الأمر بتدفقات البيانات، فإن الاستعلامات ذات النوافذ المنزلقة هي من بين أكثر الأنواع شيوعًا. تتم معالجة أحدث البيانات التي تم استلامها. يمكن أن يتغير محتوى وحجم النوافذ بمرور الوقت لأنها هياكل بيانات ديناميكية. نظراً لأن طرق النوافذ المختلفة (القائمة على الوقت أو العد) لها متطلبات محددة من حيث سياسات توزيع وإدارة البيانات، فإن مجال تدفق البيانات يتطلب خبرة خاصة وميزات محسنة نسبياً للتقنيات المتوازية التقليدية. يمكن تقليل الوقت والجهد المطلوبين للبرمجة المتوازية بمساعدة استراتيجية منظمة جيداً.

كما أنه يجعل من الأسهل فهم العلاقة بين معدل إنتاجية الحل الموازي، وزمن الوصول، ومقاييس الأداء الأخرى.

بينما تم نشر العديد من المقالات فيما يتعلق بالتوازي في استعلامات الأفق العادية، هناك قدر محدود من الأبحاث المخصصة خصيصًا للمعالجة المتوازية لاستعلامات الأفق المستمرة. تقدم هذه الدراسة برسس، وهي تقنية أفق مستمرة للمعالجات متعددة النواة مصممة خصيصًا لتدفقات البيانات القائمة على النوافذ المنزلقة. يتم إثبات فعالية التنفيذ الموازي المقترح من خلال الاختبارات التي أجريت على كل من مجموعات البيانات الواقعية والاصطناعية، والتي تشمل توزيعات النقاط المختلفة، ومعدلات الوصول، وعرض النوافذ. تظهر النتائج التجريبية لمجموعة البيانات التي تتميز بعدد كبير من الأبعاد والعناصر الأساسية تسارعًا كبيرًا.

الكلمات المفتاحية:

إدارة البيانات، البيانات الضخمة، أجهزة الاستشعار، بنية متعددة النواة، الأفق المستمر، تدفق البيانات، الأبعاد العالية، المعالجة المتوازية.

Contents

Abstract v

Contents vii

List of Acronyms xv

I Introduction 1

1 Introduction 3
- 1.1 Big Data and Skyline Processing over Data Streams (SPODS) . . . 3
- 1.2 SPODS challenges . . . 4
- 1.3 Inspiring works . . . 5
  - 1.3.1 Aviation applications . . . 5
  - 1.3.2 Predictive analytics with aviation big data . . . 6
  - 1.3.3 Aircraft sensor data streams . . . 6
  - 1.3.4 Civil Aircraft Big Data Platform . . . 7
  - 1.3.5 On-the-fly Mobility Event Detection over Aircraft Trajectories . . . 8
  - 1.3.6 Research on Aircraft Air Conditioning Performance Monitoring and Trend Predicting . . . 9
  - 1.3.7 Skyline in Network applications . . . 9
  - 1.3.8 Skyline on Social media analysis . . . 10
- 1.4 High performance SPODS applications . . . 10
  - 1.4.1 Parallelism opportunities in SPODS applications . . . 11
- 1.5 Contributions of the thesis . . . 12
- 1.6 Outline of the thesis . . . 12

II Background on Data Stream Processing 15

2 Background on query processing with Data stream 17
- 2.1 Characteristics of a *SPODS* application . . . 17
  - 2.1.1 Function state . . . 18
  - 2.1.2 Windowing approaches and State type . . . 19
- 2.2 Data Stream Processing systems . . . 20
  - 2.2.1 Data Stream Management Systems . . . 21
  - 2.2.2 Complex Event Processing systems . . . 22
  - 2.2.3 Stream Processing Engines SPE . . . 22
- 2.3 Parallelism exploitation in SPODS systems . . . 23

2.3.1 Parallelism in QPODS systems . . . 23
2.3.2 Literature approaches . . . 24
2.4 Conclusion . . . 25

III Skyline Queries 27

3 Continuous Skyline Queries 29
3.1 Entities and Attributes . . . 30
3.2 The Concept of Dominance . . . 31
3.2.1 Skyline Queries . . . 32
3.2.2 Main-Memory Computation (in-core skyline queries) . . . 33
3.2.3 Algorithms for Secondary Memory (out of core skyline queries) . . . 34
3.3 Advanced Skyline Processing . . . 34
3.3.1 Skylines in Dynamic Environments . . . 34
3.3.2 Distributed and Parallel Techniques . . . 40
3.3.3 A High-Level Approach to Parallel Programming . . . 42
3.3.4 Parallel Paradigms for Skyline Queries Over Data Stream (SPODS) . . . 44
3.3.5 Parallel Patterns for Windowed Functions . . . 53
3.3.6 Parallel Patterns Taxonomy . . . 53
3.3.7 Categories of Parallel Patterns for Windowed Functions . . . 53
3.3.8 Pane Farming . . . 55
3.3.9 Window Partitioning . . . 56
3.4 Skyline Cardinality . . . 58
3.5 Conclusion . . . 63

4 Variations of Skyline Queries 65
4.0.1 Dynamic Skyline Queries . . . 66
4.0.2 Group skyline computation . . . 67
4.0.3 Spatial Skyline Queries . . . 68
4.0.4 Metric Space Skyline Queries . . . 75
4.0.5 Constrained Skyline Query . . . 78
4.0.6 Range-Based Skyline Queries . . . 79
4.0.7 Reverse skyline queries . . . 82
4.1 Applications of Skyline-Based Queries . . . 83
4.1.1 Multi-criteria decision making . . . 83
4.1.2 Machine learning . . . 84
4.1.3 Network analysis . . . 86
4.1.4 Other interesting applications . . . 86
4.2 Conclusion . . . 89

IV Parallel Range Search Skyline 91

5 Parallel Range Search Skyline 93
5.1 Problem Definition . . . 93
5.1.1 Dimension Indexing . . . 93
5.2 Parallel RSS over data stream . . . 105
5.2.1 Parallel Range Search Skyline $PRSS$ . . . 105

5.2.2 Optimization of the dominate() Function using AVX2 110
5.2.3 Parallel implementation details 114
5.3 Performance Evaluation 115
5.3.1 Experimental Setup 115
5.3.2 Experimental Results 117
5.4 Conclusion 127

V Outlooks and conclusion **129**
5.5 Conclusion 131
5.6 List of Publications 132

List of Publications **132**

VI Bibliography **133**

Bibliography **135**

List of Figures

1.1 Skyline query over Flight Data. 6
1.2 Skyline query usage for IoT devices selections. 7
1.3 An example of skyline over data stream processing. 11

2.1 Sliding window vs Tumbling window. 20
2.2 Count Based sliding window VS Time based Sliding Window. 20
2.3 History of various DaSP Systems. 21

3.1 The concept of dominance. 32
3.2 Taxonomy of Skyline query processing over Data Stream. 35
3.3 Handling Insertions and Deletions. 35
3.4 Incomparable Points. 36
3.5 Gid-based (right) and Angle-based (left) Partitioning. 42
3.6 Pipeline parallel pattern [26]. 45
3.7 Task Farm parallel pattern [26]. 46
3.8 Map pattern with 4 cores [26]. 47
3.9 Window Farming with two cores. In the figure $|W| = 3$ and $\delta = 1$. w_i^x is the i-th window of substream X. $F(w_i)$ is the result of the processing function over a window [26]. 54
3.10 Key Partitioning with fine-grained distribution. Substream X is routed to the first Core, substream Y to the second one [26]. 55
3.11 Sliding window created with 4 different panes. Each pane is identified in 4 consecutive windows [26]. 56
3.12 Window Partitioning with 4 cores and one key. In the figure $|W| = 8$ and $\delta = 4$ [26]. 57
3.13 Correlated, anticorrelated, and independent data distributions. 58

4.1 Dynamic Skyline Query. 67
4.2 Spatial Skyline Query. 69
4.3 Nearest and the Farthest Spatial Skyline Queries. 70
4.4 Spatio-textual Skyline Query. 71
4.5 Direction-based Skyline Query. 72
4.6 Metric Skyline Query. 78
4.7 Constrained Skyline Query. 79
4.8 Type of Range-skyline Query developed in [236]. 79
4.9 Range Skyline Query. 81
4.10 Illustration of Service Workflow and Airplane Service Pruning. 85
4.11 Naive Bayesian Classifier Model. 85
4.12 Hash BBS Skyline Process. 87

4.13 Monitor Process. 88
4.14 Dynamic Skyline Sensor Selection. 89

5.1 Dimension index system. 95
5.2 Dimension index for multidimensional dataset in table 3.2. 95
5.3 Skyline Maintenance with a new Incoming Tuple. 96
5.4 Load Balancing on Cores. 105
5.5 Step-by-step Skyline Index Update. 110
5.6 Tuples comparisons using AVX2. 111
5.7 Various load-balancing strategies. 115
5.8 RSS vs PRSS (Anticorrelated Count-Based window, Different Cardinalities). . 116
5.9 RSS vs PRSS (Correlated Count-Based Window, Different Cardinalities). . . . 116
5.10 RSS vs PRSS (Independent Count-Based Window, Different Cardinalities). . . 117
5.11 RSS vs PRSS (Anticorrelated Count-Based Window, Different Dimensionalities). 118
5.12 RSS vs PRSS (Correlated Count-Based Window, Different Dimensionalities). . 118
5.13 RSS vs PRSS (Independent Count-Based Window, Different Dimensionalities). 119
5.14 PRSS scalability (Anticorrelated Count-Based Window, Different Numbers of Threads). 119
5.15 Memory consumption of RSS vs PRSS (Anti, Corr, Ind data, Different Dimensionalities). 122
5.16 Tracking Memory consumption of RSS vs PRSS. 123
5.17 RSS vs PRSS vs BskyTree (Anticorrelated Count-Based Windows, Different Cardinalities). 124
5.18 RSS vs PRSS vs BskyTree (Correlated Count-Based Windows, Different Cardinalities). 124
5.19 RSS vs PRSS vs BskyTree (Independent Count-Based Windows, Different Cardinalities). 125
5.20 RSS vs PRSS vs BskyTree (Anticorrelated Count-Based Windows, Different Dimensionalities). 125
5.21 RSS vs PRSS vs BskyTree (Correlated Count-Based Windows, Different dimensionalities). 126
5.22 RSS vs PRSS vs BskyTree (Independent Count-Based windows, Different Dimensionalities). 126

List of Tables

3.1 Summary of Notations. 32
3.2 A sample database with $A = 6$, $Z = 10$, and $m = 5$. 33
3.3 Comparison of Window Partitioning and Window Farming 57
3.4 Summary of Parallel Patterns for Skyline Queries [26]. 58
3.5 An example of approaches to sequential continuous skyline techniques over a complete data stream. 59
3.6 Examples of approaches to sequential continuous skyline techniques over Incomplete and Uncertain data streams. 60
3.7 Example of approaches to parallel continuous skyline techniques. 61

4.1 Flight ticket price and duration for group travelers. 68
4.2 Flight ticket prices, distances, and airline preferences. 71
4.3 Comparison of different skyline query variations. 84

5.1 Data Chunk Assignment 114
5.2 RSS vs Parallel RSS vs BskyTree with Real-World Datasets. 117

List of Acronyms

Acronym	Definition
SPODS	Skyline Processing Over Data Stream
QPODS	Query Processing Over Data Stream
DaSP	Data Stream Processing
CRSQs	Continuous Range-based Skyline Queries
SPM	Sectro photo meter
LBA	landmark-based
IBA	index-based
exabytes	1 million terabytes (TB) or 1 billion gigabytes (GB)
BR&T	Boeing Research & Technology
AATM	Advanced Air Traffic Management
ASDI	Aircraft Situation Display to Industry
DBMS	Database Management System
PHM	Prognostic Health Management for aircraft
AI	Artificial Intilligence
CBM	Condition-based maintenance
AL	autonomous logistics
AHM	Aircraft Health Management
AiRTHM	Aircraft Integrated Real-Time Health Monitoring
ATM	air traffic management
ADS-B	Automatic Dependent Surveillance-Broadcast
SPP	structured parallel programming
OpenMP	Open Multi-Processing
Openacc	Open Accelerators
SYCL	Standard for Heterogeneous Programming in C++
IDE	Integrated Development Environment
HPC	High-Performance Computing
PRSS	Parallel Range Search Skyline
DSMS	Data Stream Management Systems
CEP	Complex Event Processing
SPE	Stream Processing Engines
DSMS	Data Stream Management Systems

Acronym	**Definition**
DSMSs	Distributed Stream Processing Systems
SQL	Structured Query Languag
SQuAl	Stream Query Language
QoS	quality of service
IT	Information Technology
GCP	Google Cloud Platform
GPU	Graphical Processing Unit
Sky(t)	Skyline of t
BBS	Branch Bound Skyline
NN	Nearest Neighbor
P2P networks	Peer-to-Peer Networks
SSP method	Scalable Skyline Processing Method
MANETs	mobile ad-hoc networks
RP	Random Partitioning
GP	Grid-based partitioning
AP	Angle-based Partitioning
DR	Dominance Region
ADR	Anti-dominance Regions
WSN	Wirless networks

Part I

Introduction

CHAPTER

1 INTRODUCTION

Chapter content

1.1 Big Data and Skyline Processing over Data Streams (SPODS) 3

1.2 SPODS challenges 4

1.3 Inspiring works 5

1.3.1 Aviation applications 5

1.3.2 Predictive analytics with aviation big data 6

1.3.3 Aircraft sensor data streams 6

1.3.4 Civil Aircraft Big Data Platform 7

1.3.5 On-the-fly Mobility Event Detection over Aircraft Trajectories 8

1.3.6 Research on Aircraft Air Conditioning Performance Monitoring and Trend Predicting 9

1.3.7 Skyline in Network applications 9

1.3.8 Skyline on Social media analysis 10

1.4 High performance SPODS applications 10

1.4.1 Parallelism opportunities in SPODS applications 11

1.5 Contributions of the thesis 12

1.6 Outline of the thesis 12

1.1 Big Data and Skyline Processing over Data Streams (SPODS)

The rapid evolution of Big Data is transforming various domains at an unprecedented scale [1]. Sensors, infrastructure systems, and global stock markets are among the many sources of continuous data streams, with human interactions on social media also serving as significant data generators. Every day, approximately 2.5 exabytes of new data are generated, and astonishingly, 90% of all existing data has been produced within the past two years alone [2]. By 2030, it is expected that tens of billions of devices and physical hardware components will be connected to the internet, driven by the widespread adoption of sensor technology [3].

Real-time data is typically transmitted as a continuous stream, and the ability to collect and analyze it to derive insights and identify patterns is proving to be invaluable across various industries and academic fields. These applications are used in various domains and typically have strict performance requirements [4]. Examples of such applications include high-frequency

trading systems [5], healthcare [6], network security [7], and disaster management [8]. These systems require real-time processing of large volumes of data to detect anomalies and take timely corrective actions. Any delay in response is ineffective and can even lead to harmful consequences.

Time-sensitive applications cannot depend on traditional store-then-process (or batch) architectures, which are designed for environments where data is highly structured. As a result, a new method for managing and analyzing streaming data, called Query Processing Over Data Streams (QPODS), has been developed. Some of its key features, as highlighted by Babcock et al. [9] and Andrade et al. [10], are as follows:

- Data is represented not as static ”in-memory” structures, as in traditional applications, but as continuous streams that flow over time;
- Data analysis systems cannot change the way data is delivered during processing;
- Data processing must occur in real-time, or at most, with limited memory, due to the continuous input and strict performance demands. This may require the use of windowing techniques or approximate processing methods.

When processing large volumes of data, Skyline Processing over Data Streams (SPODS) is often employed. The five key characteristics of Big Data ”volume, velocity, variety, veracity, and value” are essential for understanding its complexity [11]. Volume refers to the massive amount of data generated, while velocity describes the speed at which data is produced and transmitted. Variety involves the different forms and types of data, and veracity refers to the accuracy and trustworthiness of the data. Value signifies the potential worth derived from analyzing the data. SPODS mainly focuses on handling the velocity and variety aspects of the ”Big Data Challeng” in contrast to traditional batch processing systems, which primarily address volume and variety.

1.2 SPODS challenges

From creating efficient algorithms for real-time processing to developing programming environments and runtime systems for deployment and execution, SPODS applications present several challenges in various database management contexts [12], [10]. Some of these challenges, particularly those related to modern hardware and software required to implement and run these applications, will be discussed.

The analytical results from stream processing computations need to be of high quality and delivered quickly enough to keep up with the rate at which data is being received. Therefore, SPODS applications have to utilize parallel hardware and parallel systems, such as multi/many-cores or a cluster of multicores, to achieve high throughput and low latency. the intended preference of the user are expressed by the overall performance requirements, which are preconditions on quantitative metrics presenting the application’s efficiency degree such as: reduce delays, optimized bandwidth, energy-efficient.

In this context, ”the mean response time of the application must be less than or equal to a given threshold” and ”the application must have the capacity to maintain a given throughput” are examples of common worries. The continuous style (24 h/7 d) and dynamic execution scenario of SPODS applications have an impact on them and cause unexpected shifts in their workload characteristics.

These are all problems that the HPC and data base community is familiar with, but the persistent nature of the issue calls for additional consideration here. Many SPODS algorithms

have been proposed in recent years, and a great deal of research has been conducted on this topic in both the academic and commercial sectors. This is because, in addition to being an engaging and complicated research topic, it can be of strategic importance to many different types of businesses. The SPODS community has not yet resolved all of these issues.

This thesis aims to investigate the difficulties associated with skyline computation and propose a new parallel solution to these difficulties. Here, we'll go over a number of compelling use cases for SPODS that illustrate its main features. Then we present our outlook and method for overcoming the aforementioned difficulties.

1.3 Inspiring works

In different situations, scenarios of real use that convey the aforementioned characteristics can be found. We'll go over a few of them below, but there are countless others in fields like automobiles [13], medicine [6], and the Internet of Things [14]. What kinds of uses will emerge in the coming years as a result of this "Big Data revolution" no one can guess. But one thing is certain: data stream processing is here to stay, and all this massive amount of real-time data will be worthless unless applications can quickly and accurately ingest and analyze it [10].

1.3.1 Aviation applications

The advent of algorithmic trading and high-frequency trading in the financial markets 10 years ago was a game-changer. As a result, high-frequency trading moves from traditional exchanges to digital ones, where computer programs are in charge of managing and monitoring it. Real-time data distribution services, such as Aviation Financial applications, continuously disseminate information about aircraft maintenance, safety factors, ticket prices, and sales transactions taking place across various global sale agencies [2]. Flight companies must analyze these market data feeds in order to identify trends and opportunities. Simple computations include filtering and aggregation while more complex computations involve correlating multiple streams. Among the many techniques used to manage how ticket prices will adapt in the future is the identification of recognizable patterns in aviation market charts as illustrated in Figure 1.1. Automatic trading actions can be taken to predict competitors' actions if the most recently received marketing information for a given market (say, "the last 1000") exhibits well-defined action [2].

Flight time is one of the most important design factors in aviation financial applications. If a flight company experiences significant latency when analyzing the market, the data they use to make their decisions will be outdated. However, if processing times can be guaranteed to within a few milliseconds, that could be a huge advantage over competitors. The rise of algorithmic and electronic trading, on the other hand, has resulted in a persistent acceleration of data rates over the past few years. According to recent studies [2], the maximum number of messages that can be sent in 100 milliseconds in a single stream conveying information about different market symbols is expected to rise to 400 thousand in 2025 [15]. Average rates are much lower, frequently by an order of magnitude, highlighting a highly dynamic environment where spikes are the result of outside events (such as news, public sentiment, and political decisions) and may only apply to a portion of the information market. To process this rapidly rising feed rate economically, high-performance solutions are required.

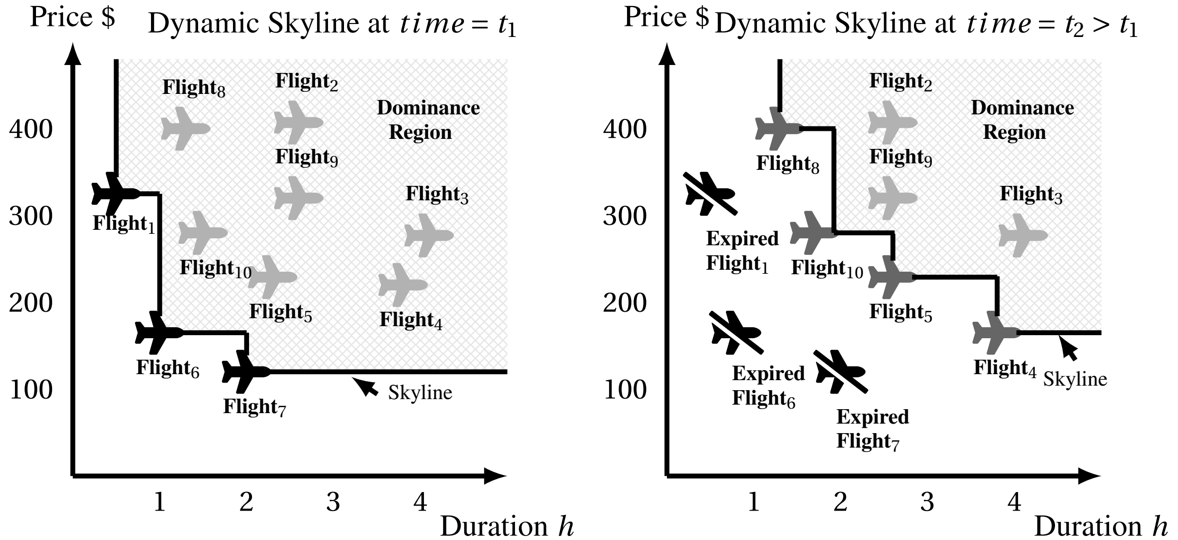


Figure 1.1: *Skyline query over Flight Data.*

1.3.2 Predictive analytics with aviation big data

Boeing Research & Technology (BR&T) created the Advanced Air Traffic Management (AATM) system, which is capable of making predictions using descriptive patterns found in a lot of aviation data [1]. For over two years, Boeing AATM has been archiving real-time information from the Aircraft Situation Display to Industry (ASDI). There is currently no straightforward method to analyze the data. It is necessary to correlate the incoming ASDI data with other flight data, and it is large and compressed. Once the ASDI data was pushed into the data warehouse, it was necessary for us to process the incoming data in near real-time so that users could begin conducting analytics.

Big data refers to information that defies conventional methods of storage and retrieval. Since it will be too big to fit on a single disk, multiple processors will need to work on it simultaneously. When used in monitoring applications, a DBMS's job is to sound the alarm whenever anything out of the ordinary occurs.

The scenario involves calculating the unique traffic volume within a boundary or airway segment at any future time in 15-minute buckets for a given airspace (sector, center, airspace volume of interest, boundary, or airway segment). Near-real-time business analytics refers to the practice of reducing the lag time between the collection of data and the implementation of a decision based on that data. Here, we're hoping to propose alternative flight paths and warn of congestion caused by flow control areas or weather if certain thresholds are met.

1.3.3 Aircraft sensor data streams

Time is a crucial factor in many contexts where prompt decisions must be made. Specifically, it necessitates real-time processing of data streams. This necessitates the real-time processing of information gathered from sensor observations. Incorporating data compression, data stream abstraction, continuous queries, and the generation of links in real time are just a few examples of the sorts of changes that need to be made to the conventional static data publishing process to meet this need.

A growing number of businesses, academic institutions, engineering fields, and even governments are investing in the race to develop effective management queries for handling what

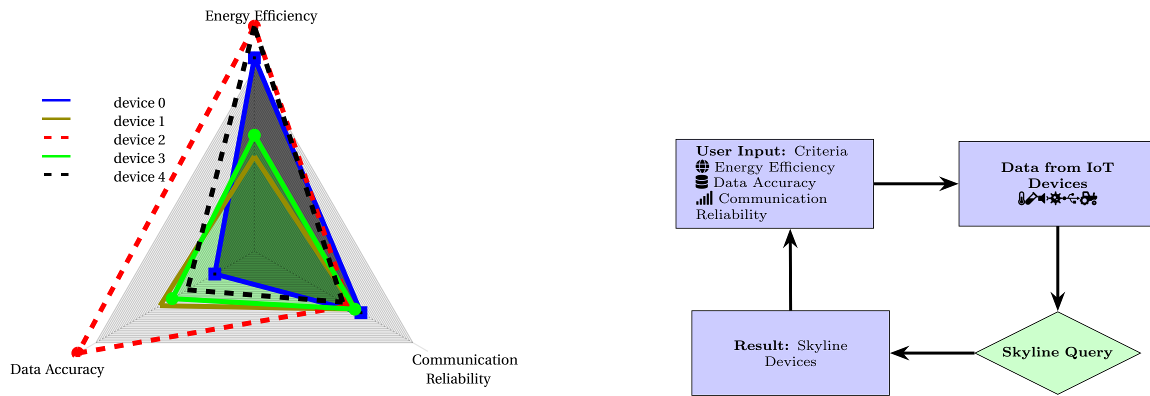


Figure 1.2: *Skyline query usage for IoT devices selections.*

is referred to as ”big data”. Large industries, such as the aviation industry, as shown in Fig. 1.2, sensor manufacturing, IoT, and others, will significantly depend on the ability to retrieve relevant patterns and information from enormous volumes of high-speed streamed data, making this a rapidly expanding and promising field.

The aviation industry was practically developed in the age of big data [16]. There are millions of components and dozens of systems in an airplane, and sensors allow for the collection and storage of massive amounts of information about their operation. Big data's innovative tools improve passengers' trips and boost airlines' bottom lines. Pratt & Whitney's big data analysis services can currently collect around 1000 parameters per engine per flight, with a projected increase to 5000 parameters per engine per flight for the next generation of geared turbofan engines [15].

1.3.4 Civil Aircraft Big Data Platform

Scheduling Flights Using Predictive Analytics

The regularity of flights has an impact on airlines revenue. When arranging flights, airlines look for ways to cut down on wait times, boost customer satisfaction, and lower overhead expenses. Weather, aircraft type, route, flight order, air traffic control, airline planning, airport security capabilities, operation model, passenger density, and passenger composition are among the primary contributors to flight delays. The civil aircraft big data platform performs essential flight arrangement tasks like real-time flight position tracking, flight arrival time forecasting, weather analysis, flight planning, and marketing analysis. The system is able to minimize both flight delays and energy usage [2].

Health and fault management for aircraft

Aircraft contain a plethora of sensors and other monitoring equipment. Large amounts of information can be collected through high-frequency, extended sampling while in flight. Cabin pressure, altitude, and fuel consumption data from a Boeing 787 flight, for instance, total more than half a terabyte. Aircraft health and performance can be tracked and potential problems identified with the help of big data analytics applied to intelligent fault diagnosis systems. The big data platform for civil aviation keeps tabs on the aircraft's deteriorating performance, forecasts its future dependability, detects mutation faults, and sounds alarms before a complete

breakdown occurs. Prognostic Health Management (PHM) technology went through two major shifts due to the rise of big data and advances in AI. First, moving from intelligent system diagnosis and prediction based on sensor data Secondly, switching from scheduled maintenance (maintenance at specific times) to maintenance based on the current state [2].

Plan for Maintenance

Intelligent aircraft maintenance is possible with the help of big data and PHM technology. The civil aviation industry can take advantage of the big data platform to analyze trends in flight and historical data for health monitoring, fault diagnosis, and fault prediction. The platform can calculate when the aircraft will need to be inspected for routine maintenance. In order to improve maintenance productivity and reduce maintenance costs, airlines can prepare for potential maintenance by doing things like making maintenance plans and ordering repair parts in advance [2].

A Big-Data-Powered Fuel-Saving System

By analyzing flight plan data, actual flight data, load balancing data, airport weather conditions, pilot operation, and engine fuel loss, the civil aircraft big data platform can aid airlines in making noise alternate field decisions. Using a large database of flights as input, the platform's prediction model can estimate how much fuel will be used during each flight. A more manageable amount of reserve oil can be carried, and the use of any oil left over after landing can be optimized to save money on fuel [2].

1.3.5 On-the-fly Mobility Event Detection over Aircraft Trajectories

Hundreds of millions of people use commercial airlines every year to go on vacation or for business. Every day, there are over 26,000 flights within Europe [17]. Worries about safety and its possible significant effect on the already congested skies have been raised as the air transport industry, which is a strategic industry, is anticipated to expand by 5% annually until 2030. Airlines must carefully plan their routes, maintenance services, crew rotations, and ticket prices due to intense competition and low profit margins resulting from rising costs of both labor and fuel [17].

In order to reduce direct operating expenses, air traffic management (ATM) systems use flight plans as an accurate representation of aircraft's intended routes. Due to the critical nature of accurate and predictable aircraft trajectories in ATM systems, a model-based approach is typically used. However, this has limitations due to inherent inaccuracies, varying sources of error, and the impact of uncertainty (weather, traffic, etc.). Yet, with today's technology, planes can be monitored in real time, either through Automatic Dependent Surveillance-Broadcast (ADS-B) messages broadcast by planes or by terrestrial radar stations. Therefore, a data-driven strategy for trajectory management may contribute to better ATM performance. Since flight plans for many planes in the sky may need to be changed at the last minute due to weather or other unforeseen circumstances, accurate and timely trajectory representation for all active flights across a large region is essential [17].

1.3.6 Research on Aircraft Air Conditioning Performance Monitoring and Trend Predicting

There are six primary aircraft systems that have a major impact on flight safety. System components include HVAC (pack), air bleed, hydraulics, flight controls, landing gear, and generators. Therefore, those systems' failures are also the primary cause of airline flight delays and cancellations [8]. The air conditioning system of A320 aircraft accounts for a high percentage of all types of malfunctions, which has a significant impact on flight punctuality. Most weather phenomena occur in the boundary layer, the lowest layer of the atmosphere. The variables and factors that influence the weather on a daily basis are numerous and inaccessible. As a result, weather forecasts can only be as accurate as the data used to create them [8].

Historically, meteorologists have relied on weather balloons and stationary weather stations to compile their data. However, weather stations have significant limitations in their measurements due to their stationary location and close proximity to the earth's surface [8]. They are limited in their ability to collect data from the boundary layer because they must remain on the ground.

Drones, on the other hand, are exempt from this rule and, with the proper authorization, can be flown at altitudes greater than 1 kilometer (3500 feet) [8]. In order to gather information from a specific location at a specific altitude, weather drones are so easy to pilot that they can be flown directly into the wind or even into storms. Drones can also be programmed to go back to their home base. This enables the installation of a wide variety of high-priced sensors and other devices on a weather drone.

Making quick and reliable decisions about how and when a specific weather scenario will impact a flight is one of the greatest obstacles faced by operators throughout all segments of aviation. With Schneider Electric's new 4D Flight Route Alerting solution, pilots and flight dispatchers can receive real-time alerts that include altitude, position, and time components to help aircraft avoid areas of heavy turbulence or severe weather. Despite advances in aviation technology, human error is still a problem, which is why we need decision-making support systems to help pilots out while they're in the air.

1.3.7 Skyline in Network applications

Maintaining control of a sizable data communications infrastructure calls for round-the-clock network monitoring. The goals of an analysis may vary greatly. For instance, traffic monitoring on traffic summaries generated by network equipment may be of interest to network administrators. Analysis of system load (e.g., how much bandwidth is used by which applications or which parts of the network) is computed frequently, as are analyses of flow characteristics (e.g., distribution of life time and size) and session characteristics [18].

In addition to infrastructure management, traffic monitoring also includes the equally pressing issue of cyber security. Data theft, economic losses, and lost productivity are all avoidable if security threats are managed. In order to combat this issue, detection systems for network intrusions were recently implemented. Detecting potential attacks in real time, they then issue alerts as quickly as possible so that countermeasures can be taken [19]. Their detection algorithms are typically complex, requiring the line speed to be able to process and analyze multiple traffic flows simultaneously. For an appropriately big corporate network, especially during working hours, similar systems have to handle streaming data from multiple sources at collected rates approaching several to tens of gigabytes per second. The desired detection latencies must be on the order of milliseconds, and meeting these requirements is essential because delayed responses

can have serious consequences for both the economy and security.

1.3.8 Skyline on Social media analysis

For establishing connections with others and sending information, social media are now indispensable. Among these is Twitter (rebrand as X) [20], a microblogging service that has quickly become a go-to for people looking to share their thoughts, break news, or just chat with strangers. It has been instrumental in reporting on and discussing major sociopolitical events like the Arab Spring and Occupy Wall Street, as well as natural disasters. As Twitter has grown in popularity, researchers and developers have found new ways to put the platform to use. For example, data mining methods can be used to increase scientists' ability to comprehend novel phenomena. Real-world event identification via Twitter reasoning is a common form of online reasoning. In the past, event identification has been carried out using static document collections that are all connected through a web of unknown occurrences. Recent Twitter projects have begun to process data in real time as it is generated. Many of them seek to locate specific types of events, like breaking news or natural disasters like earthquakes and typhoons. It's possible that emergency workers would benefit from reduced latency, especially in the event of a natural disaster. Other works attempt to solve the issue of event identification (and tweet about it) without any prior context. Here, we employ a wide range of methods, from statistical to online clustering. The most recent data, which is more relevant to end users, is usually the primary focus of these analyses [21].

Research into the relationship between tweets and the stock market is another promising avenue. Twitter is a popular platform for disseminating stock market-related news and analysis among traders, investors, analysts, and media outlets. This means that certain stocks can generate thousands of tweets per day. The volume of tweets about a stock typically rises and falls over the course of a few days but can suddenly spike if there has been a significant change in sentiment about the stock. The sentiment in tweets could be used to inform stock option trading strategies [20].

These fascinating applications must deal with Twitter's inherent characteristics. There are currently over 611 million active users on Twitter, who collectively post an average of 5,700 tweets per second [21]. In response to a specific social event, for instance, the rate of tweets on that topic may spike by multiple orders of magnitude in a short period of time. System behavior must adjust to the volume of arriving data in order to deliver timely results, as timely topics or events tend to come and go quickly.

1.4 High performance SPODS applications

Processing skyline queries over data stream can be expressed in a generalized form using a computation graph as illustrated in Figure 1.3. query inputs consist of one or more continuous data streams, and query outputs may be received as yet additional continuous updated skyline flows. The query can be broken down into its component parts thanks to internal nodes that express intermediate computations [10, 22].

In most cases, an intermediate function takes in data and processes it in accordance with its own internal logic. Both stateless and stateful functions exist. Each piece of data fed into a stateless function is handled separately. In contrast, stateful functions maintain state information as they process data streams [10]. The value of the internal state, which is constantly updated whenever a new item is received, determines the output results. The issue of having

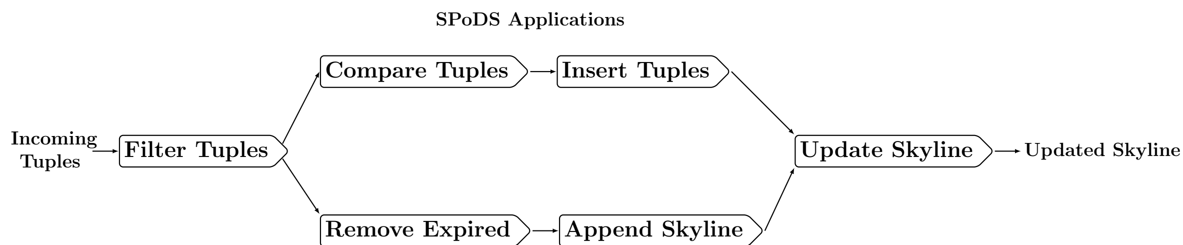


Figure 1.3: *An example of skyline over data stream processing.*

to store only a subset of continuous data streams arises from the fact that these streams could theoretically go on forever. In many contexts, more recent data is preferable to older data. A common workaround is to keep only the most recent data in a temporary window buffer and then run the skyline computation on that.

Various stream processing frameworks provide mechanisms for creating functions, streams, and so on, making it easier for developers to map their applications into a computation graph [10]. In order to boost developers' efficiency, it is common practice to supply them with a set of reusable functions that can be used in a wide variety of programs. Filters, joins, update, compare, sorts, aggregates (sum, count, max) and windowed functions are just a few examples of relational functions that transform data from one format to another, allowing it to be used within a skyline query. In addition, modern frameworks (like Apache Storm[23], Apache Flink [24], MapReduce [25]) generally support programmable functions written in a generic language. The abstract computation is then run using a collection of computers. With the help of the runtime system, computation can be mapped into a collection of interacting threads or processes, which can then be deployed and managed using the available resources (CPU cores).

1.4.1 Parallelism opportunities in SPODS applications

High throughput (i.e. applications must be capable of handling a high volume of arriving data) and low latency (i.e. results must be obtained in a short period of time) are two requirements that SPODS applications must meet. Inter-function parallelism, in which multiple functions run simultaneously on multiple processing elements, provides the first parallelism opportunity due to the independence of graph functions. The computation graph needs to be reorganized if inter-function parallelism is insufficient. The slowest sequential function must be identified and parallelized. The goal is to produce a computation graph with the same functional properties as the original graph but with some nodes transformed by internal parallelization like parallelising the dominance test or the insertion of new incoming tuples and the deletion of outdated ones [26].

In all but the most exceptional circumstances, manual parallelization is discouraged. Such low-level approaches not only limit software productivity and shorten development times, but they also prevent code and performance portability and can cause problems like race conditions since they are hard to find in complex applications. High-level approaches to parallel programming are commonly used in the high-performance computing (HPC) community to circumvent these issues [27–29]. For developers to easily put together a parallel application, the programming environment must provide them with access to high-level parallel constructs. This increases the level of abstraction which simplifies programming:

With only an abstract, high-level view of the parallel program, the programmer is free to focus on the computational aspects of the code while leaving the most important implemen-

tation decisions to the programming tool and runtime assistance. In addition, this will make the program's architecture portable, allowing for consistent performance expectations across platforms.

We argue that the SPODS setting might benefit from a structured parallel programming (SPP) method [30]. Structured parallel programming's fundamental concept is to allow the developer to specify an application in terms of preexisting parallel patterns (also known as paradigms). Parametric implementations of communications and computation patterns are well-defined, as are the parallel patterns that arise in the realization of many real-world algorithms and applications. The programmer needs only choose the appropriate pattern and then describe the sequential code when using parallel paradigms (ex: OpenMP[31], Openacc, SYCL[32]). The IDE and runtime automatically generate the remaining code. Once a bottleneck function has been identified, a suitable parallelization can be found by exploring a small, manageable group of options. The ability to accurately predict and measure the overall performance provided by the parallel computation across a range of execution and environmental conditions is a key feature of this method. Performance portability across different architectures relies on these characteristics as their foundation. Additionally, SPP makes it easier to think about the features of a parallel solution with respect to throughput, latency, and memory occupancy, in addition to reducing the effort and complexity of parallel programming. To put it simply, this is what intra-function parallelism requires [26].

When it comes to high-level parallel models, structured parallel programming is by far the most interesting [26], [33], [34],[35]. Academic institutions and HPC communities have been using it extensively for quite some time. Based on the foregoing, we think that parallel patterns are a viable option for achieving intra-function parallelism in SPODS applications.

1.5 Contributions of the thesis

Dynamic parallel strategies and reconfiguration mechanisms with well-known properties of stability, overall performance, energy-efficient and low latency are still lacking in today's approach to the development of SPODS applications. To speed up the parallel implementation of SPODS functions, we follow existing parallel patterns in the parallelization process. We use existing parallel frameworks like OpenMP to deliver easily parallel SPODS application. With these aims in mind, the thesis's research contributions can be summed up. The following are the main results:

- Finding patterns of parallelism through the identification of high bottleneck computation functions.
- An extensive literature review and a research proposal for a parallel-based strategy to provide reasonably high performance guarantees, either in terms of throughput or latency.
- The use of a shared-memory architecture to realize and implement the proposed parallel SPODS algorithm called Parallel Range Search Skyline PRSS [36].
- To evaluate the efficacy of our proposed method, extensive experimental setups on a shared memory architecture has been evaluated.

1.6 Outline of the thesis

This thesis has four main sections, accompanied by an introduction. The second Chapter addresses the concept of data stream processing. Chapter 3 focus on the concept of skyline query

and some existing skyline techniques. Chapter 4 focus on developed skyline query variations and parallel algorithms and investigating strategies and reconfiguration mechanisms for parallel SPODS patterns. In Chapter 5, we will go over our developed parallel skyline algorithm PRSS [36]. We argue for a more high-level approach to parallel application development. This chapter provides a brief overview of the PRSS algorithm. Then we provides an in-depth experimental analysis of the proposed skyline method over a real and synthetic public available data, evaluating its applicability, performance impacts, and drawbacks in the context of a multi-core, shared-memory architecture. Finally, We presents the thesis's conclusions and depicts potential future works through experimental evaluation of the proposed solutions on shared memory architectures.

Part II

Background on Data Stream Processing

CHAPTER

2 BACKGROUND ON QUERY PROCESSING WITH DATA STREAM

“Only dead fish go with the stream.”
MALCOLM MUGGERIDGE

Chapter content

2.1 Characteristics of a *SPODS* application **17**
2.1.1 Function state 18
2.1.2 Windowing approaches and State type 19
2.2 Data Stream Processing systems **20**
2.2.1 Data Stream Management Systems 21
2.2.2 Complex Event Processing systems 22
2.2.3 Stream Processing Engines SPE 22
2.3 Parallelism exploitation in SPODS systems **23**
2.3.1 Parallelism in QPODS systems 23
2.3.2 Literature approaches 24
2.4 Conclusion **25**

The fundamental ideas behind Skyline Processing over Data Stream (*SPODS*) applications and current frameworks are introduced in this chapter. The first section focuses on the key attributes of a *SPODS* application; we will introduce the terms stream and tuples as well as the key characteristics of stream processing functions, such as the existence of internal state and the notion of windowing. The programming environment, in terms of functionality, functions, and features that they provide to programmers, will be a major focus of the second part of our review of various SPODS frameworks. We’ll talk about how parallelism is used in related algorithms. This will provide a clear starting point for this thesis work by outlining the features and restrictions of the current algorithms as well as research methodologies in the field.

2.1 Characteristics of a *SPODS* application

Data is continuously and theoretically infinitely entering from external (remote) sources into a SPODS application. In order to give users prompt responses, the business logic of the applica-

tion has to handle the data in real time as it comes in.

A typical representation of a SPODS application is a direct graph with functions as its vertices and streams as its arcs [10]. The application consumes and analyzes a series of individual data items from the input stream that reflect events that occurred or, in general, useful data then outputs the result to assist the user in decision making. A tuple is the basic, or atomic, data item included in a data stream and handled by the application. The term ”tuple” will be used here to refer to input data. A group of named and typed attributes (or fields) make up its structure such as temperature, pressure, price, fuel, noise, distance,..., Each occurrence of an attribute has a corresponding value. In this way, we can think of a data stream as an endless sequence of tuples that share a common schema.

In many data stream application a physical input stream carries multiplexed tuples from different logical substreams [37]. It is typical to take into account a key attribute of the tuple in order to determine the correspondence between tuples and substreams [26]. The stock symbol to which this information refers, for instance, can be a key in trades and ticket prices sent from flight markets [38]. Similarly, in networking management, a stream can be divided into sections based on IP addresses. Different applications must independently compute on each substream based on the multiplexed streams. Referring to the earlier examples, it might be necessary to analyze data by IP source or destination or perform pattern recognition independently for each stock on the financial market [39].

In most cases, a function takes the input tuples and uses them to apply a transformation in accordance with its internal processing logic. Because of this, the function may emit fresh tuples as new streams, possibly with different schemas. The selectivity of the function is measured by how many output tuples are generated for every tuple that is consumed.

2.1.1 Function state

Multiple categories exist for stream intermediate functions. Based on the work of Andrade et al. [10], we will divide them into three categories with regard to state management:

- A stateless function is one that processes data tuple by tuple without saving any intermediate results or retaining any data structures produced by the computation. Examples of stateless functions include selection, filtering, and projection, in which the processing of one tuple does not affect the processing of any preceding tuples;

- A stateful function, on the other hand, keeps and updates an internal data structure while operating on input data. The current value of the internal state affects the outcome of the processing logic inside. Examples include sorting, joining, and the Cartesian product;

- Lastly, a keyed stateful (or partitioned) function is a crucial category of stateful function. For a keyed stream, the function performs the same calculation separately on each substream. Because of this, the data structures that maintain the internal state are partitioned off in their own places based on the relative substream.

Given the inherent difficulty that can arise when designing a parallel implementation of a stateful or partitioned stateful function, these two types of functions are the most interesting to study.

2.1.2 Windowing approaches and State type

The unbounded nature of input streams makes it impractical for a stateful function to keep track of the entire stream history. There are two approaches to fixing this issue:

- Aggarwal and Yu [40] suggest using clear data structures like synopses, sketches, histograms, and wavelets to implement the state and keep track of aggregate summary metrics. In this case, we don't keep track of tuples separately.
- Tuples must be kept in a single internal state for a variety of applications. Fortunately, in many applications, the importance of each tuple decreases over time, and recent data is where the focus should be. To solve this problem, the state can be implemented as a window buffer, where only the most recently used tuples are stored [41].

Windows are the most common abstraction for implementing SPODS functions' internal state, and many frameworks offer windowed functions. The meaning of a window can be defined in various ways. We will divide windows into two categories [10]:

a) based on what kinds of tuples the buffer can store (or the eviction policy);

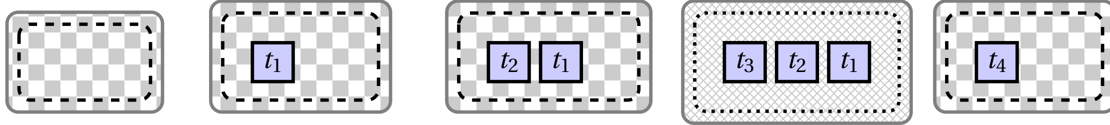
b) depending on the windows' behavior when moving over new tuples (triggering policy).

The first criterion allows for the classification of policies into many categories. Both count-based and time-based windows are extremely common. If *N* is a constant, then the window will always contain *N* consecutive stream elements, such as "the last 1000 received tuples," at any given time. The elements received in the last 30 seconds would be part of a time-based window, which contains all the elements received in the interval T. Due to the fact that the number of active tuples may change over time, the window size is not a constant. Time-based windows require each data stream element to have a timestamp attribute, either automatically from the data source or manually when it reaches the application's boundaries. Delta-based windows are less common [10], and they are defined by a delta threshold value and a tuple attribute named the delta attribute. The threshold value determines how much deviation there is between the oldest and newest tuples in the window. For instance, we can define a delta-based window that "contains all the tuples having a timestamp within one minute, one from the other," by using a timestamp attribute of the tuple.

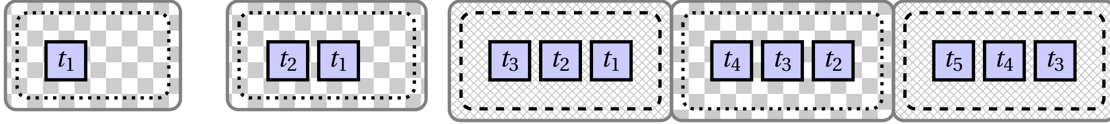
The second criterion for categorization is the window's movement. Tumbling and sliding windows are both possible. When a tumbling window is completely stuffed, the data in it is ready for analysis. Figure 2.1a depicts the eviction of all tuples from windows once processing of those windows is complete. Consequential activations of the function logic will apply to distinct tuples in this manner [10].

The most recent tuples are kept up-to-date in a continuously sliding window, however. Only the oldest tuples are removed from the window when it is full. Delta δ, a sliding factor, expresses when the function's algorithm processes the window's content, for example, upon the arrival of Delta δ tuples (Figure 2.1b). In count-based sliding window the window slides through the stream of data, always maintaining the most recent tuples for skyline computation. In this approach, the window moves after a fixed number of tuples have been processed. As new tuples arrive, the oldest ones are removed once the count limit is reached. The dynamic nature of this window ensures that only the most recent set of tuples is analyzed as illustrated in figure 2.2a.

In time-based sliding windows, tuples expire over time, which means that the function's logic may be invoked independently of a tuple's arrival, depending on the implementation. Count-based sliding windows and time-based sliding windows are the most common, but all four combinations of the previous characteristics are useful. Tuples enter and exit the window

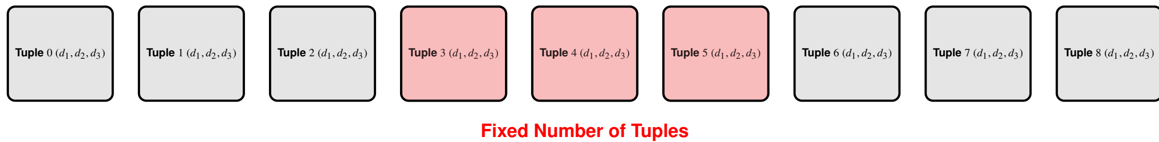


(a) *The evolution of a count-based tumbling window with size = 3. Numbers illustrate the order of incoming tuples [10].*

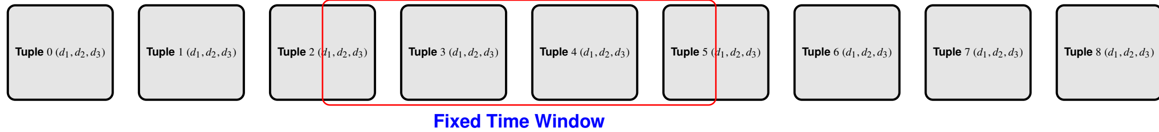


(b) *The evolution of a count-based sliding window with size = 3 and sliding factor = 2 [10].*

Figure 2.1: *Sliding window vs Tumbling window.*



(a) *Count Based sliding window*



(b) *Time based Sliding Window*

Figure 2.2: *Count Based sliding window VS Time based Sliding Window.*

based on time rather than count. As time progresses, old tuples expire, and new ones are included in the window. Unlike the count-based approach, the window continuously slides over time, ensuring that only the most recent tuples within the time window are considered as illustrated in figure 2.2b.

2.2 Data Stream Processing systems

Coding from scratch has always been the go-to method for fixing issues with data stream processing. The inflexibility, high cost of development and maintenance, and slow response time to new feature requests are the most obvious drawbacks of this strategy.

The early twenty-first century saw the re-branding and marketing of several established software technologies, including main memory database management systems and rule engines [38], to this market. Modern and generalized SPODS systems owe a great deal to the innovations and proposals made in these areas [10].

The following discussion involves a retrospective analysis of the evolution of data processing systems, tracing the trajectory from the initial Data Stream Management Systems (DSMS) and Complex Event Processing (CEP) systems to the present-day Stream Processing Engines (SPE). The present study will center on the programming environment, encompassing the various high-

level functionalities that are made available to programmers for the purpose of composing their applications. Additionally, attention will be directed towards the runtime, which refers to the underlying infrastructure that furnishes the necessary mechanisms and support for executing an SPODS application. Figure 2.3 demonstrates the various systems that the academic, open-source, and business communities have developed in recent years. This emphasizes that the subject matter is not only a captivating and demanding area of research but also holds significant strategic value for applications that rely on data and communication.

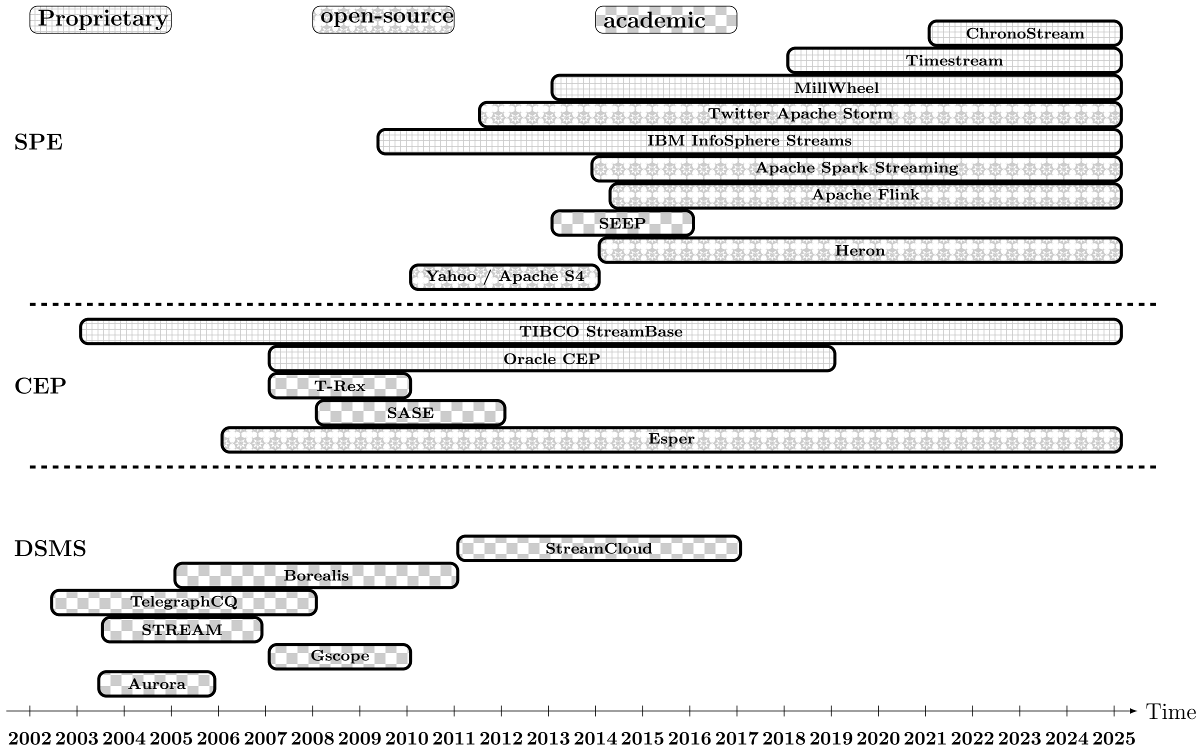


Figure 2.3: *History of various DaSP Systems.*

2.2.1 Data Stream Management Systems

Conventional database management systems (DBMS) are constructed with a durable storage mechanism that houses every relevant piece of data and receives updates that are comparatively uncommon. These systems are not optimized for efficient and consistent loading of individual data or prompt query responses [26]. The database community developed a novel class of systems with the goal of quickly and effectively handling large data streams in order to address the aforementioned constraints. This category of systems is known as Data Stream Management Systems" (DSMS). Distributed Stream Processing Systems (DSMSs) are distinguished from conventional Database Management Systems (DBMSs) due to their specialization in handling dynamic information that receives constant updates. The system is capable of performing continuous queries that operate incessantly and supply revised responses as new data is received. The users are not required to explicitly request updated information, as the system proactively notifies them in accordance with the installed queries. Distributed Stream Processing Management Systems (DSMSs) provide software developers with a declarative language similar to SQL for the purpose of specifying continuous queries.

Numerous software infrastructures for DSMS have been proposed since the early 2000s.

TelegraphCQ [42], STREAM [43], and StreamCloud [44] are among the frameworks that exemplify this category. Certain tools have been created with a specific focus area in mind, such as Gigascope (AT&T Labs) [45], which was developed specifically for the purpose of analyzing network traffic. Of the proposals presented, Aurora and Borealis are deemed to be the most significant. The collaborative development of Aurora [46] involved the participation of Brandeis University, Brown University, and MIT. The programming paradigm of the system was established in a graphical programming language known as SQuAl (Stream Query Language). The inheritance of SQUAL operators by relational algebra is a notable feature. These operators can be categorized as either single-tuple operators or windowed operators. Aurora boasts a comprehensive runtime system, which encompasses a scheduler, a quality of service (QoS) monitor, and a load shedder. The scope of the project has been expanded to explore the domain of distributed processing utilizing the Medusa framework, as documented by [46]. The exchange of information among processors occurs through a distributed data transport layer that involves flexible associations between operators and flows. The Borealis project, as documented by [12], incorporated the capabilities of the aforementioned projects while also introducing novel functionalities, including the capacity for dynamic deployment and adaptation of the application. Although their progress was impeded in the past, the Aurora and Borealis systems introduced several innovative concepts that have since been integrated into contemporary commercial SPODS (Stream Processing on Demand Systems) platforms.

2.2.2 Complex Event Processing systems

Event-driven systems are common in the IT industry [47]. All incoming data is treated as a notification of some external event that must be sorted and combined in order to make sense of it. One system that could prove useful in implementing such applications is a complex event processing (CEP) system, which takes as input a number of streams of primitive events and analyzes them to look for patterns that represent complex events whose occurrence necessitates notification to interested parties. These programs are essentially pattern-matching engines that check incoming events against an established set of rules.

From the academic community (such as Sase [48] and T-Rex [22]) to businesses (such as Oracle CEP [49] and Tibco Streambase CEP [50]) and the open-source community (such as Esper [51]), many CEP systems have been proposed, and many are still available and in production. All of them place a strong emphasis on rule-based detection tasks expressed in terms of complex patterns and provide a windowed operator to enable the programmer to specify constraints in terms of time or number of events for the validity of a match. However, the language, which forbids altering incoming data, as well as their inability to handle unstructured data or complex calculations beyond rule-based logic, place limitations on them.

2.2.3 Stream Processing Engines SPE

DSMSs and CEP engines offer concise and easy-to-implement solutions to many streaming problems, but the operational flow of an imperative language is better suited to expressing complex application logics. The need for greater latitude in application definition has led to the development of more generic frameworks, such as the Stream Processing Engine (SPE). Operators can contain user-defined code and be composed in any way [5]. At the same time, they provide a full runtime environment for the setup and maintenance of programs, usually on standard clusters. We will now examine a selection of these that together represent the cutting edge of this research area.

1. Apache Kafka Streams: A stream processing library and framework that is part of the Apache Kafka ecosystem. It provides a simple and scalable approach to process and analyze data streams in real-time [21].

2. Apache Flink: An open-source stream processing framework with strong event time support, fault-tolerance, and high throughput. It offers a rich set of APIs for building real-time applications and supports batch processing as well [24].

3. Apache Storm: A distributed real-time computation system. It provides a fault-tolerant mechanism for processing streams of data with low-latency and high throughput [23].

4. Amazon Kinesis Data Streams: A fully managed service by Amazon Web Services (AWS) for real-time data streaming and processing. It enables you to ingest, process, and analyze streaming data at scale [52].

5. Apache Spark Streaming: A component of Apache Spark that enables scalable, high-throughput, and fault-tolerant stream processing. It integrates with the Spark ecosystem and allows for the combination of batch and streaming processing [53].

6. Google Cloud Dataflow: A fully managed service on Google Cloud Platform (GCP) for both batch and stream processing. It offers a unified programming model for developing data processing pipelines [54].

7. Microsoft Azure Stream Analytics: A real-time analytics service provided by Microsoft Azure. It enables processing and analyzing streaming data from various sources, including IoT devices, logs, and social media feeds [55].

8. IBM Streams: A platform for developing and executing streaming applications. It provides a high-performance runtime environment for processing continuous data streams in real-time [56].

2.3 Parallelism exploitation in SPODS systems

To achieve the typical performance requirements, SPODS applications must utilize parallel systems [57]. This thesis takes a methodical approach to the topic of intra-function parallelism where the intra-function of the skyline computation parallelized on a shared memory machine. We will describe and analyze the ways in which SPODS algorithms currently make use of this kind of parallelism. Existing frameworks typically use overly simplistic representations of intra-function parallelism. Assigning input tuples to the replicas in a load-balanced fashion is the most common approach for stateless ones that involve replicating the function multiple times according to a parallelism degree. While there are many proposed solutions for the unkeyed case, the parallel solution for partitioned stateful functions is to use replicas, each of which works on a subset of the keys.

2.3.1 Parallelism in QPODS systems

Most modern systems (at least in theory) make quasi-transparent use of parallelism in QPODS applications. One common method for accomplishing this is through the use of parallel execution of certain high-level programming constructs.

1. Storm: A programmer can specify the level of parallelism (parallelism hint) they want Storm to use, and the system will then replicate bolts in accordance with that setting [58], [59]. Tuples can be partitioned among multiple functions of an operator to express various parallelizations. A round-robin distribution, the result of shuffling, is practical for a stateless function. However, tuples with the same key are always sent to the same replica using a field grouping distribution, which is practical for keyed functions. The programmer can define new patterns by using arbitrary grouping. The programmer must fully specify the nature of cooperation between the function's replicas if they are to interact with one another.

2. Spark Streaming: Each function in Spark Streaming is converted into a series of tasks with the correct precedencies and dependencies that are carried out as soon as possible on the available resources [60], [61]. This conforms to the user constraint that states that window-related functions must be associative. The level of parallelism here is also programmer-controlled.

3. IBM Infosphere: Developer-defined parallelism is available in IBM Infosphere Streams. A function's replica count (or channel count, in their lingo) is an annotation posed at the beginning of the function that specifies how many copies of the function must be created [34]. Using a splitter, developers can divide the streaming records using either round-robin or hash-based algorithms for keyed streams. The level of parallelism is a compile-time option. In addition, programmers can optimize application performance by mandating that parallel channels execute on separate hosts.

4. Flink: Operations on streams are decomposed into tasks in Flink, which are then placed in task slots and run as quickly as possible using the available resources [4]. There is only one task per key when evaluating a window on a keyed stream in parallel. When evaluating an unkeyed stream, a windowed function only evaluates one task at a time, resulting in sequential execution.

2.3.2 Literature approaches

The parallelization of specific computations is at the center of much of the study of the topic. Several works in the field of parallelizing continuous queries concentrate on the parallelization of a specific relational operator. Some specific applications include parallelizing join evaluation over windows [[62]; [63]; [64]] and skyline queries over windows [Lu et al. [65]; [41], [66], [67], [68]]. A quite general approach is taken in Streamcloud [44], where the programmer indicates continuous queries that are then automatically parallelized for a shared-nothing execution environment. Even for stateful functions like windowed, Streamcloud allows for intra-function parallelism. However, ad hoc solutions are put forward because of the function's restrictions (e.g., aggregates, joins, and cartesian products). Functions are replicated, and tuples are carefully distributed and collected while taking into account the semantics of the function.

Recent works in complex event processing have focused on parallelizing single rules. To facilitate the execution of rules in parallel over keyed streams, the author of [69] suggests extending the IBM SPQ language. Normal parallel processing for pattern evaluation on events with different keys is possible. For unkeyed streams, other methods attempt to parallelize rule evaluation. In [70], an evaluation algorithm is proposed that is parallelized on a GPU for windowed matches, while in [71], the parallelization focuses on commodity multicore processors.

In Wu et al. [72], the authors discuss a more general strategy for the parallelization of stateful functions in SPODS systems. To enable parallelization, we propose a distributed shared state mechanism. Tuples are typically distributed in a round-robin fashion among the available function replicas. Access to the shared state may need to be synchronized (protected by locks) because multiple replicas can run in parallel. Also included is a theoretical model for figuring out how much parallelism is optimal. This method does not effectively manage the case of partitioned stateful parallelism, and it requires the explicit use of synchronization on shared state, which can have a negative impact on performance. Schneider et al. [73] take on the same issue, but they focus on a different aspect of the problem: the generic stateless or keyed stateful function with dynamic selectivity. The proposed solution is similar to those in Storm and IIS, but in this case, the ordering of the output is also considered. While these two methods are useful for parallelizing generic states and windows, they still require sequential execution over individual windows.

Balkesen and Tatbul [74] is one of the few methods that addresses the issue of parallelizing the computation of aggregate computation on a single window. Each window is separated into non-overlapping, contiguous partitions called panes, a concept introduced by the authors in [75]. It is possible to compute sub-aggregates in parallel across panes and then aggregate the results. This obviously necessitates the computation of certain properties of the function over the window.

2.4 Conclusion

In this chapter, we provided an overview of query processing in data stream environments, emphasizing the characteristics and challenges of Stream Processing Over Data Streams (SPODS) applications. We explored key aspects such as function state management and various windowing approaches, which play a crucial role in handling continuous data streams effectively.

Furthermore, we reviewed different data stream processing systems, including Data Stream Management Systems (DSMS), Complex Event Processing (CEP) systems, and Stream Processing Engines (SPE). Each of these systems offers unique capabilities for real-time data analysis, enabling efficient handling of high-velocity data streams.

To address the growing computational demands of stream processing, we discussed parallelism exploitation in SPODS systems. We examined parallelism strategies within Query Processing Over Data Streams (QPODS) systems and highlighted key literature approaches that enhance performance by leveraging parallel architectures.

Overall, this chapter established a strong foundation for understanding the fundamental principles of data stream processing and its parallel execution. The insights presented here serve as a basis for the subsequent chapters, where we delve deeper into optimizing skyline query processing in multicore architectures.

Part III

Skyline Queries

CHAPTER

3 CONTINUOUS SKYLINE QUERIES

"We all make decisions, but in the end, our decisions make us."

KEN LEVINE

Chapter content

3.1 Entities and Attributes . . . 30

3.2 The Concept of Dominance . . . 31

3.2.1 Skyline Queries . . . 32

3.2.2 Main-Memory Computation (in-core skyline queries) . . . 33

3.2.3 Algorithms for Secondary Memory (out of core skyline queries) . . . 34

3.3 Advanced Skyline Processing . . . 34

3.3.1 Skylines in Dynamic Environments . . . 34

3.3.2 Distributed and Parallel Techniques . . . 40

3.3.3 A High-Level Approach to Parallel Programming . . . 42

3.3.4 Parallel Paradigms for Skyline Queries Over Data Stream (SPODS) . 44

3.3.5 Parallel Patterns for Windowed Functions . . . 53

3.3.6 Parallel Patterns Taxonomy . . . 53

3.3.7 Categories of Parallel Patterns for Windowed Functions . . . 53

3.3.8 Pane Farming . . . 55

3.3.9 Window Partitioning . . . 56

3.4 Skyline Cardinality . . . 58

3.5 Conclusion . . . 63

In this chapter, we'll establish the foundation for understanding some fundamental concepts in skyline-based query processing. It defines some important terms and provides an overview of the background to the principle of dominance. We examine dominance from the point of view of a database.

3.1 Entities and Attributes

In real-world applications, entities are typically modeled as objects with associated attributes. For example, a drone can be described by characteristics such as flight time, cost, and camera quality. When comparing two drones, d_1 and d_2, the selection process is straightforward if based on a single criterion. For instance, if price is the only factor considered, the optimal choice would be the drone with the lowest price, assuming all other factors remain equal.

However, solely prioritizing cost can lead to suboptimal choices. For example, the least expensive drone might have a 10-megapixel camera and weigh 5 kg, which may not align with user requirements. This limitation arises because price was the only criterion used to assess quality. As more attributes are introduced into the decision-making process, selecting the best option becomes increasingly complex.

The number of dimensions corresponds to the number of attributes, and objects are represented as points in this space. For simplicity, all attributes are assumed to have numeric values (integers or floats). Thus, an object can be represented as a point $t = (t.x_1, t.x_2, \ldots, t.x_d) \in \mathbb{R}^d$, where d is the number of dimensions and x_i is the value of the i-th dimension. However, in real-world applications, attributes may belong to different categories, requiring distinct treatment.

The number of attributes associated with objects can vary significantly depending on the application. Generally, problems are easier to solve in low-dimensional spaces than in high-dimensional ones. Many computational geometry problems can be solved in $O(n \log n)$ time in two-dimensional space, but their complexity increases exponentially with the number of dimensions, a phenomenon known as the *curse of dimensionality* [76]. Despite the presence of many attributes, users often focus on only a subset of them. For instance, a drone may have numerous characteristics, yet a specific user might only be interested in its cost and storage capacity, effectively reducing the problem to a two-dimensional space.

A common approach to handling high-dimensional data is to use a user-defined ranking function (also called a scoring function) to map high-dimensional objects to a single value [77]. This function determines the quality of an object. The simplest implementation considers all or a subset of the attributes and returns a value between 0 and 1, where values closer to 1 indicate high-quality objects, and values closer to 0 indicate low-quality ones.

The ranking function is assumed to be monotonic with respect to attribute values, meaning that if an attribute value improves, the overall score of the object also improves [77]. A straightforward example of a ranking function is the sum of attribute values. However, since summing values from different attributes can introduce bias, a weighted sum is typically used instead. Specifically, a weighted-sum ranking function can be defined as [77]:

$$F(t) = \sum_{i=1}^{d} w_i \cdot t.x_i, \quad F(t) \in [0, 1] \tag{3.1}$$

where w_i represents the weight assigned to attribute i, reflecting its importance in the ranking process.

Defining a ranking function can be challenging, especially when dealing with different attribute types or a large number of objects. Moreover, since ranking functions are typically user-defined, different functions may lead to different outcomes. This raises an important question: *What can be done if a ranking function does not exist?* The subsequent discussion will address this issue in detail.

3.2 The Concept of Dominance

Users' preferences can be expressed through a ranking function or through a directive to maximize (have the highest possible) or minimize (have the lowest possible) the attributes of the objects returned by the query. We focus on the second class of algorithmic techniques that make use of the concept of dominance to provide a natural way of ranking without requiring an explicit user-defined ranking function. To dominate signifies that one thing t is superior to another thing q. In this case, we use multi-dimensional points to symbolize three-dimensional objects. Without limiting ourselves too much, we'll say that small values are better in every dimension. It's important to keep in mind that even if larger values are preferred in some situations, we can often reframe the issue so that smaller values are preferred.

Imagine a passenger searching for an optimal flight ticket based on two main criteria as illustrated in figure 3.1:

- **Price** (p): Lower is better.
- **Flight Duration** (d): Shorter is better.

Airlines offer different flight options, some with layovers and others direct, leading to a diverse set of choices.

We represent each flight option with an airplane icon as a tuple $t_i = (p, d)$ where:

- p represents the price of the ticket.
- d represents the total flight duration.

Consider the following flight options:

A tuple $t_a = (p_a, d_a)$ dominates another tuple $t_b = (p_b, d_b)$ if:

1. $p_a \le p_b$ and $d_a \le d_b$ (better or equal in all attributes).
2. At least one of these inequalities is strict: $p_a < p_b$ or $d_a < d_b$.

Applying this rule:

- t_7 dominates t_1, t_2, t_3, t_4, t_5 because it is cheaper and/or has a shorter or equal flight duration.
- t_6 is not dominated by t_7.

Thus, the **Skyline Set** (Pareto-optimal flights) consists of t_6 and t_7.

The **dominance region** of a tuple t includes all tuples that it dominates. The dominance region of t_7 consists of flights that are:

- More expensive.
- Longer in duration.

This means that t_7 creates a dominance region including t_9, t_2, t_3, t_4, t_5, meaning these flights are **suboptimal choices**.

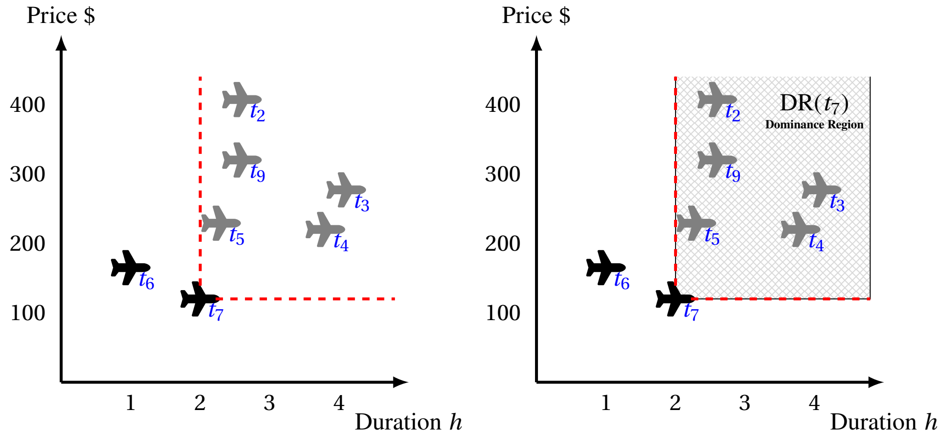


Figure 3.1: *The concept of dominance.*

3.2.1 Skyline Queries

Since Borzsonyi's [78] development of the skyline operator, this approach to extracting interesting objects from multi-dimensional datasets has become a major issue in database research. The Skyline operator's widespread effectiveness in fields as wide-ranging as quantitative economics, market research, environmental monitoring, data mining, and visualization is largely responsible for its enormous reputation. In such contexts, the database tuples are considered a set of multidimensional data points, and the skyline query determines the points that are the best choices between the different dimensions without using cumulative functions to identify the best results but instead based on the user's preferences.

Table 3.1: *Summary of Notations.*

Notation	Meaning
$\mathcal{T}$	Input data stream for processing skyline
$\mathcal{N}$	Number of dimensions of $\mathcal{T}$
A	$N-dimensional$ index
A_i	one dimensional index $(1 \leq i \leq \mathcal{N})$
Z	the size of streaming window (count or time based window)
x_1, x_2, x_3	tuples in the streaming data
s	Skyline tuple
v_x^i	dimensional value of the incoming tuple x on dimension i.
$M_{\mathcal{T}}$	Set of skyline tuples of $\mathcal{T}$
$x_1 \prec x_2$	x_1 dominates x_2
$x_1 \nprec x_2$	x_1 does not dominate x_2
$x_1 \nsim x_2$	x_1 is incomparable to x_2

Definition 1 (Dominance [79])**.** *Given x an $\mathcal{N}-$dimensional tuple, let's denote the value of the tuple x at the dimension i by $x[i]$ where $1 \leq i \leq \mathcal{N}$. let x and x' two tuples, $x[i] \prec x'[i]$ denotes that $x[i]$ is better than $x'[i]$ and $x[i] \preceq x'[i]$, it means, $(x[i] \prec x'[i]) \vee (x[i] = x'[i])$, Denotes that $x'[i]$ is not worse than $x[i]$. Therefore, $x[i] \prec x'[i] \Rightarrow x[i] \preceq x'[i]$.*

A tuple x dominates a tuple x', represented by $x \prec x'$, iff for every dimension $1 \le i \le \mathcal{N}$, we get $x[i] \preceq x'[i]$, and at a minimum one dimension $1 \le m \le \mathcal{N}$, we get $x[m] \prec x'[m]$. let x and x' two tuples, we represent $x \nprec x'$ such x does not dominate x', and we represent $x \nsim x'$ namely x and x' are incomparable, it means, $(x \nprec x') \wedge (x' \nprec x)$. We expand $\prec, \nprec, \nsim$ to sets of tuples, similarly, $x \prec X$ denotes that the tuple x dominates every tuple in X, $x \nsim X$ denotes that the tuple x is incomparable with every tuple in X.

Tuple ID	N cores	RAM (gigabytes)	Storage (gigabytes)	Camera (megapixels)	Battery (mAh)	Screen (pixels)	Skyline
$Phone_0$	8	13	220	33	3600	210	Yes
$Phone_1$	2	31	330	41	2200	310	Yes
$Phone_2$	16	58	310	36.1	4800	550	No
$Phone_3$	4	27	330	38	4000	550	Yes
$Phone_4$	8	22	290	35	2700	550	Yes
$Phone_5$	64	51	350	33	5500	420	No
$Phone_6$	4	44	190	37	1700	180	Yes
$Phone_7$	4	44	330	39	4500	670	No
$Phone_8$	6	34	370	42	3100	620	No
$Phone_9$	16	62	260	40	6100	450	No

Table 3.2: *A sample database with $A = 6$, $Z = 10$, and $m = 5$.*

Table 3.2 shows a sample sliding window of 6 dimensions ($A = 6$) that includes 10 tuples ($Z = 10$), of which 5 are skyline tuples ($|\mathcal{M}| = 5$) while the greater than order is applied to all dimensions.

Example 1. *from the set of 10 tuples $phone_0, phone_1, ..., phone_9$ in table 3.2, $phone_1 \prec phone_8, phone_4 \prec phone_2, phone_6 \prec phone_7$, $phone_6 \prec phone_9$ and $phone_0 \prec phone_5$, and no tuple dominates $phone_3$, and $phone_3$ does not dominate any tuples. Therefore the Skyline is $\mathcal{M} = \{phone_0, phone_1, phone_3, phone_4, phone_6\}$.*

Definition 2 (Icomparability [79])**.** *Given two $\mathcal{N}-$dimensional tuples $x, x\prime \in \mathcal{T}$, it is said that x and $x\prime$ are incomparable on $\mathcal{T}$, that is, if x and $x\prime$ do not dominate each other simultaneously. denoted as $x \nsim x\prime$, iff $x \nprec x\prime$ and $x\prime \nprec x$.*

This property facilitates checking if one or more tuples can be Skyline tuples. A tuple in the skyline set needs to be incomparable to all other tuples in the set.

Definition 3 (Continuous Skyline [79])**.** *Given $\mathcal{T}$ a $\mathcal{N}-$dimensional dynamic dataset, a tuple $x \in \mathcal{T}$ is a skyline tuple iff $\nexists\ x' \in \mathcal{T}$ where $x' \prec x$. The skyline $\mathcal{M}$ of dynamic dataset $\mathcal{T}$ is the total set of skyline tuples such that, $\mathcal{M} = \{x \in \mathcal{T} \mid \nexists\ x\prime \in \mathcal{T}, x\prime \prec x\}$.*

The Skyline algorithms have proven to be powerful queries in related database development studies after being the subject of extensive study and different applications such as environment monitoring [80], IoT [14], Aviation industry [81, 82] Social Media analisys [21]. Generally, the introduced skyline techniques can be grouped into algorithms that extract the skyline of a dataset $\mathcal{T}$ that can be stored in main memory(in-core algorithms) and algorithms that use secondary memory(out-of-core) [77].

3.2.2 Main-Memory Computation (in-core skyline queries)

queries to identify the skyline when the dataset can be contained in main memory. Main memory algorithms, also known as in-core algorithms, are those developed specifically for processing data that can be loaded entirely within a machine's main memory simultaneously such as

the brute-force BF skyline and the Sort-based SB skyline algorithms presented in [77]. Obviously, query processing can be done more efficiently if the entire data set can be stored in main memory [83]. But generally, the dataset is much bigger than the volume of main memory that is commonly available [84]. Therefore, effective methods are needed to obtain the skyline for data stored on disk [78].

3.2.3 Algorithms for Secondary Memory (out of core skyline queries)

Although it is preferable to launch skyline processing in main memory, there are cases in which this is not practical [84]. Since modern applications typically utilize very sizable datasets, such datasets would require more storage than is available in the machine's main memory. Because of this, there is a need for algorithmic ways to compute skylines that can be used on data sets that are stored on disk [78]. The primary solution does not require any unique indexing methods. Many datasets, for example, are collected and stored without ever being indexed. The second solution is the use of methods that rely on a particular indexing mechanism for points in multiple dimensions [77]. In fact, indexes are generally widely accessible and can be employed to speed up skyline computation [85].

- **Index-Free Techniques**: Methods that do not require to index the set of tuples $\mathcal{T}$ are called "index-free" BNL and $D\&C$ [78], $Iskyline$ [86], VP and $KISB$ [87], and Object Space Partitioning (OSP) [88]. As a result, their cost will exceed that of index-based methods. Such algorithms suffer from the problem of high-cost computations, such as presorting or point-to-point comparisons, which needs to be fixed [77].

- **Index-Based Techniques**: you can quickly get to the data you need using an index without having to search through data that's not related to your objective. It is possible that the running time of these skyline queries could be significantly accelerated by utilizing an index on particular attributes [85]. Algorithms using an $R-tree$ [89, 90], $Btree$ [91], [92], [93], [94] $SkyMap$ a trie-based index [95]. Index-basesd techniques will face challenges of availability for only a specific data type or the cost of the indexing structure that will outweigh its advantages. There are also dominance relation-based skyline query techniques that decrease comparisons between tuples, control the skyline with tree or lattice structures, and utilize the incomparability to skip unnecessary comparison tests $BSkyTree$ [96] and $BJRtree$ [97].

3.3 Advanced Skyline Processing

The literature on the topic is rich in parallel and distributed methods as well as skyline algorithms in dynamic environments, in addition to the aforementioned main-memory and secondary-memory skyline processing methods. Scalable skyline computation depends heavily on the effective use of numerous resources and the careful management of insertions and deletions as illustrated in figure 3.2.

3.3.1 Skylines in Dynamic Environments

The dynamic nature of real-world applications is the norm, not the exception. Datasets are typically dynamic in nature in practical applications, which means that object insertions and

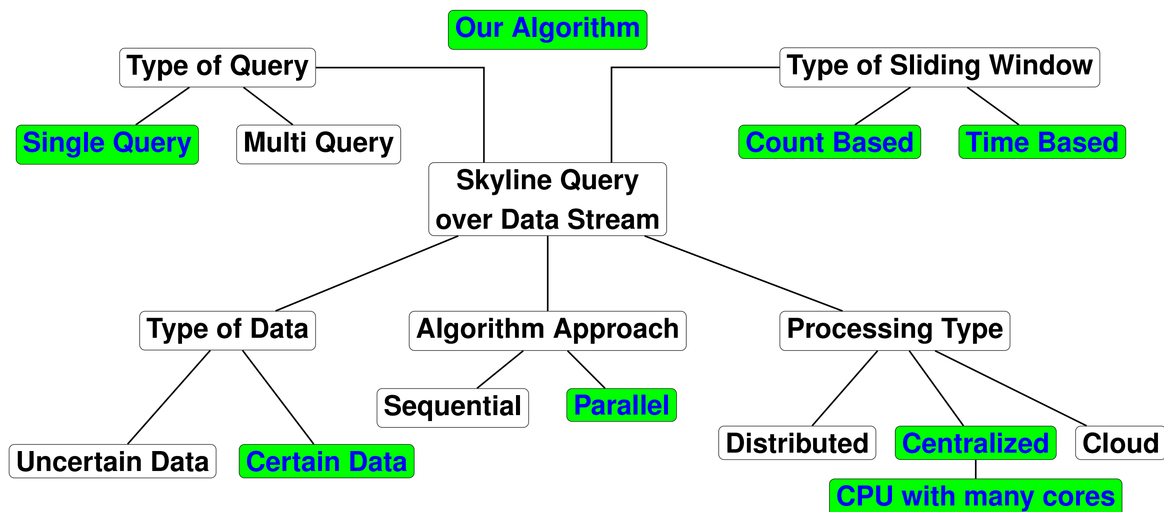
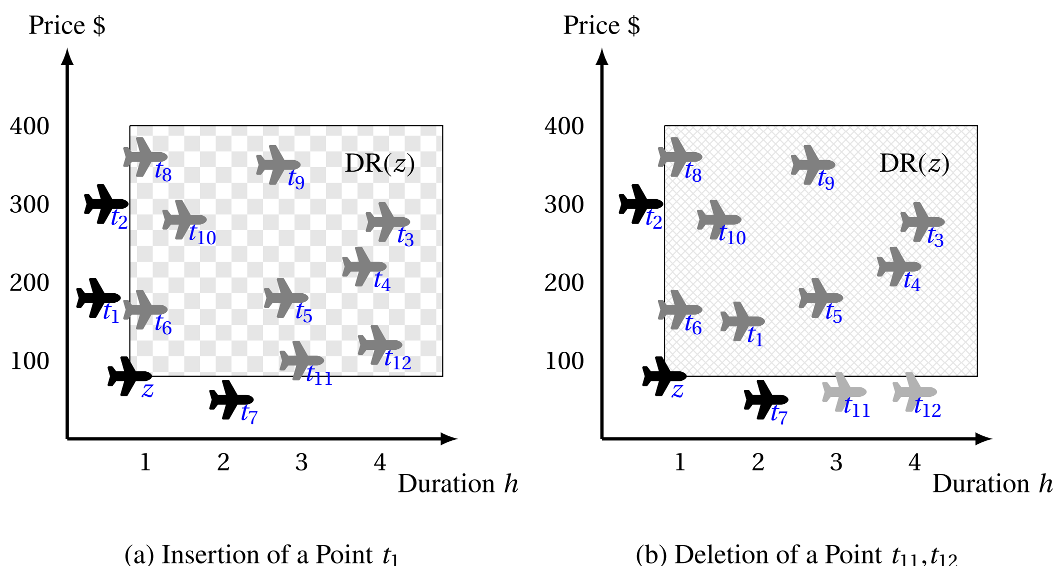


Figure 3.2: *Taxonomy of Skyline query processing over Data Stream.*

deletions can typically occur at will. The following elementary techniques are required for use in a dynamic environment:

Flight data changes over time due to delays, cancellations, and fully booked flights. A continuous skyline query dynamically updates the skyline set when new flights are added or old flights are removed as illustrated in figure 3.3.

- Handling Insertions: When a new flight tuple t_1 arrives, it is compared against existing skyline tuples. If t_1 dominates any skyline tuples, they are removed, and t_1 is added to the skyline set. Otherwise, it is discarded.
- Handling Deletions: If flights t_{11} and t_{12} expire due to cancellations or bookings, they are removed from the skyline. This may result in previously dominated tuples becoming skyline candidates again, requiring reevaluation of the skyline set.



(a) Insertion of a Point t_1 (b) Deletion of a Point t_{11}, t_{12}

Figure 3.3: *Handling Insertions and Deletions.*

Yu et al. [98] propose a window-based algorithm for skyline queries, which splits skyline queries into a large number of dynamic window queries. Lin et al. [99] propose a different sliding window method that uses an efficient pruning technique to cut down on the number of necessary elements. Tao and Papadias [100] also investigate sliding window horizons on data streams, proposing algorithms that continuously monitor data entry and perform incremental skyline maintenance.

In addition to the static dimensions, the dynamic ones are also required for a continuous skyline query. Creating a time-variant, continuously accurate skyline summary is a useful computation over such streaming data sets. In Morse et al. [101], the authors propose a skyline algorithm that works well over a continuous time interval. In Huang et al. [102], they propose using a kinetic-based data structure to simplify the processing of skyline queries for mobile objects.

By using a region-level lattice to determine the cost-optimal pivot point, the $BSkyTree$ algorithm that Lee and Hwang proposed takes into account both dominance and incomparability. The concept of incomparability helps reduce unnecessary comparisons by identifying tuples that neither dominate nor are dominated by each other. As illustrated in figure 3.4, incomparable points are highlighted, illustrating cases where direct comparisons do not lead to dominance relationships. Instead of checking every tuple against all others, skyline algorithms can use this property to:

Partition data into comparable and incomparable sets, skip comparisons between tuples that will never dominate each other and optimize search structures to reduce redundant dominance checks. The whole Partitions 1 and 3 are incomparable in the same way as illustrated in figure 3.4. (1 3). On the other hand, partition 2 can be safely removed if partition 4 is not empty.

Lee and Hwang [103] have proposed two instantiations, namely $BSkyTree{-}S$ and $BSkyTreeP$. These instantiations can be considered optimized algorithms in sorting- and partitioning-based schemes, respectively. $BSkyTree$ employed point-based space partitioning to optimize both dominance and incomparability relationships.

The $BSkyTree$ algorithm is founded on the Divide and Conquer ($D\&C$) paradigm, which involves partitioning the dataset into smaller subsets. The algorithm then proceeds to extract local skylines from each partition and subsequently merges all local skylines to produce the final skyline output.

The $BSkyTree$ algorithm is founded on the divide-and-conquer principle, whereby the dataset is partitioned into smaller subsets. Local skylines are subsequently extracted from each partition, and the resulting skylines are merged to generate the final skyline output.

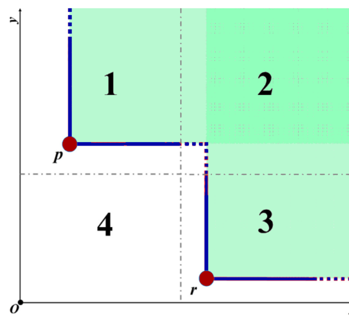


Figure 3.4: *Incomparable Points.*

The utilization of skyline queries is attracting significant interest in facilitating multi-criteria analysis within datasets of substantial size. The computation of a multidimensional subspace

skyline was at the center of Lee and Hwang's [76] study. The authors proposed a method to reduce the full space skyline, thereby enabling users to take into account the subspace skyline that aligns with their preferences. The authors have identified that there is potential for additional enhancements and have suggested a more effective algorithm based on space partitioning for the computation of Skycube, which they have named QSkycube. Skycube and $QSkycube$ are centered on devising effective techniques for computing multiple subspace skyline queries. The Lattice and Skycube data structures are utilized for the purpose of managing complexity. The $QSkyCube$ algorithm employs a hierarchical approach to generate the skyline for individual cuboids, or subsets of dimensions, through the utilization of a tree-based structure. Nevertheless, the size of the lattice may prove excessive when attempting to construct a skycube.

The Skyline query operator has proven to be a valuable tool in demanding applications, particularly in the field of data exploration. Chester et al. [104] conducted an experimental study to look into how much the skyline changes as users alter their constraints. However, the limitations imposed can significantly influence the outcome of the query, resulting in the inclusion or exclusion of skyline points. Both of these works evaluate the influence on the user resulting from their engagement with the skyline. However, neither of them offer solutions for facilitating the user's refinement of their inquiries towards their intended goal.

To handle skyline-join queries across multiple nodes, Bai et al. [105] presented the distributed skyline-join query algorithm (DSJQ). However, none of these computations handles the work with incomplete databases.

The researchers in this study are using a distributed database to compare data points in parallel to find solutions to problems more quickly. One can learn about the state of data objects and their dimensional attributes with the help of a data set skyline query. Consideration of the advantages and disadvantages of attribute values yields the dominant relationship between the data objects in the data set, which then yields the data objects not subject to the dominant relationship and the Skyline result set. But as data sizes grow, Skyline Join becomes infeasible.

To gain access to user preference profiles represented as DAGs and assist with the cache-based skyline query computation with partially ordered domains, Hsueh et al. [106] presented an extended depth-first search indexing method. However, these methods suffer from the issue of high mapping costs and are best suited to centralized environment extensions. The authors introduced a refinement process to answer queries using cached queries. Let q and $q\prime$ represent two queries, and $q.R$ and $q\prime.R$ their preferences. q is a refinement of $q\prime$ iff $q\prime.R$ inclus in $q.R$. In such case it is easy to see that $Sky_{q.R}$ inclus in $Sky_{q\prime.R}$. Let's believe that q is a completely new query submission and that Q is a cached set of already submitted queries. The key to their strategy is locating a refinement $q\prime$ in Q of q and then applying q's evaluation to q. The authors suggest using an indexing structure to track down a refinement after providing a query. This index is not comprehensive because it may miss some refinement points.

Some studies are concentrating on data transmission in an effort to decrease energy consumption [107]. The authors propose a regional division strategy that is based on a sub-region clustering strategy [107] to lay the basis for data node tree-based spatial skyline queries and parallel queries of non-spatial skyline data. Most WSN skyline query algorithms use a pruning strategy [107] to decrease the energy cost of data transmission between nodes. For multidimensional sensing data, Wang et al. [107] offer an energy-efficient skyline query method, and they also introduce several effective data reduction techniques like the dynamic filter and the

tuple-cutting strategy, among others. Additionally, such an algorithm is only useful for handling skyline queries and cannot be used for regaining access to original sensory datasets. The technique employs a node cut strategy to dynamically produce filtering tuples rather than issuing filtered queries in order to collect query results.

Using dominance size as a measure of a set's "representativeness," Bai et al. [108] proposed LDS queries. They're looking for the most dominant set possible. In particular, the volume of dominance region is one of the criteria used to evaluate a point's quality or significance. Calculated points on the skyline that maximize the dominated area or volume represente the entire skyline's "contour" in the same way. The k-representative skylines algorithm yields a predetermined number of k skyline points, which are chosen based on measurements that take into account the volume of their respective dominance regions. Li [109] introduced a circular distributed Skyline query (PDS) algorithm.

The authors [110] conducted a study and demonstrated that the Eager approach, as presented in [110], exhibits superior efficiency in terms of execution time. However, it necessitates a greater amount of memory due to the need to maintain the event list.

Bai et al. [111] presented a set of maintenance techniques for handling subspace global skyline (SGS) queries in dynamic databases.

In order to achieve efficient computation of the skyline, the [112] authors introduce a new approach for partitioning data space that is suitable for parallel and distributed skyline computation. This approach involves two phases, namely diagonal and entropy score curve-based partitioning.

Koizumi et al. [97] study has expanded upon the BJR-tree algorithm for efficient computation of skyline queries by incorporating a tree structure based on dominance relations.

The concept of skyline probability applies to the probability that a data object with uncertain attributes will be determined as a "skyline point". By setting a lower threshold for Skyline probability, it is possible to effectively exclude smaller values. When the threshold is denoted as a probability event, the corresponding skyline is commonly referred to as the "q-skyline".

The authors, Zheng et al. [113], have presented a method for computing the continuous skyline using an incremental motion model. This approach involves moving the query point incrementally in discrete time steps without any constraints or predictability. The authors' research is centered on the processing of skyline queries in Euclidean spaces. This involves the computation of the distance between objects purely based on their locations rather than taking into account the connectivity of the road network.

A novel technique, HashSkyline [114], has been introduced for executing parallel skyline queries on high-dimensional datasets utilizing GPUs, with a focus on correlation awareness. The HashSkyline algorithm employs a technique based on hashing to enhance the efficiency of skyline queries for datasets that exhibit correlation.

Huang [115] has presented a study on the processing of queries in a dynamic road network that includes the skyline. The authors' methodology for query processing relies on the dominance and distance of object attributes.

It is crucial to quickly determine whether the time-varying information has an impact on the K-nearest skyline objects ($KNSOs$) in order to handle real-time processing of time-varying information effectively. Subsequently, any necessary updates to the $KNSOs$ must be immediately implemented. In order to attain the objective, the study employs three distinct data structures that were developed in [115]. These include the object attribute dominating matrix ($OADM$),

the road distance sorted list ($RDSL$), and the skyline object expansion tree ($SOET$). These structures are utilized to effectively store pertinent information about both data objects and the road network. The primary objective of these data structures is to streamline the process of identifying the temporal information that impacts the query outcome. The $CKNN-SQ$ updating algorithm is introduced as a proficient approach for promptly assessing the latest outcome of KNSOs, utilizing data structures.

Kertiou et al. [14] noted that the information obtained from the Internet of Things (IoT) layer could be reduced and refined through the use of self-governing sensor selection in response to user requests. The utilized methodology in this context is denoted as Skyline. The methodology extracts the most crucial sensor information from a diverse array of Internet of Things (IoT) sensor data.

Authors use the dynamic skylines operator in conjunction with context data from sensors to effectively narrow the search space and then find the best sensors that meet the user's unique needs. In order to achieve scalability, the authors suggest the implementation of distributed gateways that are linked to a central server. Each gateway is responsible for handling local requests made by users.

In contrast to the fast nondominated algorithm, the dynamic skyline algorithm shows reduced time complexity. However, it is not advisable to require that users provide ideal values for sensor properties.

The method under consideration receives the user's request and performs a computation of the local dynamic skyline in order to minimize the sensor dataset. Subsequently, the acquisition of a group of sensor data that has undergone local filtration is conducted in order to calculate the dynamic skyline. Ultimately, the SAW [14] algorithm was employed to prioritize the sensor dataset.

The new cognitive problem of serendipitous discovery [116], best summed up as "unexpectantly finding surprisingly interesting points," is concerned with cell discovery via continuous skyline queries. Similar to a dominance graph, but in a temporal setting.

Using a dominance relation that can be determined through point-to-point comparisons, the data points for Object-based Space Partitioning Skyline, a balanced joint-rooted BJR-tree [116], are partitioned into regions of the multi-dimensional data space. The next step is to build a tree representing the dominance relationship between the partitioned regions so that the incomparability can be quickly and easily determined. These trees reduce the amount of time spent on dominance testing by figuring out where an input data point fits within the tree and finding other similar regions. The primary benefit of employing these techniques lies in their ability to perform dominance tests exclusively on the data points associated with the respective regions.

The skyline query techniques that rely on dominance relations have been found to effectively decrease the frequency of dominance tests. This is achieved by utilizing the dominance relationship that is obtained through the dominance test and then bypassing the need for dominance tests in cases where the items are incomparable. Nevertheless, the utilization of skyline query techniques is restricted to comparisons between two points only. Furthermore, the application of the concept of dominance relation to alternative methodologies is challenging due to the structural inter-dependencies inherent in current dominance relation-based approaches.

Han et al. [117] have proposed a method for processing dynamic skyline queries on large datasets. Their method entails retrieving tuples from dynamically sorted lists in a round-robin

fashion up until an early termination criterion is satisfied, then computing the dynamic skyline results.

Huang et al. [118] made improvements to the $CKNN-SQ$ algorithm proposed in [119] to enable its implementation in a dynamic road network. The improved algorithm utilized a combination of three data structures to facilitate the efficient updating of K-nearest skyline objects ($KNSOs$).

In the work of [4], researchers have implemented the lattice of cuboids algorithm as a means of minimizing the computational costs associated with conducting skyline queries on data streams with high dimensions. However, the time complexity of these methods tends to increase as the number of dimensions increases. The present study aims to effectively calculate skyline queries featuring multiple attraction points across more than two dimensions while maintaining a predetermined upper limit on time complexity.

The computation of skylines initially appeared based on a single point of attraction within two-dimensional systems. Over time, the skyline has been expanded beyond two dimensions and has been analyzed with multiple points of attraction.

Tang et al. [120] conduct research on skyline queries concerning mobile entities on dynamic road networks, where the weights of the edges may also receive modifications. The authors used a grid partition technique to achieve a quick response time.

Liu et al. [79] proposed a Dynamic Dimension Indexing skyline algorithm called range search skyline RSS for high dimensional data stream that supports both count and time based sliding windows.

3.3.2 Distributed and Parallel Techniques

One way to ensure scalable query processing is to make use of multiple resources. Skyline queries, like many others, may benefit significantly from parallel or distributed processing. Skyline queries can be efficiently executed in distributed environments using the index-based algorithms presented in Balke et al. [91] and Lo et al. [92]. Wu et al. [121] present another investigation into the issue of parallelizing Skyline query execution across a large number of machines by means of content-based data partitioning. The proposed algorithm, named DSL for distributed skyline query processing, finds the skyline points incrementally.

Wang et al. [122] investigate the use of P2P networks for processing skyline queries. To better regulate the peers accessed and search messages when processing skyline query, authors proposed a method called SSP that partitions and numbers the data space between the peer nodes. Wang et al. [123] present the SKYFRAME method, a generalization of the SSP method that allows skyline processing to be performed without identifying the starting peer.

Since it is nearly impossible to ensure full and accurate query answers without an exhaustive search, Hose [124] presents a study on the effective processing of skyline queries in large-scale P2P systems. To reduce the load on nodes, approximate algorithms backed by probabilistic guarantees are proposed. When obtaining precise answers to skyline queries is prohibitively expensive, approximate algorithms are proposed as an alternative, as suggested in Li et al. [125]. The proposed algorithms use heuristics based on the local semantics of peer nodes, which yield high-quality results. In addition, Hose and Vlachou [39] present a comprehensive review of skyline processing in highly distributed environments.

In order to reduce the amount of data transferred over the network connecting the peers, Vlachou et al. [124] propose a threshold-based algorithm for efficient subspace skyline processing in a P2P environment called SKYPEER.

Skyline query processing methods in mobile ad-hoc networks (MANETs) are examined in Huang et al. [126], with the primary goal of minimizing communication overhead between nodes and maximizing device throughput. Li and Xiong [127] investigate query processing and optimization strategies for WSNs. Later they proposed the SKY-SEARCH algorithm, which efficiently calculates the skyline with the highest existential probability.

Vlachou et al. [128] first proposed and designed a method to operate in shared-nothing architectures. A master node that serves as the coordinator and a group of core nodes constitute the abstract architecture. The coordinator's job is to accept requests and assign tasks to the related cores. For the remainder of the discussion, N will stand for the number of servers. It is presumed that all machines communicate via a fast network. This indicates that network latency must be considered.

The dataset P, is made up of a set of d-dimensional points that must be partitioned across N servers. Where $1 \leq i \leq N$, each partition P_i contains some subset of the points in P. For effective parallel processing, a user-friendly data partitioning scheme is required. There are three stages to the algorithm:

- Partitioning Phase: The data is partitioned among all of the available core nodes.
- Local Processing Phase: Based just on local data, each core calculates the skyline.
- Merging Phase: Based on the local results that the cores produced in the earlier phase, the coordinator calculates the final skyline result.

After deciding on a partitioning method, the remaining processing (local processing and merging) is simple. Therefore, let's examine various partitioning options with an eye toward evaluating their effect on query processing generally. Random Partitioning (RP) is the simplest and most adaptable partitioning strategy because it provides nearly equal points to each core and assigns them at random (or via a hash function, or via round-robin). Grid-based partitioning (GP), in which the address space is divided into cells and each core receives the points that fall on a particular cell, is another partitioning strategy that is very simple to implement. A single core, however, may be assigned multiple cells to process. Angle-based Partitioning (AP), proposed by Vlachou et al. [128], is a partitioning strategy that uses conical regions with respect to the coordinate system's origin to divide the address space.

Figure 3.5 provides an illustration of the distinction between GP and AP before we dive into an analysis of the effects of each partitioning method. To keep things straightforward, let's assume that the grid decomposition has to ensure that each cell has roughly the same number of points. While this isn't strictly necessary for the algorithm to work, load imbalances may seriously impact performance. There are a variety of grid structures usable, and their cell sizes need not be uniform. Using a uniform grid partitioning where each cell contains three points, as shown in Figure 3.5 (right), is one way to decompose the address space. Figure 3.5 (left) depicts the decomposition of the address space using angle-based partitioning.

Given the success of grid-based and angle-based partitioning strategies, it stands to reason that the RP strategy will perform poorly because it does not adhere to a consistent method for dividing up data among cores. The experimental results reported by Vlachou et al. [128] confirm the theoretical prediction that RP incurs a high network latency cost, which has a direct negative impact on performance. This also confirms that AP has the highest performance of any partitioning strategy tested. The authors of Vlachou et al. [128] also talk about dynamic angle-based partitioning, which is like an adaptive grid, and equi-volume angle-based partitioning, which is like a uniform grid. When it comes to runtime performance and scalability, AP

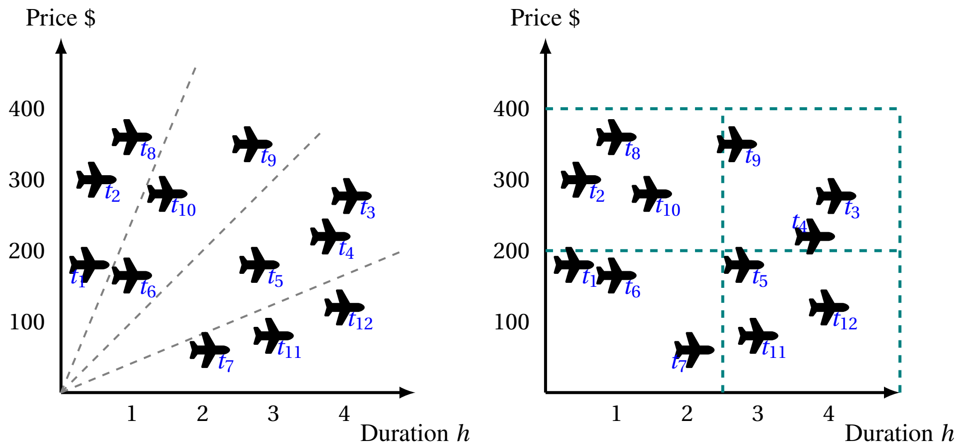


Figure 3.5: *Gid-based (right) and Angle-based (left) Partitioning.*

consistently outperforms the competition. This is true regardless of the size of the dataset or the number of dimensional dimensions present in the data.

3.3.3 A High-Level Approach to Parallel Programming

With the exponential growth in computational power, communication bandwidth, and storage capacity, parallel programming has become an essential paradigm in modern computing. Traditionally, parallel programmers relied on low-level parallelization techniques and specialized libraries to maintain complete control over parallel applications. These methods enabled manual optimization of code to maximize hardware utilization. However, such low-level approaches often impede software productivity, limit development efficiency, and hinder both code portability and performance portability [26].

Code portability refers to the ability to compile and execute the same code across multiple architectures without modification. Performance portability, on the other hand, ensures that an application can efficiently leverage different parallel architectures without requiring significant re-engineering. Given the diversity of modern computing platforms—including multicore CPUs, heterogeneous architectures, and cloud environments—ensuring performance portability has become a crucial requirement [57]. Continuous adaptation and manual tuning for each new hardware configuration are neither practical nor cost-effective for individuals or organizations [34].

The database community has traditionally addressed these challenges by adopting high-level abstraction techniques that simplify parallel programming. High-level programming models significantly reduce development time and costs by concealing hardware complexities. These models enhance productivity and ensure economic sustainability through improved portability and efficient performance [Darlington et al. [27]; Skillicorn and Talia [28]; Cole [29]; McCool et al. [33]]. To enable the development of parallel applications, programming environments must provide high-level constructs that allow developers to focus primarily on computational logic while abstracting parallel execution details.

Modern runtime environments and programming tools, such as OpenMP [129], handle critical implementation decisions, allowing a single program to leverage multiple runtime systems for different execution architectures. This abstraction facilitates architecture-independent development, enabling reliable performance predictions across diverse hardware platforms.

Various low-level mechanisms, including POSIX threads [130] and message-passing li-

braries such as MPI [131], as well as hybrid approaches like OpenMP [31], provide certain aspects of high-level parallel programming. However, effective parallel programming often requires developers to define an explicit parallel schema, which may involve restructuring sequential code to incorporate parallel directives and constructs. This necessity has led to the emergence of structured parallel programming (SPP) as a robust and compelling methodology for high-level parallel programming [Bacci et al. [132]; Vanneschi [133]; Aldinucci et al. [134]; Cole [29]; Vanneschi [135]].

Structured parallel programming is founded on the use of parallel paradigms (or patterns) to define applications. Many real-world algorithms exhibit well-defined parametric implementations of communication and computation patterns. These paradigms, which represent fundamental schemas of parallel computation, provide a structured approach to parallel software development. Leveraging these patterns enables efficient application configuration, adaptive execution, and performance optimization.

From cluster-based computing [136] to shared-memory architectures [137], and from grid computing [138] to cloud and pervasive environments [139], structured parallel programming has demonstrated its effectiveness. Our developed algorithm successfully applies a high-level parallel approach to skyline queries over data streams, demonstrating its applicability to large-scale data processing. Moreover, structured parallelism has significantly contributed to advancements in various domains, including data mining [140], signal processing [141], and computational biology [142].

By embracing high-level abstractions and structured parallel paradigms, modern parallel programming can achieve greater efficiency, maintainability, and scalability, making it an indispensable approach for future computing challenges.

The work in [143] introduced the concepts of reverse k-skyband queries and ranked reverse skyline queries, which aim to modify or relax the constraints of the conventional reverse skyline query to retrieve a greater number of objects that are considered important.

De Matteis et al. [41] introduced a parallelized algorithm designed for window-based skyline computation on modern hardware such as multicore CPUs. The authors concentrate on enhancing the efficiency of the reduction phase and establishing effective load-balancing techniques. The utilization of Skyline Influence Time is a technique employed by authors to facilitate the parallelization of the eager algorithm.

De Matteis et al. [5] have proposed a parallelization algorithm for efficient processing of skyline queries. The algorithm is designed to achieve near-optimal speedup and incorporates several load-balancing strategies.

The authors in [5] have presented a proposal for parallelizing the eager algorithm, which is based on the concept of Skyline Influence Time. Through the implementation of various load-balancing strategies, they have been able to achieve a speedup that is close to optimal. The proposals have been demonstrated to be effective and efficient through a comprehensive set of experimental results. De Matteis et al. [5] introduced join query processing, which involves incorporating multiple operators within a primary attribute and subsequently coordinating their functions. An efficient parallel continuous skyline approach has been investigated by the authors in the context of [144] where the dataset points have been sorted in accordance with the Manhattan distance metric.

In their study, Koizumi et al. [145] employed a Joint Rooted Tree data structure to retrieve the skyline periodically for a large set of input points. Each point in the set is characterized by

an entry time (Tstart) and an expiration time (Tend). Once the input point has expired, it will not be initiated again. On an Intel Core i7 processor, the authors implemented the algorithms using POSIX threads.

The investigation conducted by IsIam et al. [146] applied to the examination of parallel reverse skyline queries on shared-nothing clusters. A quad-tree was constructed to facilitate data partitioning, and the search area was minimized by expanding the concept of midpoint-based pruning. The dynamic skyline query enables users to define a query object, denoted as q, and subsequently obtain the skyline objects in relation to q.

Zhu et al. [147] introduced a parallelized variant of the Usky-base algorithm [148] named Parallel-sky. This algorithm is designed to calculate the probability of Skyline over uncertain preferences using a multicore architecture. The technique of "parallel sky" was examined under dynamic scenarios involving the addition and removal of points from a given dataset.

Zhang et al. [68] introduced the Naive Parallel Sliding Window Join (NP-SWJ) and Incremental Parallel Sliding Window Join (IP-SWJ) algorithms. The following two parallel algorithms are designed to find the skyline-join on multiple streams of data. The technique of NP-SWJ concurrently examines multiple pairs of windows in order to identify the skyline join. The IP-SWJ algorithm employs a progressive computational approach that relies on the intersection of consecutive windows.

Alami et al. [149] proposed the negative skycube to find the subspaces skyline over continuous multi-dimensional data stream.

3.3.4 Parallel Paradigms for Skyline Queries Over Data Stream (SPODS)

Parallel SPODS applications can be effectively represented using a computation graph without loss of generality. In this model, nodes correspond to intermediate functions, which serve as modular components of the overall application. Data streams act as communication channels through which computations take place. When certain functions create performance bottlenecks, they must be internally parallelized using appropriate parallelism paradigms to meet performance requirements [135, 150–152]. These paradigms exhibit the following characteristics:

- Identification of high-performance bottleneck functions.
- Enforcement of specific pre-established parallel computing patterns.
- Capability to be composed together to form highly complex computations.
- Establishment of an overall performance model.

This approach alleviates the programmer from concerns regarding the mapping of parallel computation schemes to specific parallel architectures. By utilizing performance models, efficiency metrics can be assessed as functions of various parameters, including application-specific factors (e.g., computation time and data size) and architectural properties (e.g., processor performance and memory access latency).

In this section, we examine a collection of well-established parallel patterns for query processing over data streams, with a focus on their key characteristics and performance implications.

A Basic Set of Parallel Patterns

Consider a network of air quality monitoring sensors deployed across a smart city. Each sensor collects multiple attributes:
Temperature (°C), Air pollution levels (PM2.5, CO_2, NOx, etc.), Humidity (%), Battery life (remaining energy percentage).

A skyline query helps identify the best sensors based on accuracy, energy efficiency, and strategic placement. Due to the high volume of incoming sensor data, parallel computing is essential for efficient processing. The following sections map various parallel patterns to real-world skyline query processing steps.

Pipeline:

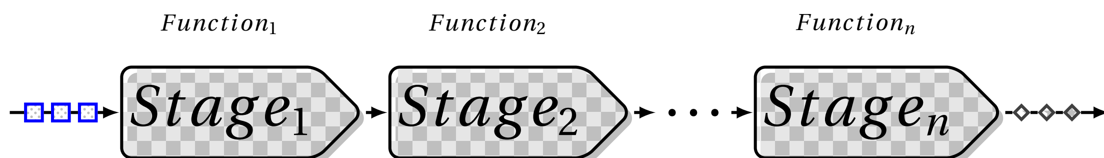


Figure 3.6: *Pipeline parallel pattern [26].*

The pipeline paradigm provides a sequential composition of functions applied to input elements, formulated as:

$$F(x) = F_n(F_{n-1}(\ldots(F_1 x)\ldots)) \tag{3.2}$$

A linear arrangement of n execution stages allows parallelization, as shown in Figure 3.6. This approach increases throughput but may introduce communication overhead between stages, potentially affecting latency [34].

Pipeline Processing:

- Breaks computation into multiple sequential stages.
- Each stage transforms data before passing it to the next stage.
- Improves throughput, but overall latency depends on the slowest stage.

Example in Skyline Query:

- Stage 1: Read sensor data from streams.
- Stage 2: Filter noisy or incomplete sensor readings.
- Stage 3: Compute skyline candidates using dominance checks.
- Stage 4: Output skyline sensors for decision-making.

Use Case: Suitable for real-time environmental monitoring, ensuring accurate and timely sensor data processing.

Parallel patterns provide structured methodologies for parallel computing. These patterns can be broadly categorized into two types:

1. Task Parallel Paradigms: These patterns operate effectively on homogeneous data streams. Parallelism is achieved by processing multiple elements concurrently, thereby increasing throughput rather than reducing computation latency for a single element [34].

2. Data Parallel Paradigms: These paradigms apply parallelization at the individual computation level, often requiring data partitioning, which leads to parallel execution of computations [34]. Unlike task parallelism, this approach can enhance computation latency.

Parallel paradigms define the interaction structure among various computational entities. These entities include cores (executors in shared memory systems), a *Splitter* that distributes data, and a *Merger* that aggregates results from cores before passing them to the output stream.

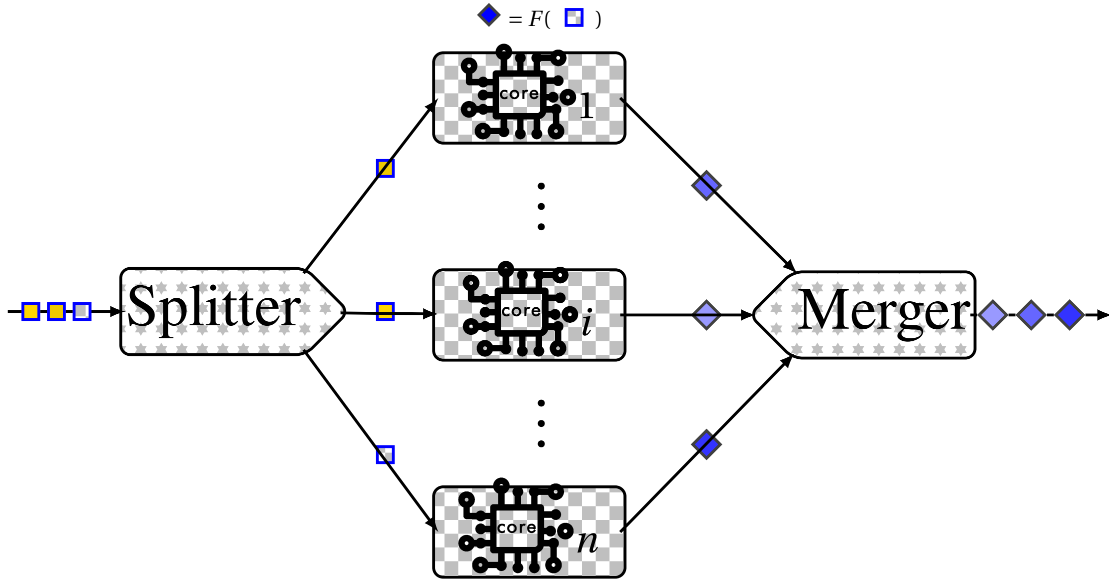


Figure 3.7: *Task Farm parallel pattern [26].*

Task Farm:

The task-farm paradigm replicates a stateless function F across multiple cores [34]. The *Splitter* assigns input elements to cores based on a scheduling policy (e.g., round-robin for uniform workloads). The *Merger* collects results and forwards them to the output stream (Figure 3.7). This pattern increases throughput while maintaining the computation latency of each element.

- Replicates the same task across multiple cores.
- A load balancer distributes sensor data to multiple processors.
- Best for independent computations with low variance.

Example in Skyline Query:

- Each core processes a batch of sensor data to check for dominance relationships.
- High parallelism is achieved when sensor readings are independent.

Use Case: Useful when pre-filtered sensor data allows skyline computations without dependencies.

Data Parallelism:

This paradigm involves both function and data partitioning, enabling independent execution of the same operation on different data partitions [34]. The *Splitter* distributes data among cores, and the *Merger* aggregates results (Figure 3.8). Since cores work independently, there is no need for direct inter-core coordination.

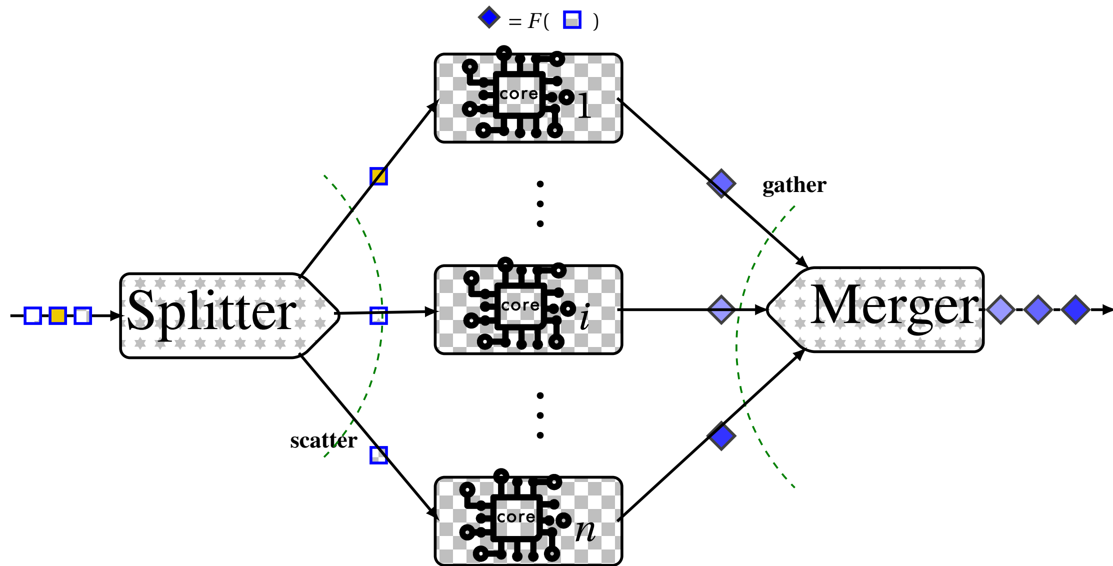


Figure 3.8: *Map pattern with 4 cores [26].*

- Splits data into partitions, processed in parallel.
- Each core operates on its assigned data chunk.
- cores may exchange intermediate results to refine skyline computations.

Example in Skyline Query:

- Divide sensor data by geographic regions (e.g., North, South, East, West).
- Compute skylines for each region in parallel.
- Merge results to obtain a global skyline.

Use Case: Optimized for large-scale smart city monitoring with regional sensor clusters. In some cases, computations exhibit dependencies across partitions, such as stencil-based computations, which require information exchange between cores. Data parallel paradigms can reduce computation latency for individual elements and increase overall throughput [34].

Reduce:

The reduce pattern applies to computations of the form:

$$y = x_1 \bigoplus x_2 \bigoplus \ldots \bigoplus x_n \tag{3.3}$$

where an associative function $\bigoplus$ is applied to aggregate input elements. The data is partitioned among cores, each performing a local reduction before a final global reduction [34].

- Computes a single aggregated result from multiple parallel computations.
- Uses an associative function (e.g., min, max, sum, average).

Example in Skyline Query:

- Each core computes a local skyline for a subset of sensors.
- A global reduction step merges all local skylines into the final skyline set.

Use Case: Used in distributed skyline computation where each processor finds partial skylines, and a central reducer merges them.

Patterns Composition

Complex structures can be constructed by combining parallel paradigms. In stream-based computations, it is common to integrate data-parallel and stream-parallel techniques. For instance, a pipeline can include one or more stages implemented as a map [34]. A widely used algorithmic approach is the ”map and reduce” paradigm [34], where a function is applied to each element of a data structure, followed by a reduction operation that aggregates the results. The MapReduce programming model, introduced by Google [153], adopts this concept. Its open-source implementation, Hadoop [154], has gained significant traction in batch processing, demonstrating the efficacy of high-level programming approaches when supported by an efficient runtime environment.

Skyline with MapReduce

MapReduce, developed by Google, simplifies the development of parallel applications capable of processing large datasets on clusters of cost-effective machines. An effective MapReduce framework should ensure load balancing across the participating machines while optimizing space utilization, CPU and I/O operations, and network transfer efficiency.

Zhang et al. [155] elaborated three distinct algorithms for skyline computation that have been suitably modified to operate within the MapReduce computational framework. The MR-BNL approach utilizes the MapReduce framework to partition the data space into subspaces of two dimensions based on the medians of each dimension. Subsequently, the local skyline for each subspace is computed using the BNL algorithm. Next, the merging of all local skylines is executed on one machine for the purpose of computing the global skyline. The MR-SFS algorithm is a modification of the MR-BNL algorithm that incorporates presorting through the implementation of a Sort-Filter-Skyline approach based on the MapReduce framework. The MR-Bitmap technique utilizes MapReduce to construct a bitmap structure, enabling parallel examination of individual objects. Both the MR-BNL and MR-SFS algorithms focus on a single machine for the purpose of computing the global skyline. Individuals may experience a dimensional curse issue whereby the size of the local skyline is significantly raised in cases where the dimensionality is high. If the bitmap structure exceeds the capacity of the main memory, MR-Bitmap necessitates a significant amount of disk space for storage. MR-Bitmap incurs the highest input and output costs.

Park et al. [156] designed a method for executing parallel dynamic SKY-M and reverse skyline RSKY-MR queries through the utilization of MapReduce. This approach involves the implementation of both a pruning approach and a data partitioning technique. The researchers employ a quad-tree index, which is generated from arbitrary subsamples of the dataset, to execute their partitioning. The optimality of quad-tree indexes for query grouping is limited due to their dependence on queries.

The SKY-MR technique begins by constructing a histogram via the sampling of a subset of the dataset. This methodology is utilized with the intention of eliminating non-skyline points in advance. The method is commonly utilized to divide data points into separate and independent regions. Subsequently, the candidate skyline points are computed for each respective region. Next, each candidate point gets an evaluation procedure for determining its classification as a skyline point. The execution of two separate MapReduce tasks is required for the implementation of this approach. However, the current methodology lacks the ability to incorporate uncertain data and may benefit from further development to address this limitation.

The MR-GPMRS [157] partitioning scheme is based on a grid structure, employs a recursive approach to divide certain dimensions of the data into multiple segments. Subsequently, skyline candidates are computed for each partition. The utilization of a bit-string by Mullesgaard et al. [157] for the purpose of grid-based partitioning has facilitated the ability to prune a greater number of data points prior to the ultimate computation of the skyline. The MR-GPMRS algorithm employs a strategy of utilizing multiple reducers in order to effectively calculate the global skyline from a set of skyline candidates. Thier study examines the impact of parallelization on skyline computation across diverse datasets with variations in dimensionality, cardinality, and execution buffers.

Zhang et al. [158] presented a parallel algorithm, named PGPS, for executing skyline queries within the MapReduce framework. The PPF-PGPS algorithm comprises three distinct phases, namely partitioning, local skyline, and global skyline. During the partitioning phase, angle-based partitioning and filtering approaches are utilized to eliminate unqualified tuples within each partition. Following this, the reduce function is applied to filter out objects subsequent to their aggregation from map functions. During the local skyline phase, PGPS employs a mapper machine to calculate a local skyline for each angle-based partition in a parallel fashion using the map function. After that, during the phase of global skyline computation, PPF-PGPS combines the individual skylines of the mappers into only one node, resulting in the ultimate global skyline, which is executed through the reduce function. A technique known as partial presort has been proposed as a way to enhance the merging performance of local skylines. Grid partitioning, then sorting, are the steps involved in the partial-presort method for dividing local skylines. The partitions located in proximity to the origin are prioritized for processing. This method is effective in quickly eliminating tuples within the dominated partitions. Despite the MapReduce Skyline algorithm's efficacy, its failure to account for workload balancing may result in a decline in algorithmic performance. According to the results of the study conducted in [159], it has been observed that the initial phase's filtering methodology is not successful for anti-correlated data.

Y. Li et al. [160] developed a system model capable of facilitating subspace skyline queries in a mobile distributed environment. Additionally, Li et al. introduced an efficient algorithm for processing skyline queries through the utilization of MapReduce. The algorithm employs the mapreduce technique to derive a significant subset of points from a complete set of skyline points within any given subspace.

Park et al. introduced the SKY-MR+ [161] framework as a means of enhancing the efficacy and scalability of SKY-MR [156]. This was achieved through the utilization of an adaptive quadtree generation method and techniques aimed at balancing the workload across machines. Sky-MR+ employs a distinct approach for constructing the Quadtree, utilizing a histogram based on the quadtree for partitioning space and implementing the dominance power filtering technique to efficiently eliminate non-skyline points. The construction of the Quadtree necessitates the presence of samples. The efficacy of these two algorithms is dependent upon the utilization of sample data points for the construction of a Quadtree.

Jia-Ling et al. [162] has introduced a parallel algorithm, MR-SKETCH, for computing the skyline using the MapReduce framework. The purpose of this algorithm is to avoid bottlenecks that may arise during the computation of the global skyline from local skylines in a sequential manner. The MR-SKETCH methodology comprises three distinct stages: a data filtering phase,

a computation phase for dominated subsets, and a final step for merging results.

Following the partitioning of a dataset, the MR-SKETCH algorithm proceeds to the filtering stage, wherein each partition selects tuples at random to preserve the set of sample points. The sample points are employed in the computation of a skyline, which serves as a filter. Following the elimination of tuples that are dominated by their respective filters, the MR-SKETCH algorithm proceeds to calculate a subset of dominated tuples for each partition. This subset includes tuples that are incapable of being a global skyline. During the merging phase, the algorithm combines the surviving tuples from both the filtering and dominated subsets for each partition. Subsequently, it eliminates the dominated subset from the surviving tuples to establish the final global skyline. The MR-SKETCH methodology formulates regulations for reducing the size of dominated subsets, thereby decreasing the network costs associated with their distribution. The experimental results indicate that MR-SKETCH outperformed other algorithms that are currently available. When performing parallel skyline computation on the MapReduce platform, the researchers' approach showed a number of limitations despite its potential benefits.

The MR-SKETCH algorithm employs a filtering strategy that relies on randomly selected filter points. In contrast, other filtering strategies deliberately select filter points to achieve more effective pruning capabilities. Thus, the efficacy of its filter points selected at random is notably inferior to alternative filtering methodologies when assessing their filtering efficacy with an equivalent number of filter points. As a result, a decrease in filtering power leads to an increase in the size of map outputs, resulting in higher computation and network costs. Furthermore, the aforementioned approach fails to account for the computational expenses incurred by dominated subsets. Consequently, as the magnitude of the dataset or the number of dimensions increases, there is a significant computational expense associated with calculating the dominated subsets.

The absence of scalability results in a reduction in the performance enhancement achieved through parallel processing. The algorithms utilized depend upon partitioning schemes that split every data dimension into just two halves. This approach poses a challenge for expanding the skyline computation as the dataset size grows, necessitating a greater number of partitions to effectively process the datasets in parallel. Despite attempts to overcome limitations by increasing the number of partitions through a generalized partitioning scheme that divides each dimension into k partitions, the approach still exhibits limitations in terms of scalability. This is due to the significant increase in network costs associated with transmitting dominated subsets as the number of partitions increases. Furthermore, in the MR-SKETCH algorithm, the process of calculating dominated subsets within a reducer necessitates a double reading of input data, once for the creation of a hash table and again for the computation of dominated subsets. As a result of this particular attribute, it is necessary to retain all input tuples of every reducer in the primary memory, given that the input values of the reducers are capable of being iterated. Therefore, the processing of large amounts of data presents a challenge.

Thirdly, the MR-SKETCH method is subject to a constraint on the extent of parallelism. The authors present a methodology to address the issue of high network costs associated with transmitting dominated subsets. This issue cannot be ignored even when partitions are generated by dividing each data dimension into only two halves. The proposed solution aims to reduce network costs. In order to address this issue, a strategy is employed whereby the calculations pertaining to lower subsets are delayed until after their transmission. Nonetheless, this methodology results in the parallel computation of dominated subsets returning to serial computation. Moreover, the outcomes of the reducers and the dominated subsets are passed on to an independent node for the purpose of merging the results towards the conclusion of MR-SKETCH. The merging step of the result is executed sequentially, which poses a bottleneck for computing the skyline as the dataset size or dimensionality increases.

Jang et al. [163] developed a filtering approach that utilizes multiple regression analysis within the MapReduce framework to decrease the number of possible objects beyond the skyline threshold. Simultaneously, the filter threshold and grid-based threshold were taken into consideration to guarantee the accuracy of the skyline.

Kim et al. [164] introduced a skyline query in the MapReduce framework, which leverages parallel computing in a distributed parallel framework to efficiently compute and update skyline query results. The methods presented suggest a new two-phase strategy within the MapReduce framework. During the initial phase, the dataset is partitioned into multiple subsets, following which local skylines are computed exclusively for the qualified subsets. During the second phase, the determination of global skyline points is performed by utilizing the local skyline points. The present paper introduces two distinct filtering techniques, namely outer-cell filtering and inner-cell filtering. The experimental results indicate that the suggested techniques exhibit suitable levels of effectiveness and scalability.

Wang et al. [165] implemented the spatial skyline computation NSSQ within the MapReduce framework, utilizing a grid-based partitioning scheme to recursively divide the data dimensions into multiple parts. The fundamental concept requires that every search region comprise multiple disks that establish a distinct region while searching for spatial skyline points. As a result, the researchers decided to focus their search on the independent region pivot, which is defined by the designated point p E P. This enabled them to transform the problem into M independent problems, with each disk that defines the search region corresponding to one of these problems. The M problems are concurrently resolved. The authors propose a solution based on MapReduce that consists of three phases. During the initial phase, the computation of CH(Q) takes place, where Q is subjected to partitioning. Next, multiple local convex hulls are generated, which are eventually combined to generate the global convex hull. During the second phase of MapReduce, the focus is on identifying the independent region pivot. This involves identifying the point $p \in P$ that has the smallest search region. The minimization of search region areas occurs first at a local level and then at a global level. In the third phase, each point of P is linked to the independent regions that determine the exploration area of the selected pivot, within which it is covered. The solution involves addressing $|E(Q)|$ independent spatial skyline problems concurrently, utilizing only the points that are included within the independent region. The use of pruning regions, which have circular ring sectors, facilitates the minimization of dominance tests. This is due to the fact that the points located within these regions are subject to dominance. The algorithm employs a pair of multi-level grids and leads to the removal of potentially duplicated skyline points. This is a necessary step as identical skylines may be derived from distinct and independent regions, resulting in duplication within the output.

in the work of [166] Li et al. demonstrate an enhancement to the location query through the utilization of the MapReduce framework during the map stage. The processing of ”big data” is facilitated in a distributed environment, and the algorithm will speed the identification of an optimal location on a map.

Hyeong-Cheol Ryu et al. [167] employed the adaptive two-level grids (TLG) technique to execute the skyline query within a single job in the MapReduce framework. In order to reduce the network communication cost between the map and reduce phases, it is recommended [25] to incorporate the primary pruning mechanism of a MapReduce job into the map phase. The

authors effectively convey the idea of creating a dynamic index during the map phase to remove a sizable portion of the dataset. Additional techniques for refinement can be implemented during the combiner and reduce phases.

Adequacy of Parallel Patterns for Data Stream Processing

A pertinent question arises: Are the aforementioned parallel patterns sufficient for handling intra-operator parallelism in SPoDS applications? The answer is: not entirely. While stream-parallel patterns remain applicable and effective for stateless computations, stateful operators introduce significant complexity.

Stateful operators in SPoDS iteratively process sequences of input values from one or more streams to produce final results. Windows define continuous segmentations of the input streams, determining subsets of elements that must be considered for output generation at any given time. Traditional stream-parallel paradigms, which treat stream elements separately, and data-parallel paradigms, which assume finite-size elements, fall short in handling such dynamic and evolving structures.

Recent research has explored Hadoop and, by extension, the MapReduce pattern as potential frameworks for real-time stream processing. Condie et al. [168] introduced MapReduce Online (HOP) to facilitate continuous querying, where reducers receive results as soon as they are produced by mapper nodes. However, small windows may suffer from low throughput due to the frequent application of reduce functions, such as aggregation operations [169].

Brito et al. [169] proposed Stream MapReduce, which remains compatible with the MapReduce API but disrupts its abstraction. Here, stateless operators are implemented as mappers, while stateful ones function as reducers, allowing for low or sliding windows. Similarly, M3 [170] introduced an in-memory data path between mappers and reducers to mitigate overhead caused by HDFS, the default distributed file system, which introduces significant delays unsuitable for streaming applications. These approaches force MapReduce into an ill-fitting paradigm, ultimately leading to suboptimal performance.

We argue that SPoDS applications require specialized enhancements in data distribution, management policies, and windowing techniques beyond those provided by conventional patterns. Achieving intra-operator parallelism necessitates partitioning windows among computing cores. Given their dynamic nature—where buffered tuples evolve due to triggering and eviction policies—window distributions must be carefully managed. Moreover, when time-based semantics are involved, the arrival rate of stream elements determines window cardinality, necessitating efficient mechanisms for data expiration and update.

Key challenges in this context include:

- Efficiently managing input streams and maintaining up-to-date windows. This requires well-defined distribution policies for incoming elements and effective expiration policies to remove outdated data.

- Ensuring balanced workload distribution among cores while keeping window partitions synchronized and evenly allocated.

Despite these challenges, we believe that parallelization in SPoDS computation remains feasible using structured parallel programming (SPP) paradigms. These paradigms simplify reasoning about throughput, latency, and memory occupancy while reducing development effort. Addressing these requirements will necessitate refining existing parallel patterns and introducing new ones tailored to the unique demands of SPoDS applications.

3.3.5 Parallel Patterns for Windowed Functions

When a SPoDS application fails to meet user-defined performance requirements, such as maintaining a specified input rate or ensuring a constant response time, it becomes necessary to restructure its execution. Bottlenecks typically arise in stateful functions, which maintain and update internal data structures while processing data. This introduces dependencies between the processing of individual tuples, making it crucial to preserve the sequential semantics of stateful functions when parallelizing them.

Several authors [26, 33–35] have advocated for the use of structured parallel patterns to address parallelization challenges due to their reusability and ease of implementation in windowed functions. Windows serve as a fundamental state abstraction in such functions, with their semantics defined by eviction and triggering policies. To generalize the concept, a window is characterized by the following attributes:

- **Window Size** ($|W|$): This can be expressed as a duration (e.g., seconds, minutes, or hours) for time-based windows or as a count of tuples for count-based windows.
- **Sliding Factor** (δ): This parameter defines the movement of the window. Similar to window size, it can be expressed in time units or in the number of tuples. A *tumbling window* is characterized by $\delta = |W|$, whereas a *sliding window* has $\delta < |W|$ [10].

It is important to note that a single tuple may belong to multiple consecutive windows due to overlap, particularly in sliding windows, where a window may include newly arriving tuples while still retaining older ones.

3.3.6 Parallel Patterns Taxonomy

Task parallelism in window-based functions can be uniquely characterized by the way input tuples are grouped and processed. Instead of treating individual tuples as tasks, modern parallel paradigms consider a task as a subset of the input stream that encompasses all tuples belonging to a given window [10]. The fundamental performance metrics for parallel SPoDS applications, as defined by experts [10], include:

Throughput and Service Time: The throughput of a window is defined as the average number of windows processed per unit time. The inverse of service time represents the average interval between the execution of two consecutive windows.

Latency: Latency refers to the average duration required to process a single task or compute a window.

Response Time: This metric quantifies the time taken to produce an output after receiving the last tuple that triggers a window computation.

De Matteis et al. [26] introduced the concept of scalability to assess the efficiency of parallel solutions:

Scalability: Scalability measures the performance improvement gained by increasing the level of parallelism. It is quantified as the ratio of throughput achieved with n parallel cores to the throughput achieved with a single core.

3.3.7 Categories of Parallel Patterns for Windowed Functions

Parallel patterns for windowed functions can be categorized into two primary paradigms:

Window-Parallel Paradigms: These patterns enable concurrent processing of multiple time windows. While they enhance throughput by allowing multiple windows to be computed simultaneously, they do not reduce the computation latency for individual windows [34].

Data-Parallel Paradigms: These patterns partition a single window across multiple parallel executors. By distributing the workload among identical processing units, they effectively reduce computation latency [34]. This approach is utilized in the parallelized version of the RSS algorithm, where windows are partitioned among the CPU's available cores [36].

As with all parallel patterns, the structure and definition of a pattern influence its impact on key performance factors such as throughput, latency, and memory usage. One particularly critical aspect is data distribution, given that windows are dynamic structures with evolving tuple content and cardinality. This distribution process can be examined from two perspectives:

- **Splitter Functionality**: The Splitter is responsible for distributing input elements to multiple cores at varying granularities. This distribution can occur at the level of individual tuples, tuple groups, or entire windows.

- **Window Assignment Policy**: This refers to the policy governing how windows are sequentially assigned to parallel processing cores.

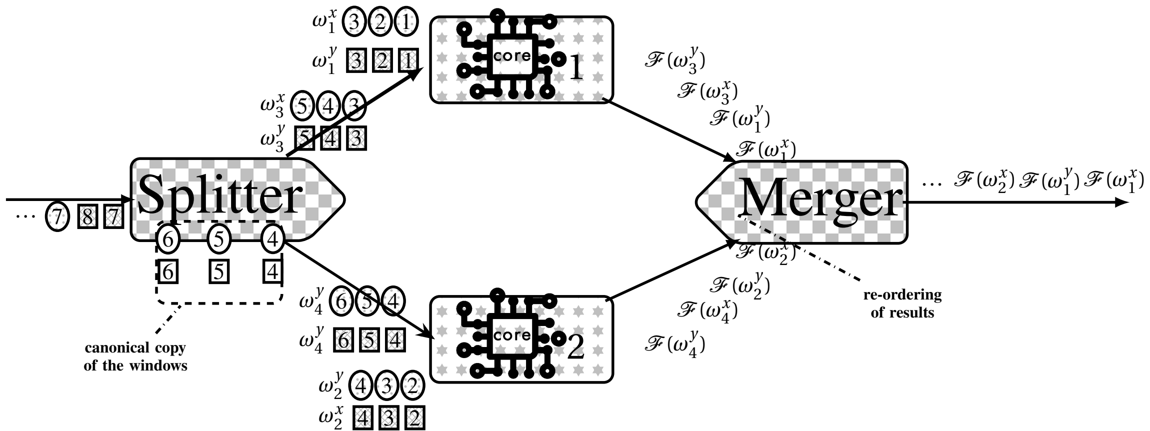


Figure 3.9: *Window Farming with two cores. In the figure $|W| = 3$ and $\delta = 1$. w_i^x is the i-th window of substream X. $F(w_i)$ is the result of the processing function over a window [26].*

Window Farming

The term *farming* in task farming is analogous to agriculture, where farmers divide land into smaller plots, cultivate crops, and harvest them. Similarly, in computing, task farming involves distributing computational tasks among multiple processing units to maximize efficiency [150].

- Uses task-farming to assign entire windows to different cores.

Example in Skyline Query:

- Each core computes the skyline for different time windows (e.g., morning, afternoon, evening).

Use Case: Efficient load balancing when analyzing time-based skyline queries. This pattern follows a straightforward intuition [34]. If each window activation involves applying a function F to its contents without dependencies on other windows—whether from the same or different logical streams—then a simple adaptation of the traditional task farm pattern can be utilized (illustrated in Figure 3.9).

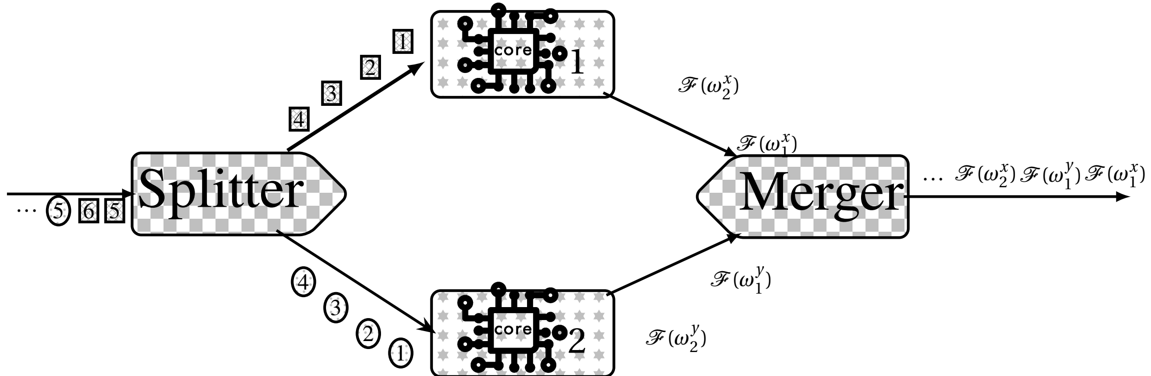


Figure 3.10: *Key Partitioning with fine-grained distribution. Substream X is routed to the first Core, substream Y to the second one [26].*

Key Partitioning

Key partitioning, a specialized form of window farming, employs a restricted assignment policy to distribute keys across processing cores. Given a set of keys K, the goal is to create n partitions, where n corresponds to the number of available cores. Each window associated with a particular key is consistently assigned to the same core, ensuring efficient and organized processing [34].

If window distribution is uniform, core workloads remain balanced. However, with fine-grained distribution, cores actively manage window boundaries (see Figure 3.10). The merger component then collects and orders the results, preserving partial ordering by default.

- Distributes data based on keys, ensuring related elements go to the same core.

Example in Skyline Query:

- Partition sensor data based on sensor ID or location.
- Ensures that all readings from a single sensor are processed by the same core.

Use Case: Used in stateful streaming skyline queries that track historical sensor trends.

3.3.8 Pane Farming

For centralized processing of sliding window aggregates, Li et al. [75] proposed a pane-based approach to improve efficiency by sub-aggregating and sharing computation. A parallel implementation of this approach was later introduced by Balkesen and Tatbul [74], leading to a pattern with desirable throughput, latency, and memory characteristics.

Each window consists of non-overlapping, continuous partitions called *panes*, with a pane size defined as $\sigma_p = \gcd(|W|, \delta)$. Each window w comprises r disjoint panes: $w = \{P_1, \ldots, P_r\}$, where $r = |W|/\sigma_p$.

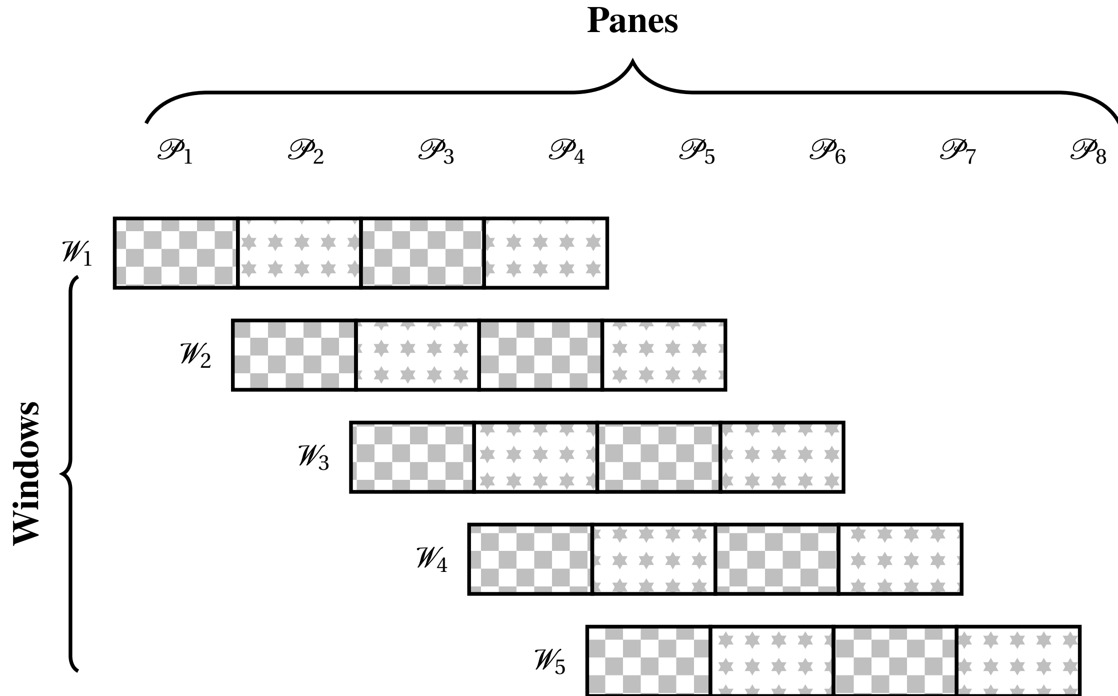


Figure 3.11: *Sliding window created with 4 different panes. Each pane is identified in 4 consecutive windows [26].*

- Optimizes sliding windows by breaking them into smaller panes.
- Reduces redundant computations for overlapping windows.

Example in Skyline Query:

- Instead of recomputing a full skyline every second, reuse previous computations for overlapping time intervals.

Use Case: Best for continuous skyline queries with small sliding windows, reducing computational overhead.

3.3.9 Window Partitioning

Window partitioning extends the map-reduce paradigm to data streams. If the computation can be expressed as a *map* function over window elements, the window can be divided among n cores, each responsible for computing the function F on their partition. A subsequent *reduce* phase may be required for aggregation.

- Divides incoming data streams into sliding or tumbling windows.
- Assigns different windows to parallel cores.

Example in Skyline Query:

- Sensors generate continuous air quality readings.

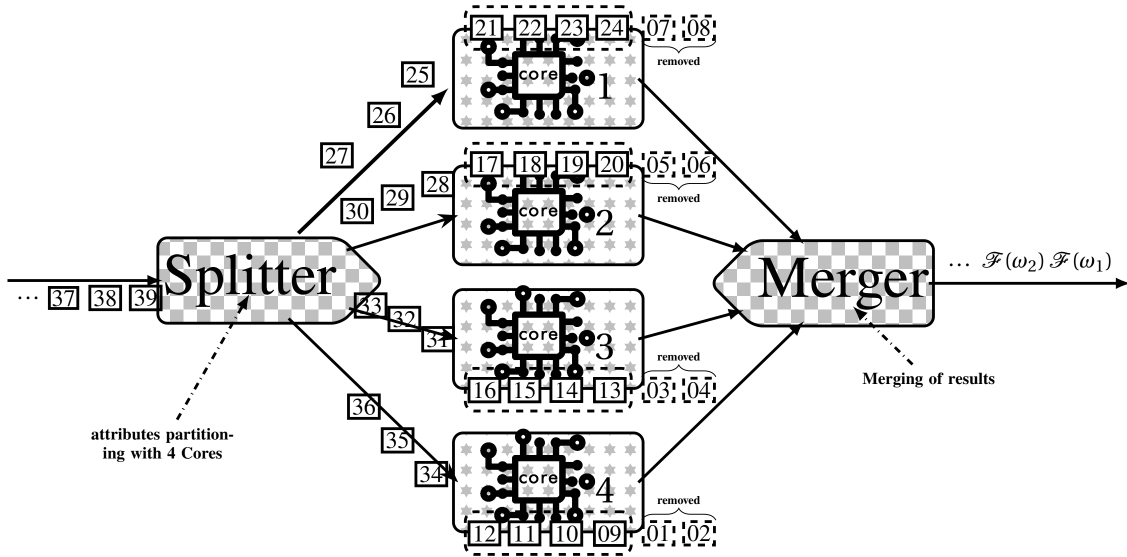


Figure 3.12: *Window Partitioning with 4 cores and one key. In the figure* $|W| = 8$ *and* $\delta = 4$ *[26].*

- Skyline queries are computed for recent windows (e.g., last 10 minutes).
- Each window computation runs independently.

Use Case: Suitable for real-time air quality monitoring with continuous updates.

Figure 3.12 illustrates this pattern, where tuples are assigned to two cores in a round-robin fashion. This approach is feasible if cores can process their partitions independently. Otherwise, alternative distributions ensuring data integrity are necessary. In a keyed scenario, each core maintains a partition for its logical substream.

The window partitioning pattern is particularly useful when computations follow a map operation, potentially followed by a reduction step. Iterative applications of map-reduce functions are also supported, including for keyed streams. The pattern enhances latency and throughput in proportion to the number of active cores. As tuples are partitioned without replication, load balancing challenges may arise if processing time varies significantly based on data values.

Comparison of Window Partitioning and Window Farming: Table 3.3 presents the difference between Window Partitioning and Window Farming patterns and table 3.4 summarises well known Parallel Patterns for Skyline Queries.

Feature	Window Partitioning	Window Farming
Data Splitting	Splits a single window into partitions	Assigns entire windows to different cores
Parallelism Focus	Within a window	Across multiple windows
Use Case	Single-window parallel skyline queries	Multi-window parallel skyline queries

Table 3.3: *Comparison of Window Partitioning and Window Farming*

Summary of Parallel Patterns for Skyline Queries

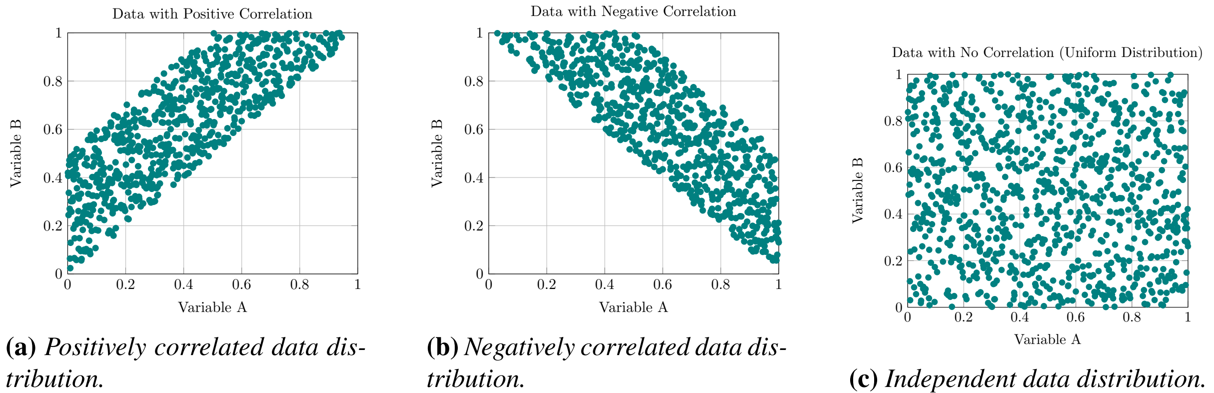
Tables 3.5, 3.6 and 3.7 summarize several developed skyline queries and their characteristics over complete and incomplete datasets.

Parallel Pattern	Definition	Example in Skyline Query
Pipeline	Sequential stages	Preprocessing → Skyline filtering → Output
Task-Farm	Independent tasks on cores	Parallel skyline computation for independent sensor data
Data Parallelism	Partitioned data processed independently	Skyline per geographic region
Reduce	Aggregates results	Merging regional skyline results
Window Partitioning	Parallelizes within a window	Skyline for one time window split across cores
Key Partitioning	Assigns data based on key	Ensuring same sensor data is processed by one core
Window Farming	Parallelizes across windows	Skyline for morning, afternoon, night in parallel
Pane Farming	Optimizes sliding windows	Avoid recomputing skylines for overlapping windows

Table 3.4: *Summary of Parallel Patterns for Skyline Queries [26].*

3.4 Skyline Cardinality

The skyline query allows users to specify whether they want attribute values minimized or maximized. Despite its apparent simplicity, the skyline query has significant computational implications. One of the fundamental challenges is determining the cardinality of the skyline set, denoted as $|SKY(P)|$, which is influenced by both the number of dimensions and the data distribution. The skyline cardinality can range between a single point and the total number of points in the dataset [78]:



(a) *Positively correlated data distribution.*

(b) *Negatively correlated data distribution.*

(c) *Independent data distribution.*

Figure 3.13: *Correlated, anticorrelated, and independent data distributions.*

$$1 \leq |SKY(P)| \leq |P| \tag{3.4}$$

Understanding the factors that affect skyline cardinality is essential for efficient query processing and optimization.

Impact of Data Distribution: The distribution of data significantly influences skyline cardinality. Figure 3.13 illustrates three common types of data distributions—correlated, uniform, and anticorrelated—in a two-dimensional space. These distributions are widely used to evaluate the performance of skyline query algorithms.

- **Correlated Distribution (Figure 3.13a):** In this distribution, a high value in one dimension corresponds to a high value in another. Since the points tend to align along a diagonal, skyline cardinality remains low. In an extreme case, all points lie on a straight line, making the problem effectively one-dimensional.

Approach year	Computation Space	Data Type	Technique	Query Type	Processing Type	Window type	Data Partitioning
Lazy, Eager 2006[100]	Partial	Complete	-	skyline	Centralized	Both	-
LookOut 2007[101]	-	Complete	-	skyline	Centralized	Time-based	-
MSQ 2008[171]	-	Complete	metric index	metric skyline	Centralized	-	-
FSQW 2009[172]	-	Complete	-	Frequent skyline	Centralized	Time-based	Angle-based
BOCS 2010[90]	Partial	Complete	Index R-tree	skyline	Distributed	time-based	horizontally split data
mMR-SUDS 2013[173]	-	Complete	-	-	Cloud	Time-based	-
skyline group 2015[174]	-	Complete	-	group skyline	Centralized	Time based	-
DSJQ 2016[105]	-	Complete	-	Skyline join	Distributed	-	-
k-LDS, PBA, ϵ-greedy 2016[108]	-	Complete	-	k Representative	Centralized	Count based	-
LookOut, Lazy, Eager 2016[110]	-	Complete	-	skyline	Centralized	Both	-
Gsky 2018[120]	Partial	Complete	Clustering	top-k,multi-Query	Centralized	Count-Time	grid-based
BJR-tree, NDcache 2019[116]	-	Complete	Index BJR-tree	skyline	Centralized	-	Object-based
NSCt framework 2020[21]	-	Complete	-	negative-skyline	Centralized	Both	-
IMSS, OIMSS, PMSS 2020[175]	-	Complete	-	Mutual skyline	Centralized	-	-
Dynamic k-dominant Skyline 2021[176]	-	Complete	Clustering	k-dominant	Centralized	-	-

Table 3.5: *An example of approaches to sequential continuous skyline techniques over a complete data stream.*

Approach year	Computation Space	Data Type	Technique	Query Type	Processing Type	Window type	Data Partitioning
Replacement, Bucket, Iskyline 2008[86]	Full space	Incomplete	Clustering Non-Index	skyline	Centralized	Count-Time	-
OURS 2009[177]	-	Uncertain	-	Probabilistic Skyline	Centralized	-	-
SSKY, MSKY, QSKY 2009[178]	-	Uncertain	-	q-skyline	Centralized	Both	-
DCRS 2009[179]	-	Uncertain	DC-tree index	Reverse skyline	Distributed	-	-
TDG algorithm 2010[180]	-	Uncertain	Clustering	Probabilistic	Centralized	Time based	-
RBSSQ 2016[181]	Full space	Incomplete	Clustering Index	skyline	Centralized	-	-
SIDS 2013[182]	Full space	Incomplete	Sorting Index	skyline	Centralized	-	-
SPQ 2017[183]	Partial	Incomplete	Sorting Index	skyline	distributed	-	-
ESB, UBB, BIG, IBIG 2015[184]	Full space	Incomplete	Clustering Index	top-k	Centralized	-	-
pnN, pmnN 2015[185]	-	Uncertain	-	n-of-N Skyline	Centralized	Time based	-
Baseline, DAG, BIB 2016[186]	Full space	Incomplete	Clustering Index	K-dominant	Centralized	-	-
uncertain reverse skyline 2017[187]	-	Uncertain	R-tree Index midpoint based	bichromatic Reverse	Distributed	-	Angle-based
GSS+ 2018[188]	-	Incomplete	-	Spatial skyline	Centralized	-	-
PFSIDS 2021[189]	Partial	Incomplete	Clustering	top-k,multi-Query	Centralized	Count-Time	Angle-based
CrowdSJ, PSJCrowd, ASJCrowd 2021[190]	-	Incomplete	- Index	skyline join	Distributed	-	-

Table 3.6: *Examples of approaches to sequential continuous skyline techniques over Incomplete and Uncertain data streams.*

Approach year	Data Type	Technique	Skyline Variant	Processing Type	Window Variant	Data Partitioning
CMS, AMS, DMS, APS 2013[191]	Uncertain	-	Probabilistic	Cloud	-	-
SPM, APM, DPM 2014[67]	Uncertain	-	skyline	Cloud	count based	-
DPF, PSS 2014[66]	Uncertain	grid index	skyline	Centralized	-	-
JRtree skyline 2015[145]	complete	sorting	skyline join	Distributed	-	-
APSS,CPSS,CUDA-Based PSS 2015[41]	Complete	-	Probabilistic skyline	Centralized	-	-
skyline Manhattan distance 2015[144]	Complete	-	-	Centralized	-	-
parallel eager 2016[5]	Complete	-	skyline	Centralized	Both	-
Q^+tree 2016[192]	Uncertain	Quad Tree Indexing	Reverse	Distributed	-	-
HashSkylineGPU 2017[193]	complete	Clustering	hash skyline	Centralized	-	-
uncertain reverse skyline 2017[187]	Uncertain	R-tree Index midpoint based	bichromatic Reverse	Distributed	-	Angle-based
parallel uncertain n-of-N 2018[194]	Uncertain	-	n-of-N skyline	Centralized	-	-
NP-SWJ, IP-SWJ 2018[68]	complete	Clustering	skyline join	Distributed	-	-
parallel uncertain n-of-N sky 2019[195]	Uncertain	-	n-of-N skyline	Centralized	-	-
PKDS 2019[196]	Uncertain	Capability Index	k-dominant	Centralized	-	-
pruning+monitoring reverse sky 2019[197]	Uncertain	-	Reverse	Distributed	-	Angle-based
dySky 2021[198]	Complete	-	skyline	Centralized	-	-
Basic, Parallel_Basic, Parallel_PS 2021[199]	Complete	-	skyline	Centralized	-	-

Table 3.7: *Example of approaches to parallel continuous skyline techniques.*

- **Uniform Distribution (Figure 3.13c):** Here, points are evenly spread across the space. This results in a higher skyline cardinality than the correlated distribution, but typically lower than the anticorrelated distribution. Although real-world datasets rarely exhibit perfect uniformity, this distribution is commonly used as a baseline for algorithm analysis.

- **Anticorrelated Distribution (Figure 3.13b):** In this case, a high value in one dimension corresponds to a low value in another. This distribution tends to produce the highest skyline cardinality. In the extreme case, where all points form a wide Pareto front, every point may become a skyline point, leading to $|SKY(P)| = |P|$.

Impact of Dimensionality: The number of dimensions (d) plays a crucial role in determining skyline cardinality. For a fixed dataset, increasing the number of dimensions generally results in a larger skyline set:

$$|SKY(P_d)| \leq |SKY(P_{d+1})| \tag{3.5}$$

For example, in a one-dimensional dataset where all values are unique, there is only one skyline point. However, as d increases, the number of skyline points grows. A theoretical estimation of skyline cardinality is given by Bentley et al. [200], who derived an approximation based on assumptions of uniformity and independence:

$$|SKY(P)| = O((\ln n)^{d-1}) \tag{3.6}$$

Buchta [201] provided a more refined bound:

$$|SKY(P)| = \frac{(\ln n)^{d-1}}{(d-1)!} \tag{3.7}$$

Further refinements by Godfrey [202] and Tiakas et al. [203] offer more precise estimates, which are crucial for cost estimation and query optimization.

Challenges of High Skyline Cardinality:

For high-dimensional datasets, skyline cardinality can grow significantly. According to Börzsönyi et al. [78], for a dataset with $n = 100,000$ points in a ten-dimensional space, skyline cardinality can reach approximately:

- $25,000$ points for a uniform distribution.

- $75,000$ points for an anticorrelated distribution.

Such large skyline sets are impractical for manual inspection, necessitating techniques to reduce skyline result sizes.

Techniques for Reducing Skyline Cardinality:

Several strategies can be employed to mitigate the issue of excessive skyline cardinality:

- **Ranking-Based Reduction:** Valkanas et al. [204] proposed ranking skyline points based on dominance relationships. For instance, returning the top-k skyline points based on a ranking function can significantly reduce result size.

- **Dimensionality Reduction:** Tao et al. [100] and Xia et al. [205] suggested computing skylines in a reduced number of dimensions ($d' < d$). Since higher dimensionality increases skyline size, reducing the number of attributes can help control cardinality.

- **Diversity-Based Selection:** Tao [206] and Valkanas et al. [207] introduced diversity-aware skyline selection. This method ensures that the selected skyline points are well spread across the space, enhancing representativeness while limiting redundancy.

By applying these techniques, it is possible to obtain a manageable subset of skyline points without losing essential information for decision-making.

3.5 Conclusion

This chapter has provided a comprehensive exploration of continuous skyline queries, detailing their fundamental concepts, computation techniques, and advanced processing methods. Beginning with an overview of entities and attributes, we established the foundational principles of skyline queries, including the concept of dominance, dataset representation, and dominance regions. These fundamental aspects are critical for understanding how skyline queries identify optimal data points in multidimensional spaces.

We then examined various computation strategies, distinguishing between in-core (main-memory) and out-of-core (secondary memory) skyline processing techniques. This distinction is particularly relevant for handling large-scale datasets where memory constraints necessitate efficient algorithms. Additionally, we discussed skyline queries in dynamic environments, highlighting the challenges posed by evolving datasets and the need for adaptive computational techniques.

Given the increasing volume of data and the necessity for real-time processing, we explored distributed and parallel approaches to skyline computation. We introduced high-level parallel programming models and specific parallel paradigms, such as SPODS, which enable efficient skyline computations over streaming data. Furthermore, we presented a taxonomy of parallel patterns tailored for windowed functions, including pipeline processing, task farming, data parallelism, and various windowing strategies. To illustrate these concepts practically, we provided a real-world example using environmental sensor data, demonstrating how different parallel processing patterns can optimize skyline computation in smart city applications.

A critical challenge in skyline computation is the rapid growth of skyline cardinality as the number of dimensions increases. We analyzed the impact of data distribution on skyline result sizes, differentiating between correlated, uniform, and anticorrelated datasets. Moreover, we examined mathematical models that estimate skyline cardinality, which are essential for optimizing query execution and reducing computational complexity.

In summary, this chapter has highlighted the theoretical foundations, computational strategies, and optimization challenges of skyline queries in continuous and high-dimensional settings. The insights provided here lay the groundwork for further research into efficient skyline computation techniques, particularly in real-time, parallel, and distributed environments. Future work may focus on refining parallel models, improving skyline estimation techniques, and developing adaptive algorithms for dynamic and large-scale datasets.

CHAPTER

4 VARIATIONS OF SKYLINE QUERIES

"Repetition makes us feel secure and variation makes us feel free."

ROBERT HASS

Chapter content

4.0.1 Dynamic Skyline Queries . . . 66
4.0.2 Group skyline computation . . . 67
4.0.3 Spatial Skyline Queries . . . 68
4.0.4 Metric Space Skyline Queries . . . 75
4.0.5 Constrained Skyline Query . . . 78
4.0.6 Range-Based Skyline Queries . . . 79
4.0.7 Reverse skyline queries . . . 82
4.1 Applications of Skyline-Based Queries . . . 83
4.1.1 Multi-criteria decision making . . . 83
4.1.2 Machine learning . . . 84
4.1.3 Network analysis . . . 86
4.1.4 Other interesting applications . . . 86
4.2 Conclusion . . . 89

Due to their broad range of applications in areas like multi-preference analysis and decision-making, skyline queries have attracted a lot of attention. Several research groups have proposed expanding the definition of skyline based on variations of the dominance relationship in order to better meet the needs of various applications. This chapter looks at how the traditional skyline query has been modified to generate various variants of dominance-based queries in light of the growing number of proposed alternatives. It examines the many features that remain unchanged in a variant of the dominance relationship, which keeps the original benefits but allows for greater adaptability to new uses. It also explores alternative dominance arrangements proposed by scholars and obtained by altering the conventional dominance arrangement's characteristics.

4.0.1 Dynamic Skyline Queries

A *dynamic skyline query* extends the traditional skyline query by considering a reference point that can change dynamically. Instead of comparing points directly using fixed attributes, the query first transforms the data according to a given reference and then applies the skyline computation.

Consider a traveler who wants to purchase a flight ticket. The traveler has two preferences: (1) minimize the ticket price and (2) minimize the flight duration. However, the traveler's location may vary (e.g., they might be at different airports). This means that the perceived advantage of a flight depends on the traveler's current location.

If the traveler is currently in location LocX, then the dynamic skyline will compute the skyline using only relevant flights (e.g., those from LocX or those that require a short additional journey to the departure airport). This dynamic consideration ensures that the query result is adapted to the user's specific context.

Historically, Papadias et al.'s [89] dynamic skyline query is among the earliest and most widely used variants of the classical skyline query. Processing algorithms for the classical skyline query assume static attribute values for database objects; processing algorithms for the dynamic skyline query, on the other hand, either calculate the attributes of the data objects "on-the-fly" during the execution of their algorithmic steps or pre-process them based on the static attribute values of the objects. The concept of dynamic dominance is central to this query type. A data object t is considered to dynamically dominate a different object r in relation to a query point q if and only if t is closer to q than r is on at least one axis and is not further away from q than r is on all the other axes, under the assumption that lower values are preferable on all the axes (attributes) of the space. To determine how close an object is along every axis, Authors use the Euclidean distance between the query point and the object's respective static attributes. Therefore, in this field, unlike in static domains, the skyline of a dataset is generated dynamically based on a user's preference q. In other words, the classical skyline query is a special case of the dynamic skyline query with the query point q at the space's origin, provided that all attributes have only positive values.

The example of a traveler looking for inexpensive flight ticket price with optimal flight duration is illustrated in Figure 4.1. The query point $q(qx, qy)$ in the figure may represent the preference expressed for a flight ticket costing qy dollars (y-axis) that takes qx durations. All qualified tickets that are not dominated by another in relation to q must be obtained in order to find the "ideal" ticket q that meets the passenger's budget and personal preferences.

A skyline analysis will suggest the most compatible match. Every ticket's data point t is projected onto a new space where its coordinate in each dimension is equal to the absolute difference between the data point t and the query point q. That's why the dynamic duration between two tickets $t(tx, ty)$ is equal to $|ty - qx|$ and the dynamic price between two tickets $t(tx, ty)$ is equal to $|tx - qx|$. All tickets outside of q's quadrant A will have their dynamic attributes calculated as if they were in q's quadrant A. A "static" skyline in the projected space consists of the points $t1, t6$, and $t10$, which are the tickets that do not currently dominate any other ticket in the data set in relation to the query predicate q.

Papadias et al. [89] propose an efficient algorithm to process the dynamic skyline query, which is an extension of the well-known BBS algorithm for handling the static skyline query given an indexed dataset using the R-tree. The only difference between this and the BBS algorithm is that the R-tree entries are now added into the heap and ordered based on their mindist distance to the query point, which mindist is determined on-the-fly when the entry is considered for the first time (recall that in the classical BBS algorithm, the R-tree entries are inserted into

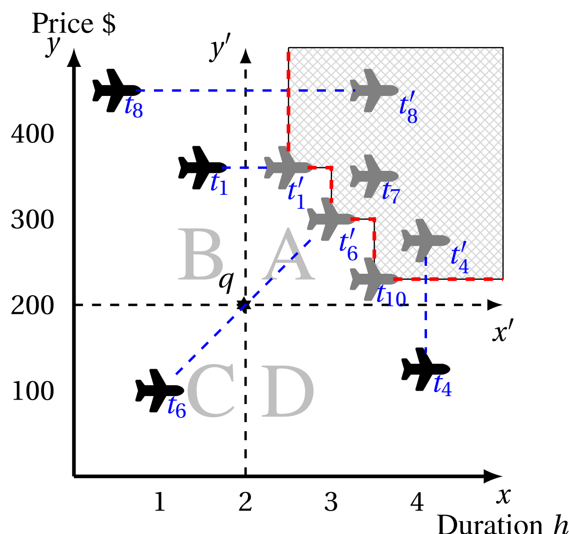


Figure 4.1: *Dynamic Skyline Query.*

the heap and ordered according to their distance to the origin point of the space). It takes little effort to adapt some other well-known methods for processing the static skyline to processing the dynamic skyline query, too. For unindexed data sets, Borzsonyi et al.'s [78] simple extension of the BNL algorithm calculates all dynamic coordinates on the fly and then evaluates each data point independently.

To safely prune parts of the dataset and speed up the execution of any future queries, Sacharidis et al. [208] propose a cache-aware algorithm that makes use of the results of the most valuable of the past executions of the dynamic skyline query. On top of that, Han et al. [117] recently developed a strategy to efficiently process the dynamic skyline query over massive data by first retrieving the tuples in dynamic sorted lists in a round-robin fashion till an early termination condition is satisfied and then computing the dynamic skyline results.

Finally, several recent research efforts, such as Zeighami et al.[19] and Wang et al. [209], have introduced frameworks to address the demand for a secure skyline calculation in encrypted datasets for services in which dealing with sensitive data is an essential concern, such as healthcare or data outsourcing. Since the dynamic skyline set must be located through processing the encrypted data objects, this formulation of the problem necessitates that this framework's primary objective be security. In addition, the proposed query processing algorithm needs to be protected by strong guarantees that no sensitive information (either related to the stored data or to the preferences of the query issuer) will be leaked to any third party spying on the data during its execution.

4.0.2 Group skyline computation

A *group skyline query* extends skyline queries to groups of points rather than individual points. The goal is to select a group of objects such that each group member collectively dominates other possible groups based on certain criteria.

Consider a travel agency that wants to offer bundled flight packages for group travelers. The agency must select a set of flights where the collective ticket prices and travel durations form an optimal skyline.

For instance, assume a group of three travelers must book tickets from different airlines, and their preferences are to minimize the total cost and average duration. Given the following flight

options:

Traveler	Ticket Price ($)	Duration (hours)
T1 (Flight A)	400	6
T2 (Flight B)	450	5.5
T3 (Flight C)	420	5

Table 4.1: *Flight ticket price and duration for group travelers.*

A group skyline computation would identify sets of flights where the total ticket price and the average duration are minimized, avoiding combinations that are dominated by others.

Et al. [83] conducted a study on the group skyline query, which relies on the dominance relationship between groups of equal size. The evaluation of the dominance relationship is conducted based on the collective attributes' values.

The authors found the dominance relationship between groups by using the aggregate points in the conventional way, with a single aggregate point representing each group. The aggregate point is the result of applying an aggregate function to the attribute values of each individual point in the set. However, only a small subset of aggregate functions, such as SUM, MIN, and MAX, have been discussed in prior works for calculating aggregate points. To aggregate the points, [83] employed SUM. SUM intuitively represents the sum total of a group's individual strengths. Due to the inability to capture all Pareto optimal groups, the skyline groups constructed using these methods are incomplete.

Using the same attribute values of k points to form a group, [83] defined and studied the group skyline query, and then compared the dominance relation between the groups using conventional dominance. Some aggregate functions, like SUM, were the most popular among the calculate functions used in these tasks. In practice, it can be challenging to determine which aggregate function is best.

Using a hash table, dominance graph, and matrix to store dominance information and incrementally update the results, the paper [210] developed an effective skyline group algorithm in a data stream, but it was not helpful to the Pareto optimal groups.

Over the data stream in a wireless sensor network, Dong et al. [211] looked at the problem of locating G-Skyline groups. The authors provided the sharing strategy and then proposed two algorithms (PAA and PEA) for efficiently computing new G-Skyline groups whenever a new point arrives or an existing point expires.

4.0.3 Spatial Skyline Queries

Spatial skyline queries focus on skyline computations where objects have spatial locations. The skyline is determined based on spatial attributes, such as distance to a reference point.

Consider a traveler choosing a flight based on two factors: (1) minimizing ticket price and (2) minimizing the distance between the destination airport and their hotel as illustrated in figure 4.2.

A spatial skyline query returns flights that are not dominated by others in terms of both price and distance. In this case, flights t1, t6 and t7 might be in the skyline, while other flights $t2, t3, t4, t5, t8, t9, t10$ is dominated because it has both a higher price and a greater distance than t6.

Every dynamic attribute is calculated as the distance of the data objects to a query point in the multi-source spatial skyline query, which is an extension of the dynamic skyline query. To be more precise, the spatial skyline query, given a set T of data points and a set Q of query points, returns those points of T that are not spatially dominated by any other point of T. The distance between the data objects to all query points is used to determine the spatial dominance. In other words, a data point t is considered to spatially dominate another point r with respect to Q if and only if $d(t,q) \leq d(r,q_i)$ for all $q_i \in Q$ and$d(t,q_j) < d(r,q_j)$ for some $q_j \in Q$, where $d(t,q)$ is the Euclidean distance between p and q as illustrated in figure 4.2.

Business planning, crisis management, vacation planning, recommender systems, etc. are just some of the many possible applications of spatial skyline queries. It's possible, for instance, that a traveler would rather stay at one of the hotels in the city's spatial skyline than at one of the attractions, provided that the attractions remain in their usual locations.

As far as we know, the first people to investigate the spatial skyline query analysis problem were Sharifzadeh and Shahabi [212]. Specifically, they proposed two index-based algorithms, the B^2S^2 (branch and bound spatial skyline) algorithm and the VS^2 (Voronoi-based spatial skyline) algorithm to process this query quickly and effectively. B^2S^2 uses a top-to-bottom traversal of the conventional R-tree spatial index to look for candidate points along the spatial skyline. After identifying a candidate point on the spatial skyline, B^2S^2 uses the R-tree's expansion to reach the node with the smallest mindist distance to the identified data point and then verifies the node's dominance relative to the other candidate points.

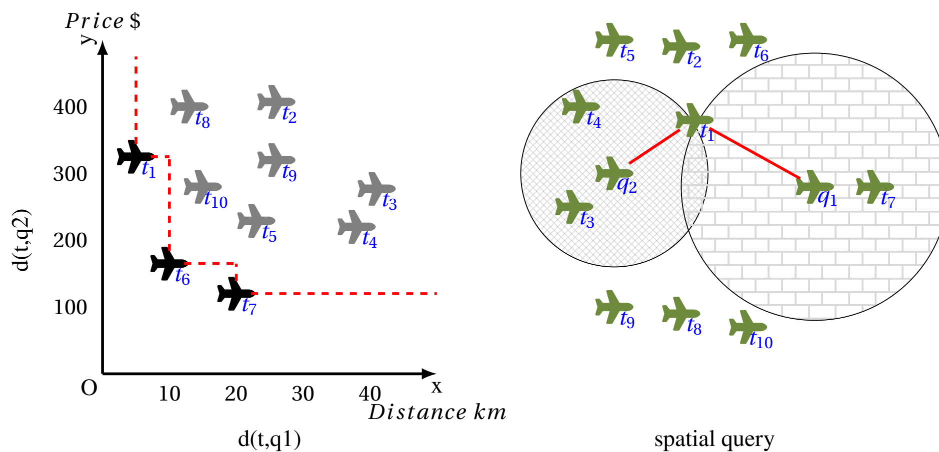


Figure 4.2: *Spatial Skyline Query.*

The $VS2$ algorithm, on the other hand, uses a Voronoi diagram built from the data points provided as input. To maintain locality, the input data points are sorted into disk pages based on their Hilbert values. Following the end of the convex hull calculation, VS2 begins its search with the data points closest to the query points and proceeds to the neighbors of the data points it has already visited using the Voronoi diagram. In order to determine spatial skyline dominance, VS2 compares each data point to all the spatial skylines discovered thus far. Until all Voronoi cells (or data points) that could contain spatial skyline points have been visited, the process will continue. Because of the high computational cost, Son et al. [213] modified VS2 by performing fewer spatial dominance tests.

While the aforementioned techniques examine the spatial skyline query using the Euclidean (or $L2$) distance, this measure is inefficient in constraint-based environments such as urban neighborhoods or road systems. Using the Manhattan (or $L1$) distance as an example, Son et al. [214] propose a method for computing the spatial skyline in a grid network, with a focus on

urban residential areas. However, Deng et al. [215] looked into the spatial skyline query issue in transportation networks. In this context, we can think of the shortest path distance between the data and query points as the equivalent of the Euclidean distance. Given a set P of data objects and a set Q of query points on a road network, we can map each data object onto a $|Q|$-dimensional point, where the value of the i-th dimension indicates the shortest path from the object to the i-th query point. The data objects that are not dominated by the $|Q|$ dimensions are then retrieved by the multi-source skyline query. Safar et al. [216] offer a different approach to the same issue.

Nearest and Farthest Spatial Skyline Queries

Nearest and Farthest Spatial Skyline Queries extend spatial skyline queries by focusing on objects that are either closest or farthest from a reference point, while still considering skyline criteria as illustrated in figure 4.3.

A traveler may want to book a flight not only based on price but also based on proximity to a major city. Depending on their goal, they may be interested in either:

- **Nearest Spatial Skyline:** Finding flights with the lowest ticket prices that land at the closest airport to the city center.
- **Farthest Spatial Skyline:** Finding flights that are inexpensive but land at the farthest airport (e.g., to avoid congestion or for better transfer options).

For the *Nearest Spatial Skyline Query*, t1, t6 and t7 might be in the skyline, as they offer a balance of cost and proximity as presented in figure 4.3. For the *Farthest Spatial Skyline Query*, t2, t9 and t3 might be a dominant option since it offers a low price while being farther from the city center.

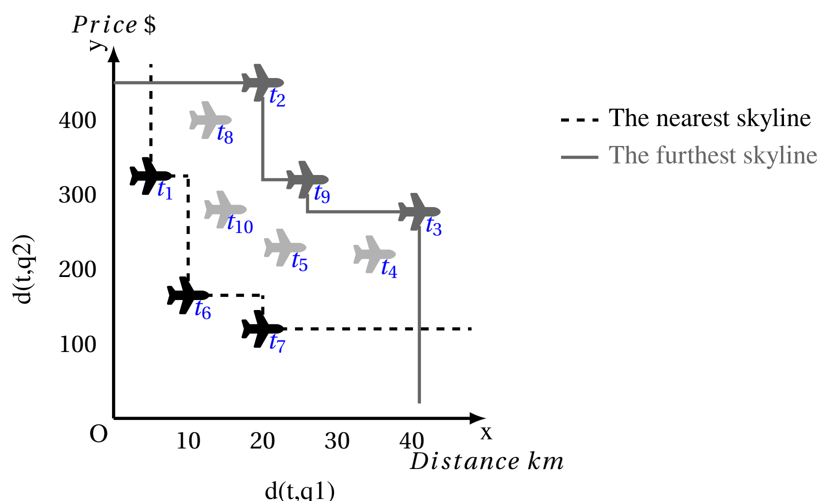


Figure 4.3: *Nearest and the Farthest Spatial Skyline Queries.*

You et al. [217] present the farthest spatial skyline query, which obtains the data points that are farther than all the other data points from the query points and thus satisfy the opposite goal of identifying eligible points that are the furthest from location query points. This query also has a wide range of applications, including but not limited to assisting with decision-making processes, trip and event planning, facility location, crisis management, and so on.

Fort et al. [218] also investigate the same topic. When making recommendations to a mobile user (a single query point), Kodama et al. [219] use the spatial skyline query to account for the user's location and preferences when suggesting nearby objects, such as restaurants. The non-spatial attributes of the objects here serve as a proxy for the user's preferences. Since this approach incorporates non-spatial attributes into the spatial skyline query, it has a wide variety of potential uses, such as location recommendation, trip planning, and advertising.

Spatio-Textual Skyline Query

A *spatio-textual skyline query* extends traditional skyline queries by considering both spatial attributes (e.g., distance) and textual relevance. This variation is particularly useful in applications where users search for spatially relevant objects that also match specific textual keywords as illustrated in figure 4.4 and table 4.2.

Consider a traveler who wants to book a flight while considering two factors: (1) minimizing ticket price and (2) minimizing the distance from the destination airport to their hotel. Additionally, the traveler prefers specific airlines based on past experience or loyalty programs.

Given the following flight options:

Flight	Ticket Price ($)	Distance to Hotel (km)	Airline
t1	320	7	Air France
t6	180	10	Lufthansa
t7	140	22	Air France

Table 4.2: *Flight ticket prices, distances, and airline preferences.*

A spatio-textual skyline query ensures that flights are selected not only based on price and distance but also on the textual keyword constraint (e.g., preferred airline). If the traveler prefers *Air France*, the skyline will prioritize flights t6 and t7 while eliminating those that are dominated as illustrated in figure 4.4.

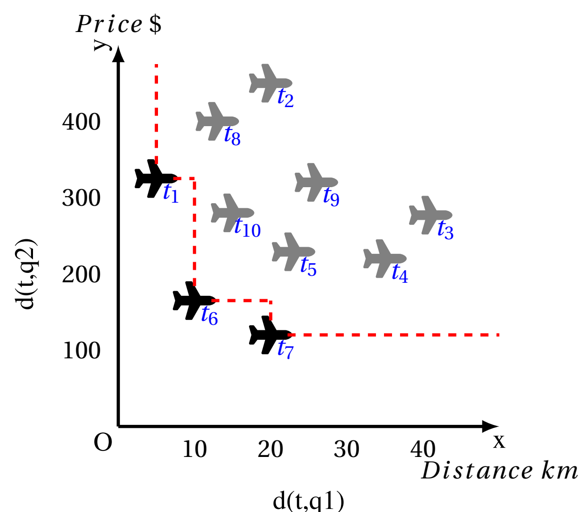


Figure 4.4: *Spatio-textual Skyline Query.*

In their paper, Shi et al. [220] propose an extension of the above problem in which the query takes into account multiple query points; they refer to this form of query as a spatio-textual skyline query. As a result, the skyline points in this query can be chosen based on both

their distance from a set of locations and their relevance to a location of set keywords. Modern applications have made it common practice to elaborate on preferences with textual descriptions, so this query is specially developed for such situations.

Direction-Based Skyline Query

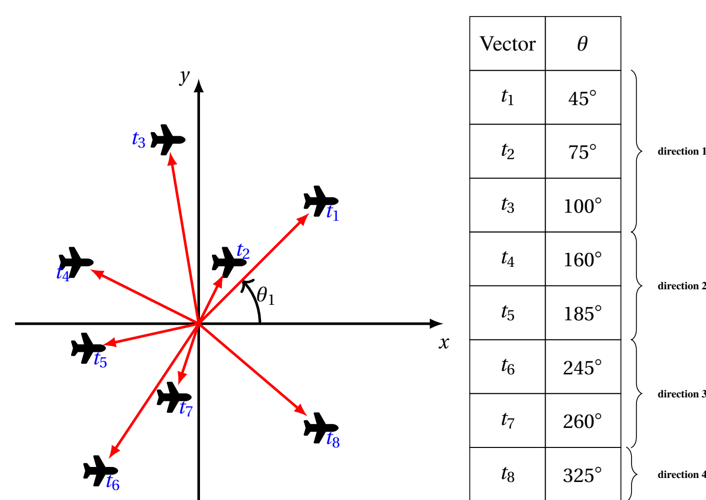
A *direction-based skyline query* incorporates directional constraints into skyline computation. This is particularly relevant in transportation applications where users are interested in flights heading in a specific direction as illustrated in figure 4.5.

Consider a traveler who needs a flight that (1) minimizes ticket price, (2) minimizes the distance from the destination airport to their hotel, and (3) heads toward a specific geographic region, such as Europe or North America.

If the traveler specifically wants flights heading toward *Europe*, the skyline query will filter out non-European flights and return t1 and t3 as the optimal choices.

The direction-based spatial skyline query introduced by Guo et al. [221] retrieves the nearest points around a mobile user (a single query point) by considering not only the distance between these points and the mobile user but also the direction. In Figure 4.5, there are eight potential destinations surrounding the query point q (origin point), which represents the user's location. In the diagram, q is the source of the vectors $t1$ till $t8$. If the angle between two vectors is less than some specified acceptable threshold, for instance, less than 60 degrees, then the two vectors are considered to be pointing in the same direction. This indicates that points $t2$ and $t3$ are also moving in the same direction as $t1$, as shown in figure 4.5 $t2$ is the most favorable recommendation because it is the closest to q out of $t1, t2,$ and $t3$. $t2$ also dominates the other two data points in the same direction. The same holds true for the comparisons between points 5 and 4, 7 and 6, and 8, with no other data point moving in the same direction. Points 2, 5, 7, and 8 make up the direction-based spatial skyline in relation to q because no other object dominates them.

For situations where the user is moving in a straight line, the solution proposed by Guo et al. [221] for direction-based spatial skyline analysis allows for maintaining continuous spatial skyline results.



Vector	θ
t_1	45°
t_2	75°
t_3	100°
t_4	160°
t_5	185°
t_6	245°
t_7	260°
t_8	325°

Figure 4.5: *Direction-based Skyline Query.*

Shen et al. [222] conducted an in-depth analysis of the aforementioned work, and they concluded that it presented issues that burdened its effectiveness and could not be ignored. Too

many data points may be returned by the method described above by Guo et al. if an acceptance threshold is set so that a small angle (say 10 degrees) indicates that two data points belong in the same direction. Alternatively, if the acceptance threshold angle is set to greater values (for example, 60 deg), then only a small number of data objects will be contained in the direction-based spatial skyline (hence the too few answers problem), which is also disappointing in terms of obtaining a sufficient selection of surrounding data objects for the user.

In order to overcome these limitations, Shen et al. [222] proposed a new method for solving the direction-based spatial skyline problem that retrieves the most relevant points in the closest area. Shen et al. also proposed algorithms for the same query problem that can manage and retrieve non-dominated surrounding data objects of arbitrary shape in relation to the user's location.

Wang et al. [165] noted that due to the exponential growth of spatial data, it is no longer feasible to use a single processing node to answer the spatial skyline query on massive spatial datasets. To deal with the typical spatial skyline query on massive datasets, they developed a state-of-the-art parallel framework based on Map Reduce. The method begins by determining the convex hull of the query points. The authors then suggested the idea of independent regions, which they use to classify input data points into distinct groups. In the end, we calculate the local spatial skyline in each autonomous region simultaneously and build the global spatial skyline from the union of all the local spatial skyline sets.

Multi-source skyline queries in Euclidean space have been the subject of research in [212]. Unfortunately, this work is based on Euclidean distance, which prevents its use for network distances. In this paper, the authors discuss relative skyline queries, in which the user-provided data points rather than the static attribute values of the data space determine the minimization. Dynamic skyline queries, or relative skyline queries, are also used.

The processing of skyline queries on road networks has gained a lot of attention in recent years. Deng et al. [215] introduce the multi-source skyline query (or MSQ for short) by applying the concept of the spatial skyline sharifzadeh2006spatial to road networks. Given a set of m query objects and a set of n data objects in a road network, we can map each data object o to an m-dimensional point, where the value of the i-th dimension represents the road distance between o and the i-th query object. Then, MSQ gets the vantage points from the skyline that aren't dominated by m. Deng et al. [215] suggest a group of algorithms: the Euclidean Distance Constraint (EDC), the Lower Bound Constraint (LBC), and the Collaborative Expansion (CE) to address the MSQ issue. EDC and LBC both use Euclidean distance as a minimum road distance to cut down on data objects in order to speed up the search process. Concerning CE, the pruning strategy involves first searching the road network for potential skyline objects, beginning with the m query objects. When an item has been seen m times, we can get rid of the ones that have never been seen before. But these algorithms may produce an excessive number of candidates, leading to an inefficient calculation of travel time and fuel costs. The authors take into account a number of query objects concurrently.

Numerous situations in which there is inherent data uncertainty are distributed, such as when integrating information from multiple, distributed data sources using fuzzy similarity scores. In contrast to the standard skyline query, processing a skyline query in road networks necessitates considering the location of the query object.

As the shortest path lengths on road networks from data to query objects were defined, the dynamic attributes of each mapped point were also defined. The developed algorithms can only be used for road network applications and not for generic metric skyline retrieval in any other metric space.

By demonstrating the efficiency of a pruning dynamic skyline query, Chen et al. [171] not only solved dynamic skyline queries in metric spaces but also provided an appropriate pruning technique to calculate dynamic skyline.

When applied to spaces with dynamic attributes defined in metric spaces, the proposed method calculates the dynamic skyline. To resolve metric skyline queries via metric index, they proposed a pruning mechanism that is both efficient and effective. To safely and efficiently prune the dataset (since distance computation is very expensive in metric spaces), the authors used a triangle-based pruning technique that includes the triangle inequality property.

You et al. investigated the furthest spatial skyline query ($FSSQ$) problem from the standpoint of facility or business locations in [217]. These spatial queries are useful for finding places far from problematic areas, such as unpleasant facilities or competitors.

Initially, the issue is addressed through the utilization of a basic algorithm, followed by the presentation of a more streamlined and effective progressive algorithm. The latter exhibits superior performance in comparison to the former by leveraging spatial locality. Additionally, they have devised an effective algorithm that allows for a balance between precision and expediency. They invert the concept of proximity to represent distance. A point of interest, denoted as p, within the set P is considered desirable if it satisfies the condition of being the farthest point from all query points within the set Q, without any other point $p\prime$ in P being farther. In contrast to the spatial skyline points in close proximity that tend to cluster within the confines of the query convex hull, those located at greater distances are more evenly distributed throughout the data space. The authors demonstrate the absence of duality between the nearest and farthest spatial skylines.

There are two suggested solutions. One that uses an R-tree to implement [217]'s Improved Distributed Skyline (IDS) algorithm, which transforms the original problem into an n-dimensional general skyline problem. The Branch and Bound Farthest Spatial Skyline ($BBFS$) improves upon the original proposal by making use of the geometric properties of the problem, which the initial one did not.

The attributes of objects in real-world applications are constantly evolving. In order to accommodate location-based queries for objects with time-varying attributes, Huang [223] developed the continuous d-Skyline query. By fusing two delicate data structures (the object attribute control matrix and the road distance sorting list), we are able to address the issue of $Cd\epsilon - SQ$ attributes evolving over time in the road network.

Sohail et al. [224] propose two new kinds of queries that add social relevance components to the semantics of traditional spatial queries.

In recent times, the spatial skyline problem has experienced expansion in various dimensions, including but not limited to uncertainty and semantics. Elmi and Min [188] have devised effective methodologies for computing the spatial skyline over imprecise data regarding a collection of query points situated in diverse locales. The utilized data structures are R-tree and Voronoi diagrams.

The present study introduces an algorithm [218] designed to address spatial queries within the Euclidean plane. This is achieved through the consolidation of geometric attributes associated with both the nearest and farthest spatial skyline queries.

In contrast, [225] investigated the issue of range-based skyline queries ($CRSQs$) in road networks. However, it has been argued by [225] that the current methodologies employed for the

continuous skyline query impose constraints on the scope of the skyline query, restricting it to a particular region within the road network. Consequently, this limitation results in the absence of valuable query outcomes. Three algorithms were proposed to decrease the number of intersection nodes in the road network. These algorithms include the intersection node aggregation algorithm ($INAA$), the link remolding algorithm (LMA), and the link fitting algorithm (LFA).

Parallel Spatial Skyline queries

Parallel spatial skyline queries optimize skyline computation by leveraging parallel processing techniques. This is essential when dealing with large datasets, such as global flight schedules.

Consider an airline booking system that processes thousands of flight options daily. The system must compute skyline queries based on ticket price and distance while ensuring fast response times.

Parallelization Approach:

1. Partition the flight dataset into smaller subsets, each handled by a different processor.
2. Compute local skylines within each subset in parallel.
3. Merge local skylines to derive the final skyline set.

The study conducted by Siddique et al. [226] applies to the effective maintenance of all outcomes of k-dominant skyline queries. The utilization of dataset properties has been employed by the authors to facilitate the processing of skyline queries, a task that can prove to be difficult for datasets that are both update-intensive and voluminous.

In their publication, Wenly et al. [165] present a parallel solution for the $NSSQ$ problem utilizing the MapReduce technique. The authors employ a grid-based partitioning scheme to recursively divide the dimensions of the data into multiple parts. The framework presented applies to the utilization of MapReduce in the computation of independent regions following the generation of a query's convex hull. The ultimate skyline is the combination of all the regions that constitute the skyline.

The authors have noted that due to the exponential increase in spatial data, it is no longer feasible to execute the spatial skyline query on extensive spatial datasets using a single processing node. An advanced parallel framework based on MapReduce was introduced to tackle the conventional spatial skyline query on datasets of significant scale. The methodology initially computes the convex hull of the queried points. The authors propose the idea of independent regions, which entails partitioning the input data points according to their corresponding independent regions. Ultimately, the concurrent computation of the local spatial skyline within each autonomous area is executed, culminating in the establishment of the comprehensive spatial skyline through the combination of all local spatial skyline sets.

The spatial-GPU [227] methodology employs a multi-level approach to separate autonomous regions for the purpose of examining potential data points. Skyline queries are evaluated concurrently in various autonomous regions. Hence, this approach has the capability to operate concurrently with GPUs or MapReduce. This methodology has been specifically developed for a specific kind of skyline query known as spatial skylines. The solution's applicability to general skyline queries is not readily extensible.

4.0.4 Metric Space Skyline Queries

Metric space skyline queries extend skyline computation to non-Euclidean spaces where distances are defined by a metric function instead of traditional Euclidean distance as illustrated in

figure 4.6.

Consider a traveler booking a flight based on two criteria: (1) minimizing ticket price and (2) minimizing an alternative distance metric, such as travel time instead of geographical distance.

Instead of using Euclidean distance, this skyline query considers *travel time* as the metric function, leading to a different optimal selection than a standard spatial skyline query.

Chen and Lian [228] proposed the metric skyline query as an extension of the spatial skyline query. This query involves computing the dynamic attributes of each data object using a set of dimension functions. The dissimilarity between spatial and metric skyline queries originates mainly from the utilization of diverse distance functions in the latter, which encompasses not only the Euclidean distance function but also other metric functions. The metric skyline query stands in contrast to the spatial skyline query in that it is a more all-encompassing query type, not limited to spatial data application domains where objects can be expressed as Euclidean vectors.

To enhance the precision of an image similarity search, multiple query images could be employed. These pictures might come from a scene in a video sequence or from footage of a place that various security cameras have recorded. This instance showcasing the practicality of the metric skyline query involves the mapping of images onto a metric space, where the similarity between any two images can be evaluated using metrics such as the Hausdorff distance Huttenlocher et al. [229]. Thus, in a given set of query images, image p is said to dominate image r if it exhibits greater or equal similarity to all query images compared to image r. The metric skyline query aims to retrieve images from a database that do not dominate any other images in relation to a specific set of query images.

In a real-world scenario, the database of user profiles (data points) in one online social network may be vulnerable to an attack using the metric skyline query after identification and validation of the user profiles (query points) maintained by a particular physical entity across various online social networks. This attack aims to identify potentially suitable profiles, including the profile of the aforementioned physical entity in the said online social network. The aforementioned scenario exemplifies the challenge of establishing connections between diverse social identities across disparate virtual social platforms. In this problem, a mixed function can be utilized to perform a distance-based comparison between any two user profiles. The function will determine the overall weighted similarity score of the text-based personal attributes of the profiles, utilizing a text similarity comparison metric as surveyed in Elmagarmid et al. [230]. Additionally, the function will consider the profile images, if available, in the two separate profiles using the Hausdorff comparison metric. This approach follows a profile correlation model similar to that proposed in Kokkos et al. [231].

It is crucial to keep in mind that the processing of skyline queries in a metric space is unable to utilize any geometric information for the purpose of guiding the pruning process. The applicability of certain techniques, such as the method used to obtain the spatial skyline in Euclidean space, is limited in the context of generic metric skyline scenarios. In the context of a metric space, it is necessary for the distance function $d()$ utilized in the metric skyline query to adhere to four distinct properties when considering any given set of three points $p, r,$ and q. The distance metric d satisfies four properties: (i) positivity, which states that $d(p,q)$ is greater than or equal to zero; (ii) identity, which states that $d(p,r)$ equals zero if and only if p equals r; (iii) symmetry, which states that $d(p,q)$ equals $d(q,p)$; and (iv) the triangle inequality, which states that $d(p,r)$ is less than or equal to the sum of $d(p,q)$ and $d(q,r)$. Consequently, these aforementioned properties represent the exclusive means by which to enable the metric skyline search.

Chen and Lian [228] proposed the metric skyline query and utilized indices in the metric

space to execute the query. The utilization of indices can effectively diminish the search space by promptly eliminating unqualified data objects. The proposed approach employs the M-tree, a dynamic access technique for data objects in a metric space that was introduced in Skopal's [232] work. The M-tree algorithm employs a technique wherein a subset of data objects is designated as centers or local pivots of hyper-spheres. The remaining objects are then distributed among these hyperspheres to construct a well-proportioned and condensed hierarchy. The illustration presented in Figure 4.6 showcases a compact M-tree that exists within a two-dimensional metric space. This M-tree comprises ten distinct data points, namely $t1, t2, \ldots,$ and $t10$, along with minimum bounding circles $e0, e1, \ldots,$ and $e6$ that have been constructed over these data entities. $e0$ is a representation of the M-tree's root. The Euclidean distance is utilized as the similarity measure for the purpose of demonstration. The given instance presupposes the presence of two distinct inquiry points, namely $q1$ and $q2$.

By building the M-tree from its root up and using the BBS algorithm, Papadias et al. [89] proposed a best-first method of searching for metric skyline points. The algorithm retrieves the root node $e0$ from the tree in the first stage, then inserts its descendant nodes $e1$ and $e2$ into an auxiliary heap with the structure (entry; key). The key value is computed as the aggregate of the minimum distances between the entry (e.g., $e1$) and each query point. Given that $q1$ and $q2$ are located within the cycle of $e1$, the first step entails removing entry $e1$ from the heap and then inserting its descendant nodes $e3$ and $e4$ into the heap. Following that, the element $e3$ is extracted from the heap, and its constituent data entries, namely $t1$ and $t2$, are subsequently reinserted into the heap. Subsequently, the heap is subjected to a pop operation on $t2$, owing to the fact that this particular data point exhibits a comparatively smaller sum of metric distances to $q1$ and $q2$. Consequently, $t2$ is incorporated into the metric skyline set. Subsequently, the element $e4$ is extracted from the heap, leading to the return of its associated data entries $t3, t4,$ and $t5$ to the heap. Subsequently, the element denoted as $t3$ is extracted from the heap. As per the available data entries, it possesses the minimum metric distance to $q2$, and thus, it is incorporated into the metric skyline set. Subsequently, employing an analogous methodology, the heap is queried to retrieve the data entry $t1$. It is noteworthy that the attribute vector of p2 surpasses that of $t1$, thereby leading to the elimination of $t1$. Upon completion of the procedure, which follows the identical steps of the conventional BBS algorithm, the metric skyline set is constituted by the points $t2$ and $t3$. The utilization of the M-tree and the BBS algorithm offers a significant benefit in that certain branches of the tree can be eliminated, thereby circumventing the typically resource-intensive computation of metric distance functions between the query points and a substantial portion of the dataset.

The approach put forth in Chen and Lian's [228] work shares some similarities with the metric skyline query processing method that Skopal and Lokoc [233] propose. The primary distinction lies in the replacement of the M-tree with the PM-tree Skopal [232] in the latter approach, which is likewise a proficient metric access technique. The operational mechanism involves the utilization of a series of hyper-rings, centered by pivots, to further truncate the original hyperspheres of the M-tree. This results in a more condensed region volume for each node of the tree.

Fuhry et al. [234] identified noteworthy correlations in the metric space between the skyline query and other commonly used similarity queries, including the nearest neighbor and range queries. The techniques devised by the authors leverage these relationships in order to effectively eliminate non-skyline points from the search space. Based on the aforementioned discovery, the authors put forth optimized algorithms that aim to decrease the frequency of dominance tests and the computation of costly metric distance functions during the process.

Jiang et al. [235] proposed the concept of a top-k combinatorial metric skyline query. This

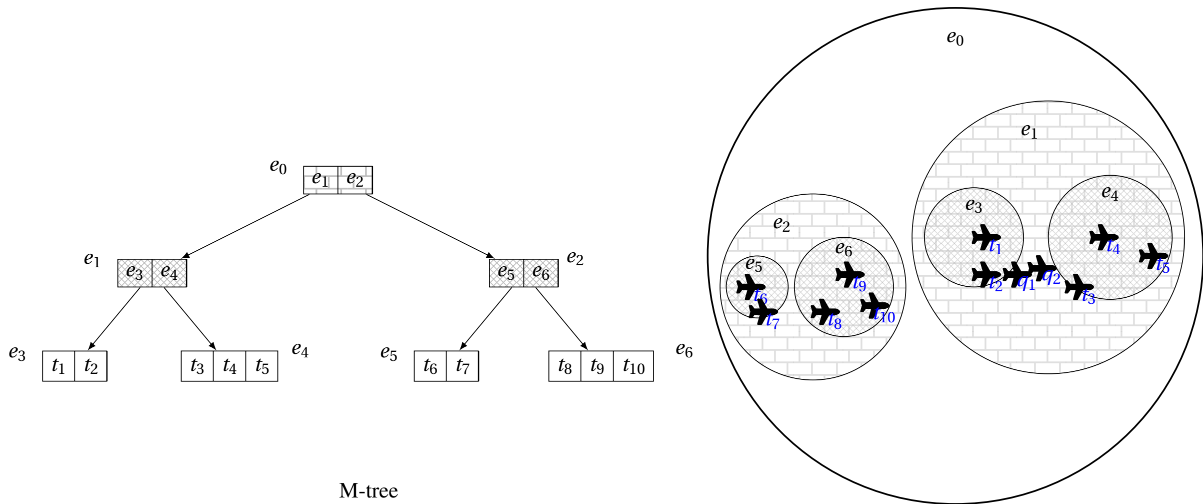


Figure 4.6: *Metric Skyline Query.*

query is designed to identify the optimal k combinations of data points in a metric space based on a strictly monotonic preference function. The objective is to ensure that each of these combinations includes a specified point p in its metric skyline. This inquiry is specifically suited for facility location scenarios, as exemplified by an enterprise endeavoring to identify suitable locations for two novel franchise stores in close proximity to an existing warehouse, denoted as "p". From a pool of eligible locations, one can choose among several available options. In order to mitigate redundant competition among internal divisions and expand their consumer base, it is recommended that both of the proposed sites be situated at a considerable distance from any of the company's current establishments. In order to adhere to a location strategy of this nature, it is imperative that the metric skyline of point p be present in the combinations of the two locations. It is noteworthy to mention that the algorithms discussed in the preceding section have solely concentrated on individual data objects and have not taken into account combinations of data objects. Consequently, they are not applicable to the specific problem at hand.

In summary, it is evident that the quantity of query points utilized in a metric skyline query scenario, as well as in the spatial skyline query scenario previously addressed, constitutes a significant parameter that necessitates meticulous calibration in each application. In the scenario where a single query point is employed, the metric skyline query is transformed into the conventional 1-nearest neighbor query. In contrast, when the quantity of query points increases, it is probable that the magnitude of the skyline will significantly expand, resulting in its limited or negligible utility to the query initiator. Hence, in order to furnish a discerning outcome that holds significance for the query issuer, it is recommended that the metric skyline query commence with a limited number of query points, such as a maximum of five query points, contingent upon the nature of the problem at hand.

4.0.5 Constrained Skyline Query

A *constrained skyline query* extends the traditional skyline by imposing additional constraints on the query results. These constraints could be based on user preferences, predefined conditions, or business rules.

Consider a traveler who wants to book a flight while minimizing both ticket price and flight duration as illustrated in figure 4.7. However, the traveler imposes the following constraints:

- The ticket price must not exceed 380 dollars.

- The flight duration must be at most 3.7 hours.

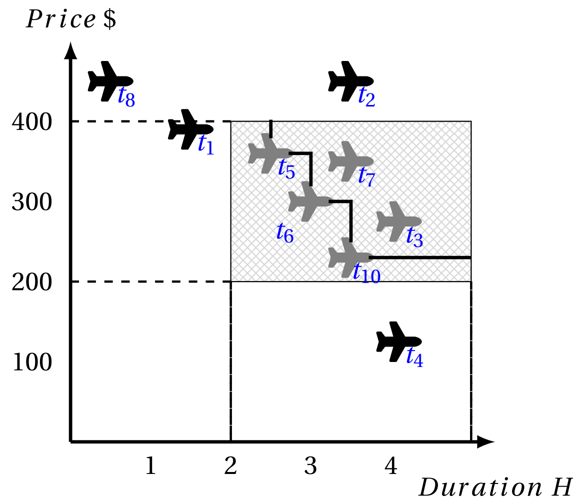


Figure 4.7: *Constrained Skyline Query.*

Applying the constrained skyline query, flight t2 and t8 are eliminated due to their high price, leaving flights t5, t6, and t10 as the optimal choices as illustrated in figure 4.7.

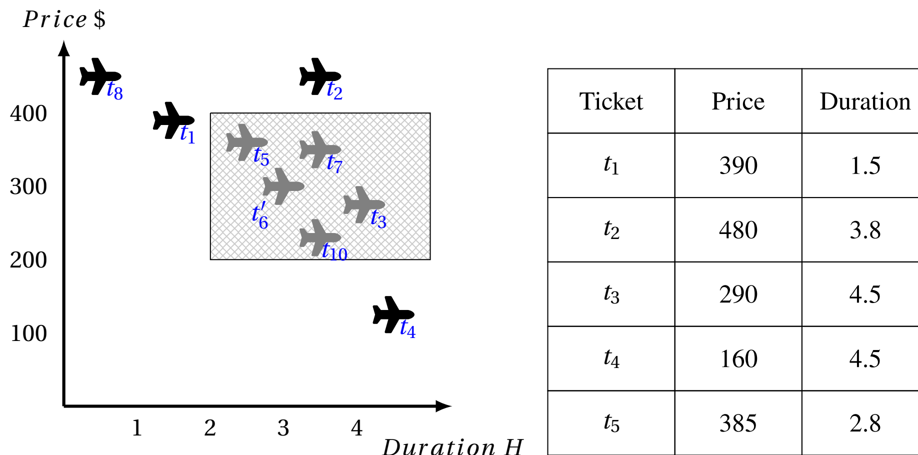
4.0.6 Range-Based Skyline Queries

Range-based skyline queries allow users to specify a range for attributes to restrict the skyline computation within a certain interval.

Suppose a traveler is interested in flights where:

- The ticket price is between $200 and $380.
- The flight duration is between 2 and 4 hours.

Given the same flight dataset as before, only flights that fall within these ranges will be considered for skyline computation as illustrated in figure 4.8.



Ticket	Price	Duration
t_1	390	1.5
t_2	480	3.8
t_3	290	4.5
t_4	160	4.5
t_5	385	2.8

Figure 4.8: *Type of Range-skyline Query developed in [236].*

The range skyline search is a query type that is oriented towards preferences and combines the characteristics of the conventional range query and the point-based skyline query. As a result, it is expected that none of the current point-based skyline search algorithms are appropriate for this particular query. In 2003, Papadias and et al. [89] proposed the constrained skyline query. This query modifies the conventional skyline query by restricting the output to only those data points that fall within a predetermined range of coordinate values and are considered to be the most significant.

The study conducted by Chen et al. [237] aimed to expand upon previous research by addressing scenarios in which data of interest is spread across various locations in unorganized distributed environments.

Huang et al. [102] introduced the concept of continuous monitoring of the conventional point-based skyline. This approach involves the constant movement of both the query point and the data points along a line at a uniform speed in all dimensions.

Their suggested solution avoids having to calculate the skyline from scratch each time. Instead, potential changes are made each time or after new incoming tuples, ensuring that the skyline query result is updated and continuously accessible. The method can be especially helpful in real-time applications like video games and digital war systems, such as when a player in a field-fighting game wants to keep an eye on the enemies who are moving close to them and pose the greatest threat in terms of a number of factors (health, firepower, tactical acumen, and so on).

Lai et al. [238] additionally focus on the distributed domain over mobile wireless sensor networks, where only the data points within a specified maximum distance (radius) of the query point q can be candidates for the conventional point-based skyline query. The query node q extends the range-based skyline query to its neighbor nodes in the first step of the suggested method. After that, the neighbors of q derive their local range-skyline results using their own local data and return them to the query node q. Q examines the dominance relations between all of the candidate data points after receiving the local range-skyline results from its neighbors. Q then derives the complete set of range-skylines. Later, Lai et al. [239] expand their strategy to account for node mobility because the sensor nodes' movement may result in the answer changing frequently.

A *range-based skyline query with respect to a range query* Q considers only those points that satisfy a predefined query range before performing skyline computation as illustrated in figure 4.9.

In their study, Rahul and Janardan [236] presented a technique aimed at retrieving the local skyline of data points situated in a specified query region on the xy-plane. The method focuses on identifying all data points within a small neighborhood of interest.

In a more general setting, the authors partition the s-dimensional space that denotes the data point coordinates and the t-dimensional space that represents the data point features in order to identify the most suitable data points using the conventional skyline search approach. The skyline set is defined as the subset of data points that fall within a specified range of values for the first s attributes and are non-dominated with respect to the last t attributes.

The use of privacy-preserving range-based skyline queries in mobile applications has become necessary due to the growing significance of maintaining confidentiality of user preferences and location.

In their study, Lin et al. [240] proposed a model for estimating the approximate location of the user issuing a query in a 2-dimensional space. They then proceeded to calculate the skyline by establishing a dominance relationship between two data points. This relation was based on the distance of the data points to the imprecise range-based location of the user, as well as the

monotonic order of any other attribute that was available.

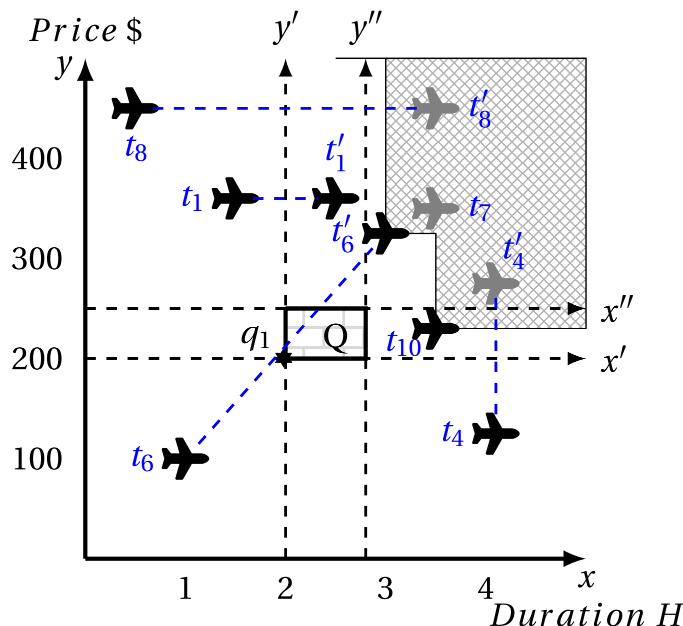


Figure 4.9: *Range Skyline Query.*

The study conducted by Lin et al. [240] was the first study to focus on range-based skyline queries in mobile environments. The authors presented two algorithms, namely I-SKY and N-SKY. The I-SKY algorithm is based on an index. The proposed approach involves the computation of the skyline scope for individual objects based on their dominance relations prior to query processing. Authors utilize the MX-CIf quadtree to index the skyline scope, which is independent of the query issuer. By traversing the index tree, a range-based skyline query can be efficiently processed. N-SKY is an algorithm that is not indexed and is designed to handle highly dynamic data sets. It achieves this by converting a range-based skyline query into multiple segment-based skyline queries. Their methodology is only relevant to Euclidean space due to the difference between the road network and Euclidean space.

The incremental construction of the I-SKY index was developed as a solution to address the issue associated with the continuous RSQ query. The probabilistic RSQ query problem was also investigated by them. The scope of their work is extensive, as it encompasses both stationary and dynamic entities. The comparable approach was similar to the approach to authenticating Skyline queries.

Rather than specifying an exact location or following a specific line segment, the authors assumed that the query point a was moving in a predefined spatial range. The incremental version of the line-based skyline solution has been developed to handle the movement of the querying objects, thereby reducing the size of the result set and the computational cost.

Since the aforementioned method only takes Euclidean space into account, Fu et al. [241] expanded it to take into account objects moving through a road network's space. This range-based skyline query that protects user privacy also accepts the spatial range Q of the user's hidden location as an input. However, in this instance, the shortest path distance to each potential query point q of the user's location and the non-spatial attributes of the objects are used to define the dominance relationship between these data objects. The authors also take into account scenarios where users and data objects move over time on a road network. Fu et al. [241] look

into continuous range-based skyline queries (CRSQs) in road networks, and they propose two effective algorithms: landmark-based (LBA) and index-based (IBA). They define the dominance relation between two data points based on both their distance to the query and the monotonic order in any other attribute, and then compute the skyline by modeling the approximate location of the moving query as a range on the 2-d road network. Their method focuses exclusively on the "static" skyline and the spatial network domain.

By handling the query as a range in every dimension rather than as a point, Wang et al. [242] were the first to propose an algorithm for processing dynamic skyline queries. A data point can only be in the range skyline given the query range if it is not dynamically dominated by any other data point with respect to every query point in the given range. The method prunes the vast majority of data points that cannot be included in the dynamic skyline using a grid index and a variation of the well-known Z-order curve. The final range skyline set is able to be obtained utilizing a non-index skyline processing technique, like the *SFS* algorithm proposed by Chomicki et al. [243].

Finally, Tzouramanis et al. [244] extend the previous work in a number of ways while focusing on a more general case. First, their method takes into account that a data point can be in the range skyline if it is not dynamically dominated by any other data point in relation to any specific query point in the d-dimensional range, and thus is not required to be in relation to all of the query points in the range. The second point is that their suggested solution specifies the sub-region (within the given range) to which a data point may belong in the range skyline set. Last but not least, their method does not require the implementation of additional procedures or the use of non-index skyline processing algorithms to remove false hits as the method of Wang et al. [242] is required to do. Instead, it uses a traditional spatial index to prune all the data points that do not belong to the dynamic skyline in relation to the given range.

The aforementioned modifications greatly expand the scope of potential domains in which the range skyline query model can be implemented.

In their study of range-based skyline queries in road networks, Miao et al. [245] focused on "why not" questions. They proposed three solutions: (1) adjusting the query range; (2) altering the why-not point's attributes; and (3) making changes to both. Keep in mind that when we talk about their "range," we're talking about distance.

4.0.7 Reverse skyline queries

Reverse skyline queries identify points that consider a given query point as part of their skyline. This is useful for scenarios where one wants to determine how competitive a particular option is.

Consider a new flight option F_{new} with a ticket price of $470 and a duration of 6.5 hours. The reverse skyline query finds all flights that would consider F_{new} as part of their skyline.

Lian and Chen [246] investigate efficient and accurate probabilistic reverse skyline query processing on monochromatic and bichromatic data. There is a probability distribution function for each object.

The probabilistic reverse furthest skyline (PRFS) was proposed in [246], which is an extension of the work from [247] and takes into account the scenario in which preference minimization is sought after rather than maximization. Another proposal is a twist on the probabilistic reverse skyline (PRS) query that provides the k most probable objects. The authors also looked into a ranking method for PRS query results called top-k reverse skyline, which retrieves the k points with the most dynamic skylines at the highest probabilities.

Using DC-trees as an index and the pruning technique for reducing the search space, Zhu et

al. [179] proposed a reverse skyline query algorithm. Sensor node power is extremely valuable in wireless sensor networks (WSNs). In such conditions, the reverse skyline algorithm must not only perform well but also minimize power consumption. Due to the constant flow of new information, it can be challenging to keep an index on a dynamic dataset up-to-date. In this manner, we implement DCRS, a divide-and-conquer algorithm for reverse skyline retrieval on data streams. The DCRS employs efficient pruning methods to reduce the search space by using the DC-Tree as the index.

In [248], the authors investigate the issue of WSN reverse skyline computation. The reduction of wasteful transmissions is the foundation of their proposed energy-efficient strategy. The authors also made some evaluations regarding range and reverse skyline queries.

For the multiple-objective decision analysis of these sensing data, skyline query algorithms are widely adopted as one of the main monitoring methods in safe production monitoring and disaster early-warning applications [248]. Power-grid transmission lines, for instance, are prone to icing in ice-disaster warning systems under conditions like low wind speed, low temperature, and high humidity. To detect iced or soon-to-be iced transmission lines, the centralized control sends a skyline query of the speed, temperature, and humidity dimensions to sensor networks installed in the power grid's transmission lines.

Reverse k-skyband [143] query and best skyline object management and identification. The former method is based on a weighted object search pattern, and it has been tested with five different algorithms. The threshold value required by most methods is frequently difficult to configure. Higher settings cause data loss, while lower settings decrease the quality of the final result. In order to return more engaging objects, the proposed method attempts to modify or relax the conditions imposed by the initial reverse skyline query.

Lim developed a parallel Skyline query algorithm that uses angle partitioning of data sets [197] to achieve better load balancing. In order to mesh the data, the algorithm first projects it onto a hypersphere and then uses the angle coordinates to do so.

Banaei Kashani investigated the SSP algorithm [249], a distributed version of the Skyline query algorithm used in the BATON overlay network. The algorithm arranges the baton network's nodes in a balanced binary tree. There is a range of data objects that each node is responsible for processing. The SSP algorithm maps multi-dimensional data to a dimensional order address using the Z-order address.

Summary of Skyline Query Variations: Table 4.3 summarizes the differences between the skyline query variations discussed above.

4.1 Applications of Skyline-Based Queries

Several useful programs employ the dominance idea. Many different kinds of software rely on dominance-based query processing capabilities as elementary components for more advanced tasks. Multi-criteria decision-making, machine learning, and network analysis are three examples of broad fields of application.

4.1.1 Multi-criteria decision making

The skyline operator identifies a group of undominated points. In this sense, the skyline points make excellent candidates to support our decision if we have to make one based on a partic-

Skyline Variation	Definition
Constrained Skyline Query	Applies user-defined constraints before skyline computation.
Range-Based Skyline Query	Considers only points within a specified range.
Range-Based Skyline with Query Q	Processes only data points satisfying a range query before skyline computation.
Reverse Skyline Query	Identifies points that consider the query point as part of their skyline.
Continuous Skyline Query	Maintains and updates the skyline dynamically as new data points arrive in a streaming environment.
Group-Based Skyline Query	Identifies the best group of objects instead of individual objects by considering group dominance.
Spatial Skyline Nearest Query	Computes the skyline of objects based on their nearest distance to a given query location.
Spatial Skyline Farthest Query	Determines the skyline of objects based on their farthest distance from a query point.
Metric Skyline Query	Generalizes the skyline query for complex metric spaces where traditional Euclidean distance may not apply.

Table 4.3: *Comparison of different skyline query variations.*

ular dataset. The field of multi-criteria decision making and quality of service (QoS) evaluation encompasses a large portion of the applications that use the skyline or other variations of dominance-based queries in the recent literature.

Service composition

the authors of Wang et al. [250] suggest a method for service composition by combining reinforcement learning (RL) and skyline computation. Building an effective process based on user-defined preference criteria is the main goal of this study. Effective service selection is the end result of this work.

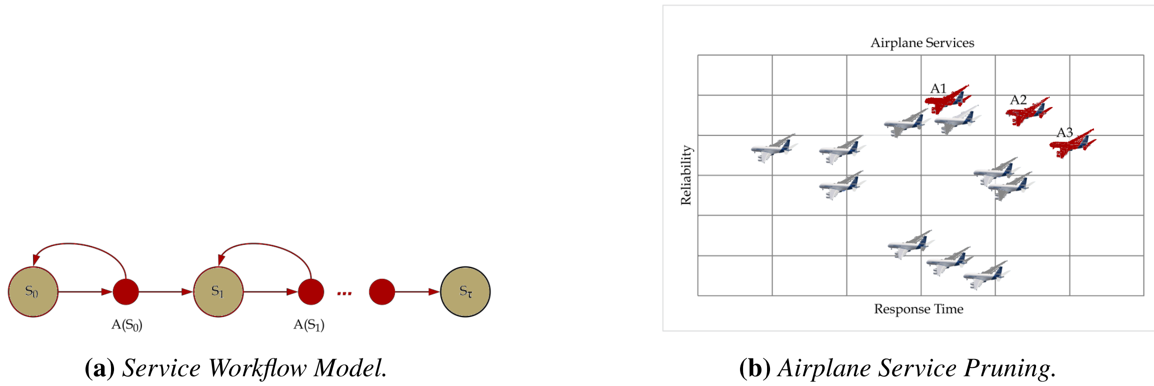
Assume a service provider offers a travel itinerary based on the user's preferences. The flight schedule will specify lodging and transportation details. Each of these parameters may include a variety of options. For instance, the transportation parameter may include the modes of transportation of airplane, train, and ship, with a variety of services available for each. The characteristics of the same option's services are the same. Response time, throughput, and reliability are just a few examples of the services offered by an airplane. The number of candidates will be lowered when the skyline algorithm is used, and only services that are not dominated will remain. Figure 4.10a provides a service pruning example.

Figure 4.10b shows the process as a Web Service Composition Markov Decision Process ($WSC-MDP$) model. The process has two parts: one uses RL to choose the path with the greatest rewards, and the other uses skyline computation to minimize the search space of the various candidates.

4.1.2 Machine learning

Intrusion detection

The work of Abdelkader et al. [251] is an extremely fascinating application that combines the skyline operator with machine learning and computer security. In network security systems, intrusion detection systems (IDS) are critical. The main purpose of these systems is to detect or predict unauthorized activity. The system's alert misfires, such as false negatives (real attacks



(a) *Service Workflow Model.* **(b)** *Airplane Service Pruning.*

Figure 4.10: *Illustration of Service Workflow and Airplane Service Pruning.*

that went unnoticed) or false positives (activities that were mistakenly flagged as attacks), should be reduced in order to lessen the threat. Additionally, because a person may review the alerts, a high rate of false-positive alerts will overwhelm the reviewer and raise the likelihood that a real attack event will go unnoticed.

Given this, anomaly detection and misuse detection are two categories of intrusion detection. While misuse detection identifies the system's normal behavior and distinguishes between normal and abnormal behavior, anomaly detection looks for attacks with known signatures. The suggested model, which is shown in Figure 4.11, is built on two levels. In terms of accuracy, detection rate, and false alert rate, the first level has the best classifiers. The decision is based on three main criteria and the skyline computation:

Increase the accuracy, increase the detection rate and decrease the false alerts rate.

The second level employs a naive Bayesian classifier to make the final determination after factoring in the first level's findings.

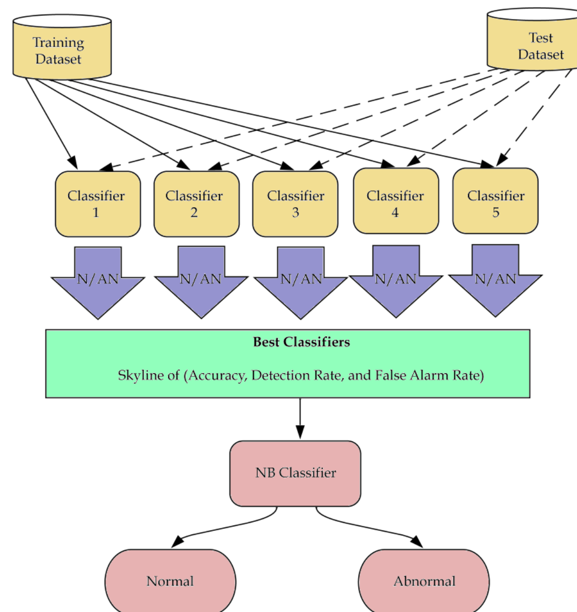


Figure 4.11: *Naive Bayesian Classifier Model.*

4.1.3 Network analysis

Everywhere you look, there are networks! They are used in many application domains because of how well they can model interactions between entities. In this section, we examine a few significant uses of network analysis that call for the use of dominance.

Graph clustering

Dhifli et al. [252] propose a graph clustering approach for mining clusters in large attributed graphs based on the dominance relationship. Each skyline solution receives simultaneous optimization for a number of fitness functions, each of which is defined over the graph topology or over a specific set of attributes obtained from various data sources.

The proposed method is experimentally tested using a sizable protein-protein interaction network of the human interactome that has been enriched with sizable collections of heterogeneous attributes related to cancer. The authors of Dhifli et al. [252] noted that the following summarizes their methodology:

- In order to create potential candidate graph clusters, disjoint powerset systems are first built from potential node combinations.
- Then, a personalized initialization algorithm that prefers the density and connectivity of the first candidate clustering solutions builds a set of chromosomes.
- Then, iteratively create new optimized chromosomes using personalized genetic operators in the initial population. The evaluation of these new chromosomes then takes place using a variety of objective functions that are defined over the graph topology and/or node or edge attributes. Encoding and decoding functions are also defined to map the contents of the chromosomes to the appropriate sets (i.e., candidate clusters) in the corresponding powerset system. A set of approximate non-dominated graph clustering solutions based on the skyline operator is obtained after repeating this procedure for a predetermined number of iterations.

4.1.4 Other interesting applications

Applications from other fields, such as image retrieval, scientometrics, chemical process monitoring, wireless routing, sensor selection, e-commerce, and indoor route search, are briefly discussed in this section. We also go over how database management systems can accommodate queries that are based on dominance.

Image retrieval

An image retrieval application from Georgiadis et al. [253] is the first illustration. To take advantage of the natural qualities of images, the skyline is used in this case. The most crucial image characteristics are encoded into a straightforward vector representation by feature vectors or descriptor vectors, which are typically created from images. Depending on the transformation or feature extraction technique used on the raw data, these vectors are typically high-dimensional, ranging from a few tens to hundreds or even thousands. Only a portion of the data from the original image such as color information, shape, texture, or a combination of these factors remains in the final vector.

Without establishing any similarity or distance function, the intrinsic skyline of images can be captured by locating the skyline using their descriptor vectors. This study's practical application is that the similarity of the entire image collection can be grouped or clustered using the seeds found in the skyline. In order to support scalability, the proposed method combines a number of skyline methods with four cutting-edge hashing algorithms for effective data partitioning and indexing in secondary memory. Figure 4.12 shows the suggested model.

In the clustering evaluation, the outcomes of two fundamental clustering algorithms are compared in order to evaluate the initial cluster centers' distance and similarity when the algorithm is started randomly versus when the skyline items start it. According to the study's results, cluster centers created using skyline items as seeds are comparable to and close to centers created using random initialization. In the current evaluation, it has also been noted that starting the algorithm with skyline items causes the clustering to converge more quickly.

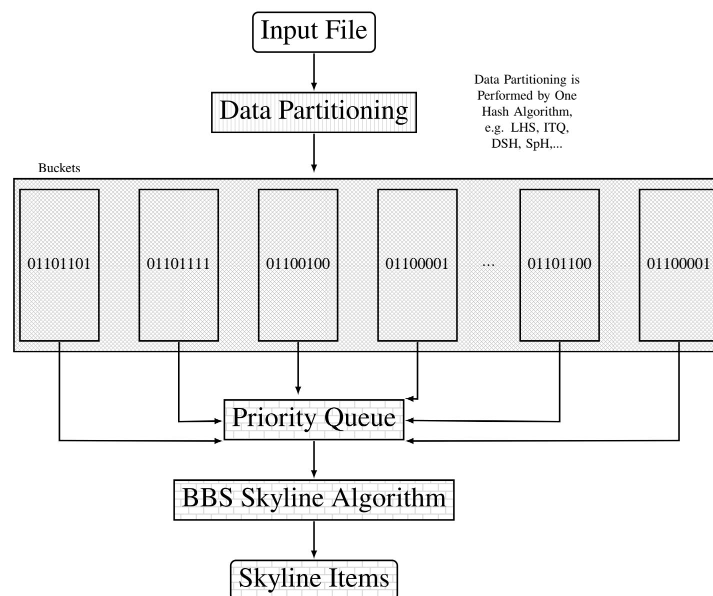


Figure 4.12: *Hash BBS Skyline Process.*

Monitor of chemical process

The next application is from the chemical process sector. The Wang et al. [254] paper proposes a method to keep track of the operational state of chemical equipment. The system has a wireless sensor network that gathers data from the ongoing chemical reaction. A distributed environment is used to collect and store the data. The authors suggest a monitoring strategy that can quickly identify abnormal equipment in order to identify a process fault. The dynamic skyline query, which is used to locate equipment that is approaching its threshold for potential explosion, is at the center of the process. The early detection of a condition like this allows for timely prevention. Figure 4.13 shows how the monitor works.

Wireless routing

According to Yakine et al. [255], a wireless ad hoc network uses the skyline operator to efficiently route communications. Without the use of infrastructure or centralized administration, a wireless ad hoc network can communicate. An ad-hoc network's main function is to facilitate

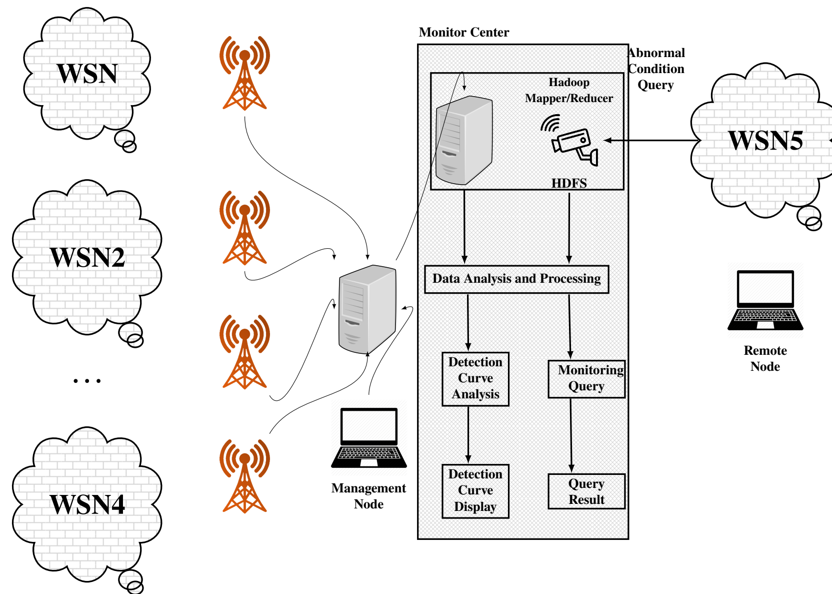


Figure 4.13: *Monitor Process.*

communication among its nodes, which include both routing and end-point nodes. If communication between nodes that are not directly connected is about to occur, an intermediary node must be used. Every node must choose a single-hop path because of the network's dynamic nature. In order to find routes in a wireless network that respect QoS parameters like hop count, delay, bandwidth, and cost, the proposed method uses the skyline computation.

Sensor selection

Other IoT systems can make use of the data produced by the IoT sensors. The difficult part is deciding which sensors, out of a very large number, are best suited to the user's needs. The authors of Kertiou et al. [14] address this issue. The suggested solution is demonstrated in the example below. There are likely three gateways G1, G2, and G3 each of which controls a number of sensors. Attributes such as ID, location, type, accuracy, dependability, and cost are present in the sensors. These attributes each have a unique set of values for each sensor. The following processes take place when a user requests a specific kind of sensor:

1. The server gets the request.
2. Each gateway calculates the local dynamic skyline in accordance with the preferences given by the user.
3. The process involves combining all of the dynamic skylines received by the server into a single global dynamic skyline.
4. The user applies the weights for each proximity-based requirement before accessing the sensor values in ranked order.

The above example is demonstrated in Figure 4.14.

Indoor route search

The application that follows aims to solve the issue of route planning in an indoor environment. Salgado's [256] authors use the skyline query to identify undominated routes. The

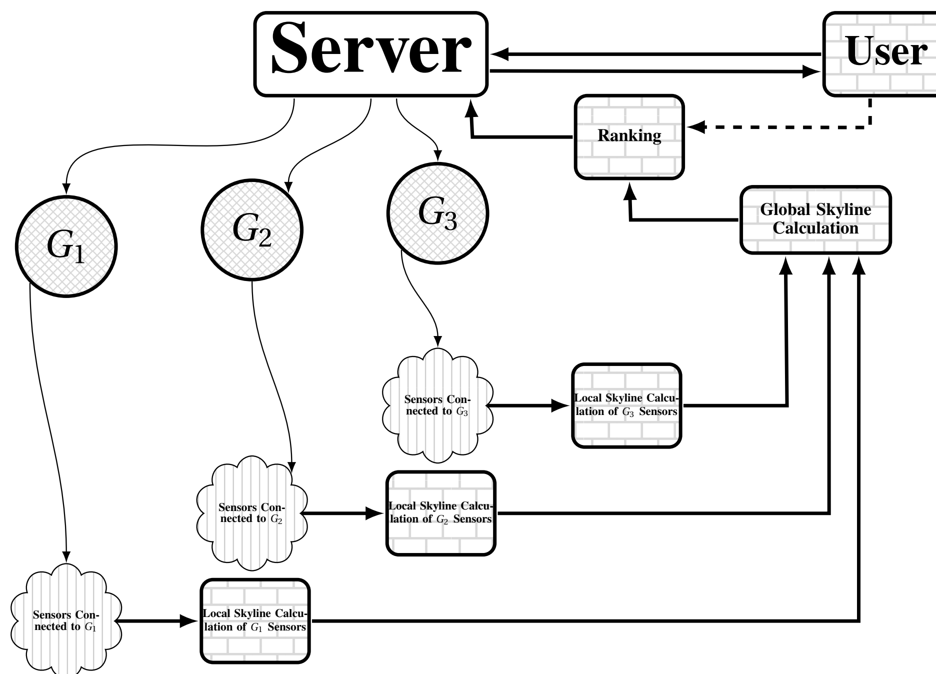


Figure 4.14: *Dynamic Skyline Sensor Selection.*

study suggests a new route search issue called Skyline Routes (KSR) Query, which stands for keyword-aware skyline routes. KSR's route planning queries differ from traditional route planning queries in that they take into account the number of stores the proposed route should contain in addition to distance and time considerations. For instance, a customer wants to purchase three items while visiting a sizable shopping mall. The user might get a plan telling her to visit three different stores if she issues a traditional route hop query that returns the fastest or shortest route. However, it may take longer to wait at three different counters than it would to purchase each item separately. In order to find the best route, one parameter to take into account is the number of stores a route visits. To respond to KSR queries, the authors of Salgado [256] suggest an exact algorithm. Despite the fact that the problem is demonstrated to be NPhard, the suggested solution is effective if there are few keywords, which is the majority of the time.

4.2 Conclusion

The concept of a skyline query has garnered significant interest due to its fundamental role in expressing information requests and facilitating efficient retrieval of vast, multidimensional datasets. Nonetheless, while the traditional skyline definition has proven successful in certain applications, it falls short in addressing a variety of other applications that possess unique characteristics and complexities. As a result, there is a need for adaptations and expansions to the conventional skyline definition. In response to this identified requirement, numerous research groups have introduced novel adaptations to the traditional skyline problem within the past twenty years, along with proficient algorithms for handling these adaptations in relational databases.

The present chapter has centered on various types of skyline queries, namely dynamic skyline queries, spatial skyline queries, metric skyline queries, and range-based skyline queries, which are considered prominent extensions of the conventional skyline query in the field of databases. These inquiries have garnered significant interest due to their improved capacity to

address contemporary needs and their applicability to a diverse array of fields, including but not limited to multi-preference analysis and decision-making support, business planning, stock market trading, advertisement, healthcare, molecular biology, geographic information systems, location-aware computing, trip advising and event planning, facility location and place recommendation, traffic networks, physical environment monitoring, crisis management, and e-games. These intriguing novel inquiries demonstrate the need for more sophisticated methodologies, and the notable advancements they contain portend a promising future. Numerous research avenues are yet to be explored, which will concentrate on enhancing the performance of current query processing algorithms through advancements in their pruning power or computational speed. Alternatively, researchers may investigate new queries in emerging domains that necessitate preference-based computation.

Part IV

Parallel Range Search Skyline

CHAPTER

5 PARALLEL RANGE SEARCH SKYLINE

> *"When the terrain disagrees with the map trust the terrain."*
>
> SWISS ARMY PROVERB

Chapter content

5.1 Problem Definition **93**

5.1.1 Dimension Indexing 93

5.2 Parallel RSS over data stream **105**

5.2.1 Parallel Range Search Skyline *PRSS* 105

5.2.2 Optimization of the dominate() Function using AVX2 110

5.2.3 Parallel implementation details 114

5.3 Performance Evaluation **115**

5.3.1 Experimental Setup 115

5.3.2 Experimental Results 117

5.4 Conclusion **127**

5.5 Conclusion **131**

5.6 List of Publications **132**

5.1 Problem Definition

Within this section, we will initially introduce the symbols and explanations of the terminology employed in this document.

5.1.1 Dimension Indexing

The *RSS* (Range Search Skyline) method relies on the indexing of dataset dimensions based on the total order of each dimension. This enables obtaining the skyline without the need for dominance comparisons involving the entire dataset or the entire current skyline [79]. While determining skyline tuples, notice that the name Range Search represents the bounded search range [79].

To find the skyline of the dataset in table 3.2 using a basic skyline technique, we would compare smartphones based on each feature. A smartphone is considered in the skyline if it is not worse than any other smartphone in all features and strictly better in at least one feature. Let's consider the comparison:

we'll use the dominance relationship. A Phone `A` dominates Phone `B` if `A` is not worse than `B` in any dimension and is strictly better in at least one dimension. First, let's go through each phone and check if it is dominated by any other phone. If a phone is not dominated, it belongs to the skyline. we Start with an empty set for the skyline. For each phone, check if it is dominated by any other phone in the current skyline. If not, add it to the skyline.

`Phone1`: Add to the skyline (no phones dominate it). `Phone2`: Add to the skyline (no phones dominate it). `Phone3`: Add to the skyline (no phones dominate it). `Phone4`: Dominated by `Phone3` (`Phone3` has equal or better values in all dimensions).
`Phone5`: Dominated by `Phone7` (`Phone7` has equal or better values in all dimensions).
`Phone6`: Add to the skyline (no phones dominate it). `Phone7`: Add to the skyline (no phones dominate it). `Phone8`: Dominated by `Phone6` (`Phone6` has equal or better values in all dimensions).
`Phone9`: Dominated by `Phone7` (`Phone7` has equal or better values in all dimensions).
`Phone10`: Dominated by `Phone5` (`Phone5` has equal or better values in all dimensions).

The Final Skyline: `Phone1`: (8, 12, 64, 5, 3000, 500) `Phone2`: (10, 16, 128, 8, 4000, 600) `Phone3`: (12, 20, 256, 10, 4500, 700) `Phone6`: (7, 10, 32, 4, 2500, 450) `Phone7`: (14, 22, 512, 12, 5000, 800). These phones form the skyline based on the given dominance relationships. They are not dominated by any other phones in the dataset.

Definition 4. *(Dimension Index[79]) Let $\mathcal{T}$ be an $\mathcal{N}$-dimensional($\mathcal{N}$ attributes) continuous dataset. The dimension index A of $\mathcal{T}$ is constructed by the set of $\mathcal{N}$ sorted collections of index entries. such that each index entry represents a single tuple, that includes a header pointer that points to the header of the tuple, the value at this dimension and a dimension pointer that points to the dimension value in the next dimension of the same tuple. All index entries of the same dimension $A_i \in A$, $1 \le i \le \mathcal{N}$, are sorted by value with the preference order $\prec$.*

Figure 5.1 demonstrates the data structure of the dimension index, all index entries $n_k^1, n_k^2, \ldots, n_k^d$ of the same tuple x_k are linked on all dimensions $1, 2, \ldots k$. the node hdr_k represents the header of the tuple x_k. for example, n_*^2 represents the value of a tuple $*$ on the dimension 2. This data structure of linked index entries permits quick tuple browsing from any dimension.

Theorem 1. *(conditions under which a tuple is classified as a skyline tuple [79]) Given A the attribute(dimension) index of an $\mathcal{N}-$dimensional continuous dataset $\mathcal{T}$, let $A_i \in A$ be a random sub-index of A, and x be a tuple. Therefore, x is a skyline tuple iff $\nexists\ x' \in \mathcal{T}$ where $(x'[i] = x[i]) \wedge (x' \prec x)$ and one of the conditions below is respected: (1) $\nexists\ p \in \mathcal{T}$ where $p[i] \prec x[i]$, or (2) $\forall m \in \mathcal{M}$ where $m[i] \prec x[i] \Rightarrow m \nprec x$.*

Fig. 5.2 illustrates the dimension indexing of RSS algorithm for multidimensional dataset in table 3.2 that includes 6 sub-indexes $A_1, A_2, \ldots, A_6$ and 10 tuples in all. The dashed lines represents dimension links between index entries of the same tuple.

Based on Theorem 1, we can determine from A_6, A_5, and A_3 independently that $Phone_6$ is a skyline tuple. From dimension A_4, we obtain both $Phone_0[4] = Phone_5[4]$ and there is no tuple in this dimension that is better than $Phone_0$ and $Phone_5$, for which both $Phone_0 \prec Phone_5$ and $Phone_5 \prec Phone_0$ must be checked to find if $Phone_0$ and $Phone_5$ are skyline tuples (actually we have $Phone_0 \prec Phone_5$ and $Phone_5 \nprec Phone_0$).

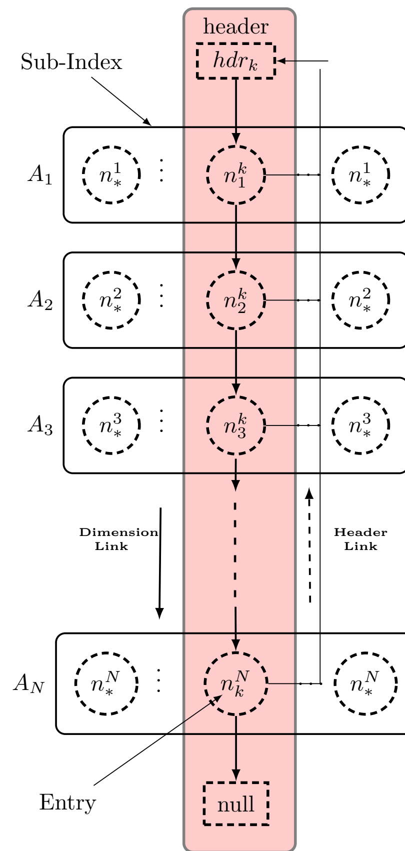


Figure 5.1: *Dimension index system.*

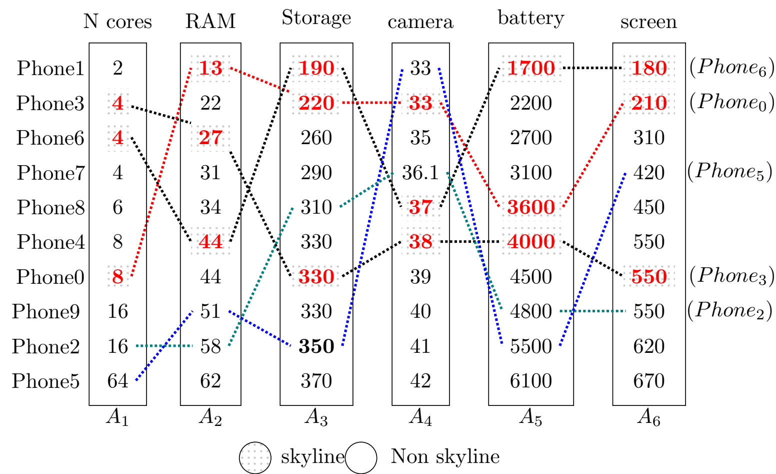


Figure 5.2: *Dimension index for multidimensional dataset in table 3.2.*

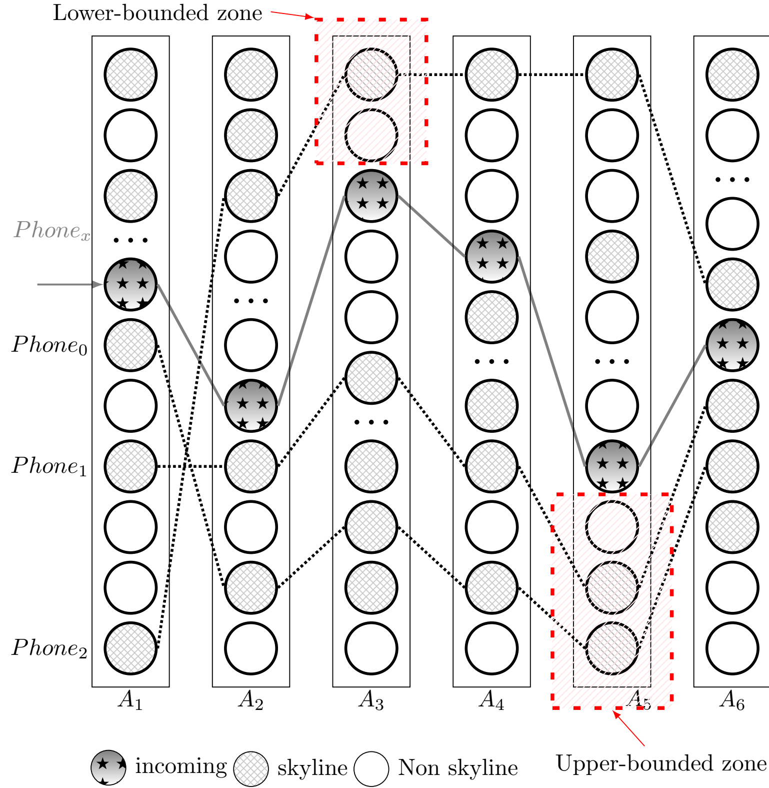


Figure 5.3: *Skyline Maintenance with a new Incoming Tuple.*

If we use Theorem 1 only on dimension A_2, $Phone_0$ is directly a skyline tuple, additionally we have $(Phone_0[2] \prec Phone_5[2]) \wedge (Phone_0 \prec Phone_5)$, so $Phone_5$ is not a skyline tuple; if we continue down, the next entry is not skyline tuple none of them until we reach $Phone_3$, where $(Phone_0[2] < Phone_3[2]) \wedge (Phone_0 \nprec Phone_3)$, therefore $Phone_3$ is a skyline tuple. Finally, with dimension A_6 only, 3 consecutive entries include the dimension value 550 so these 3 tuples have to be first locally compared to extract local skyline tuples, in this example, $Phone_3$, then we apply Theorem 1. this skyline tuples locally identified in dimensions A_4 and A_6 are called the local skyline. clearly, each distinct dimension value is a local skyline.

To maintain the correctness of the skyline when new incoming tuples are received based on Theorem 1 Liu et al.[79] introduced a technique to update the dimension index, as illustrated in Fig. 5.3, where they assumed that all values at each dimension are distinct. To keep the example simple, the symbol … represents the rest of the index entries (values). Let's assume an incoming tuple $\texttt{Phone}_x$ enters our dataset, We start by locating the index entry positions of $\texttt{Phone}_x$ on all dimensions with reference to the preference order $\prec$ (as the blue star pattern entries). Next we identify a lower-bounded dimension A_{lower} that includes the lowest number of skyline tuples $Phone_l$(lower bounded skyline) provided that $Phone_l[A_{lower}] \prec Phone_x[A_{lower}]$. Then we use theorem 1 to compare the incoming tuple $\texttt{Phone}_x$ with these lower-bounded skyline tuples and check if $\texttt{Phone}_x$ is a skyline tuple. As illustrated in Fig. 5.3 comparing the incoming tuple $\texttt{Phone}_x$ to the lower-bounded skyline in dimension A_3, where $\texttt{Phone}_x$ and $Phone_2$ are incomparable, making $\texttt{Phone}_x$ a skyline tuple. Theorem 1 can be used to filter any existing skyline tuples that could be dominated by the incoming tuple $\texttt{Phone}_x$.

If $\texttt{Phone}_x$ is a skyline tuple, we identify the upper-bounded dimension of $\texttt{Phone}_x$ that contains the lowest number of skyline tuples $Phone_u$ where $Phone_x[A_{upper}] \prec Phone_u[A_{upper}]$.

If $Phone_x \prec Phone_u$ for every tuple $Phone_u$ in the upper-bounded skyline, later $Phone_u$ will be cleared out of the skyline, else There is no need for upper-bounded skyline identification. In Fig. 5.3, skyline tuples $Phone_1$ and $Phone_2$ are dominated by $\mathtt{Phone}_x$, so they will be cleared out of the skyline.

To identify the upper-bounded A_{upper} and lower-bounded A_{lower} dimensions for an incoming tuple $\mathtt{Phone}_x$, Liu et al.[79] used the following estimation equation:

$$A_{lower} = arg_{A_i} min(|\frac{(v_x^i - min(A_i))}{(max(A_i) - min(A_i))}|)$$

$$A_{upper} = arg_{A_i} max(|\frac{(v_x^i - min(A_i))}{(max(A_i) - min(A_i))}|)$$

v_x^i : dimensional value of the incoming tuple x on dimension i. Here When the number of dimensions (attributes) increases, parallelizing the building of the dimension index and the identification of upper-bounded A_{upper} and lower-bounded A_{lower} dimensions in addition to the parallel insertion and removal of tuples will boost the whole skyline computation's performance.

The estimation formulas for the lower and upper bounds in [79] are used to determine the most promising dimensions for processing.

1. Lower Bound Estimation Formula:

$$A_{\text{lower}} = \arg\min_{A_i} \left(\left| \frac{v_t^i - \min(A_i)}{\max(A_i) - \min(A_i)} \right| \right)$$

2. Upper Bound Estimation Formula:

$$A_{\text{upper}} = \arg\max_{A_i} \left(\left| \frac{v_t^i - \min(A_i)}{\max(A_i) - \min(A_i)} \right| \right)$$

The goal of these formulas is to select the dimensions that are most promising for maintaining efficient skyline computation. This involves normalizing the value of the incoming tuple with respect to the range of values in each dimension.

The term $\frac{v_t^i - \min(A_i)}{\max(A_i) - \min(A_i)}$ normalizes the value v_t^i of the incoming tuple in dimension i to a range between 0 and 1. This is important because it allows for comparison across different dimensions, which might have vastly different scales.

By considering the absolute value $|\cdot|$, the formulas focus on how far the normalized value v_t^i is from the extremes (minimum and maximum) of the values seen so far in that dimension.

The lower bound formula selects the dimension A_i where the normalized value v_t^i is closest to the minimum value. This is because in skyline computations, dimensions where the incoming tuple is close to the minimum are less likely to be dominated by other tuples. Hence, $\arg\min$ is used to find the dimension where this deviation is minimal.

Conversely, the upper bound formula selects the dimension A_i where the normalized value v_t^i is closest to the maximum value. This dimension is most likely to have a strong influence on whether the tuple is dominated, as it is far from being minimal in that dimension. Hence, $\arg\max$ is used to find the dimension where this deviation is maximal.

- For each dimension i, normalize the value of the incoming tuple v_t^i:

$$\text{Normalized value} = \frac{v_t^i - \min(A_i)}{\max(A_i) - \min(A_i)}$$

- Absolute Deviation: Calculate the absolute deviation of the normalized value from the extremes (0 and 1).
- Minimize for Lower Bound: Select the dimension where this deviation is minimal:

$$A_{\text{lower}} = \arg\min_{A_i} \left(\left| \frac{v_t^i - \min(A_i)}{\max(A_i) - \min(A_i)} \right| \right)$$

- Maximize for Upper Bound: Select the dimension where this deviation is maximal:

$$A_{\text{upper}} = \arg\max_{A_i} \left(\left| \frac{v_t^i - \min(A_i)}{\max(A_i) - \min(A_i)} \right| \right)$$

- Lower Bound: Dimensions where the value is close to the minimum are less likely to be dominated, making it a lower bound for potential skyline membership.
- Upper Bound: Dimensions where the value is close to the maximum have higher chances of being dominated, setting an upper bound for skyline consideration.

The authors [79] derived these formulas based on the need to efficiently prune the search space by focusing on the most promising dimensions. The normalization process ensures that comparisons are scale-independent, and the use of absolute deviation helps in identifying dimensions where the tuple stands out, either as a potential skyline point (lower bound) or as a likely dominated point (upper bound).

The estimation formulas for lower and upper bounds are intended to prioritize the dimensions that are most likely to help quickly determine whether a new incoming tuple is a skyline point. By focusing on these dimensions, the algorithm can efficiently prune non-promising tuples, reducing computational overhead.

Step-by-Step Derivation:

- Normalization is a crucial step to ensure that comparisons across different dimensions (which may have different scales) are fair and meaningful.

  - For a given dimension i:
  - v_t^i is the value of the incoming tuple t in dimension i.
  - $\min(A_i)$ and $\max(A_i)$ are the minimum and maximum values seen so far in dimension i.

  The normalized value of v_t^i in dimension i is calculated as:

$$\text{Normalized value} = \frac{v_t^i - \min(A_i)}{\max(A_i) - \min(A_i)}$$

  This normalization scales the value v_t^i to a range between 0 and 1, where: - 0 corresponds to $\min(A_i)$ - 1 corresponds to $\max(A_i)$

- The absolute deviation of the normalized value from 0 and 1 indicates how close v_t^i is to the minimum or maximum value in that dimension. This helps in understanding the potential of the tuple being a skyline point.

$$\left| \frac{v_t^i - \min(A_i)}{\max(A_i) - \min(A_i)} \right|$$

- If the deviation is small (close to 0), v_t^i is near the minimum value in that dimension.
- If the deviation is large (close to 1), v_t^i is near the maximum value in that dimension.

- Lower Bound Calculation: For the lower bound estimation, the goal is to identify the dimension where v_t^i is closest to the minimum value. This dimension is less likely to have tuples that dominate v_t^i, making it a strong candidate for being part of the skyline.

  - We use the $\arg\min$ function to find the dimension A_{lower} that minimizes the absolute deviation:

  $$A_{\text{lower}} = \arg\min_{A_i} \left(\left| \frac{v_t^i - \min(A_i)}{\max(A_i) - \min(A_i)} \right| \right)$$

  This selects the dimension where the normalized value is closest to 0 (the minimum value).

- Upper Bound Calculation: For the upper bound estimation, the goal is to identify the dimension where v_t^i is closest to the maximum value. This dimension is more likely to have tuples that dominate v_t^i, making it a less likely candidate for being part of the skyline.

  - We use the $\arg\max$ function to find the dimension A_{upper} that maximizes the absolute deviation:

  $$A_{\text{upper}} = \arg\max_{A_i} \left(\left| \frac{v_t^i - \min(A_i)}{\max(A_i) - \min(A_i)} \right| \right)$$

  This selects the dimension where the normalized value is closest to 1 (the maximum value).

  - Lower Bound (Minimization): By choosing the dimension where v_t^i is closest to the minimum value, we are identifying the dimension in which the incoming tuple is least likely to be dominated. This helps in quickly including potential skyline points.
  - Upper Bound (Maximization): By choosing the dimension where v_t^i is closest to the maximum value, we are identifying the dimension in which the incoming tuple is most likely to be dominated. This helps in quickly excluding non-promising tuples.

  Methods Used:
  1. Normalization: To handle different scales in different dimensions. 2. Absolute Deviation: To measure how far the tuple's value is from the extremes (minimum or maximum) in a normalized space. 3. Optimization Functions ($\arg\min$ and $\arg\max$): To identify the most promising dimensions for inclusion (lower bound) or exclusion (upper bound).

Theorem 1: Let A be the dimension index of a d-dimensional database D, $A_i \in A$ be an arbitrary sub-index of A, and t be a tuple. Then, t is a skyline tuple if and only if $\nexists r \in D$ such that $(r[i] = t[i])$ and $(r \prec t)$ and one of the following conditions is satisfied: **1.** $\nexists u \in D$ such that $u[i] \prec t[i]$, or **2.** $\forall s \in S$ such that $s[i] \prec t[i] \Rightarrow s \nprec t$.

Proof: To show that a tuple t is a skyline tuple if and only if no other tuple r exists such that r equals t in one dimension and dominates t, and one of two conditions holds.

- Skyline Tuple: A tuple t is part of the skyline if no other tuple in the database D dominates t. - Dominance: Tuple r dominates tuple t ($r \prec t$) if r is at least as good as t in all dimensions and better in at least one dimension.

1. Conditions Without Repeated Dimension Values:
- Consider an arbitrary dimension i, where $1 \leq i \leq d$.
- Assume no tuples r exist such that $r[i] = t[i]$.

Condition (1): - t appears at the top of A_i. - If $\nexists u \in D$ such that $u[i] \prec t[i]$, then no tuple is better than t in the i-th dimension. - Thus, t is a skyline tuple because no tuple dominates t in any dimension.

Condition (2): - Let s be a skyline tuple such that $t[i] \prec s[i]$.
- If $s \nprec t$ for all skyline tuples s such that $s[i] \prec t[i]$, then t is incomparable to any skyline tuple s.
- Therefore, t must be a skyline tuple because no skyline tuple s can dominate t.

2. Conditions With Repeated Dimension Values: - Now, consider the case where $\exists r \in D$ such that $r[i] = t[i]$.

Condition (1) Fails: - Condition (1) does not account for $(r[i] = t[i])$ and $r \prec t$. - Simply stating $\nexists u \in D$ such that $u[i] \prec t[i]$ does not exclude $r \prec t$ when $r[i] = t[i]$.

Condition (2) Fails: - Condition (2) does not cover the case where $r \in S$ (skyline set). - Stating $\forall s \in S$ such that $s[i] \prec t[i] \Rightarrow s \nprec t$ does not prevent r from dominating t if $r[i] = t[i]$.

3. Establishing Correctness: - For conditions (1) and (2) to be valid, we must ensure that for any $r \in D$ such that $r[i] = t[i]$, $r \nprec t$.
- This ensures that no tuple r can dominate t when they are equal in dimension i, making t a skyline tuple.

- The proof combines the conditions without repeated dimension values and extends them to handle cases where dimension values are repeated.
- The key insight is ensuring no tuple r with $r[i] = t[i]$ can dominate t.
- This comprehensive approach confirms that the specified conditions correctly identify skyline tuples.

This step-by-step breakdown aligns with the logical flow in the proof, confirming that t is a skyline tuple if and only if it satisfies the given conditions, preventing any other tuple from dominating it while sharing the same dimension value.

Theorem 2: The update of non-skyline tuples dominated by expired tuples listed in Algorithm 1 (lines 7-14) is correct and complete.

Proof: If a tuple x is dominated by a tuple s and s is incomparable with the incoming tuple t, then s and x must expire before t. Therefore, x will not be considered while removing t.

1. Skyline Queries:
- A skyline query retrieves a set of points from a dataset such that each point is not dominated by any other point.
- A point p dominates another point q if p is as good or better in all dimensions and strictly better in at least one dimension.

2. Sliding Window Model:
- In data streams, the sliding window model is used to handle only the most recent data points.
- Tuples (data points) within this window are considered for skyline computation.

3. Algorithm 1 Overview:
- The algorithm processes incoming tuples to maintain the skyline over a sliding window.
- When a new tuple t arrives, the algorithm updates the skyline by adding t and removing any tuples that are now dominated by t.

Explanation of Theorem 2:

1. Expired Tuples: - Tuples that are outside the sliding window are considered expired.
- Expired tuples are removed from the skyline.

2. Dominated Tuples: - If a tuple x is dominated by an expired tuple s, x should be reevaluated because s is no longer part of the dataset.

3. Correctness and Completeness: - The theorem asserts that the process of updating the skyline by removing dominated tuples is both correct (no false positives or negatives) and com-

plete (all necessary tuples are considered).

Proof Steps:

1. Dominance and Expiry:

- Assume a tuple x is dominated by a tuple s.
- If s is incomparable with the incoming tuple t (meaning s and t do not dominate each other), the relationship between x and s remains unaffected by t.

2. Expiration Before t:

- Both s and x must expire before t because they are older than the current window's scope.
- Once s expires, x is reevaluated to check if it can now be a part of the skyline or if it is dominated by another tuple.

3. Tuple Removal:

- When t arrives, tuples dominated by t are removed.
- Since s and x are already expired, they are no longer considered for removal or inclusion in the skyline.

- Correctness: The procedure correctly identifies tuples that are dominated by expired tuples and ensures they are not considered part of the skyline. - Completeness: The procedure considers all relevant tuples, ensuring that no skyline tuple is missed.

Efficiency Analysis:

- Time Complexity: The estimation of bounded dimensions requires $O(d)$ time, where d is the number of dimensions. This is efficient compared to the exact computation of skyline sizes.
- Skyline Maintenance: Given a sliding window W and a skyline size M, the dominance test requires $O(d)$ time, making the overall process efficient for dynamic data streams.

The theorem ensures that the algorithm correctly and completely updates the skyline by handling expired tuples and dominated tuples appropriately. The use of estimation formulas helps in maintaining efficiency while ensuring that the most relevant dimensions are considered for skyline computation.

To thoroughly understand Theorem 3 and its proof, let's break down the theorem and each part of the proof step by step.

Theorem 3: In the worst case, RSS inserts an incoming skyline tuple t in time

$$O\left(d\log|W| + 2(d+M) + \frac{M}{d}(d + \frac{(|W|-M)(d+M)}{d})\right).$$

Proof: To describe and justify the time complexity of inserting a new skyline tuple t in the RSS algorithm.

Components of the Time Complexity Expression:

1. Insertion into Sub-Indexes:

$$O(d\log|W|)$$

- The RSS algorithm maintains d B-tree based sub-indexes, one for each dimension.
- Inserting t into each of these sub-indexes requires $O(\log|W|)$ time, where $|W|$ is the size of the sliding window.
- Since there are d dimensions, the total time for this step is $O(d\log|W|)$.

2. Finding Lower-Bounded and Upper-Bounded Dimensions:

$$O(2d)$$

- If t is a skyline tuple, we need to determine its lower-bounded and upper-bounded dimensions.
- This involves calculating the estimation formulas (1) and (2), which require $O(d)$ time each.
- Thus, finding both bounds takes $O(2d)$ time.

3. Dominance Tests with Bounded Skylines:

$$O(2d+2M)$$

- According to Theorem 1, for each bounded skyline, $\frac{M}{d}$ dominance tests are required.
- Since there are two bounded skylines (lower-bounded and upper-bounded), the total time is $O(2d+2M)$.

4. Handling Dominated Tuples: - In the worst case, t may dominate all $\frac{M}{d}$ tuples in the upper-bounded skyline.
- For each of these tuples, we need to find their lower-bounded dimension, which takes $O(d)$ time.
- Each upper-bounded skyline tuple may dominate $\frac{(|W|-M)}{d}$ non-skyline tuples.
- For each of these non-skyline tuples, we need to perform $O(d)$ time operations to find the lower-bounded dimension and potentially update the skyline.

Detailed Steps of the Proof:

1. Insertion into Sub-Indexes:
- The incoming tuple t is inserted into d B-tree based sub-indexes.
- Each insertion requires $O(\log|W|)$ time.
- Since there are d dimensions, the total time for this step is:

$$O(d\log|W|)$$

2. Finding Lower-Bounded and Upper-Bounded Dimensions:
- To determine if t is a skyline tuple, the algorithm needs to find its lower-bounded and upper-bounded dimensions using the estimation formulas.
- Each calculation takes $O(d)$ time, so both calculations take:

$$O(2d)$$

3. Dominance Tests with Bounded Skylines:
- If t is a skyline tuple, it needs to be tested against the lower-bounded and upper-bounded skylines.
- Each skyline has $\frac{M}{d}$ tuples.
- Performing $\frac{M}{d}$ dominance tests for each bounded skyline (lower and upper) takes:

$$O(2\cdot\frac{M}{d}) = O(2M)$$

- Adding the time for finding the bounded dimensions, the total time is:

$$O(2d+2M)$$

4. Handling Dominated Tuples:
- In the worst case, t may dominate all $\frac{M}{d}$ tuples in the upper-bounded skyline.
- For each of these $\frac{M}{d}$ tuples, finding the upper-bounded dimension takes $O(d)$ time:

$$O(d\cdot\frac{M}{d}) = O(M)$$

- Each of these upper-bounded skyline tuples can dominate $\frac{(|W|-M)}{d}$ non-skyline tuples.
- For each of these non-skyline tuples, the algorithm performs $O(d)$ time operations to determine if they become skyline tuples:

$$O\left(\frac{|W|-M}{d}\cdot d\right) = O(|W|-M)$$

- Each dominance test involves an additional $O(d+M)$ time in the worst case, so the total time is:

$$O\left(\frac{(|W|-M)}{d}\cdot(d+M)\right)$$

Combining the Components:

- The total time complexity is the sum of all these components:

$$O(d\log|W|)+O(2d)+O(2M)+O(M)+O\left(\frac{(|W|-M)}{d}\cdot(d+M)\right)$$

- Simplifying the expression:

$$O\left(d\log|W|+2d+2M+M+\frac{(|W|-M)(d+M)}{d}\right)$$

- Since $O(2d)$ and $O(M)$ are lower-order terms compared to others, they can be combined for a more concise representation:

$$O\left(d\log|W|+2(d+M)+\frac{M}{d}(d+\frac{(|W|-M)(d+M)}{d})\right)$$

The proof of Theorem 3 justifies that the time complexity for inserting a skyline tuple t in the worst case involves:
- Inserting t into the sub-indexes.
- Finding the lower-bounded and upper-bounded dimensions.
- Performing dominance tests with the bounded skylines.
- Handling any non-skyline tuples that may be affected by t.

This step-by-step analysis ensures that the time complexity expression accurately reflects the algorithm's performance in the worst-case scenario.

Theorem 4:

In the worst case, RSS deletes an expired skyline tuple t in time:

$$O\left(d+\frac{(|W|-M)(d+M)}{d}+d\log|W|\right)$$

Proof: To describe and justify the time complexity for deleting an expired skyline tuple t in the RSS algorithm.

Components of the Time Complexity Expression

1. Finding the Upper-Bounded Dimension:

$$O(d)$$

- This step involves determining the upper-bounded dimension of the tuple t.
- Using the estimation formula, this takes $O(d)$ time.

2. Testing Non-Skyline Tuples:

$$O\left(\frac{(|W|-M)}{d}(d+M)\right)$$

- According to the proof of Theorem 3, $\frac{(|W|-M)}{d}$ non-skyline tuples in the upper-bounded zone need to be tested in the worst case.
- Each test requires $O(d+M)$ time.

3. Removing Index Entries:

$$O(d \log |W|)$$

- Finally, the algorithm needs to remove d index entries from the B-trees.
- Removing each entry takes $O(\log |W|)$ time, so removing d entries takes $O(d \log |W|)$ time.

Detailed Steps of the Proof:

1. Finding the Upper-Bounded Dimension:
- When deleting a skyline tuple t, the algorithm needs to find the upper-bounded dimension.
- This involves calculating the estimation formula:

$$A_{\text{upper}} = \arg\max_{A_i} \left| \frac{v_t^i - \min(A_i)}{\max(A_i) - \min(A_i)} \right|$$

- This calculation takes $O(d)$ time.

2. Testing Non-Skyline Tuples:
- In the worst case, the expired skyline tuple t may have dominated $\frac{(|W|-M)}{d}$ non-skyline tuples.
- Each of these non-skyline tuples needs to be tested to determine if they can now become part of the skyline.
- Each test involves:
- $O(d)$ time to find the lower-bounded and upper-bounded dimensions.
- $O(M)$ time to test if the tuple becomes part of the skyline.
- Thus, each test takes $O(d+M)$ time, and for $\frac{(|W|-M)}{d}$ non-skyline tuples, the total time is:

$$O\left(\frac{(|W|-M)}{d}(d+M)\right)$$

3. Removing Index Entries:
- To delete t, the algorithm needs to remove its entries from the d B-tree based sub-indexes.
- Each removal operation from a B-tree takes $O(\log |W|)$ time.
- Removing d entries takes:

$$O(d \log |W|)$$

Combining the Components:

- The total time complexity for deleting an expired skyline tuple t is the sum of all these components:

$$O(d) + O\left(\frac{(|W|-M)}{d}(d+M)\right) + O(d \log |W|)$$

- Simplifying the expression, we get:

$$O\left(d + \frac{(|W|-M)(d+M)}{d} + d \log |W|\right)$$

The proof of Theorem 4 justifies that the time complexity for deleting an expired skyline tuple t in the worst case involves:
- Finding the upper-bounded dimension of t.
- Testing non-skyline tuples that may become skyline tuples after t is deleted.
- Removing the index entries of t from the sub-indexes.

This step-by-step analysis ensures that the time complexity expression accurately reflects the algorithm's performance in the worst-case scenario.

5.2 Parallel RSS over data stream

5.2.1 Parallel Range Search Skyline $PRSS$

This section outlines the process of developing a parallel version of the sequential RSS algorithm, which we have named parallel RSS ($PRSS$) [36].. Within $PRSS$, the arrival of new tuples triggers the incremental update of the skyline and the removal of expired tuples. $PRSS$ updates the skyline incrementally by adding each incoming tuple when the window is not yet full. The algorithm efficiently removes expired tuples, changes dominance relationships, and adjusts the skyline for filled windows. $PRSS$ utilizes dominance lists to keep track of tuples that are directly dominated, enabling efficient updates when tuples are added or removed. $PRSS$ examines the

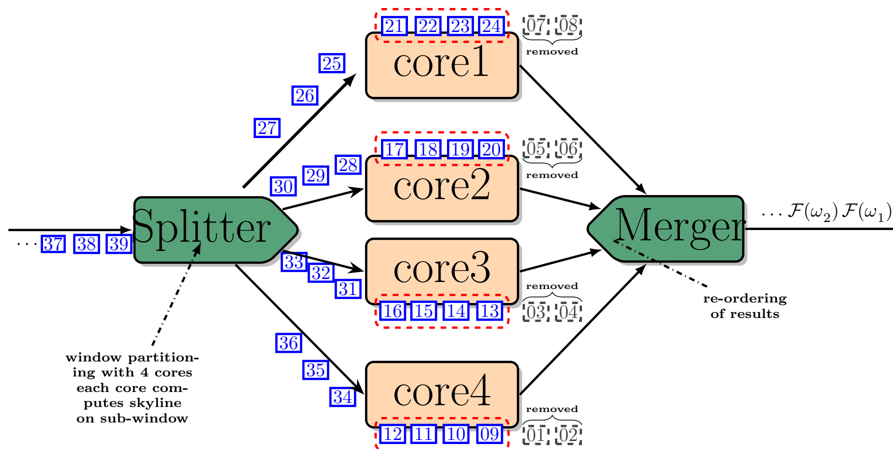


Figure 5.4: *Load Balancing on Cores.*

collection of expired skyline tuples to determine if any dominated tuples could potentially be recent skyline tuples. Subsequently, eliminate the tuples that have reached their expiration date and incorporate the newly inputted tuples into the window. This algorithm 1, referred to as $PRSS$, discusses the concept of $PRSS$. The inputs for this process include a data stream T, a window size Z, and a dimensionality N (representing the number of attributes), the start and end of the chunk given for each processor core. As the window Z moves over the data stream T, the algorithm maintains the updated skyline index S after it receives each tuple x.
$PRSS$ begins by initializing two indices: Dimension Index $A = \emptyset$: An empty index used to manage the tuples based on their dimensions (Used to store the index entries for each dimension). Skyline Index Skyline Index $S = \emptyset$: An empty set that will store and keep track of the current skyline tuples. $PRSS$ continuously processes tuples from the data stream T. For each new tuple x read from the stream, the following steps are performed: The technique of $PRSS$ begins by removing expired tuples from the dimension index A (lines 5-17). The function $PRSS$ retrieves the set V of all expired tuples based on the type of the window Z (either count-based or time-based) at line 4. The set E consists of all tuples that are directly dominated by each expired skyline tuple $v \in V$ (line 7). This identification is done using the $parallelDominatedTuples$ method.

Subsequently, the variable v is deactivated in A to guarantee that subsequent dominance tests remain unaffected (Mark v as inactive in the dimension index A) . Disabling and Removing Tuples will ensure the integrity of the index and skyline by properly handling the expired tuples and maintaining an accurate skyline set. The $PRSS$ function utilizes the $parallelRangeSearch$ algorithm (Algorithm 2) to identify additional skyline tuples for each tuple $e \in E$ (lines 9-13).

If the element v belongs to the set S, the $PRSS$ algorithm removes v from the skyline index S at line 14. In order to stop the deletion process of the tuple, $PRSS$ permanently deletes v from the set A at line 16. Following that, $PRSS$ initiates the process of inserting incoming tuples into A (lines 18-25). The $PRSS$ function is called on line 18 to determine whether x is a skyline tuple using the $PRangeSearch$ algorithm. If the incoming tuple x is a skyline, the $PRSS$ algorithm calls the $parallelDominatedTuples$ function to retrieve the tuples R that are directly dominated by x. It then proceeds to simultaneously remove each tuple $r \in R$ in parallel, as indicated in lines 19-22. Ultimately, the $PRSS$ appends the value x to the set S at line 23 and concurrently inserts x into the set A at line 25. The attributes of an incoming tuple x will be distributed among the CPU cores (workers), as shown in Figure 5.4. If $PRSS$ is unable to retrieve any expired tuples, it will promptly proceed with the insertion of new tuples.

The $parallel\ RangeSearch$ algorithm determines whether a given tuple x is a skyline tuple by checking against the current dimension index A and updating the skyline index S as necessary. The algorithm begins by initializing the skyline index S .

If x passes the initial dominance check, the algorithm retrieves the skyline tuples S for the lower bounded dimension. It iterates over each skyline tuple s in S . If any of these skyline tuples dominate x , the algorithm returns false. Lower Bounded Range Search: It identifies the lower bounded dimension for the tuple x using the LowerBoundedDimension function. This dimension helps in locating the relevant blocks of tuples that could potentially dominate x. Determine the lower bounded dimension "lower": $lower \longleftarrow LowerBoundedDimension(A, x)$ this function identifies the dimension with the least value where x is bounded.

Summary of PRSS steps:

1. Initialization: Set up empty indices for dimensions and skylines.
2. Process Each Tuple in Data Stream: Identify and handle expired tuples, If an expired tuple is in the skyline, update the skyline by re-evaluating dominated tuples. Insert the incoming tuple, Determine if it should be part of the skyline. Update the dimension index and skyline accordingly.

Key Functions: $ExpiredTuples(A, \mathcal{Z})$: Identifies tuples that have expired based on the window W.

$DominatedTuples(A, v)$: Finds tuples directly dominated by a given tuple.

$PRangeSearch(A, x)$: Checks if a tuple is a skyline tuple and updates necessary indices.

The $parallel\ RangeSearch$ method is demonstrated in Algorithm 2, which concurrently determines the upper and lower bounds of the skyline for an incoming tuple x in the dimension index A. Upon the arrival of a new tuple x in our datasets, the $PRangeSearch$ algorithm initiates by determining the dimension with the lowest lower bound for x (as outlined in Algorithm 3) in order to perform dominance tests (line 2).

According to Theorem 1, PRangeSearch employs the embedded BNL algorithm to compare x with all tuples B that share the same value at the same dimension (line 3). The algorithm retrieves the block of tuples B from the dimension index A corresponding to this lower bounded dimension. ($B \longleftarrow GetBlock(x, A_{lower})$) The block is a subset of tuples in the dimension index A that are relevant for comparison with x in the lower dimension.

If x is determined to be a local skyline, the function $PRangeSearch$ will return true. Otherwise, the function will stop executing (lines 4-6). uses a Block Nested Loop (BNL) method to check if any tuple in block B dominates x . If any tuple in B dominates x , the algorithm returns false, indicating that x is not a skyline tuple. BNL checks if x can be a skyline tuple within the block B. If BNL returns false, x is not a skyline tuple, and the function exits early.

If x is a local skyline, the $PRangeSearch$ algorithm also concurrently identifies the lower-bounded skyline S for the tuple x (line 7), and subsequently compares S with x to determine if x is a skyline. If the condition is not met, the $PRangeSearch$ function terminates immediately (lines 8-12), and returns false.

If x is identified as a skyline tuple, the $PRangeSearch$ identifies the upper bounded dimension for x (line 13) using the $UpperBoundedDimension(A, x)$ function in order to eliminate skyline tuples that are ultimately dominated by x. This helps in locating another set of relevant blocks of tuples.

The skyline tuples b that have the same dimension value as x (lines 14-20) need to be removed from the skyline index. $PRangeSearch$ Fetches the block B associated with the upper bounded dimension: It retrieves the block of tuples B from the dimension index A corresponding to this upper bounded dimension. $B \longleftarrow GetBlock(x, Aupper)$ the block is a subset of tuples in the dimension index A that are relevant for comparison with x in the upper dimension. Check each tuple b in block B: For each tuple b in block B , if b is part of the current skyline and x dominates b , the algorithm updates the dominance list to reflect that x now dominates b . It then removes b from the skyline index S . The algorithm retrieves the skyline tuples S for the upper bounded dimension. It iterates over each skyline tuple s in S . If x dominates any of these skyline tuples, it updates the dominance list $UpdateDominanceList(x, b)$ accordingly and removes the dominated skyline tuples from S. for each tuple b in B, if b is in the skyline index S and x dominates b ($x \prec b$), update the dominance list for x and remove b from S.

The $PRangeSearch$ algorithm identifies the upper-bounded skyline S of x (line 21) and removes any skyline tuple that is dominated by x (lines 22-27). If x is a skyline, the function $PRangeSearch$ will return a value of true. Retrieve the skyline tuples in the upper bounded dimension: $S \longleftarrow UpperBoundedSkyline(Aupper, x)$ this function gets the set of skyline tuples that are relevant to the upper bounded dimension upper. Check dominance against tuples in S: For each skyline tuple s in S, if x dominates s ($x \prec s$), update the dominance list for x and remove s from S.

$PRangeSearch$ algorithm returns true indicating that x is a skyline tuple: If x passes all these checks without being dominated by any other tuple, and after updating the skyline index S as necessary, the algorithm concludes that x is a skyline tuple and returns true.

in summary $PRangeSearch$ algorithm does Lower Bounded Range Search where it identifies the lower bounded dimension. Fetches relevant block and checks dominance using BNL. Compares against current skyline tuples and updates if necessary. Then does Upper Bounded Range Search where it identifies the upper bounded dimension. Fetches relevant block and compares against current skyline tuples. Updates dominance lists and skyline tuples as necessary. Finally Returns True: Concludes that x is a skyline tuple if all checks pass.

Key Functions: $LowerBoundedDimension(A, x)$ / $UpperBoundedDimension(A, x)$: Determines the lower/upper dimension bounds for x. $GetBlock(x, A)$: Retrieves the relevant block of tuples for comparison. $BNL(B, x)$: Checks if x can be a skyline tuple within the block B. $LowerBoundedSkyline(A, x)$ / $UpperBoundedSkyline(A, x)$: Retrieves the skyline tuples in the lower/upper bounded dimension. The $UpdateDominanceList()$ function, as shown in Algorithm 2, is responsible for updating the dominance indexes when the incoming tuple dominates the existing skyline tuples. When the function $UpdateDominanceList(x, b)$ is called, the dominance index of b is appended to the dominance index of x.

By systematically checking both lower and upper bounded dimensions, $PRangeSearch$ algorithm ensures efficient skyline tuple maintenance by narrowing down the range of dominance checks and updating the skyline index continuously.

Algorithm 1: PRSS (Parallel Range Search for Stream)

```
   Input  : Data stream 𝒯, Window 𝒵, Dimensionality 𝒩, start, end
   Output : Instant update of skyline S.
 1 Dimension Index A = ∅
 2 Skyline Index S = ∅   // deletes expired tuples from the dimension index A.
 3 while t = start and t < end do
 4     V ⟵ ExpiredTuples(A, 𝒵)  // Gets the set V of all expired tuples w.r.t
       Count or Time-based window Z.
 5     foreach v ∈ V do
 6         if v ∈ S then
 7             E ⟵ DominatedTuples(A, v)      // For each expired skyline tuple
               v ∈ V, PRSS finds by DominatedTuples the set E of all tuples
               directly dominated by v
 8             Disable v ∈ A            // Disables v ∈ A for not affecting following
               dominance tests.
 9             foreach e ∈ E do
10                 if PRangeSearch(A, e) then
11                     S ⟵ S ∪ {e}   // For each tuple e ∈ E, PRSS calls PRangeSearch
                       to determine new skyline tuples
12                 end
13             end
14             S ⟵ S \ {v}    // To finish tuple deletion, PRSS removes u from the
               skyline index S if v ∈ S
15         end
16         Remove v from A           // and definitively removes v from A.
17     end
18     // PRSS inserts the incoming tuple x into A.
19
20     if PRangeSearch(A, x) then
21         R ⟵ DominatedTuples(A, x) // If x is a skyline tuple, PRSS calls
           DominatedTuples that returns the set R of the tuples directly
           dominated by x
22
23         foreach r ∈ R do
24             Remove r from A              // and removes each tuple r ∈ R.
25         end
26         S ⟵ S ∪ {x}                    // Finally, PRSS adds x to S
27     end
28     foreach x ∈ 𝒩 do
29         Insert x to A             // and commits the insertion of x to A.
30     end
31 end
```

Algorithm 2: Parallel RangeSearch

Input : Dimension index A, tuple x, Dimensionality $\mathcal{N}$.

Output : **true** if x is a skyline tuple.

1. Skyline Index S
2. $lower \longleftarrow LowerBoundedDimension(A, x)$ // *PRangeSearch* first calculates the lower-bounded dimension for x in order to perform dominance tests.
3. $B \longleftarrow GetBlock(x, A_{lower})$ // *PRangeSearch* tests whether x is a local skyline in the set B of all tuples having the same dimension value by the embedded BNL algorithm, which returns true if x is a local skyline.
4. **if not** $BNL(B, x)$ **then**
5.   **return false** // otherwise PRangeSearch stops.
6. **end**
7. $S \longleftarrow LowerBoundedSkyline(A_{lower}, x)$ // If x is a local skyline, *PRangeSearch* further retrieves the lower-bounded skyline S for x.
8. **foreach** $s \in S$ **do**
9.   **if** $s \prec x$ **then**
10.   **return false** // then tests x is a true skyline against S: if not, *PRangeSearch* stops by returning false.
11.   **end**
12. **end**
13. $upper \longleftarrow UpperBoundedDimension(A, x)$ // If x is found being a skyline tuple, *PRangeSearch* calculates the upper-bounded dimension for x in order to eliminate skyline tuples eventually dominated by x.
14. $B \longleftarrow GetBlock(x, A_{upper})$ // At this step, repeated dimension values shall also be taken into account:
15. **foreach** $b \in B$ **do**
16.   **if** $b \in S$ **and** $x \prec b$ **then**
17. $UpdateDominanceList(x, b)$ // if x dominates any skyline tuple b having the same dimension values
18. $S = S \setminus \{b\}$ // then b must be removed from the skyline index.
19.   **end**
20. **end**
21. $S \longleftarrow UpperBoundedSkyline(A_{upper}, x)$ // *PRangeSearch* retrieves the upper-bounded skyline S for x.
22. **foreach** $s \in S$ **do**
23.   **if** $x \prec s$ **then**
24. $UpdateDominanceList(x, s)$ // removes all skyline tuples dominated by x.
25. $S = S \setminus \{s\}$
26.   **end**
27. **end**
28. **return true** // *PRangeSearch* returns true if x is a skyline tuple.

However, After processing new incoming tuples, computing the updated skyline, and removing expired data points, it is essential to maintain an efficient indexing structure. Now Let's see how Our algorithm updates the dimensional index to ensures that the index remains accurate and efficient, facilitating rapid skyline computations in a continuously evolving data environment.

Figure 5.5 shows how dominance indexes are updated while an incoming tuple 38 dominates existing skyline tuples, as an example a count-based window W = 15 is concerned. The update

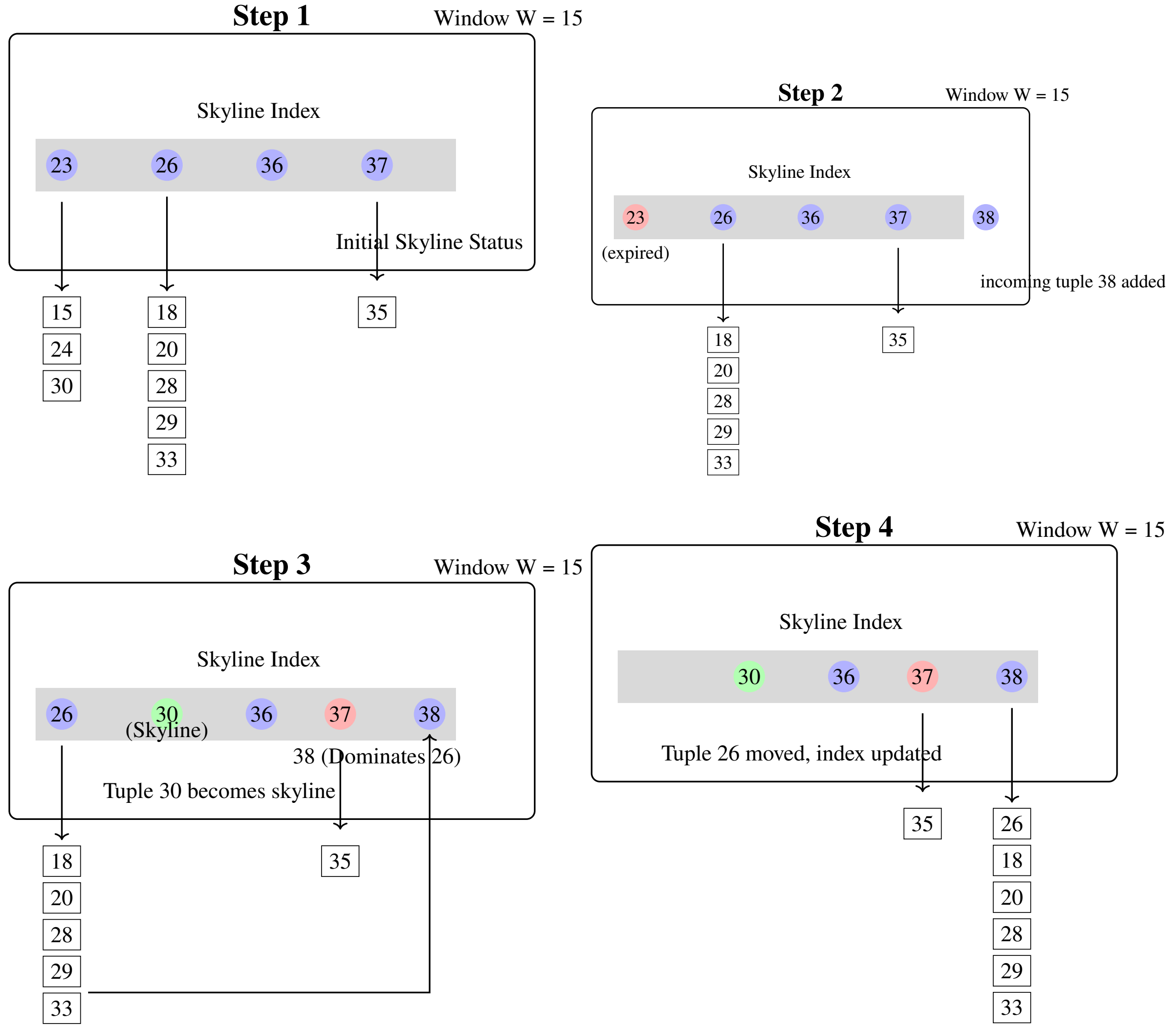


Figure 5.5: *Step-by-step Skyline Index Update.*

of dominance indexes can be illustrated by the following steps (let the gray band depict the skyline index and assume that the incoming tuple 38 is a skyline tuple): (a) the incoming tuple 38 makes the current earliest tuple 23 expired;

(b) while removing the expired tuple 23, assume that tuple 30 becomes skyline tuple, and assume that the incoming tuple 38 dominates the skyline tuple 26;

(c) tuple 26 is appended to the tuple 38 and removed from the skyline index and its dominance entries are moved to tuple 38.

5.2.2 Optimization of the dominate() Function using AVX2

The dominate() function Indicates whether the first tuple dominates the second tuple. It Iterates through each element (dimension) of the tuples. Compares the corresponding elements of the two tuples: If the element in the first tuple (”*p1”) is greater than the element in the second tuple (”*p2”), the first tuple does not dominate, and ”false” is returned. If the element in the first tuple is less than the element in the second tuple and dominance hasn’t been established (”!dominating”), it sets ”dominating” to ”true”. If the iteration completes without returning ”false”, it means the first tuple dominates the second tuple if ”dominating” is ”true”. Returns the result.

Koizumi et al. [145] used Intel SSE/AVX instructions to decrease the number of instructions in

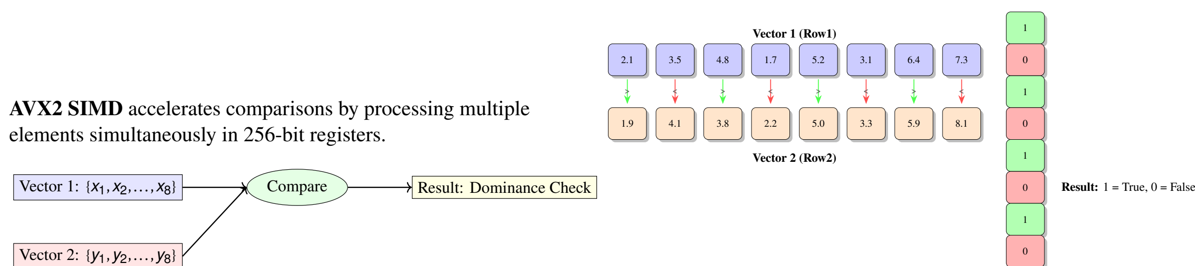


Figure 5.6: *Tuples comparisons using AVX2.*

the dominates() function. In our case we used the AVX2 which is an abbreviation for Advanced Vector Extensions 2, which serves as an extension to the x86 instruction set architecture. AVX2 is specifically engineered to enhance the speed of vectorized calculations, especially when it comes to SIMD (Single Instruction, Multiple Data) operations. AVX2 introduces vector registers that are 256 bits wide, which is double the width of the registers in SSE and AVX. This enables the concurrent processing of a greater number of data elements within a single instruction [257] as illustrated in figure 5.6. AVX2 is a component of a wider range of improvements in SIMD technology that focuses on utilizing parallelism in contemporary processors. SIMD instructions enable the execution of a single instruction on multiple data elements simultaneously, resulting in notable performance enhancements for specific computational tasks [145].

Listing 5.1: *AVX2 source code of dominates() function*

```
1  // assume width is always a multiple of 8
2  inline bool dominate_avx2_x8(double *row1, double *row2, size_t width) {
3    __m256d dominating = _mm256_setzero_pd();
4
5    for (size_t i = 0; i < width; i += 8) {
6      __m256d vec1 = _mm256_loadu_pd(&row1[i]);
7      __m256d vec2 = _mm256_loadu_pd(&row2[i]);
8
9      // Compare vectors
10     __m256d cmp_result = _mm256_cmp_pd(vec1, vec2, _CMP_GT_OQ);
11
12     // Logical OR with dominating
13     dominating = _mm256_or_pd(dominating, cmp_result);
14
15     // Check for dominance
16     if (!_mm256_testz_pd(cmp_result, cmp_result)) {
17       return false;
18     }
19   }
20
21   // Check if any element in the dominating vector is true
22   return !_mm256_testz_pd(dominating, dominating);
23 }
```

The $dominate_avx2()$ function is an AVX2-optimized version of the original "dominate" function As shown in Listing 5.1. It leverages AVX2 intrinsics to perform SIMD operations, processing multiple elements in parallel. Compares corresponding elements of two tuples using AVX2 vectorized instructions. Takes advantage of AVX2's ability to execute multiple comparisons in a single instruction. Generates a mask indicating the dominance relationship for

each element. Combines individual element comparisons using AVX2 intrinsics. This AVX2-optimized function enhances the efficiency of dominance checking in the context of skyline computations, leveraging advanced vectorized instructions for improved parallelism.

Algorithm 3: dominate($Tuple_1, Tuple_2, \mathcal{N}$)

```
   Input  : Tuple1, Tuple2, Dimensionality N.
   Output : True or False.
1  procedure dominate(dataset; a: index; b: index)
2  flag ← 0
3  for d = 0 to N − 1 do
4  |  if dataset[a][d] > dataset[b][d] then
5  |  |  return 0
6  |  end
7  |  else if dataset[a][d] < dataset[b][d] then
8  |  |  flag ← 1
9  |  end
10 end
11 return flag
```

The time complexity of the following AVX2-optimized code is O(m/8), where 'm' is the width of the vectors being compared. The implementation harnesses the power of Intel AVX2 instructions, which enable Single Instruction, Multiple Data (SIMD) operations for 256-bit operands. In the context of AVX2, the $_mm256_setzero_pd()$ function initializes a $256bit$ AVX register ($__m256d$) with all elements set to zero. This register, named $dominating$, is used to accumulate comparison results during the vectorized processing of the input vectors. The code iterates over the vectors in chunks of eight elements at a time, leveraging the 256-bit wide AVX2 registers. For each iteration, two 256-bit vectors (vec1 and vec2) are loaded from memory using the $_mm256_loadu_pd$ intrinsic. This enables simultaneous loading of eight double-precision floating-point values. The $_mm256_cmp_pd$ intrinsic is then employed to compare corresponding elements of $vec1$ and $vec2$. The comparison mode $_CMP_GT_OQ$ checks if the values in $vec1$ are greater than those in $vec2$. The resulting vector (cmp_result) holds the comparison outcomes.

To determine if any dominance exists in the current set of eight elements, a logical OR operation is performed between the 'dominating' register and cmp_result. This update ensures that if any element in the current chunk indicates dominance, it will be reflected in the 'dominating' register. Following the vectorized comparison, the code checks if any element in the 'dominating' register is non-zero using the $_mm256_testz_pd$ intrinsic. If true, it means that no dominance was found in the current chunk, and the loop continues to the next set of eight elements. If false, the function returns false as dominance is detected. Finally, the function returns true if any element in the 'dominating' register is non-zero after processing all chunks, indicating the presence of dominance in the entire vector. This AVX2-optimized implementation significantly improves performance by exploiting parallelism in the comparison operations, leading to reduced instruction count and enhanced efficiency in processing large datasets.

Algorithm 4: Main Algorithm for Computing Skylines in Parallel

Input : Window $\mathcal{Z}$, Dimensionality $\mathcal{N}$, $datatype$, $datasize$, $number_of_cores$
Output : $globalSkyline[\,]$

```
1  data ← readDataStream(datasize, N, datatype)
2  step ← datasize / number_of_cores + 1
3  localSkylines[ ]
4  # pragma omp parallel for schedule(dynamic)
5  for i ← number_of_cores − 1 to 0 do
6  |            // i * step start of the data chunk
7  |           // min((i+1) * step, window) Chunk End
8  |   localSkyline[ ] ←
   |    PRSS(data, dimensionality, window, i * step, min((i+1) * step, window))
9  |      // Combine local skylines outside the parallel region
10 |   critical section: append localSkyline[ ] to localSkylines[ ]
11 end
12 globalSkyline[ ] = combine_skylines(localSkylines[ ])
```

Understanding Data Chunk Assignment to Cores:
The main algorithm 4 distributes data chunks to multiple cores using OpenMP for parallel execution.

- The dataset has `datasize` tuples.
- The number of available cores is `number_of_cores`.
- The **step size** for each core is computed as:

$$step = \frac{datasize}{number_of_cores} + 1 \tag{5.1}$$

  This ensures that all tuples are processed, even if `datasize` is not perfectly divisible by `number_of_cores`.

- Each core `i` is assigned a **contiguous chunk** of data:

$$\text{Start index} = i \times step$$
$$\text{End index} = \min((i+1) \times step, window)$$

The Parallel Loop Execution:
`#pragma omp parallel for schedule(dynamic)`
For(i ← `number_of_cores` - 1 To 0)
This loop executes **in parallel** across the available cores.

- `#pragma omp parallel for schedule(dynamic)`:
  - Each iteration runs independently on a different core.
  - **Dynamic scheduling** ensures efficient workload balancing.
- **Loop direction** (`number_of_cores - 1` **to** `0`):
  - Processing starts from the highest indexed core.

Each core calls:

$$localSkyline[] \leftarrow PRSS(data, dimensionality, window, i*step, min((i+1)*step, window))$$

Synchronization and Combining Local Skylines: After computing `localSkyline[]`:

```
# Critical section to avoid race conditions
append localSkyline[] to localSkylines[]
```

Finally, local skylines are merged:

```
globalSkyline[] = combine_skylines(localSkylines[])
```

Execution Example:

For `datasize = 1000` and `number_of_cores = 4`:

$$step = \frac{1000}{4} + 1 = 251 \tag{5.2}$$

Core Index (i)	Start Index	End Index
Core 3	753	1000
Core 2	502	753
Core 1	251	502
Core 0	0	251

Table 5.1: *Data Chunk Assignment*

Summary of Execution:

1. The dataset of 1000 tuples is divided into 4 chunks.
2. Each core processes its chunk **independently**.
3. **Local skylines** are computed per chunk.
4. Local skylines are merged to form the **global skyline**.

5.2.3 Parallel implementation details

Utilizing multi-core platforms can effectively reduce the execution time for continuous Skyline computation. We focus on studying optimization techniques that are specifically related to the distribution of computation across multi-core processors [144]. We examine the impact of various load-balancing strategies. In our proposed implementation, the workloads are distributed among the cores. Once a core completes its computation, it takes on a portion of the remaining tasks. Consequently, an effective load balancing is ensured. Furthermore, based on the number of cores that are available, we use two distinct solutions for loop iterations scheduling among threads as illustrated in figure 5.7.

1. **Static Scheduling:** iterations of the loop are divided into chunks at compile time. Each thread is assigned a fixed-size chunk of iterations to process before the loop begins. The chunk size remains constant throughout the execution of the loop. Static scheduling is often used when the workload per iteration is known to be uniform, and the overhead of chunk distribution is negligible compared to the overall computation.

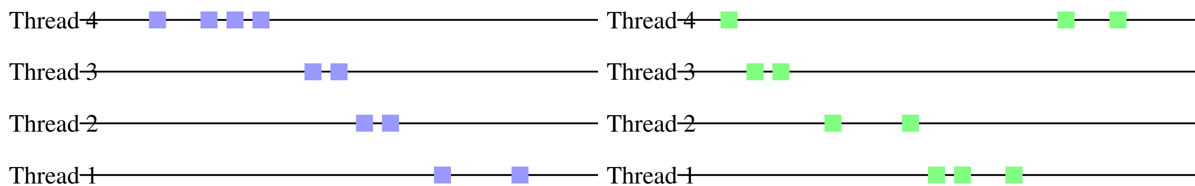


Figure 5.7: *Various load-balancing strategies.*

2. **Dynamic Scheduling:** iterations of the loop are distributed dynamically among the available threads at runtime. Threads request work (iterations) dynamically from a shared pool of iterations until all iterations are completed. The chunk size may vary dynamically based on the workload of each thread and the availability of iterations in the pool. Dynamic scheduling can be beneficial when the workload per iteration is unpredictable or when the workload varies significantly across iterations.

We employed OpenMP to parallelize the calculation of local skylines across a sliding window. The data is partitioned into segments according to the number of cores ($core_number$), and each core processes a subset of the data. Within the parallel region, every core initializes a skyline instance and calculates the local skyline for its designated data subset. The local skylines are stored in a temporary vector called localSkyline. Following the parallel region, a critical section is employed to merge the individual skylines into a collective localSkylines vector. The global skyline is computed by utilizing the combined local skylines outside the parallel region.

5.3 Performance Evaluation

5.3.1 Experimental Setup

We assess the performance of our PRSS system using both count and time-based windows on synthetic and real data streams. The PRSS and RSS implementations are written in C++ and compiled using g++ (v11.3.0) with the -O3 optimization flag. The OpenMP API (v5.0) library is employed for implementing multi-threading algorithms.The simulations were executed on an Intel i7 4.2 GHz processor, which has 6 cores, along with 16 GB of DDR4 RAM. The operating system used was Windows 10. We assess the scalability and runtime efficiency of our algorithm by conducting a comprehensive experimental simulation, comparing it to the sequential RSS algorithm. The PRSS c++ source code has been published on GitHub as well[1]. In order to compare our parallel algorithm with its sequential counterpart RSS, we employed the original implementation developed by the original author[2]. Datasets:We employed the Skyline benchmarking data generator[3]. The parameters utilized in the current skyline techniques are employed to directly compare the outcomes. As an illustration, we vary the dimensionality d, ranging from 8 to 64, and the cardinality, varying from n = 100k to 500k. In addition, we conduct benchmarking using real datasets, specifically the Covertype[4] and weather[5] datasets. **Covertype:** The Covertype dataset contains data on cartographic variables, including elevation,

[1] https://github.com/avionicscode/multicore-prss, accessed on August, 2023
[2] https://github.com/skyline-sdi/sdi-rss, accessed on October , 2022
[3] http://pgfoundry.org/projects/randdataset, accessed on February, 2020
[4] https://doi.org/10.24432/C50K5N, accessed on August, 2023
[5] https://crudata.uea.ac.uk/cru/data//hrg/tmc/, accessed on August, 2023

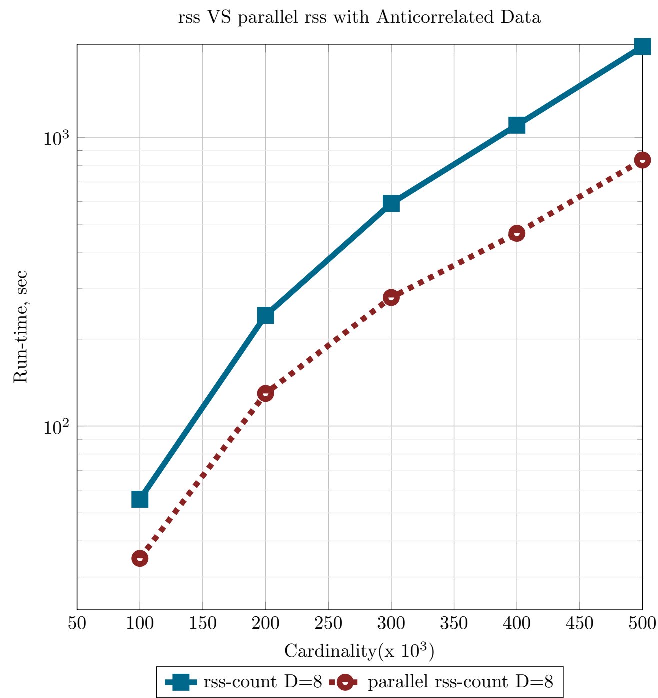


Figure 5.8: *RSS vs PRSS (Anticorrelated Count-Based window, Different Cardinalities).*

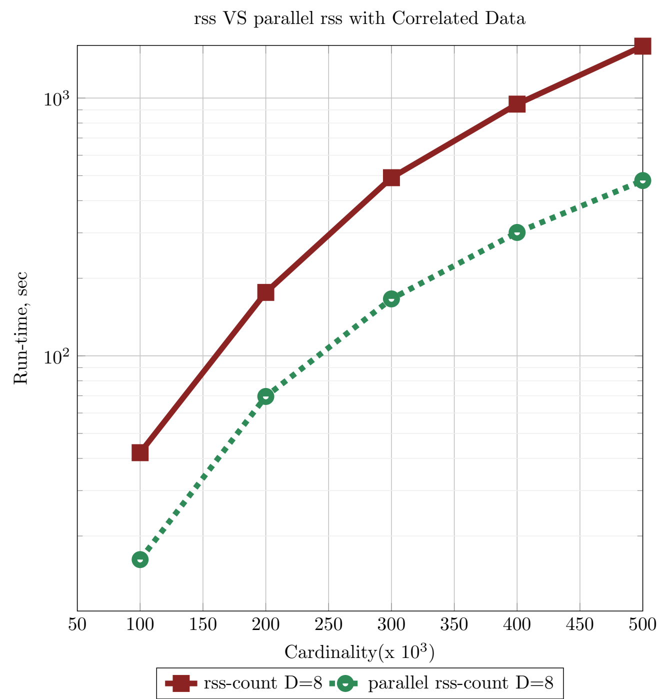


Figure 5.9: *RSS vs PRSS (Correlated Count-Based Window, Different Cardinalities).*

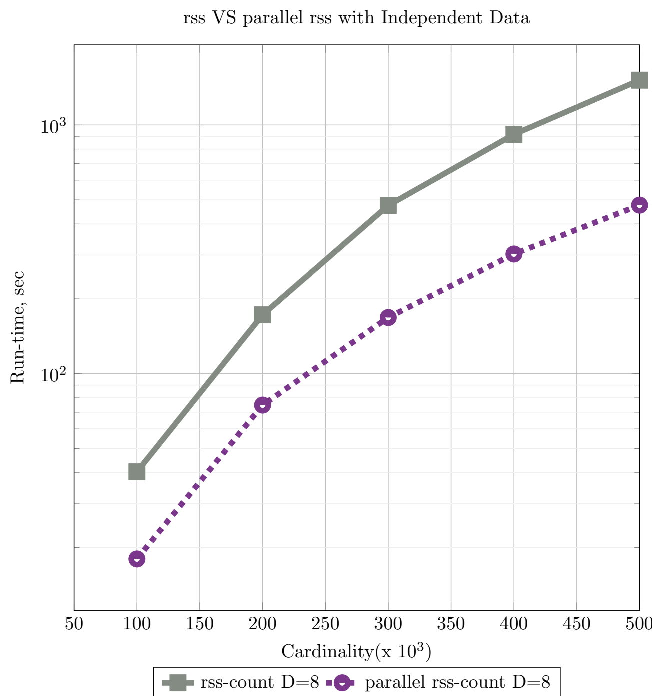


Figure 5.10: *RSS vs PRSS (Independent Count-Based Window, Different Cardinalities).*

proximity to the nearest roadway, and slope. The data is accessible for grid cells with dimensions of 30 m x 30 m in the Roosevelt National Forest located in Colorado, USA. Skyline points represent forested areas that have unique cartographic characteristics.

Weather: The Weather dataset offers monthly data on precipitation, along with the geographical coordinates (latitude and longitude) and elevation for 566,268 terrestrial locations across the globe. Each record corresponds to a cell that measures 10 degrees of latitude and longitude. A skyline record refers to a distinct pattern of months that experience unusually high levels of rainfall based on its three-dimensional location, with a preference for higher and northeastern areas.

5.3.2 Experimental Results

Data Name	Window Size	Dimensions Number	RSS Time (sec)	PRSS Time (sec)	BskyTree Time (sec)
Covertype	100000	10	62.9839	41.3117	3.978
Weather	100000	15	54.6236	21.2429	140.563

Table 5.2: *RSS vs Parallel RSS vs BskyTree with Real-World Datasets.*

Runtime Efficiency Comparison

To evaluate the performance of our proposed PRSS (Parallel Range Search Skyline) algorithm, we conducted extensive experiments comparing it with the sequential RSS algorithm and the

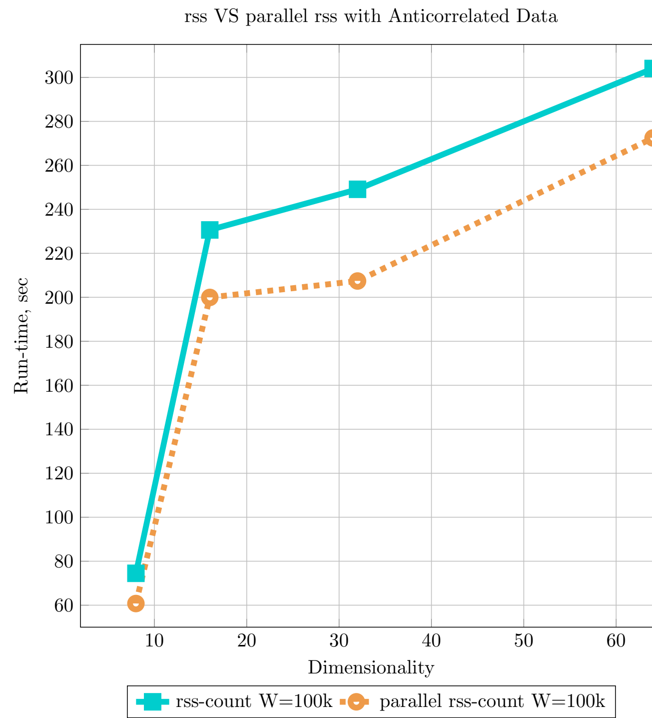


Figure 5.11: *RSS vs PRSS (Anticorrelated Count-Based Window, Different Dimensionalities).*

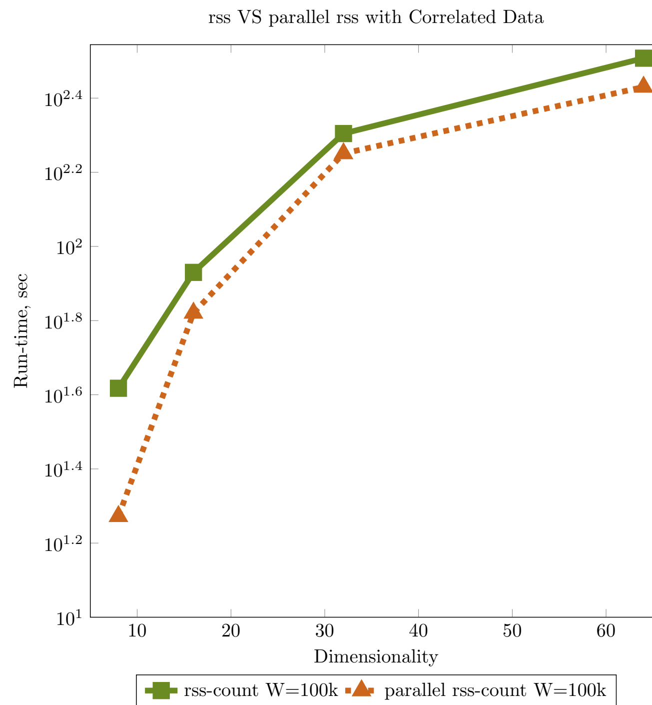


Figure 5.12: *RSS vs PRSS (Correlated Count-Based Window, Different Dimensionalities).*

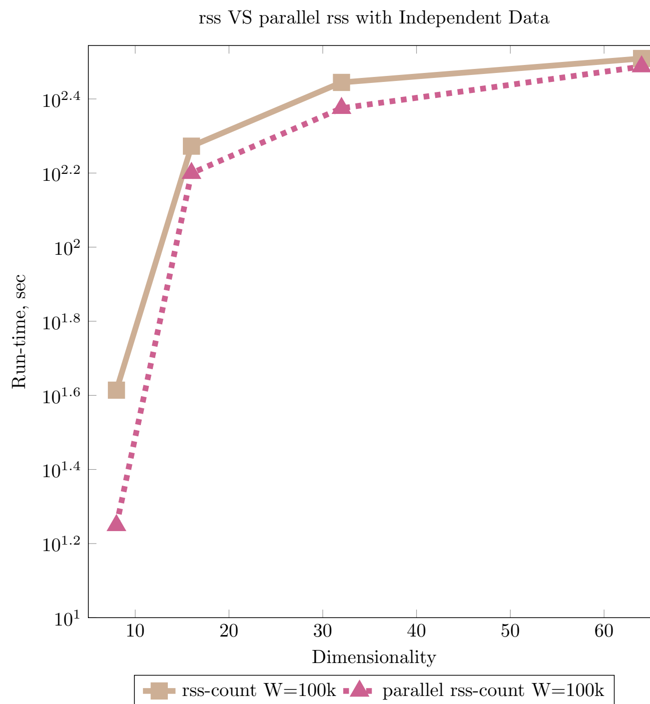


Figure 5.13: *RSS vs PRSS (Independent Count-Based Window, Different Dimensionalities).*

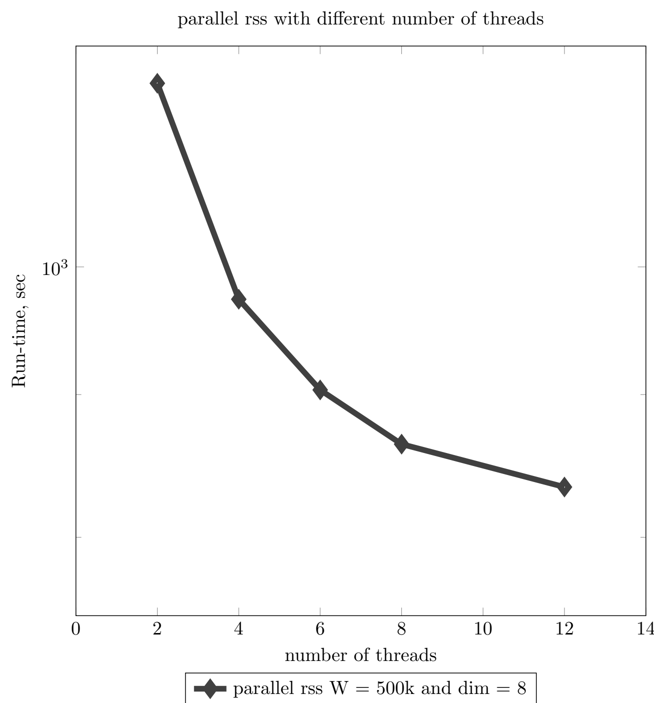


Figure 5.14: *PRSS scalability (Anticorrelated Count-Based Window, Different Numbers of Threads).*

well-known BskyTree algorithm. Our benchmarking focuses on different cases involving varying dimensionalities, cardinalities, and numbers of processing threads.

The runtime in seconds displayed in the figures accounts for the entire skyline maintenance process, including:

- Reception of new tuples.
- Computing the skyline.
- Updating the skyline with new tuples.
- Maintaining skyline indices.
- Removing expired tuples from the sliding window.

For example, if the window size is set to 500K, our experiments initially process 450K tuples to extract the skyline. Upon receiving an additional 50K tuples, the skyline is updated accordingly. This comprehensive measurement ensures a realistic evaluation of skyline query processing in streaming environments.

Initially, a comparison is made between the run-time efficiency of PRSS and the sequential RSS method. The comparison results of the algorithms are displayed in Figures 5.8, 5.9, 5.10 and 5.11, 5.12, 5.13. The lines represent the quantity of attributes present in the datasets, while the bars indicate the duration of the running times, measured in seconds. The experiment utilized sliding window sizes ranging from 100k to 500k tuples for the count-based window, and from 8 to 64 for the dimensionality.

- **Effect of data Dimensionality:** Figures 5.11, 5.12 and 5.13 demonstrates benchmarking results of RSS and PRSS with a fixed cardinality of 100K while increasing dimensionality from 10 to 60 with anti-correlated, correlated, and independent data, respectively.
- PRSS consistently outperforms RSS in all cases, demonstrating better scalability with higher dimensionalities. However, the key observation lies in the rate of this increase. Parallel RSS consistently outperforms its sequential counterpart, showcasing a notable improvement in speed. This improvement is particularly significant when dealing with datasets of higher dimensionality, as demonstrated by the steeper slopes in the execution time curves of PRSS. The parallelization strategy implemented in PRSS efficiently handles the increased computational load induced by higher dimensionality and cardinality, resulting in superior scalability.
- **Effect of data Cardinality:** Figures 5.8, 5.9 and 5.10 presents the benchmarking results for PRSS and RSS with a fixed dimensionality of 8 while varying the window size from 100K to 500K with anti-correlated, correlated, and independent data, respectively.
- Our PRSS algorithm consistently outperforms RSS by leveraging parallel processing, reducing execution time as the window size increases.
- Effect of real dataset: In order to demonstrate the efficiency of Skyline queries, it is necessary to evaluate them using real datasets that may contain duplicate values. In table 5.2, using real data for weather condition monitoring with 15 attributes, our parallel RSS algorithm continues to demonstrate superior performance compared to the sequential RSS algorithm. Hence, our parallel version exhibits superiority not only on synthetic dataset

but also on real-world data. In real-world scenarios where datasets may contain duplicate values and exhibit more complex structures, the parallel version of the algorithm maintains its efficiency. The ability of PRSS to handle real-world datasets underscores its practical applicability and reinforces the significance of the parallelization strategy employed.

- Effect of number of threads: In Fig. 5.14, We showcase the capacity of PRSS to effectively utilize the complete parallel processing capabilities of the CPU. This is achieved by executing the parallel version of PRSS on a dataset with a fixed window size and dimensionality, while varying the number of CPU threads employed from 1 to 12. Observing a fixed dataset window size (100k) and dimensionality (8), it is evident that the execution time of PRSS decreases almost linearly with an increase in the number of threads. The decrease in execution time is nearly linear as the number of threads increases from 1 to 12, highlighting the efficiency of parallelization. This behavior indicates that PRSS can efficiently distribute the workload across multiple threads, minimizing idle time and maximizing CPU utilization. The scalability of PRSS is crucial in scenarios where computational resources can be dynamically allocated, providing a significant advantage over the sequential approach, especially in modern multicore architectures. This demonstrates the effective utilization of the CPU's parallel processing capabilities by PRSS.

- Consistency Across Diverse Datasets: One noteworthy aspect of PRSS's performance is its consistency across diverse datasets. Whether dealing with synthetic datasets of varying dimensionality and cardinality or real-world datasets with complex structures, PRSS consistently outperforms its sequential counterpart. This consistency emphasizes the robustness of the parallelization strategy employed in PRSS, showcasing its adaptability to different data characteristics and scenarios.

  In conclusion, the experimental results and expanded explanations collectively provide a comprehensive understanding of the effectiveness and advantages of the parallel skyline algorithm (PRSS) over its sequential counterpart. The consistent performance across diverse datasets, scalability with the number of threads, and applicability to real-world scenarios highlight the robustness and practicality of the proposed parallelization strategy.

Memory Consumption Comparison

It's important to understand that efficiency in parallel computing is not solely determined by memory consumption. While higher memory usage may raise concerns, the efficiency of parallelization should be evaluated based on various factors, including speedup, scalability, and resource utilization.

Here are some points to consider:

1. Speedup: based on our Runtime Efficiency Comparison our parallel algorithm demonstrates significant speedup compared to the sequential version, it indicates that our parallelization approach is effective in utilizing multiple processing units. This speedup can lead to faster overall execution times, which is a primary goal of parallel computing.

2. Scalability: A well-designed parallel algorithm should scale efficiently with an increasing number of threads. Scalability refers to the ability of the parallel algorithm to maintain or improve performance as the system size grows. our parallel algorithm exhibits good scalability, thus the parallelization approach is well-suited for larger problem sizes.

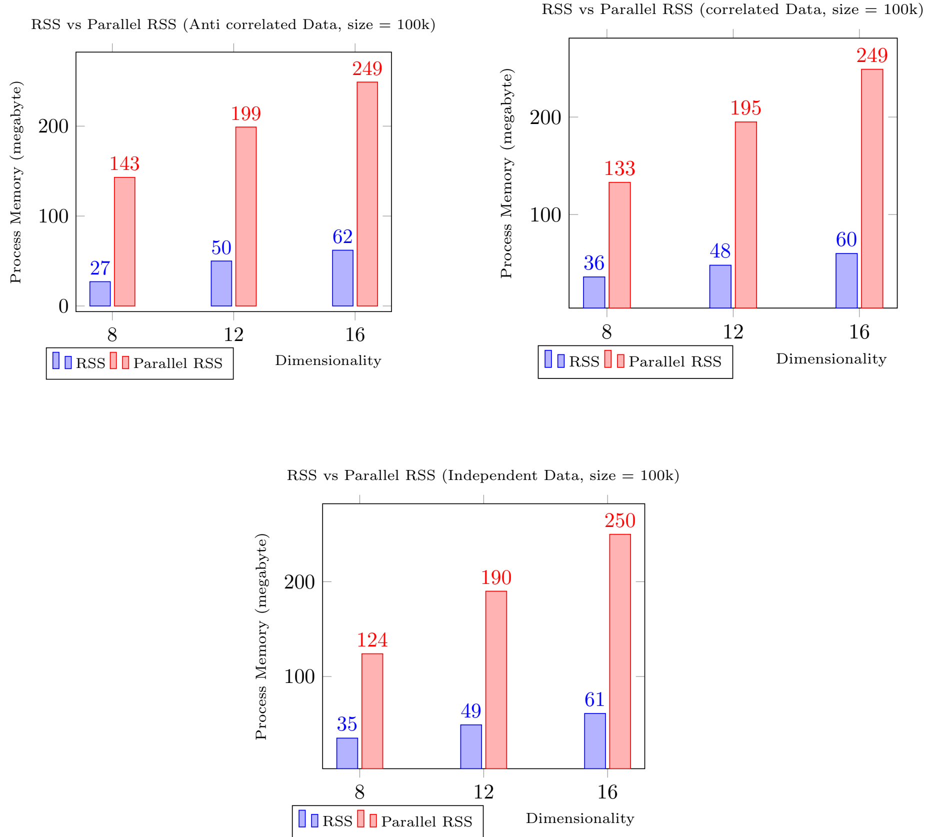


Figure 5.15: *Memory consumption of RSS vs PRSS (Anti, Corr, Ind data, Different Dimensionalities).*

3. Resource Utilization: While higher memory consumption may raise concerns, it's essential to assess whether the memory usage is justified by the performance gains achieved through parallelization. Efficient utilization of available resources, including CPU cores and memory, is a key aspect of parallel computing. our parallel algorithm effectively leverages resources to achieve better performance, thus the higher memory usage is acceptable.

4. Trade-offs: Parallel computing often involves trade-offs between factors such as memory consumption, speedup, and scalability. It's essential to consider these trade-offs in the context of our specific application requirements and system constraints. The increase in memory usage for PRSS algorithm has a reasonable trade-off for significant performance improvements.

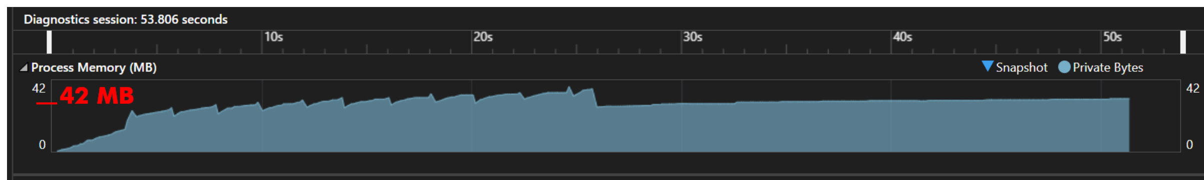
Liu et al. [79] provides insights into the memory consumption of the sequential RSS algorithm based on the data structures used, specifically a dynamic dimension index based on B-trees. Let's break down the key points of the analysis:

1. Tuple Memory Requirements: Each d-dimensional tuple requires 24xd bytes to store all index entries, including values, header links, and dimension links. Additionally, a header for each tuple requires 20 bytes to store a tuple ID, a dominance list pointer, and a skyline flag.

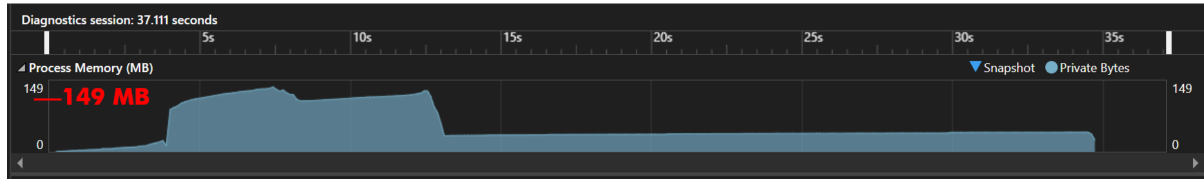
2. Space Complexity of B-tree: The space complexity of the B-tree is O(N), where N is the number of nodes. Assuming 2 64-bit pointers per node to maintain the tree, the total memory required by the dynamic dimension index is calculated as 16 x N x (24 x d + 20) bytes.

3. Memory Requirement Example: The analysis provides an example stating that 100 MB of memory space is sufficient to handle a 10 K window of 20-dimensional streaming data.

Using Microsoft Visual Studio's memory tool, we analyzed the Impact of Parallelism on



(a) *Memory consumption of Sequential RSS*



(b) *Memory consumption of Parallel RSS*

Figure 5.16: *Tracking Memory consumption of RSS vs PRSS.*

Memory Consumption of PRSS as illustrated in figure 5.16. While PRSS offers significant performance improvements, it does come with higher memory usage, particularly due to the parallel overhead, including thread stack space and duplicated data across CPU cores. However, this trade-off is justified by the algorithm's faster processing times, especially in high-demand, real-time applications. And as we see the memory usage rises only at the begining Fig. 5.16b as each thread allocates its private variables but stabilizes as the algorithm progresses. PRSS employs dynamic memory allocation to optimize resource usage based on the dataset size and workload, minimizing memory overhead. Additionally, cache optimization reduces memory access times, and memory management techniques prevent leaks, ensuring stability. Overall, while PRSS consumes more memory, the performance gains in execution speed and scalability make this trade-off worthwhile.

our parallel PRSS algorithm consumes more memory than the sequential RSS while running faster can be attributed to several factors:

1. Parallel Overhead: When we parallelize The RSS algorithm using OpenMP, it creates multiple threads to execute computation in parallel. Each thread requires its own stack space, which adds to the overall memory consumption. Additionally, OpenMP might allocate additional resources for thread management and synchronization, contributing to increased memory usage.

2. Data Duplication: In parallel computing, some data needs to be duplicated or partitioned across threads to ensure each thread has access to the required data independently. This replication or partitioning can lead to higher memory consumption compared to the sequential version, where data might be processed in-place.

3. Parallelization Overheads: - Each thread running the sequential RSS algorithm on a chunk of data will incur memory overhead for thread stack space and local variables used within the thread. - The number of threads spawned in parallel and the size of the data chunks processed by each thread will determine the overall memory consumption during parallel execution.

4. Combining Global Results: - After processing chunks of data in parallel, PRSS combines the global results outside the parallel region. This step requires additional memory for storing the aggregated results before the final skyline computation.

Despite the increased memory consumption, our parallel PRSS version can still run faster due to the parallel execution of computation across multiple threads. This increased speed can result from exploiting available CPU resources more effectively, reducing overall execution time despite the higher memory footprint.

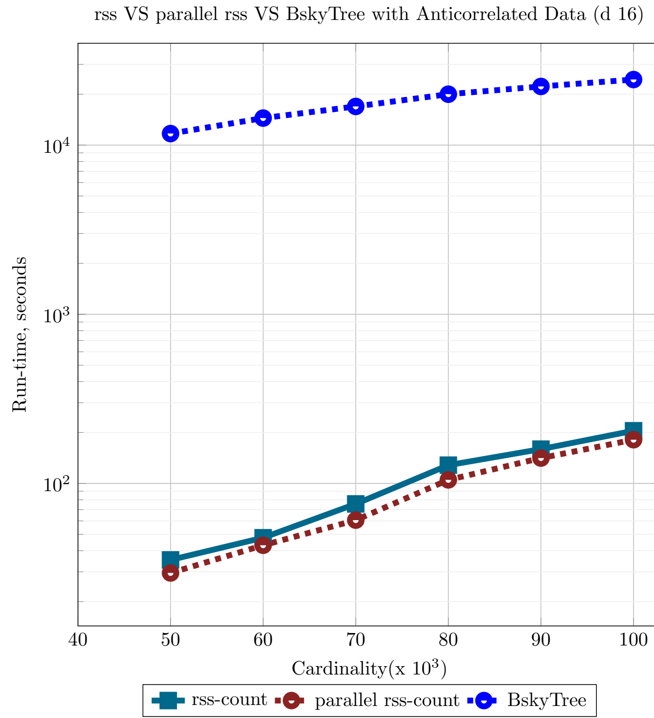


Figure 5.17: *RSS vs PRSS vs BskyTree (Anticorrelated Count-Based Windows, Different Cardinalities).*

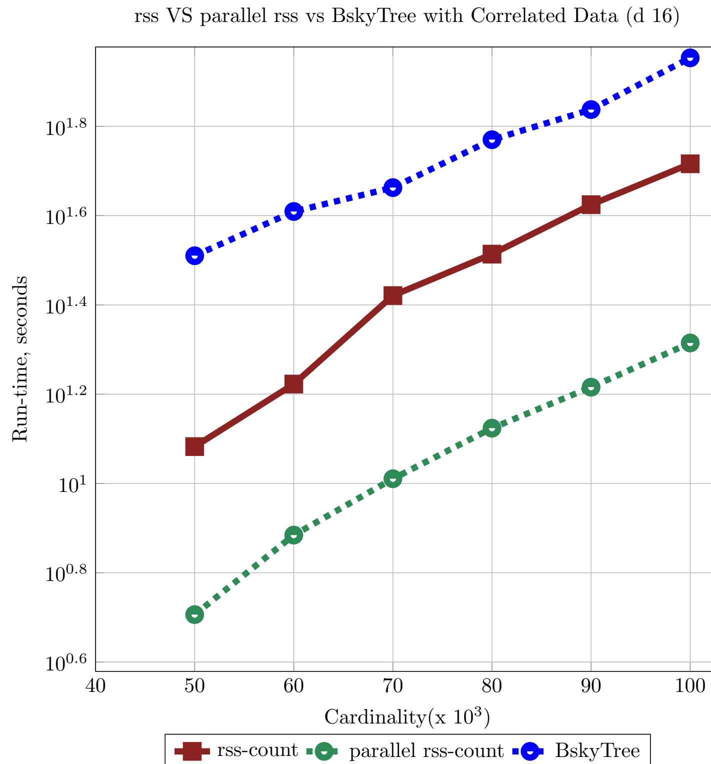


Figure 5.18: *RSS vs PRSS vs BskyTree (Correlated Count-Based Windows, Different Cardinalities).*

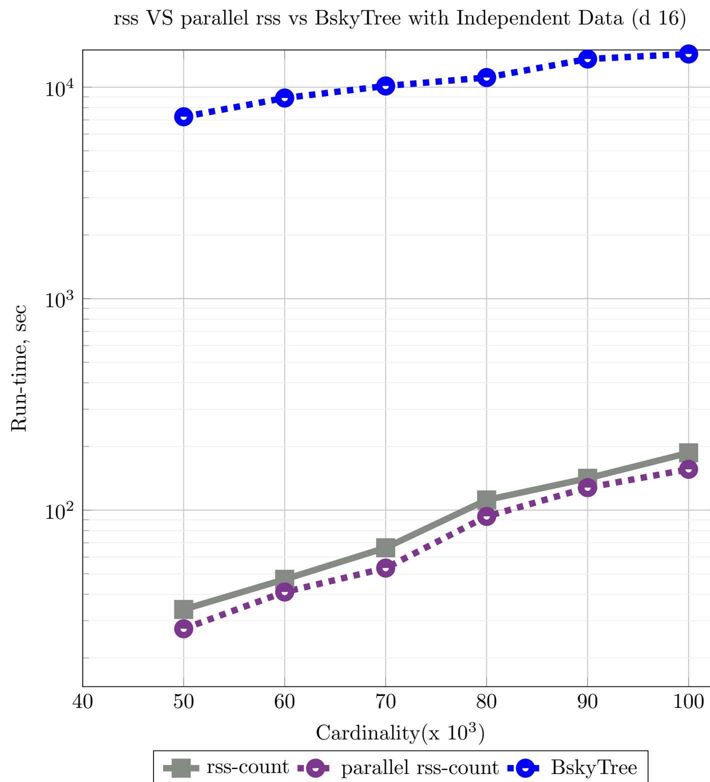


Figure 5.19: *RSS vs PRSS vs BskyTree (Independent Count-Based Windows, Different Cardinalities).*

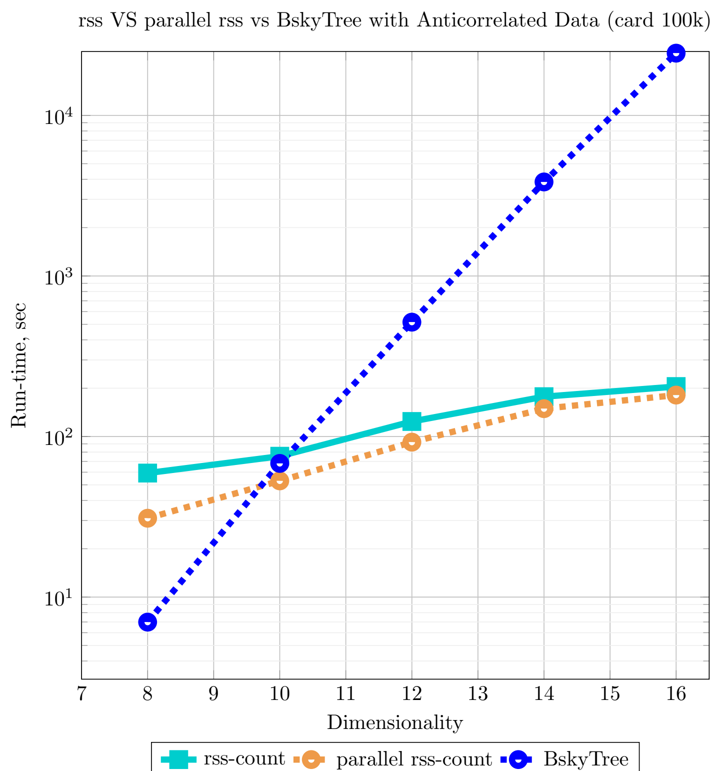


Figure 5.20: *RSS vs PRSS vs BskyTree (Anticorrelated Count-Based Windows, Different Dimensionalities).*

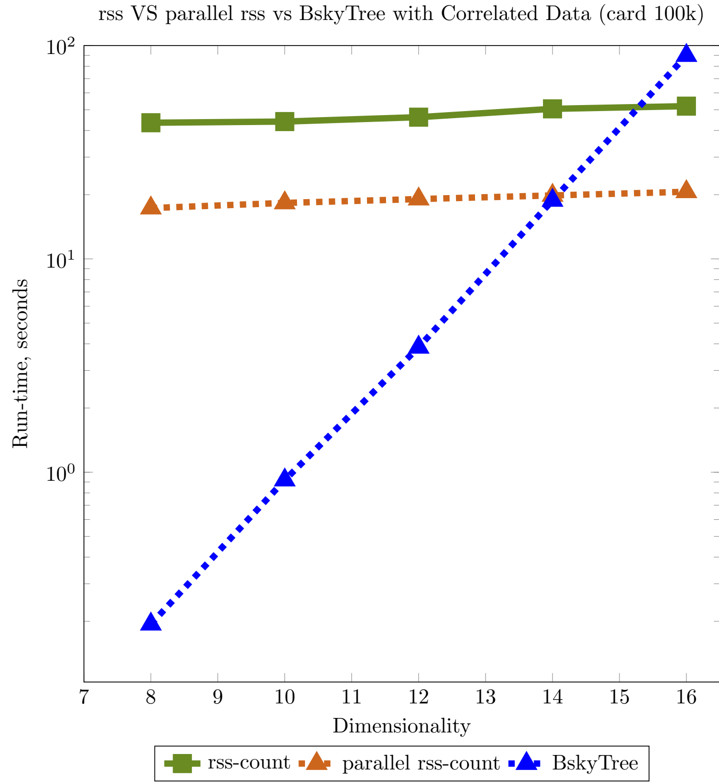


Figure 5.21: *RSS vs PRSS vs BskyTree (Correlated Count-Based Windows, Different dimensionalities).*

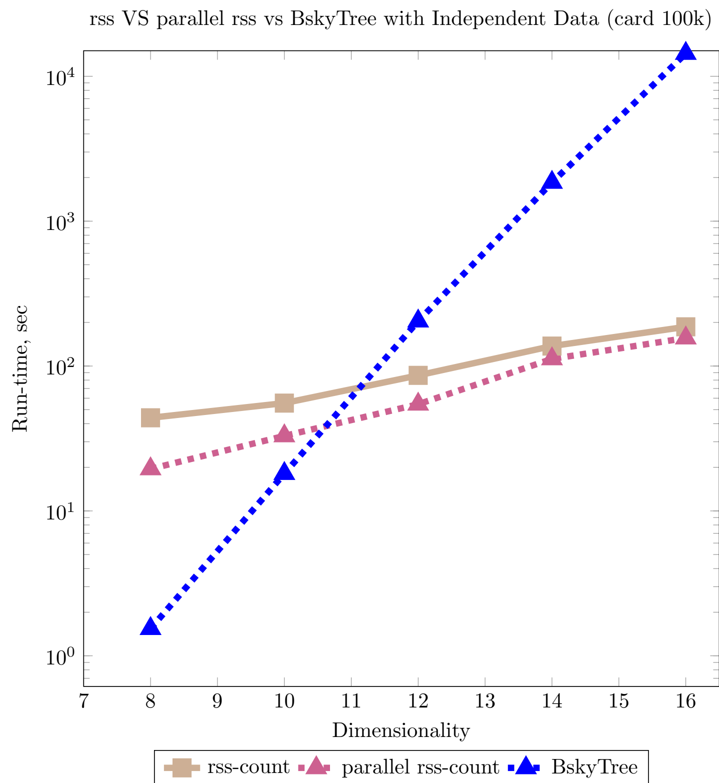


Figure 5.22: *RSS vs PRSS vs BskyTree (Independent Count-Based windows, Different Dimensionalities).*

Comparison of RSS and PRSS with BSkyTree

To assess the performance of our parallel algorithm, we compare it (i) to a baseline approach which computes the skyline using state of the art algorithm BSkyTree [96]. The goal of this comparison is to show that (i) without any index structure, the best skyline algorithm known so far is unable to handle multidimensional skyline queries when the dimensionality is moderately large in a streaming context on both synthetic datasets and real world datasets.

First, the effects of (1) dimensionality and (2) cardinality of data have been evaluated. For (1), the cardinality is fixed to 100K and for (2), the dimensionality of data is fixed to 16.

PRSS and RSS outperform BSkyTree in high dimensionality domains on AC ($d > 10$) and Corr data ($d > 14$) and Independent data ($d > 12$) but is less efficient than BSkyTree in low dimensionalities. PRSS systematically outperforms RSS and BSkyTree on all data.

We point out two observations from this experiment:

1. PRSS is faster with more than one order of magnitude in most cases where $d > 10$.
2. state of the art algorithm BSkyTree algorithm takes a lot of time to extract skyline when dimensionality $d > 10$ thus it is unefficient in high dimensionality domains.

RSS vs PRSS vs BskyTree on Count-Based Windows with Different Cardinalities: Figures 5.17, 5.18 and 5.19 shows the comparison of RSS, PRSS and BskyTree with a fixed dimensionality of 16 and varying window sizes from 50K to 100K with anti-correlated, correlated, and independent data, respectively.

PRSS significantly outperforms both RSS and BskyTree due to its parallel execution and efficient indexing techniques.

RSS vs PRSS vs BskyTree on Count-Based Windows with Different Dimensionalities: Figures 5.20, 5.21 and 5.22 shows the comparison of RSS, PRSS and BskyTree with a fixed cardinality of 100K while varying dimensions from 8 to 16 with anti-correlated, correlated, and independent data, respectively.

The PRSS algorithm continues to outperform the sequential RSS and BskyTree algorithms, maintaining superior efficiency as dimensionality increases.

5.4 Conclusion

The results confirm that PRSS is significantly faster than RSS and BskyTree in all tested scenarios. By effectively utilizing parallel processing, PRSS reduces runtime across varying window sizes, dimensionalities, and dataset distributions. The scalability analysis further demonstrates that PRSS efficiently leverages multicore architectures, making it a robust solution for skyline computation in high-dimensional data streams.

PRSS achieves superior performance through its novel indexing mechanism that dynamically adjusts to varying dimensionalities, ensuring efficient tuple organization. The window partitioning strategy enables effective load balancing, allowing PRSS to distribute workload evenly across processing cores. Optimizations such as AVX2 instruction sets significantly accelerate tuple comparisons, reducing computational overhead. Furthermore, PRSS scales well as additional threads are introduced, maintaining its efficiency in high-throughput environments.

Experimental results with real-world datasets validate PRSS's effectiveness, showcasing its adaptability to diverse data distributions. Its ability to handle large-scale skyline queries efficiently makes PRSS a compelling choice for practical applications requiring real-time skyline computation. These advantages collectively establish PRSS as a state-of-the-art solution, outperforming both the sequential RSS algorithm and the well-known BskyTree algorithm.

Part V

Outlooks and conclusion

5.5 Conclusion

In this study, we have systematically investigated the parallelization of a well-known sequential Skyline algorithm on a multicore processor, aiming to enhance its scalability and overall performance. Our research introduced the Parallel Range Search Skyline (PRSS) algorithm, a highly efficient method specifically designed to extract skyline points from continuous data streams. To optimize dominance tests within the sliding window, we employed a dynamic dimension indexing technique, which significantly reduced computational overhead and improved processing speed. Through extensive experimentation, we evaluated the PRSS algorithm under various conditions, utilizing count-based sliding windows on both synthetic and real-world datasets with high dimensionality. The results demonstrated a substantial performance gain over both the sequential implementation and the state-of-the-art BSkyTree algorithm, underscoring the practical utility of our approach in real-world applications.

The significance of our study extends beyond performance improvements, as it highlights the broader potential of parallel computing in large-scale, high-dimensional data processing. The effectiveness of multicore architectures in real-time skyline computation, as showcased in our experiments, presents new opportunities for scalable and high-performance data stream analysis. In particular, our work emphasizes the importance of efficient parallelization strategies in handling the inherent complexities of continuous and dynamic data environments. The insights gained from this research can inform the development of future skyline query processing techniques in domains such as financial analytics, transportation systems, and sensor-based monitoring.

Despite the advancements introduced in this study, several challenges remain open for future exploration. One key limitation lies in the potential overhead introduced by parallel processing in cases where data distributions lead to load imbalances among processing cores. Addressing this issue requires the development of adaptive load-balancing mechanisms to further optimize parallel skyline computation. Additionally, while our PRSS algorithm leverages AVX2 instructions for dominance test optimization, future work could explore the integration of more advanced vectorization techniques or specialized hardware accelerators to further enhance performance.

Furthermore, a promising direction for extending this research is the deployment of PRSS on heterogeneous computing architectures, such as Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs). These architectures offer the potential for even greater parallelism and energy efficiency, making them well-suited for real-time applications involving high-velocity and high-volume streaming data. Additionally, investigating skyline computation in the presence of incomplete and uncertain data remains an important avenue for future study. Extending PRSS to support probabilistic skyline queries and other uncertainty-aware models would significantly enhance its applicability in real-world scenarios.

In conclusion, this research contributes to both theoretical and practical advancements in

skyline computation over data streams, demonstrating the feasibility and effectiveness of parallelization techniques in multicore environments. By bridging the gap between parallel computing and skyline query processing, our work paves the way for further innovations in high-performance big data analytics. The promising results obtained in this study reinforce the critical role of parallelism in managing complex data-intensive applications, providing a strong foundation for future developments in this field.

5.6 List of Publications

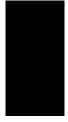
This Section lists the publications that have resulted from the research conducted during my PhD.

[1] Walid Khames, Allel Hadjali, and Mohand Lagha. "Parallel continuous skyline query over high-dimensional data stream windows". In: *Distributed and Parallel Databases* (2024), pp. 1–56.

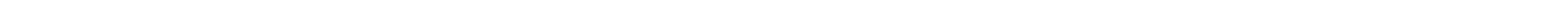
[2] Walid Khames, Allel Hadjali, and Mohand Lagha. "Skyline Computation on Multicore Architectures: A Survey". In: *2020 International Conference on Data Analytics for Business and Industry: Way Towards a Sustainable Economy (ICDABI)*. IEEE. 2020, pp. 1–6.

Part VI

Bibliography



Bibliography



Bibliography

[1] Samet Ayhan et al. "Predictive analytics with aviation big data". In: *2013 Integrated Communications, Navigation and Surveillance Conference (ICNS)*. IEEE. 2013, pp. 1–13 *Cited on pages 3, 6.*

[2] Sujie Li et al. "Civil aircraft big data platform". In: *2017 IEEE 11th International Conference on Semantic Computing (ICSC)*. IEEE. 2017, pp. 328–333 *Cited on pages 3, 5, 7, 8.*

[3] Anastasia Yastrebova et al. "Future networks 2030: Architecture and requirements". In: *2018 10th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. IEEE. 2018, pp. 1–8 *Cited on page 3.*

[4] Lena Rudenko and Markus Endres. "Real-time skyline computation on data streams". In: *New Trends in Databases and Information Systems: ADBIS 2018 Short Papers and Workshops, AI* QA, BIGPMED, CSACDB, M2U, BigDataMAPS, ISTREND, DC, Budapest, Hungary, September, 2-5, 2018, Proceedings 22*. Springer. 2018, pp. 20–28 *Cited on pages 3, 24, 40.*

[5] Tiziano De Matteis, Salvatore Di Girolamo, and Gabriele Mencagli. "Continuous skyline queries on multicore architectures". In: *Concurrency and Computation: Practice and Experience* 28.12 (2016), pp. 3503–3522 *Cited on pages 4, 22, 43, 61.*

[6] Songnian Zhang et al. "PPsky: Privacy-Preserving Skyline Queries with Secret Sharing in eHealthcare". In: *GLOBECOM 2022-2022 IEEE Global Communications Conference*. IEEE. 2022, pp. 5469–5474 *Cited on pages 4, 5.*

[7] Xiaofeng Ding et al. "Efficient and privacy-preserving multi-party skyline queries over encrypted data". In: *IEEE Transactions on Information Forensics and Security* 16 (2021), pp. 4589–4604 *Cited on page 4.*

[8] Yumei Luo, Honghua Zhao, and Binhua Xiong. "Research on Air Conditioning Performance Monitoring and Trend Prediction of A320 Aircraft Based on Big Data Analysis". In: *2021 IEEE 3rd International Conference on Civil Aviation Safety and Information Technology (ICCASIT)*. IEEE. 2021, pp. 375–379 *Cited on pages 4, 9.*

[9] Brian Babcock et al. "Models and issues in data stream systems". In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2002, pp. 1–16 *Cited on page 4.*

[10] Henrique CM Andrade, Buğra Gedik, and Deepak S Turaga. *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press, 2014 *Cited on pages 4, 5, 10, 11, 18–20, 53.*

[11] Quoc-Cuong To, Juan Soto, and Volker Markl. "A survey of state management in big data processing systems". In: *The VLDB Journal* 27.6 (2018), pp. 847–872 *Cited on page 4.*

[12] Michael Stonebraker, U?ur Çetintemel, and Stan Zdonik. "The 8 requirements of real-time stream processing". In: *ACM Sigmod Record* 34.4 (2005), pp. 42–47 *Cited on pages 4, 22.*

[13] Zhi Cai et al. "Continuous Road Network-Based Skyline Query for Moving Objects". In: *IEEE Transactions on Intelligent Transportation Systems* 22.12 (2020), pp. 7383–7394 *Cited on page 5.*

[14] Ismail Kertiou et al. "A dynamic skyline technique for a context-aware selection of the best sensors in an IoT architecture". In: *Ad Hoc Networks* 81 (2018), pp. 183–196 *Cited on pages 5, 33, 39, 88.*

[15] Sujie Li, Guigang Zhang, and Jian Wang. "Civil aircraft health management research based on big data and deep learning technologies". In: *2017 IEEE International Conference on Prognostics and Health Management (ICPHM)*. IEEE. 2017, pp. 154–159 *Cited on pages 5, 7.*

[16] Kathrin Rodriguez Llanes, Marco Antonio Casanova, and Noel Moreno Lemus. "From sensor data streams to linked streaming data: a survey of main approaches". In: *Journal of Information and Data Management* 7.2 (2016), pp. 130–130 *Cited on page 7.*

[17] Kostas Patroumpas, Nikos Pelekis, and Yannis Theodoridis. "On-the-fly mobility event detection over aircraft trajectories". In: *Proceedings of the 26th ACM SIGSPATIAL international conference on advances in geographic information systems*. 2018, pp. 259–268 *Cited on page 8.*

[18] Thomas Plagemann et al. "Using data stream management systems for traffic analysis–a case study–". In: *Passive and Active Network Measurement: 5th International Workshop, PAM 2004, Antibes Juan-les-Pins, France, April 19-20, 2004. Proceedings 5*. Springer. 2004, pp. 215–226 *Cited on page 9.*

[19] Sepanta Zeighami, Gabriel Ghinita, and Cyrus Shahabi. "Secure dynamic skyline queries using result materialization". In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE. 2021, pp. 157–168 *Cited on pages 9, 67.*

[20] Johan Bollen, Huina Mao, and Xiaojun Zeng. "Twitter mood predicts the stock market". In: *Journal of computational science* 2.1 (2011), pp. 1–8 *Cited on page 10.*

[21] Karim Alami and Sofian Maabout. "A framework for multidimensional skyline queries over streaming data". In: *Data & Knowledge Engineering* 127 (2020), p. 101792 *Cited on pages 10, 23, 33, 59.*

[22] Gianpaolo Cugola and Alessandro Margara. "Complex event processing with T-REX". In: *Journal of Systems and Software* 85.8 (2012), pp. 1709–1728 *Cited on pages 10, 22.*

[23] *Apache Storm*. https://storm.apache.org/ *Cited on pages 11, 23.*

[24] *Apache Flink*. https://flink.apache.org/ *Cited on pages 11, 23.*

[25] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113 *Cited on pages 11, 51.*

[26] Tiziano De Matteis. "Parallel Patterns for Adaptive Data Stream Processing". PhD thesis. PhD thesis, University of Pisa, Italy, 2016 *Cited on pages 11, 12, 18, 21, 42, 45–47, 53–58.*

[27] John Darlington et al. "Parallel skeletons for structured composition". In: *Proceedings of the Fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 1995, pp. 19–28 *Cited on pages 11, 42.*

[28] David B Skillicorn and Domenico Talia. "Models and languages for parallel computation". In: *Acm Computing Surveys (Csur)* 30.2 (1998), pp. 123–169 *Cited on pages 11, 42.*

[29] Murray Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming". In: *Parallel computing* 30.3 (2004), pp. 389–406 *Cited on pages 11, 42, 43.*

[30] Hyeonseung Im, Jonghyun Park, and Sungwoo Park. "Parallel skyline computation on multicore architectures". In: *Information Systems* 36.4 (2011), pp. 808–823 *Cited on page 12.*

[31] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55 *Cited on pages 12, 43.*

[32] Jose Carlos Romero et al. "SkyFlow: Heterogeneous streaming for skyline computation using FlowGraph and SYCL". In: *Future Generation Computer Systems* 141 (2023), pp. 269–283 *Cited on page 12.*

[33] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012 *Cited on pages 12, 42, 53.*

[34] Tiziano De Matteis and Gabriele Mencagli. "Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach". In: *International Journal of Parallel Programming* 45.2 (2017), pp. 382–401 *Cited on pages 12, 24, 42, 45–48, 53–55.*

[35] David del Rio Astorga et al. "A C++ generic parallel pattern interface for stream processing". In: *Algorithms and Architectures for Parallel Processing: 16th International Conference, ICA3PP 2016, Granada, Spain, December 14-16, 2016, Proceedings*. Springer. 2016, pp. 74–87 *Cited on pages 12, 53.*

[36] Walid Khames, Allel Hadjali, and Mohand Lagha. "Parallel continuous skyline query over high-dimensional data stream windows". In: *Distributed and Parallel Databases* (2024), pp. 1–56 *Cited on pages 12, 13, 54, 105.*

[37] Mark Sullivan and Andrew Heybey. "A system for managing large databases of network traffic". In: *Proceedings of USENIX*. 1998 *Cited on page 18.*

[38] Henrique Andrade et al. "Scale-up strategies for processing high-rate data streams in System S". In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE. 2009, pp. 1375–1378 *Cited on pages 18, 20.*

[39] Katja Hose and Akrivi Vlachou. "A survey of skyline processing in highly distributed environments". In: *The VLDB Journal* 21 (2012), pp. 359–384 *Cited on pages 18, 40.*

[40] Charu C Aggarwal and Philip S Yu. "A survey of synopsis construction in data streams". In: *Data streams: models and algorithms* (2007), pp. 169–207 *Cited on page 19.*

[41] Tiziano De Matteis, Salvatore Di Girolamo, and Gabriele Mencagli. "A multicore parallelization of continuous skyline queries on data streams". In: *European Conference on Parallel Processing*. Springer. 2015, pp. 402–413 *Cited on pages 19, 24, 43, 61.*

[42] Sirish Chandrasekaran et al. "TelegraphCQ: continuous dataflow processing". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 2003, pp. 668–668 *Cited on page 22.*

[43] STREAM Group et al. *STREAM: The stanford stream data manager*. Tech. rep. Stanford InfoLab, 2003 *Cited on page 22.*

[44] Vincenzo Gulisano et al. "Streamcloud: An elastic and scalable data streaming system". In: *IEEE Transactions on Parallel and Distributed Systems* 23.12 (2012), pp. 2351–2365 *Cited on pages 22, 24.*

[45] Chuck Cranor et al. "Gigascope: A stream database for network applications". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 2003, pp. 647–651 *Cited on page 22.*

[46] Daniel J Abadi et al. "Aurora: a new model and architecture for data stream management". In: *the VLDB Journal* 12 (2003), pp. 120–139 *Cited on page 22.*

[47] Opher Etzion and Peter Niblett. *Event processing in action*. Manning Publications Co., 2010 *Cited on page 22.*

[48] Eugene Wu, Yanlei Diao, and Shariq Rizvi. "High-performance complex event processing over streams". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 2006, pp. 407–418 *Cited on page 22.*

[49] *Oracle Stream Analytics*. https://www.oracle.com/middleware/technologies/stream-processing.html *Cited on page 22.*

[50] *TIBCO® Streaming*. https://www.tibco.com/products/tibco-streaming *Cited on page 22.*

[51] *Esper - EsperTech*. https://www.espertech.com/esper/. Mar. 2022 *Cited on page 22.*

[52] *Amazon Kinesis Data Streams*. https://aws.amazon.com/kinesis/data-streams/ *Cited on page 23.*

[53] *Apache Spark*. https://spark.apache.org/ *Cited on page 23.*

[54] SPT Krishnan et al. "Google cloud dataflow". In: *Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects* (2015), pp. 255–275 *Cited on page 23.*

[55] Anindita Basak et al. *Stream Analytics with Microsoft Azure: Real-time data processing for quick insights using Azure Stream Analytics*. Packt Publishing Ltd, 2017 *Cited on page 23.*

[56] Chuck Ballard et al. *Ibm infosphere streams: Assembling continuous insight in the information revolution*. IBM Redbooks, 2012 *Cited on page 23.*

[57] Walid Khames, Allel Hadjali, and Mohand Lagha. "Skyline Computation on Multi-core Architectures: A Survey". In: *2020 International Conference on Data Analytics for Business and Industry: Way Towards a Sustainable Economy (ICDABI)*. IEEE. 2020, pp. 1–6 *Cited on pages 23, 42.*

[58] Anders Fredrik Ulvig Kiær. "Skyline Computing over multiple Data Streams with a Storm Cluster." MA thesis. NTNU, 2014 *Cited on page 24.*

[59] Ze Deng et al. "Spatial-keyword skyline publish/subscribe query processing over distributed sliding window streaming data". In: *IEEE Transactions on Computers* 71.10 (2022), pp. 2659–2674 *Cited on page 24.*

[60] Jimmy Ming-Tai Wu et al. "Mining Skyline Patterns from Big Data Environments based on a Spark Framework". In: *Journal of Grid Computing* 21.2 (2023), p. 22 *Cited on page 24.*

[61] Ioanna Papanikolaou. "Distributed Algorithms for Skyline Computation using Apache Spark". In: (2020) *Cited on page 24.*

[62] Bugra Gedik et al. "Grubjoin: An adaptive, multi-way, windowed stream join with time correlation-aware cpu load shedding". In: *IEEE Transactions on Knowledge and Data Engineering* 19.10 (2007), pp. 1363–1380 *Cited on page 24.*

[63] Jens Teubner and Rene Mueller. "How soccer players would do stream joins". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 2011, pp. 625–636 *Cited on page 24.*

[64] Daniele Buono, Tiziano De Matteis, and Gabriele Mencagli. "A high-throughput and low-latency parallelization of window-based stream joins on multicores". In: *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE. 2014, pp. 117–126 *Cited on page 24.*

[65] Hua Lu, Yongluan Zhou, and Jonas Haustad. "Efficient and scalable continuous skyline monitoring in two-tier streaming settings". In: *Information Systems* 38.1 (2013), pp. 68–81 *Cited on page 24.*

[66] Xiaoyong Li et al. "Parallelizing skyline queries over uncertain data streams with sliding window partitioning and grid index". In: *Knowledge and Information Systems* 41.2 (2014), pp. 277–309 *Cited on pages 24, 61.*

[67] Xiaoyong Li et al. "Parallel skyline queries over uncertain data streams in cloud computing environments". In: *International Journal of Web and Grid Services* 10.1 (2014), pp. 24–53 *Cited on pages 24, 61.*

[68] Jinchao Zhang et al. "Efficient algorithms of parallel Skyline join over data streams". In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2018, pp. 184–199 *Cited on pages 24, 44, 61.*

[69] Martin Hirzel. "Partition and compose: Parallel complex event processing". In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. 2012, pp. 191–200 *Cited on page 24.*

[70] Gianpaolo Cugola and Alessandro Margara. "Low latency complex event processing on parallel hardware". In: *Journal of Parallel and Distributed Computing* 72.2 (2012), pp. 205–218 *Cited on page 24.*

[71] Cagri Balkesen et al. “Rip: Run-based intra-query parallelism for scalable complex event processing”. In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. 2013, pp. 3–14 *Cited on page 24.*

[72] Sai Wu et al. “Parallelizing stateful operators in a distributed stream processing system: how, should you and how much?” In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. 2012, pp. 278–289 *Cited on page 25.*

[73] Scott Schneider et al. “Safe data parallelism for general streaming”. In: *IEEE transactions on computers* 64.2 (2013), pp. 504–517 *Cited on page 25.*

[74] Cagri Balkesen and Nesime Tatbul. “Scalable data partitioning techniques for parallel sliding window processing over data streams”. In: *International workshop on data management for sensor networks (DMSN)*. 2011 *Cited on pages 25, 55.*

[75] Jin Li et al. “No pane, no gain: efficient evaluation of sliding-window aggregates over data streams”. In: *Acm Sigmod Record* 34.1 (2005), pp. 39–44 *Cited on pages 25, 55.*

[76] Jongwuk Lee and Seung-won Hwang. “Toward efficient multidimensional subspace skyline computation”. In: *The VLDB Journal* 23 (2014), pp. 129–145 *Cited on pages 30, 37.*

[77] Apostolos N Papadopoulos et al. “Skylines and Other Dominance-Based Queries”. In: *Synthesis Lectures on Data Management* 15.2 (2020), pp. 1–158 *Cited on pages 30, 33, 34.*

[78] Stephan Borzsony, Donald Kossmann, and Konrad Stocker. “The skyline operator”. In: *Proceedings 17th international conference on data engineering*. IEEE. 2001, pp. 421–430 *Cited on pages 32, 34, 58, 62, 67.*

[79] Rui Liu and Dominique Li. “Dynamic Dimension Indexing for Efficient Skyline Maintenance on Data Streams”. In: *International Conference on Database Systems for Advanced Applications*. Springer. 2020, pp. 272–287 *Cited on pages 32, 33, 40, 93, 94, 96–98, 122.*

[80] Hua Lu, Yongluan Zhou, and Jonas Haustad. “Continuous skyline monitoring over distributed data streams”. In: *Scientific and Statistical Database Management: 22nd International Conference, SSDBM 2010, Heidelberg, Germany, June 30–July 2, 2010. Proceedings 22*. Springer. 2010, pp. 565–583 *Cited on page 33.*

[81] Atish Das Sarma et al. “Randomized multi-pass streaming skyline algorithms”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 85–96 *Cited on page 33.*

[82] Fan Guo et al. “Robust and Automatic Skyline Detection Algorithm Based on MSSDN”. In: *Journal of Advanced Computational Intelligence and Intelligent Informatics* 24.6 (2020), pp. 750–762 *Cited on page 33.*

[83] Hyeonseung Im and Sungwoo Park. “Group skyline computation”. In: *Information Sciences* 188 (2012), pp. 151–169 *Cited on pages 34, 68.*

[84] Cheng Sheng and Yufei Tao. "On finding skylines in external memory". In: *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems*. 2011, pp. 107–116 *Cited on page 34.*

[85] Markus Endres and Erich Glaser. "Indexing for Skyline Computation: A Comparison Study". In: *Flexible Query Answering Systems: 13th International Conference, FQAS 2019, Amantea, Italy, July 2–5, 2019, Proceedings 13*. Springer. 2019, pp. 31–42 *Cited on page 34.*

[86] Mohamed E Khalefa, Mohamed F Mokbel, and Justin J Levandoski. "Skyline query processing for incomplete data". In: *2008 IEEE 24th international conference on data engineering*. IEEE. 2008, pp. 556–565 *Cited on pages 34, 60.*

[87] Yunjun Gao et al. "Processing k-skyband, constrained skyline, and group-by skyline queries on incomplete data". In: *Expert Systems with Applications* 41.10 (2014), pp. 4959–4974 *Cited on page 34.*

[88] Shiming Zhang, Nikos Mamoulis, and David W Cheung. "Scalable skyline computation using object-based space partitioning". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 2009, pp. 483–494 *Cited on page 34.*

[89] Dimitris Papadias et al. "An optimal and progressive algorithm for skyline queries". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 2003, pp. 467–478 *Cited on pages 34, 66, 77, 80.*

[90] Shengli Sun et al. "Efficient monitoring of skyline queries over distributed data streams". In: *Knowledge and information systems* 25.3 (2010), pp. 575–606 *Cited on pages 34, 59.*

[91] Wolf-Tilo Balke, Ulrich Güntzer, and Jason Xin Zheng. "Efficient distributed skylining for web information systems". In: *Advances in Database Technology-EDBT 2004: 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004 9*. Springer. 2004, pp. 256–273 *Cited on pages 34, 40.*

[92] Eric Lo et al. "Progressive skylining over web-accessible databases". In: *Data & Knowledge Engineering* 57.2 (2006), pp. 122–147 *Cited on pages 34, 40.*

[93] Kian-Lee Tan, Pin-Kwang Eng, Beng Chin Ooi, et al. "Efficient progressive skyline computation". In: *VLDB*. Vol. 1. 2001, pp. 301–310 *Cited on page 34.*

[94] Ken CK Lee et al. "Z-SKY: an efficient skyline query processing framework based on Z-order". In: *The VLDB Journal* 19 (2010), pp. 333–362 *Cited on page 34.*

[95] Joachim Selke and Wolf-Tilo Balke. "SkyMap: a trie-based index structure for high-performance skyline query processing". In: *International Conference on Database and Expert Systems Applications*. Springer. 2011, pp. 350–365 *Cited on page 34.*

[96] 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*. 2010, pp. 195–206 *Cited on pages 34, 127.*

[97] Kenichi Koizumi et al. "BJR-tree: fast skyline computation algorithm for serendipitous searching problems". In: *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2017, pp. 272–282 *Cited on pages 34, 38.*

[98] Jing Yu, Xin Liu, and Guo-hua Liu. “A window-based algorithm for skyline queries”. In: *Sixth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT’05)*. IEEE. 2005, pp. 907–909 *Cited on page 36.*

[99] Xuemin Lin et al. “Stabbing the sky: Efficient skyline computation over sliding windows”. In: *21st International Conference on Data Engineering (ICDE’05)*. IEEE. 2005, pp. 502–513 *Cited on page 36.*

[100] Yufei Tao and Dimitris Papadias. “Maintaining sliding window skylines on data streams”. In: *IEEE Transactions on Knowledge and Data Engineering* 18.3 (2006), pp. 377–391 *Cited on pages 36, 59, 62.*

[101] Michael Morse, Jignesh M Patel, and William I Grosky. “Efficient continuous skyline computation”. In: *Information Sciences* 177.17 (2007), pp. 3411–3437 *Cited on pages 36, 59.*

[102] Zhiyong Huang et al. “Continuous skyline queries for moving objects”. In: *IEEE transactions on knowledge and data engineering* 18.12 (2006), pp. 1645–1658 *Cited on pages 36, 80.*

[103] Jongwuk Lee and Seung-Won Hwang. “Scalable skyline computation using a balanced pivot selection technique”. In: *Information Systems* 39 (2014), pp. 1–21 *Cited on page 36.*

[104] Sean Chester, Michael L Mortensen, and Ira Assent. “On the Suitability of Skyline Queries for Data Exploration.” In: *EDBT/ICDT Workshops*. 2014, pp. 161–166 *Cited on page 37.*

[105] Mei Bai et al. “Skyline-join query processing in distributed databases”. In: *Frontiers of Computer Science* 10.2 (2016), pp. 330–352 *Cited on pages 37, 59.*

[106] Yu-Ling Hsueh, Chia-Chun Lin, and Chia-Che Chang. “An efficient indexing method for skyline computations with partially ordered domains”. In: *IEEE Transactions on Knowledge and Data Engineering* 29.5 (2017), pp. 963–976 *Cited on page 37.*

[107] Yan Wang et al. “An energy-efficient skyline query for massively multidimensional sensing data”. In: *Sensors* 16.1 (2016), p. 83 *Cited on page 37.*

[108] Mei Bai et al. “Discovering the k representative skyline over a sliding window”. In: *IEEE Transactions on Knowledge and Data Engineering* 28.8 (2016), pp. 2041–2056 *Cited on pages 38, 59.*

[109] He Li and Jaesoo Yoo. “Efficient continuous skyline query processing scheme over large dynamic data sets”. In: *ETRI Journal* 38.6 (2016), pp. 1197–1206 *Cited on page 38.*

[110] Alexander Tzanakas, Eleftherios Tiakas, and Yannis Manolopoulos. “Skyline algorithms on streams of multidimensional data”. In: *East European Conference on Advances in Databases and Information Systems*. Springer. 2016, pp. 63–71 *Cited on pages 38, 59.*

[111] Mei Bai et al. “The subspace global skyline query processing over dynamic databases”. In: *World Wide Web* 20.2 (2017), pp. 291–324 *Cited on page 38.*

[112] Aziz Nasridinov, Jong-Hyeok Choi, and Young-Ho Park. “A two-phase data space partitioning for efficient skyline computation”. In: *Cluster Computing* 20 (2017), pp. 3617–3628 *Cited on page 38.*

[113] Jiping Zheng, Jialiang Chen, and Haixiang Wang. “Efficient geometric pruning strategies for continuous skyline queries”. In: *ISPRS International Journal of Geo-Information* 6.3 (2017), p. 91 *Cited on page 38.*

[114] Boseon Yu, Wonik Choi, and Ling Liu. "Exploring correlation for fast skyline computation". In: *The Journal of Supercomputing* 73.11 (2017), pp. 5071–5102 *Cited on page 38.*

[115] Yuan-Ko Huang. "Within skyline query processing in dynamic road networks". In: *ISPRS International Journal of Geo-Information* 6.5 (2017), p. 137 *Cited on page 38.*

[116] Kenichi Koizumi et al. "BJR-tree: fast skyline computation algorithm using dominance relation-based tree structure". In: *International Journal of Data Science and Analytics* 7 (2019), pp. 17–34 *Cited on pages 39, 59.*

[117] Xixian Han, Bailing Wang, and Guojun Lai. "Dynamic skyline computation on massive data". In: *Knowledge and Information Systems* 59.3 (2019), pp. 571–599 *Cited on pages 39, 67.*

[118] Yuan-Ko Huang, Chien-Pang Lee, and Cheng-Yuan Tsai. "Evaluating KNN-skyline queries in dynamic road networks". In: *2018 27th Wireless and Optical Communication Conference (WOCC)*. IEEE. 2018, pp. 1–2 *Cited on page 40.*

[119] Yuan-Ko Huang, Chia-Heng Chang, and Chiang Lee. "Continuous distance-based skyline queries in road networks". In: *Information Systems* 37.7 (2012), pp. 611–633 *Cited on page 40.*

[120] Yingfeng Tang and Shiping Chen. "Supporting Continuous Skyline Queries in Dynamically Weighted Road Networks". In: *Mathematical Problems in Engineering* 2018 (2018) *Cited on pages 40, 59.*

[121] Ping Wu et al. "Parallelizing skyline queries for scalable distribution". In: *Advances in Database Technology-EDBT 2006: 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006 10*. Springer. 2006, pp. 112–130 *Cited on page 40.*

[122] Shiyuan Wang et al. "Efficient skyline query processing on peer-to-peer networks". In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2006, pp. 1126–1135 *Cited on page 40.*

[123] Shiyuan Wang et al. "Skyframe: a framework for skyline query processing in peer-to-peer systems". In: *The VLDB Journal* 18 (2009), pp. 345–362 *Cited on page 40.*

[124] Katja Hose. "Processing skyline queries in P2P systems". In: *VLDB 2005 PhD Workshop*. Citeseer. 2005, pp. 36–40 *Cited on page 40.*

[125] Huajing Li, Qingzhao Tan, and Wang-Chien Lee. "Efficient progressive processing of skyline queries in peer-to-peer systems". In: *Proceedings of the 1st international conference on Scalable information systems*. 2006, 26–es *Cited on page 40.*

[126] Zhiyong Huang et al. "Skyline queries against mobile lightweight devices in MANETs". In: *22nd International Conference on Data Engineering (ICDE'06)*. IEEE. 2006, pp. 66–66 *Cited on page 41.*

[127] Jianzhong Li, Shuguang Xiong, et al. "Efficient Pr-skyline query processing and optimization in wireless sensor networks". In: *Wireless Sensor Network* 2.11 (2010), p. 838 *Cited on page 41.*

[128] 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*. 2008, pp. 227–238 *Cited on page 41.*

[129] Renato B Hoffmann et al. "OpenMP as runtime for providing high-level stream parallelism on multi-cores". In: *The Journal of Supercomputing* (2022), pp. 1–22 *Cited on page 42.*

[130] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997 *Cited on page 42.*

[131] William Gropp et al. "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel computing* 22.6 (1996), pp. 789–828 *Cited on page 43.*

[132] Bruno Bacci et al. "P3 L: A structured high-level parallel language, and its structured support". In: *Concurrency: practice and experience* 7.3 (1995), pp. 225–255 *Cited on page 43.*

[133] Marco Vanneschi. "The programming model of ASSIST, an environment for parallel and distributed portable applications". In: *Parallel computing* 28.12 (2002), pp. 1709–1732 *Cited on page 43.*

[134] Marco Aldinucci, Marco Danelutto, and Paolo Teti. "An advanced environment supporting structured parallel programming in Java". In: *Future Generation Computer Systems* 19.5 (2003), pp. 611–626 *Cited on page 43.*

[135] Marco Vanneschi. "High performance computing: parallel processing models and architectures". In: *High performance computing*. Pisa University Press. 2014, pp. 1–552 *Cited on pages 43, 44.*

[136] Marco Danelutto. "Efficient support for skeletons on workstation clusters". In: *Parallel Processing Letters* 11.01 (2001), pp. 41–56 *Cited on page 43.*

[137] Marco Aldinucci et al. "Fastflow: High-Level and Efficient Streaming on Multicore". In: *Programming multi-core and many-core computing systems* (2017), pp. 261–280 *Cited on page 43.*

[138] Omer F Rana and Jose Cardoso Cunha. *Grid computing: software environments and tools*. Springer Science & Business Media, 2007 *Cited on page 43.*

[139] Athanasios V Vasilakos et al. *Autonomic Computing and Communications Systems: Third International ICST Conference, Autonomics 2009, Limassol, Cyprus, September 9-11, 2009, Revised Selected Papers*. Vol. 23. Springer, 2010 *Cited on page 43.*

[140] Massimo Coppola and Marco Vanneschi. "High-performance data mining with skeleton-based structured parallel programming". In: *Parallel Computing* 28.5 (2002), pp. 793–813 *Cited on page 43.*

[141] Daniele Buono et al. “Performance analysis and structured parallelisation of the space–time adaptive processing computational kernel on multi-core architectures”. In: *International Journal of Parallel, Emergent and Distributed Systems* 29.5 (2014), pp. 460–498 *Cited on page 43.*

[142] Marco Aldinucci et al. “Parallel stochastic systems biology in the cloud”. In: *Briefings in Bioinformatics* 15.5 (2014), pp. 798–813 *Cited on page 43.*

[143] Yunjun Gao et al. “On processing reverse k-skyband and ranked reverse skyline queries”. In: *Information Sciences* 293 (2015), pp. 11–34 *Cited on pages 43, 83.*

[144] Ehsan Montahaie et al. “Efficient continuous skyline computation on multi-core processors based on manhattan distance”. In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE. 2015, pp. 56–59 *Cited on pages 43, 61, 114.*

[145] Kenichi Koizumi, Mary Inaba, and Kei Hiraki. “Efficient implementation of continuous skyline computation on a multi-core processor”. In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE. 2015, pp. 52–55 *Cited on pages 43, 61, 111.*

[146] Md Saiful Islam et al. “Q+ tree: An efficient quad tree based data indexing for parallelizing dynamic and reverse skylines”. In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 2016, pp. 1291–1300 *Cited on page 44.*

[147] Haoyang Zhu et al. “Parallelization of skyline probability computation over uncertain preferences”. In: *Concurrency and Computation: Practice and Experience* 29.18 (2017), e4201 *Cited on page 44.*

[148] Arun K Pujari et al. “Efficient computation for probabilistic skyline over uncertain preferences”. In: *Information Sciences* 324 (2015), pp. 146–162 *Cited on page 44.*

[149] Karim Alami et al. “The negative skycube”. In: *Information Systems* 88 (2020), p. 101443 *Cited on page 44.*

[150] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989 *Cited on pages 44, 54.*

[151] Susanna Pelagatti. “A methodology for the development and the support of massively parallel programs”. In: *BULLETIN-EUROPEAN ASSOCIATION FOR THEORETICAL COMPUTER SCIENCE* 50 (1993), pp. 540–540 *Cited on page 44.*

[152] S Orlando et al. “P3L: a Structured High-level Parallel Language and its Structured Support””. In: *Concurrency: Practice and Experience* 7.3 (1995), pp. 225–255 *Cited on page 44.*

[153] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified data processing on large clusters”. In: (2004) *Cited on page 48.*

[154] *Apache Hadoop*. https://hadoop.apache.org/ *Cited on page 48.*

[155] Boliang Zhang, Shuigeng Zhou, and Jihong Guan. “Adapting skyline computation to the mapreduce framework: Algorithms and experiments”. In: *Database Systems for Adanced Applications: 16th International Conference, DASFAA 2011, International Workshops: GDB, SIM3, FlashDB, SNSMW, DaMEN, DQIS, Hong Kong, China, April 22-25, 2011. Proceedings 16*. Springer. 2011, pp. 403–414 *Cited on page 48.*

[156] Yoonjae Park, Jun-Ki Min, and Kyuseok Shim. "Parallel computation of skyline and reverse skyline queries using mapreduce". In: *Proceedings of the VLDB Endowment* 6.14 (2013), pp. 2002–2013 *Cited on pages 48, 49.*

[157] Kasper Mullesgaard et al. "Efficient Skyline Computation in MapReduce." In: *EDBT*. 2014, pp. 37–48 *Cited on page 49.*

[158] Ji Zhang et al. "Efficient parallel skyline evaluation using MapReduce". In: *IEEE Transactions on Parallel and Distributed Systems* 27.7 (2015), pp. 1996–2009 *Cited on page 49.*

[159] Heri Wijayanto et al. "LShape Partitioning: Parallel Skyline Query Processing Using MapReduce". In: *IEEE Transactions on Knowledge and Data Engineering* 34.7 (2020), pp. 3363–3376 *Cited on page 49.*

[160] Yuanyuan Li et al. "Efficient subspace skyline query based on user preference using MapReduce". In: *Ad Hoc Networks* 35 (2015), pp. 105–115 *Cited on page 49.*

[161] Yoonjae Park, Jun-Ki Min, and Kyuseok Shim. "Efficient processing of skyline queries using mapreduce". In: *IEEE Transactions on Knowledge and Data Engineering* 29.5 (2017), pp. 1031–1044 *Cited on page 49.*

[162] Jia-Ling Koh et al. "MapReduce skyline query processing with partitioning and distributed dominance tests". In: *Information Sciences* 375 (2017), pp. 114–137 *Cited on page 49.*

[163] Miyoung Jang, Youngho Song, and Jae-Woo Chang. "A parallel computation of skyline using multiple regression analysis-based filtering on MapReduce". In: *Distributed and Parallel Databases* 35 (2017), pp. 383–409 *Cited on page 51.*

[164] Junsu Kim and Myoung Ho Kim. "An efficient parallel processing method for skyline queries in MapReduce". In: *The Journal of Supercomputing* 74 (2018), pp. 886–935 *Cited on page 51.*

[165] Wenlu Wang et al. "Efficient parallel spatial skyline evaluation using mapreduce". In: *Proceedings of the 20th international conference on extending database technology*. 2017 *Cited on pages 51, 73, 75.*

[166] Chen Li, Asif Zaman, Yasuhiko Morimoto, et al. "MapReduce-based computation of area skyline query for selecting good locations in a map". In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 4779–4782 *Cited on page 51.*

[167] Hyeong-Cheol Ryu and Sungwon Jung. "MapReduce-based skyline query processing scheme using adaptive two-level grids". In: *Cluster Computing* 20 (2017), pp. 3605–3616 *Cited on page 51.*

[168] Tyson Condie et al. "MapReduce online." In: *Nsdi*. Vol. 10. 4. 2010, p. 20 *Cited on page 52.*

[169] Andrey Brito et al. "Scalable and low-latency data processing with stream mapreduce". In: *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE. 2011, pp. 48–58 *Cited on page 52.*

[170] Ahmed M Aly et al. "M3: Stream processing on main-memory mapreduce". In: *2012 IEEE 28th International Conference on Data Engineering*. IEEE. 2012, pp. 1253–1256 *Cited on page 52.*

[171] Lei Chen and Xiang Lian. "Dynamic skyline queries in metric spaces". In: *Proceedings of the 11th international conference on extending database technology: advances in database technology*. 2008, pp. 333–343 *Cited on pages 59, 74.*

[172] Zhenjie Zhang et al. "Minimizing the communication cost for continuous skyline maintenance". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 2009, pp. 495–508 *Cited on page 59.*

[173] Wang Hanning et al. "Efficient processing of continuous skyline query over smarter traffic data stream for cloud computing". In: *Discrete Dynamics in Nature and Society* 2013 (2013) *Cited on page 59.*

[174] Aziguli Wulamu et al. "Processing Skyline Groups on Data Streams". In: *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. IEEE. 2015, pp. 935–942 *Cited on page 59.*

[175] Tao Jiang et al. "Efficient column-oriented processing for mutual subspace skyline queries". In: *Soft Computing* 24.20 (2020), pp. 15427–15445 *Cited on page 59.*

[176] Zhiyun Zheng et al. "k-dominant Skyline query algorithm for dynamic datasets". In: *Frontiers of Computer Science* 15 (2021), pp. 1–9 *Cited on page 59.*

[177] Mikhail J Atallah and Yinian Qi. "Computing all skyline probabilities for uncertain data". In: *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2009, pp. 279–287 *Cited on page 60.*

[178] Wenjie Zhang et al. "Probabilistic skyline operator over sliding windows". In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE. 2009, pp. 1060–1071 *Cited on page 60.*

[179] Ling Zhu, Cuiping Li, and Hong Chen. "Efficient computation of reverse skyline on data stream". In: *2009 International Joint Conference on Computational Sciences and Optimization*. Vol. 1. IEEE. 2009, pp. 735–739 *Cited on pages 60, 83.*

[180] Hui Zhu Su, En Tzu Wang, and Arbee LP Chen. "Continuous probabilistic skyline queries over uncertain data streams". In: *International Conference on Database and Expert Systems Applications*. Springer. 2010, pp. 105–121 *Cited on page 60.*

[181] Mohammad Shamsul Arefin and Yasuhiko Morimoto. "Skyline sets queries for incomplete data". In: *AIRCC's International Journal of Computer Science and Information Technology* 4.5 (2016), pp. 67–80 *Cited on page 60.*

[182] Rahul Bharuka and P Sreenivasa Kumar. "Finding skylines for incomplete data". In: *Proceedings of the Twenty-Fourth Australasian Database Conference-Volume 137*. 2013, pp. 109–117 *Cited on page 60.*

[183] Yan Wang et al. "Skyline preference query based on massive and incomplete dataset". In: *IEEE Access* 5 (2017), pp. 3183–3192 *Cited on page 60.*

[184] Xiaoye Miao et al. "Top-k dominating queries on incomplete data". In: *IEEE Transactions on Knowledge and Data Engineering* 28.1 (2015), pp. 252–266 *Cited on page 60.*

[185] Wenjie Zhang et al. “Probabilistic n-of-N skyline computation over uncertain data streams”. In: *World Wide Web* 18.5 (2015), pp. 1331–1350 *Cited on page 60.*

[186] Xiaoye Miao et al. “K-dominant skyline queries on incomplete data”. In: *Information Sciences* 367 (2016), pp. 990–1011 *Cited on page 60.*

[187] Md Saiful Islam et al. “Computing influence of a product through uncertain reverse skyline”. In: *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. 2017, pp. 1–12 *Cited on pages 60, 61.*

[188] Sayda Elmi and Jun-Ki Min. “Spatial skyline queries over incomplete data for smart cities”. In: *Journal of Systems Architecture* 90 (2018), pp. 1–14 *Cited on pages 60, 74.*

[189] Chuang-Ming Liu, Denis Pak, and Ari Ernesto Ortiz Castellanos. “Priority-Based Skyline Query Processing for Incomplete Data”. In: *Proceedings of the 25th International Database Engineering & Applications Symposium*. 2021, pp. 204–211 *Cited on page 60.*

[190] Linlin Ding et al. “CrowdSJ: Skyline-join query processing of incomplete datasets with crowdsourcing”. In: *IEEE Access* 9 (2021), pp. 73216–73229 *Cited on page 60.*

[191] Xiaoyong Li et al. “Parallelizing probabilistic streaming skyline operator in cloud computing environments”. In: *2013 IEEE 37th Annual Computer Software and Applications Conference*. IEEE. 2013, pp. 84–89 *Cited on page 61.*

[192] Md Saiful Islam et al. “Q+ tree: An efficient quad tree based data indexing for parallelizing dynamic and reverse skylines”. In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 2016, pp. 1291–1300 *Cited on page 61.*

[193] Haoyang Zhu et al. “Parallelization of skyline probability computation over uncertain preferences”. In: *Concurrency and Computation: Practice and Experience* 29.18 (2017), e4201 *Cited on page 61.*

[194] Jun Liu et al. “Parallel n-of-N skyline queries over uncertain data streams”. In: *International Conference on Database and Expert Systems Applications*. Springer. 2018, pp. 176–184 *Cited on page 61.*

[195] Jun Liu et al. “Parallelizing uncertain skyline computation against n-of-N data streaming model”. In: *Concurrency and Computation: Practice and Experience* 31.4 (2019), e4848 *Cited on page 61.*

[196] Xiaoyong Li et al. “Parallel k-dominant skyline queries over uncertain data streams with capability index”. In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE. 2019, pp. 1556–1563 *Cited on page 61.*

[197] Jongtae Lim, Kyoungsoo Bok, and Jaesoo Yoo. “A continuous reverse skyline query processing scheme for multimedia data sharing in mobile environments”. In: *Multimedia Tools and Applications* 78.20 (2019), pp. 28357–28373 *Cited on pages 61, 83.*

[198] Karim Alami and Sofian Maabout. “A Partitioning Approach for Skyline Queries in Presence of Partial and Dynamic Orders”. In: *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLIX*. Springer, 2021, pp. 70–96 *Cited on page 61.*

[199] Liang Kuang Tai, En Tzu Wang, and Arbee LP Chen. "Finding the most profitable candidate product by dynamic skyline and parallel processing". In: *Distributed and Parallel Databases* 39 (2021), pp. 979–1008 *Cited on page 61.*

[200] Jon Louis Bentley et al. "On the average number of maxima in a set of vectors and applications". In: *Journal of the ACM (JACM)* 25.4 (1978), pp. 536–543 *Cited on page 62.*

[201] Christian Buchta. "On the average number of maxima in a set of vectors". In: *Information Processing Letters* 33.2 (1989), pp. 63–65 *Cited on page 62.*

[202] Parke Godfrey. "Skyline cardinality for relational processing: how many vectors are maximal?" In: *Foundations of Information and Knowledge Systems: Third International Symposium, FoIKS 2004 Wilheminenburg Castle, Austria, February 17-20, 2004 Proceedings 3*. Springer. 2004, pp. 78–97 *Cited on page 62.*

[203] Eleftherios Tiakas, Apostolos N Papadopoulos, and Yannis Manolopoulos. "On estimating the maximum domination value and the skyline cardinality of multi-dimensional data sets". In: *International Journal of Knowledge-Based Organizations (IJKBO)* 3.4 (2013), pp. 61–83 *Cited on page 62.*

[204] George Valkanas, Apostolos N Papadopoulos, and Dimitrios Gunopulos. "Skyline Ranking à la IR." In: *EDBT/ICDT Workshops*. 2014, pp. 182–187 *Cited on page 62.*

[205] Tian Xia et al. "Online subspace skyline query processing using the compressed skycube". In: *ACM Transactions on Database Systems (TODS)* 37.2 (2012), pp. 1–36 *Cited on page 62.*

[206] Yufei Tao. "Diversity in Skylines." In: *IEEE Data Eng. Bull.* 32.4 (2009), pp. 65–72 *Cited on page 63.*

[207] George Valkanas, Apostolos N Papadopoulos, and Dimitrios Gunopulos. "Skydiver: a framework for skyline diversification". In: *Proceedings of the 16th International Conference on Extending Database Technology*. 2013, pp. 406–417 *Cited on page 63.*

[208] Dimitris Sacharidis, Panagiotis Bouros, and Timos Sellis. "Caching dynamic skyline queries". In: *Scientific and Statistical Database Management: 20th International Conference, SSDBM 2008, Hong Kong, China, July 9-11, 2008 Proceedings 20*. Springer. 2008, pp. 455–472 *Cited on page 67.*

[209] Weiguo Wang et al. "An efficient secure dynamic skyline query model". In: *arXiv preprint arXiv:2002.07511* (2020) *Cited on page 67.*

[210] Xi Guo et al. "Efficient processing of skyline group queries over a data stream". In: *Tsinghua Science and Technology* 21.1 (2016), pp. 29–39 *Cited on page 68.*

[211] Leigang Dong et al. "G-skyline query over data stream in wireless sensor network". In: *Wireless Networks* 26 (2020), pp. 129–144 *Cited on page 68.*

[212] Mehdi Sharifzadeh and Cyrus Shahabi. "The spatial skyline queries". In: *Proceedings of the 32nd international conference on Very large data bases*. 2006, pp. 751–762 *Cited on pages 69, 73.*

[213] Wanbin Son et al. "Spatial skyline queries: An efficient geometric algorithm". In: *Advances in Spatial and Temporal Databases: 11th International Symposium, SSTD 2009 Aalborg, Denmark, July 8-10, 2009 Proceedings 11*. Springer. 2009, pp. 247–264 *Cited on page 69*.

[214] Wanbin Son, Seung-Won Hwang, and Hee-Kap Ahn. "MSSQ: Manhattan spatial skyline queries". In: *Information Systems* 40 (2014), pp. 67–83 *Cited on page 69*.

[215] Ke Deng, Xiaofang Zhou, and Heng Tao Shen. "Multi-source skyline query processing in road networks". In: *2007 IEEE 23rd international conference on data engineering*. IEEE. 2006, pp. 796–805 *Cited on pages 70, 73*.

[216] Maytham Safar, Dalal El-Amin, and David Taniar. "Optimized skyline queries on road networks using nearest neighbors". In: *Personal and Ubiquitous Computing* 15 (2011), pp. 845–856 *Cited on page 70*.

[217] Gae-won You et al. "The farthest spatial skyline queries". In: *Information Systems* 38.3 (2013), pp. 286–301 *Cited on pages 70, 74*.

[218] Marta Fort, J Antoni Sellarès, and Nacho Valladares. "Nearest and farthest spatial skyline queries under multiplicative weighted Euclidean distances". In: *Knowledge-Based Systems* 192 (2020), p. 105299 *Cited on pages 71, 74*.

[219] Kazuki Kodama et al. "Skyline queries based on user locations and preferences for making location-based recommendations". In: *Proceedings of the 2009 International Workshop on Location Based Social Networks*. 2009, pp. 9–16 *Cited on page 71*.

[220] Jieming Shi, Dingming Wu, and Nikos Mamoulis. "Textually relevant spatial skylines". In: *IEEE Transactions on Knowledge and Data Engineering* 28.1 (2015), pp. 224–237 *Cited on page 71*.

[221] Xi Guo, Yoshiharu Ishikawa, and Yunjun Gao. "Direction-based spatial skylines". In: *Proceedings of the Ninth ACM International Workshop on Data Engineering for Wireless and Mobile Access*. 2010, pp. 73–80 *Cited on page 72*.

[222] Bojie Shen et al. "Direction-based spatial skyline for retrieving surrounding objects". In: *World Wide Web* 23 (2020), pp. 207–239 *Cited on pages 72, 73*.

[223] Yuan-Ko Huang. "Continuous dε-Skyline Queries for Objects with Time-Varying Attribute in Road Networks". In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. IEEE. 2017, pp. 439–446 *Cited on page 74*.

[224] Ammar Sohail, Muhammad Aamir Cheema, and David Taniar. "Social-aware spatial top-k and skyline queries". In: *The Computer Journal* 61.11 (2018), pp. 1620–1638 *Cited on page 74*.

[225] Zhi Cai et al. "Speed and Direction Aware Skyline Query for Moving Objects". In: *IEEE Transactions on Intelligent Transportation Systems* 23.4 (2020), pp. 3000–3011 *Cited on page 74*.

[226] Md Anisuzzaman Siddique et al. "Multicore Based Spatialk-dominant Skyline Computation". In: *2012 Third International Conference on Networking and Computing*. IEEE. 2012, pp. 188–194 *Cited on page 75*.

[227] Wenlu Wang et al. “A scalable spatial skyline evaluation system utilizing parallel independent region groups”. In: *The VLDB Journal* 28 (2019), pp. 73–98 *Cited on page 75.*

[228] Lei Chen and Xiang Lian. “Efficient processing of metric skyline queries”. In: *IEEE Transactions on Knowledge and Data Engineering* 21.3 (2008), pp. 351–365 *Cited on pages 76, 77.*

[229] Daniel P Huttenlocher, Gregory A. Klanderman, and William J Rucklidge. “Comparing images using the Hausdorff distance”. In: *IEEE Transactions on pattern analysis and machine intelligence* 15.9 (1993), pp. 850–863 *Cited on page 76.*

[230] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. “Duplicate record detection: A survey”. In: *IEEE Transactions on knowledge and data engineering* 19.1 (2006), pp. 1–16 *Cited on page 76.*

[231] Athanasios Kokkos, Theodoros Tzouramanis, and Yannis Manolopoulos. “A hybrid model for linking multiple social identities across heterogeneous online social networks”. In: *SOFSEM 2017: Theory and Practice of Computer Science: 43rd International Conference on Current Trends in Theory and Practice of Computer Science, Limerick, Ireland, January 16-20, 2017, Proceedings 43*. Springer. 2017, pp. 423–435 *Cited on page 76.*

[232] Tomás Skopal. “Pivoting M-tree: A Metric Access Method for Efficient Similarity Search.” In: *DATESO*. Vol. 4. 2004, pp. 27–37 *Cited on page 77.*

[233] Tomás Skopal and Jakub Lokoc. “Answering Metric Skyline Queries by PM-tree.” In: *DATESO*. Citeseer. 2010, pp. 22–37 *Cited on page 77.*

[234] David Fuhry, Ruoming Jin, and Donghui Zhang. “Efficient skyline computation in metric space”. In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. 2009, pp. 1042–1051 *Cited on page 77.*

[235] Tao Jiang et al. “Incremental evaluation of top-k combinatorial metric skyline query”. In: *Knowledge-Based Systems* 74 (2015), pp. 89–105 *Cited on page 77.*

[236] Saladi Rahul and Ravi Janardan. “Algorithms for range-skyline queries”. In: *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*. 2012, pp. 526–529 *Cited on pages 79, 80.*

[237] Lijiang Chen, Bin Cui, and Hua Lu. “Constrained skyline query processing against distributed data sites”. In: *IEEE Transactions on Knowledge and Data Engineering* 23.2 (2010), pp. 204–217 *Cited on page 80.*

[238] Chuan-Chi Lai, Zulhaydar Fairozal Akbar, and Chuan-Ming Liu. “A cooperative method for processing range-skyline queries in mobile wireless sensor networks”. In: *Proceedings of the Sixth International Conference on Emerging Databases: Technologies, Applications, and Theory*. 2016, pp. 1–8 *Cited on page 80.*

[239] Chuan-Chi Lai et al. “Distributed continuous range-skyline query monitoring over the internet of mobile things”. In: *IEEE Internet of Things Journal* 6.4 (2019), pp. 6652–6667 *Cited on page 80.*

[240] Xin Lin, Jianliang Xu, and Haibo Hu. "Range-based skyline queries in mobile environments". In: *IEEE Transactions on Knowledge and Data Engineering* 25.4 (2011), pp. 835–849 *Cited on pages 80, 81.*

[241] Xiaoyi Fu et al. "Continuous range-based skyline queries in road networks". In: *World Wide Web* 20 (2017), pp. 1443–1467 *Cited on page 81.*

[242] Wen-Chi Wang, En Tzu Wang, and Arbee LP Chen. "Dynamic skylines considering range queries". In: *Database Systems for Advanced Applications: 16th International Conference, DASFAA 2011, Hong Kong, China, April 22-25, 2011, Proceedings, Part II 16*. Springer. 2011, pp. 235–250 *Cited on page 82.*

[243] Jan Chomicki et al. "Skyline with presorting: Theory and optimizations". In: *Intelligent Information Processing and Web Mining: Proceedings of the International IIS: IIPWM'05 Conference held in Gdansk, Poland, June 13–16, 2005*. Springer. 2005, pp. 595–604 *Cited on page 82.*

[244] Theodoros Tzouramanis et al. "The range skyline query". In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 2018, pp. 47–56 *Cited on page 82.*

[245] Xiaoye Miao et al. "On efficiently answering why-not range-based skyline queries in road networks". In: *IEEE Transactions on Knowledge and Data Engineering* 30.9 (2018), pp. 1697–1711 *Cited on page 82.*

[246] Xiang Lian and Lei Chen. "Reverse skyline search in uncertain databases". In: *ACM Transactions on Database Systems (TODS)* 35.1 (2008), pp. 1–49 *Cited on page 82.*

[247] Xiang Lian and Lei Chen. "Monochromatic and bichromatic reverse skyline search over uncertain databases". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 213–226 *Cited on page 82.*

[248] Guoren Wang et al. "Energy-efficient reverse skyline query processing over wireless sensor networks". In: *IEEE Transactions on Knowledge and Data Engineering* 24.7 (2011), pp. 1259–1275 *Cited on page 83.*

[249] Farnoush Banaei-Kashani et al. "Efficient maximal reverse skyline query processing". In: *GeoInformatica* 21 (2017), pp. 549–572 *Cited on page 83.*

[250] Hongbing Wang et al. "Integrating reinforcement learning and skyline computing for adaptive service composition". In: *Information Sciences* 519 (2020), pp. 141–160 *Cited on page 84.*

[251] Abdelkader Alem, Youcef Dahmani, and Bendaoud Mebarek. "Skyline computation for improving naïve Bayesian classifier in intrusion detection system". In: *Journal homepage: http://iieta. org/journals/isi* 24.5 (2019), pp. 513–518 *Cited on page 84.*

[252] Wajdi Dhifli, Nour El Islem Karabadji, and Mohamed Elati. "Evolutionary mining of skyline clusters of attributed graph data". In: *Information Sciences* 509 (2020), pp. 501–514 *Cited on page 86.*

[253] Nikolaos Georgiadis et al. "Skyline-based dissimilarity of images". In: *Journal of Intelligent Information Systems* 53 (2019), pp. 509–545 *Cited on page 86.*

[254] Shiqing Wang et al. "Surveillance Methods of Running Condition of Chemical Equipments Based on Skyline". In: *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE. 2016, pp. 2246–2250 *Cited on page 87.*

[255] Fadoua Yakine, Manar Abourezq, and Abdellah Idrissi. "Skyline method in wireless adhoc networks routing". In: *WSEAS Transactions on Communications* 15 (2016), pp. 137–146 *Cited on page 87.*

[256] Chaluka Salgado. "Keyword-aware skyline routes search in indoor venues". In: *Proceedings of the 9th ACM SIGSPATIAL International Workshop on Indoor Spatial Awareness*. 2018, pp. 25–31 *Cited on pages 88, 89.*

[257] Kenneth S Bøgh et al. "Template skycube algorithms for heterogeneous parallelism on multicore and gpu architectures". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 447–462 *Cited on page 111.*