

Imperial College London
Department of Computing

Enhancing Stream Data Processing: System Optimizations and Learned Indexes

Liang Liang

Submitted in part fulfillment of the requirements for the degree of
Doctor of Philosophy in Computing from Imperial College London and
the Diploma of Imperial College, January 2025

Statement of Originality

This thesis comprises the work conducted from 2020 to 2024 in the Department of Computing at Imperial College London. I declare that the work presented in this thesis is my own; all joint work is documented in Section 1.4.

Copyright

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Abstract

This thesis aims to optimize stream data processing, which is important for real-time data analysis and decision-making. Stream data’s inherent properties, including unbounded size, high volume, and variable velocities, impose significant challenges on processing systems. These systems must continuously evolve to meet the requirements of modern stream data processing. This thesis presents optimizations to enhance the functionality, scalability, and performance of stream data processing at both the system and algorithmic levels.

At the system level, we focus on a stream system, dispel4py, designed for scientific workload computation. We enhance the scalability and state management of dispel4py by developing *dynamic allocation*, *dynamic auto-scaling* and *hybrid* optimizations. Specifically, *dynamic allocation* allows dispel4py to scale for each task depending on the workload demands, and *dynamic auto-scaling* enables the entire workload to scale with fewer or more resources to maintain the performance while achieving cost efficiency. Furthermore, *hybrid* enables dispel4py to support stateful tasks and scaling simultaneously. Comprehensive experiments validate the scalability, portability, and performance of these three optimizations.

At the algorithm level, our focus shifts to Index-Based Window Processing (IBWP). Recently, learned indexes integrating machine learning models to enhance query performance present a promising alternative to traditional index structures. Motivated by this trend, we explore how learned indexes can effectively support search while maintaining updates for high-velocity data streams. However, the challenge lies in the inherent limitations of current updatable learned indexes. These limitations are often inherited from their traditional tree-based structures, which are cumbersome and impede update performance. To overcome these limitations, we pioneered the use of innovative queue-style flat structures, which significantly enhance update efficiency and reduce the index footprint. Based on the flat structures, we propose FLIRT and SWIX, designed for sequential IBWP and generic IBWP, respectively. Our experiments demonstrate that they effectively manage their respective IBWPs, outperforming all baselines.

Acknowledgements

I would like to use a metaphor that I often think of: *the whole world is just a stream window for all living people*. I want to first thank those who brought me into this “window,” my parents.

To better recall and describe events and experiences, we tend to subjectively segment our lives into different periods. Subjectively and idealistically, I will partition this vast “window” according to what I consider as important periods. The period from 2020 to 2024, during which I pursued my PhD, is especially significant for me, not only because we pay more attention to new experiences and data but, more importantly, because this period has significantly reformed me in many ways. I want to thank everyone I have met during this sub-window known as my PhD period, starting with the person who opened this sub-window for me, Prof. Thomas Heinis. I have learned many valuable qualities from him, and I thank him for guiding and inspiring me to reach the boundary of this sub-window. I am also grateful to our research group members, Guang Yang, Dr. Ali Hadian, and Luis Alberto Croquevielle. I especially want to thank Guang Yang, who is not only a dedicated collaborator in academia but also a trustworthy friend in life. Furthermore, I want to thank Dr. Rosa Filgueira, who has not only helped me academically but has also influenced my outlook on life and academia.

I extend my gratitude to my family and friends who have supported me throughout this PhD journey. Though not all names are listed, know that your impact is deeply appreciated.

Finally, I want to give special thanks to my fiancée Yan Yan for accompanying and tolerating me.

In this sub-window, I have accumulated enough wonderful memories to give me the courage to move on to the next sub-window.

Dedication

To my parents and my grandparents.

‘Panta rhei.’ (‘Everything flows.’)

— Heraclitus

Contents

Statement of Originality	i
Abstract	v
Acknowledgements	vii
1 Introduction	1
1.1 Motivation and Objectives	1
1.2 Research Questions and Contributions	3
1.3 Thesis Structure	6
1.4 Publications	7
2 Background	10
2.1 Overview	10
2.2 Stream Data	11
2.2.1 Characteristics of Stream Data	11
2.2.2 Requirements of Stream Data Processing	12
2.2.3 Stream Data Model	14
2.2.4 Stream Window Techniques	16

2.3	Stream System	18
2.3.1	Evolution of Stream System	18
2.3.2	State Management	22
2.3.3	Scaling Strategy	24
2.4	Index	26
2.4.1	Index Concepts	26
2.4.2	Traditional Index	27
2.4.3	Learned Index	31
2.4.4	Stream Index	35
3	Optimization of Stream-based Workflow System	38
3.1	Overview of dispel4py	38
3.2	Motivation	41
3.2.1	Motivation for dynamic allocation	41
3.2.2	Motivation for auto-scaling allocation	43
3.2.3	Motivation to restore the supporting stateful computing in dynamic dispel4py	45
3.3	Background	46
3.3.1	Redis	46
3.3.2	Related Work	47
3.4	Optimization towards Scalability and Statefulness	49
3.4.1	The <i>dynamic allocation</i> Optimization	49
3.4.2	The <i>dynamic auto-scaling</i> Optimization	52

3.4.3	The <i>hybrid</i> Optimization for Statefulness	55
3.5	Useases	57
3.5.1	Internal Extinction of Galaxies Workflow	57
3.5.2	Seismic Cross-Correlation Workflow	58
3.5.3	Sentiment Analyses for News Articles Workflow	58
3.6	Evaluation	59
3.6.1	Experiment Setup	59
3.6.2	Evaluation on the Internal Extinction of Galaxies workflow	62
3.6.3	Evaluation on the Seismic Cross-Correlation workflow	65
3.6.4	Evaluation on Sentiment Analyses for News Articles workflow	66
3.6.5	Analysis of <i>dynamic auto-scaling</i>	69
3.6.6	Key Insights	71
3.7	Conclusion	72
4	An Efficient Learned Index for Sequential Stream Processing	73
4.1	Motivation	73
4.2	Design Overview	75
4.3	FLIRT	78
4.3.1	Structures	78
4.3.2	Operations	82
4.4	Parallel FLIRTs	88
4.4.1	Partition-based FLIRT	89
4.4.2	Shared FLIRT	91

4.5	Evaluation	93
4.5.1	Experiment Setup	93
4.5.2	Self Analysis	97
4.5.3	Comparative Analysis	99
4.5.4	Parallel FLIRTs Analysis	104
4.5.5	Key Insights	107
4.6	Conclusion	107
5	A Versatile Learned Index for Stream Windows	108
5.1	Motivation	108
5.2	Design Overview	110
5.3	SWIX	112
5.3.1	Structure of SWIX	112
5.3.2	Search	114
5.3.3	Insertion	116
5.3.4	Deletion	118
5.3.5	Bulk-load	119
5.3.6	Retrain	121
5.3.7	Auto-tuning	122
5.4	Parallel SWIX (PSWIX)	123
5.4.1	Overview of PSWIX	123
5.4.2	Operations of PSWIX	124
5.5	Evaluation	126

5.5.1	Experiment Setup	127
5.5.2	Self Analysis	129
5.5.3	Comparative Analysis	131
5.5.4	Evaluation on Parallel Learned Index	137
5.5.5	Key Insights	139
5.6	Conclusion	140
6	Conclusion	141
6.1	Summary of Thesis Achievements	141
6.2	Future Work	143
6.2.1	System-level Future Work	143
6.2.2	Algorithm-level Future Work	144
	Bibliography	145

List of Tables

2.1	First and modern generation stream systems addressing key requirements. . . .	20
2.2	Baseline updatable learned indexes	34
3.1	Abbreviations of optimization techniques.	59
3.2	Performance comparison based on the Internal Extinction of Galaxies workflow between techniques (A and B). The <i>runtime</i> ratio is the <i>runtime</i> of mapping A over the <i>runtime</i> of mapping B. Similarly, the <i>process time</i> ratio is the total process time elapsed ratio between A and B. The ratios are prioritized by the metric in the "Prioritized By" column, and the [Mean, Std] shows the average and standard deviation for all <i>runtime</i> and <i>process time</i> ratios.	66
3.3	Performance comparison based Seismic Cross-Correlation workflow between <i>dynamic auto-scaling</i> techniques and their baselines.	68
3.4	Performance comparison based Sentiment Analyses for News Articles workflow between stateful techniques (<i>hybrid-redis</i> and <i>multi</i>).	69

List of Figures

1.1	Classification of research questions (into system and algorithmic levels) and how thesis contributions fit in.	3
1.2	Design trade-offs between current stream index and where the ideal stream index fits in.	6
2.1	Overview of the structure of the background chapter. Yellow highlights topics discussed in Section 2.2. Similarly, Red indicates topics in Section 2.3, and Green describes topics in Section 2.4	11
2.2	Theoretical stream model where data is indicated in green, deleted tuples are outlined in red, and inserted tuples are outlined in green.	14
2.3	Data-flow stream model, in which each tuple includes data in bold and <i>timestamp</i> in italics.	15
2.4	Demonstration of a continuous fixed-size window, specifically a count-based window with a size of 3.	16
2.5	Demonstration of a continuous variable-size window, where the time interval is set to 10 (time) units.	17
2.6	Illustration of late data for an event time window, where the data of 21 at t_7 arrives after t_8	17
2.7	Demonstration of scaling strategies in parallel stream systems.	25
2.8	Illustration of B+tree.	30

2.9	Demonstration of common learned index structure.	31
2.10	Comparison between the difference in search bound of algorithmic and learned models.	32
2.11	Illustration of the predict and correct process in a learned index.	32
2.12	Demonstration of an Index-based Window Processing (IBWP) workload.	36
3.1	dispel4py overview.	39
3.2	Example of the default workload allocation allocation using MPI with 12 processes.	40
3.3	Comparison of Workflow Distributions in Processing Elements (PEs): (a) Uniform Distribution and (b) Skewed Distribution . In both distributions, the first PE (PE_0) is assigned a consistently low workload and is executed only once, resulting in similar behavior across both figures. The last PE (PE_3) also maintains a relatively low and consistent workload in both distributions. The primary difference lies in PE_1 and PE_2	42
3.4	Total idle time for different workload types.	42
3.5	Efficiency against number of processes for different workload types.	44
3.6	The <i>dynamic allocation</i> optimization. Each process can freely access the global queue to update data and tasks.	49
3.7	The <i>dynamic auto-scaling</i> optimization. The auto-scaler can dynamically activate or deactivate processes based on the monitoring framework.	52
3.8	Hybrid optimization for stateful operation. Processes categorized into stateless and stateful manage corresponding PEs, optimizing workflow execution. Stateful processes maintain dedicated private queues.	56
3.9	Internal Extinction of Galaxies workflow.	57
3.10	Seismic Cross-Correlation workflow.	58
3.11	Sentiment Analyses for News Articles Workflow.	59

3.12	Workload Performance for Internal Extinction of Galaxies using the <i>server</i> with up to 16 processes.	63
3.13	Workload Performance for Internal Extinction of Galaxies using the <i>cloud</i> with up to 16 processes.	64
3.14	Workload Performance for Internal Extinction of Galaxies using the <i>HPC</i> with up to 64 processes.	65
3.15	Experiments on Seismic Cross-Correlation over different platforms.	67
3.16	Experiments on Sentiment Analyses for News Articles over different platforms. .	68
3.17	Active Size and the monitoring metrics (Queue Size and Avg Idle Time) running with Internal Extinction of Galaxies workflow and Seismic Cross-Correlation workflow.	70
4.1	Illustration of the sequential IBWP workload.	74
4.2	Trade-off between search and update in sequential IBWP for algorithmic and learned indexes.	74
4.3	FLIRT overview, illustrating the system’s hierarchical design. The Coordinator Level is the <i>CoordQ</i> . Below this, the Segment Level is comprised of multiple Linear Prediction Queues (<i>LPQ</i>), each equipped with a linear model $F_i(key)$. Key operations include ‘Enqueue’ for data input and ‘Dequeue’ for output. The ‘Auto-Tuner & Segmentation’ dynamically refines these models and segmentation to optimize system responsiveness to changes in data distribution.	76
4.4	Shrinking cone algorithm for segmentation.	79
4.5	Steps of the search operation in FLIRT.	82
4.6	Steps for enqueueing into FLIRT.	84
4.7	Implicit deletion during dequeuing of FLIRT. <i>Left</i> : Non-empty segment flagging expired keys. <i>Right</i> : Erasing the entire segment.	85
4.8	Overview of the structure of PPFLIRT.	89

4.9	Illustration of thread transition in PPFLIRT when the enqueue thread becomes full and the dequeue thread becomes empty.	90
4.10	Overview of the structure of PSFLIRT.	92
4.11	Distribution of datasets in the SOSD benchmark [93].	95
4.12	Evaluation of the effectiveness of FLIRT’s auto-tuner.	98
4.13	Overall performance of FLIRT.	99
4.14	Performance of indexes under varying window sizes.	101
4.15	Performance of indexes under varying number of executions.	102
4.16	Performance of indexes under varying scan ranges.	103
4.17	Performance of indexes under three sets of read-write ratios.	103
4.18	Performance of parallel search between parallel and serial FLIRTs.	105
4.19	Performance of parallel updates between parallel and serial FLIRTs.	106
4.20	Performance under various degrees of query skewness.	106
5.1	Demonstration of a generic IBWP workload in which keys are indexed for querying, and keys’ (validness) time-to-live in the index (window) depends on the non-indexed timestamp	109
5.2	Illustration of how different indexes support a generic IBWP workload, where non-stream methods require additional queries or secondary indexes. Note that IMTree and SWIX do not require these mechanisms; hence, they stay in the same location in both figures.	110
5.3	Illustration of the piecewise linear functions used in SWIX for segmenting keys.	111
5.4	SWIX Structure Overview: The top-level <i>DPmeta</i> coordinates the <i>DPsegs</i> , which manage data segments using linear models at the lower level. Both <i>DPmeta</i> and <i>DPseg</i> are constructed on the <i>DParr</i> . Additionally, <i>DPseg</i> integrates a buffer to enhance update performance.	114

5.5	Steps for searching a key of 700 in SWIX, where red indicates implicit delete operations.	115
5.6	Types of insertions after finding the insertion position in <i>DPseg</i>	116
5.7	<i>SD Algorithm</i> for bulk-loading during cold-starts.	120
5.8	Illustration of the rendezvous batch retrain process when the orange segment triggers a retrain.	121
5.9	Structure of parallel SWIX.	124
5.10	Synchronization between threads when <i>DPmeta</i> is updated.	125
5.11	Synchronization required when <i>DPmeta</i> triggers a retrain.	126
5.12	Key distribution of <i>lognormal</i> and <i>wiki</i> datasets from the SOSD benchmark[93].	127
5.13	Performance of the auto-tuner in terms of latency and memory usage compared to static values of <i>tol</i>	130
5.14	The changes in error with the number of updates when applying the auto-tuner.	130
5.15	Summary of the average memory overhead and latency for all indexes and datasets.	131
5.16	Speed vs size performance of indexes under varying window sizes.	132
5.17	Speed vs size performance of indexes under varying scan ranges.	133
5.18	Evaluation of indexes under varying degrees of skewness. The top row shows the performance in terms of latency. The bottom three rows display the memory usage under continuous updates.	134
5.19	Latency performance for three read-to-write ratios.	135
5.20	Performance of indexes under varying scan ranges in a sequential IBWP workload.	136
5.21	Performance of parallel indexes for three read-to-write ratios.	138
5.22	Parallel evaluation under a sequential IBWP workload.	139

Chapter 1

Introduction

1.1 Motivation and Objectives

Stream data processing has become a necessity, especially in the context of the Internet of Things (IOT). By 2025, an estimated 30% (52.5 zettabytes) of the global data sphere will consist of real-time data that requires immediate processing [132]. Stream data processing plays an important role in both industry and academic research [134]. In the industrial sections, applications, such as online market dynamics and real-time consumer behaviour analysis, rely on real-time stream processing of big data for timely and quick business decisions [7]. On the other hand, in academia, tasks such as environmental monitoring and geologic research require processing terabytes of stream data from sensors daily [79]. These growing demands necessitate the development of stream data processing technologies and require optimizations at the system and algorithm levels to enhance performance, scalability, and efficiency to accommodate large volumes of diverse data streams in different domains [76].

Stream processing systems have evolved over the last two decades [59]. The first-generation of stream systems was designed and implemented in the early 2000s; representative examples include prototypic systems such as NiagaraCQ [37] and Aurora [18], and commercial systems such as IBM System S [23]. The architecture of these first-generation systems, known as the data stream management system (DSMS), is derived from the database management sys-

tem (DBMS). Building on top of database management design concepts and principles, these systems focus on stream-specific features such as stream window query management and complex event processing (CEP). Entering the 2010s, with the rise of MapReduce [44] and cloud computing, modern-generation systems turned towards distributed, data-parallel computing in shared-nothing architectures [59], which demonstrate new features such as stateful computing, scalability, and fault tolerance. Examples include Spark Streaming [10, 162], Flink [9], and Google’s Dataflow [72].

The development of stream processing has led to changes in several requirements. In this thesis, we focus on three of them. The first change is that modern stream systems have explicitly supported state management [29] since it ensures the integrity of results in a distributed environment and contributes to the overall performance. Another is that modern systems have deployed advanced scaling technology to enhance performance and handle dynamic workloads while maintaining cost efficiency, which is becoming increasingly critical for sustainability [32]. Lastly, the demand for accuracy in query results has become paramount, departing from the early designs that emphasized speed at the expense of precision through approximation approaches [59]. Achieving accurate and fast results requires efficient data management techniques, like indexes, to enhance query performance in stream processing.

Optimisation in modern generation systems is necessary to accommodate the evolution of stream processing, particularly in state handling and scalability [131, 124]. To accommodate the first requirement, state management modules must correctly and efficiently support stateful tasks. To achieve the second requirement, systems require enhanced scalability to scale the performance and improve the cost efficiency [103]. These two questions can be seen as system-level optimization. However, to meet the requirements for the third question, we need to look into algorithmic-level optimizations to design efficient indexes for fast and exact queries. One possible and promising solution could be the learned index, which can accelerate the querying profane by learning the data distribution. Therefore, we argue that optimizing a modern system is not as simple as designing one module; it requires a synergy of system-level efforts and algorithm-level optimizations.

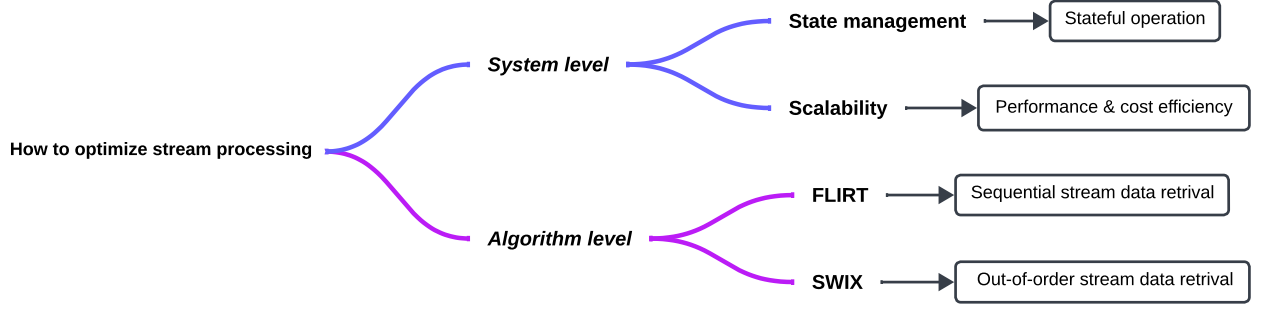


Figure 1.1: Classification of research questions (into system and algorithmic levels) and how thesis contributions fit in.

In essence, this thesis introduces three types of optimizations at the system and algorithmic levels, encompassing scaling strategies, state management, and index structures. These optimizations are designed to improve the efficiency and effectiveness of real-time data processing across various domains. The scalability strategies enable systems to adapt dynamically to changing workloads, crucial for industries like finance where market conditions fluctuate rapidly [142]. State management optimizations are crucial for ensuring data consistency and reliability in distributed systems, which are commonly employed in e-business [114]. Finally, advanced index structures improve the speed and accuracy of data retrieval, which is beneficial on real-time content analysis such as high frequency trading [3]. These advances have implications not only in industry but also in academia, where they can drive innovation and new discoveries. For example, in seismology, improved data processing capabilities allow quicker and more accurate earthquake predictions, potentially saving lives by improving response times [99].

1.2 Research Questions and Contributions

In this study, as shown in Figure 1.1, we aim to optimize stream data processing at both the system level and algorithm level by answering the following two questions:

1. *At the system level, how can state management optimization be designed and implemented for a stream workflow system, and scaling is achieved according to diverse platforms?*
2. *At the algorithm level, how can learned indexes be designed to enhance stream processing,*

particularly in addressing challenges including updatability and temporal ordering, and what performance gains can learned index bring to stream window processing?

For question 1, one concrete stream workflow system of interest is dispel4py [57]. It is a scientific workflow-specific stream system that aims to bridge the gap between stream processing and scientific research. The motivation of such a system is straightforward: There are gaps in applying stream processing technologies to scientific research, especially in scenarios requiring high-performance computing (HPC) for compute-intensive tasks based on stream data [21]. Researchers from a non-systems background usually find it difficult to efficiently leverage HPC infrastructure in stream applications. To bridge the gap, dispel4py has been developed based on the concepts of DISPEL (Data-Intensive Systems Process Engineering Language) [112], which allows users to focus on their work without being distracted from managing computing infrastructure [58].

The original architecture of dispel4py featured a static assignment of processing elements (PEs) to computational resources. This approach did not account for fluctuations in data rates or varying task demands, limiting the system’s ability to be flexible and scalable. To overcome these challenges and improve both performance and cost-effectiveness, we incorporated dynamic scaling methodologies. Notably, *dynamic allocation* and *dynamic auto-scaling* allow dispel4py to adjust resources dynamically in response to real-time workload requirements [99]. This integration necessitated a comprehensive overhaul of the system’s framework, as the original design did not accommodate such dynamic capabilities.

The implementation of *dynamic auto-scaling* presented a significant challenge mainly due to the need to efficiently balance resource scaling in diverse computational environments without compromising performance. This spurred the creation of an innovative strategy for dynamically managing workload allocation, guaranteeing optimal resource use according to the system’s instant requirements. Additionally, the introduction of dynamic enhancements necessitated state management capabilities during scaling operations. To enable stateful processing in dynamic allocation settings, we have designed a lightweight state management strategy called *hybrid*. This strategy helps to manage state information between distributed nodes, ensuring the con-

sistency and reliability of stream data processing in dispel4py [103].

At the algorithmic level, as mentioned above, the transition to exact query results requires efficient indexing techniques. There are existing works to leverage traditional indexing techniques, like hash tables and trees in query processing in stream systems [19]. Recently, learned indexes have gained much attention. They utilize machine learning models to optimize access and storage and are seen as an alternative to traditional indexes [94]. However, learned indexes have not yet been utilised in stream processing due to being problematic when it comes to updates. With the development of the updatable learned index, high-performance updatable learned indexes, such as ALEX [49] and PGM [56], are proposed. These indexes demonstrate competitive performance relative to traditional indexing methods, which makes the learned indexes a viable option to improve query performance in highly dynamic scenarios.

To integrate learned indexes into stream systems effectively, we must address common challenges for updatable learned indexes, such as adapting to rapidly changing data distributions, performing low-latency updates and retraining, and ensuring consistent performance across diverse workloads [157]. Specifically, we need to carefully manage two critical trade-offs when designing an ideal stream index: the search-update and time-space balances. As Figure 1.2 suggests, an ideal stream index should adeptly navigate these trade-offs, aiming to achieve a balance where both space and time efficiency are optimized for update-heavy streaming workloads while maintaining low retrieval latency. These trade-offs and their strategic management will be further discussed in Section 2.4.3. In addition, in stream processing, learned indexes must also effectively maintain the temporal sequence of timestamps, further introducing complexities.

To address these challenges, we developed two innovative learned indexes, FLIRT and SWIX, for the effective execution of stream window processing: FLIRT is designed for handling the incoming streams of sequential data like time series over a window. It leverages a piecewise linear model to reduce lookup time and utilizes a queue-assembling structure to enable efficient enqueueing and dequeuing in scenarios where data arrive in a predictable arrival order. However, SWIX is designed for out-of-order key insertions and is able to work with the generic window

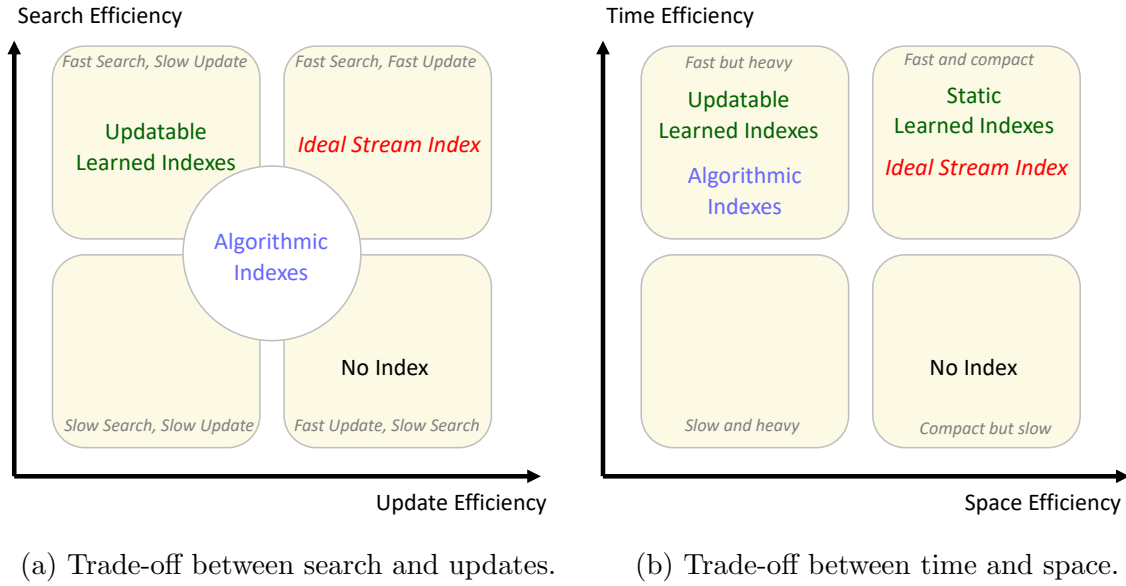


Figure 1.2: Design trade-offs between current stream index and where the ideal stream index fits in.

processing settings, in which data arrival times do not necessarily reflect the inherent order of data events. Specifically, SWIX uses advanced buffering and dynamic structures that enable fast query execution with low memory consumption. The experimental analysis shows both FLIRT and SWIX have demonstrated exceptional performance in their respective applications, endorsing the feasibility of adopting the learned index in stream application.

1.3 Thesis Structure

The rest of the thesis is organized as follows:

- Chapter 2 introduces concepts relevant to stream data processing. We trace the evolution of stream systems and then focus on developing state management strategies and scaling strategies. Furthermore, we give a brief introduction to traditional index structures, learned indexes, and stream-specific indexes for index techniques.
- Chapter 3 introduces dispel4py and presents the design and implementation of stateful management and auto-scaling techniques. We evaluate the performance impact of these optimizations in handling dynamic workloads.

- Chapter 4 proposes FLIRT, the first learned index designed to optimize the performance of stream system queries. We discuss the architectural design of FLIRT, operational mechanics and performance in stream processing situations where data arrives sequentially.
- Chapter 5 extends stream learned indexes to handle the out-of-order arrival of data by proposing a versatile stream index, SWIX. We dive into the design considerations of SWIX before demonstrating that SWIX is both a space- and time-efficient index through detailed performance evaluations.
- Chapter 6 concludes the thesis and outlines further research work.

1.4 Publications

The following peer-reviewed publications [99, 103, 161, 101], completed during my PhD, contribute to the research presented in this thesis:

1. **Liang Liang**, Rosa Filgueira, Yan Yan, and Thomas Heinis. 2022. **Scalable Adaptive Optimizations for Stream-based Workflows in Multi-HPC-clusters and Cloud Infrastructures**. *Future Generation Computer Systems, (FGCS)*, 2022.

Contribution statement: Liang Liang: Conceptualization, Methodology, Software, Writing, Investigation. Rosa Filgueira: Data curation, Conceptualization, Supervision, Validation, Reviewing, Funding acquisition. Yan Yan: Visualization, Analysis. Thomas Heinis: Supervision, Resources, Reviewing.

2. **Liang Liang**, Heting Zhang, Guang Yang, Thomas Heinis, and Rosa Filgueira. **Optimization towards Efficiency and Stateful of dispel4py**. *In Proceedings of the SC'23 Workshops of The International Conference on High-Performance Computing, Network, Storage, and Analysis (SC'23)*, 2023.

Contribution statement: Liang Liang: Investigation, Conceptualization, Methodology, Software, Writing, Visualization, Analysis. Zhang, Hething: Investigation, Conceptualization, Software. Guang Yang: Investigation, Validation, Reviewing. Thomas Heinis:

Supervision, Resources, Reviewing. Rosa Filgueira: Investigation, Data curation, Conceptualization, Supervision, Validation, Reviewing.

The source code for the aforementioned two projects is available at https://github.com/NoPuzzle/dispel4py_autoscaling. The foundational dispel4py framework was developed by Rosa Filgueira and her collaborators. I led the implementation of all new mappings and optimizations proposed by the aforementioned papers, as well as spearheaded the validation, unit testing, and execution of experiments.

3. Guang Yang, **Liang Liang**, Ali Hadian, and Thomas Heinis. **Flirt: A Fast Learned Index for Rolling Time Frames**. *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2023.

Contribution statement: Guang Yang: Investigation, Conceptualization, Methodology, Validation, Software, Writing, Analysis. Liang Liang: Investigation, Conceptualization, Methodology, Writing, Validation, Visualization, Analysis. Ali Hadian: Methodology, Validation, Reviewing. Thomas Heinis: Supervision, Resources, Reviewing.

4. **Liang Liang**, Guang Yang, Ali Hadian, Luis Alberto Croquevielle, and Thomas Heinis. **Swix: A Memory-efficient Sliding Window Learned Index**. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2024.

Contribution statement: Liang Liang: Investigation, Conceptualization, Methodology, Software, Writing, Validation, Visualization, Analysis. Guang Yang: Investigation, Conceptualization, Methodology, Validation, Software, Writing, Analysis. Ali Hadian: Methodology, Validation, Reviewing. Luis Alberto Croquevielle: Conceptualization, Analysis, Validation, Reviewing. Thomas Heinis: Supervision, Resources, Reviewing.

The source code for the aforementioned two projects is available at <https://github.com/SWIXProject/SWIX>. This repository was jointly developed by Guang Yang and me through pair programming on Parallel FLIRTs, SWIX, and PSWIX. Specifically, I was responsible for implementing the root layout, search, and bulk-loading components of the code. I led the validation and unit testing processes. All experiments were conducted collaboratively.

The following peer-reviewed publications [102, 160], completed during my PhD, are not included in this thesis:

1. **Liang, Liang**, Guang Yang, Thomas Heinis, David Taniar. **SOJA: A memory-efficient Small-large Outer Join for MPI**. *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2021.
2. Guang Yang, Ahirup Ghosh, **Liang Liang**, and Thomas Heinis. **In-network Approximate and Efficient Spatiotemporal Range Queries on Moving Objects**. *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2024.

Chapter 2

Background

2.1 Overview

This chapter provides the essential background for the thesis, depicted in Figure 2.1. Stream data concepts are comprehensively introduced in Section 2.2, where we go through the defining characteristics (Section 2.2.1), processing requirements (Section 2.2.2), and models (Section 2.2.3) of stream data. We then discuss the window techniques in Section 2.2.4. These discussions highlight a clear need for stream processing systems and the role of indexing structures in enhancing system performance. We then showcase the stream system in Section 2.3. We first trace the historical development of stream systems, outlining their evolving capabilities to meet the demands of stream processing requirements described in Section 2.3.1, followed by a focus on state management and scaling strategy of stream systems in Section 2.3.2 and Section 2.3.3 respectively. In Section 2.4, we shift focus to algorithmic enhancements in stream processing by discussing details on traditional indexes, learned indexes, and stream indexes in Section 2.4.2, Section 2.4.3 and Section 2.4.4, respectively.

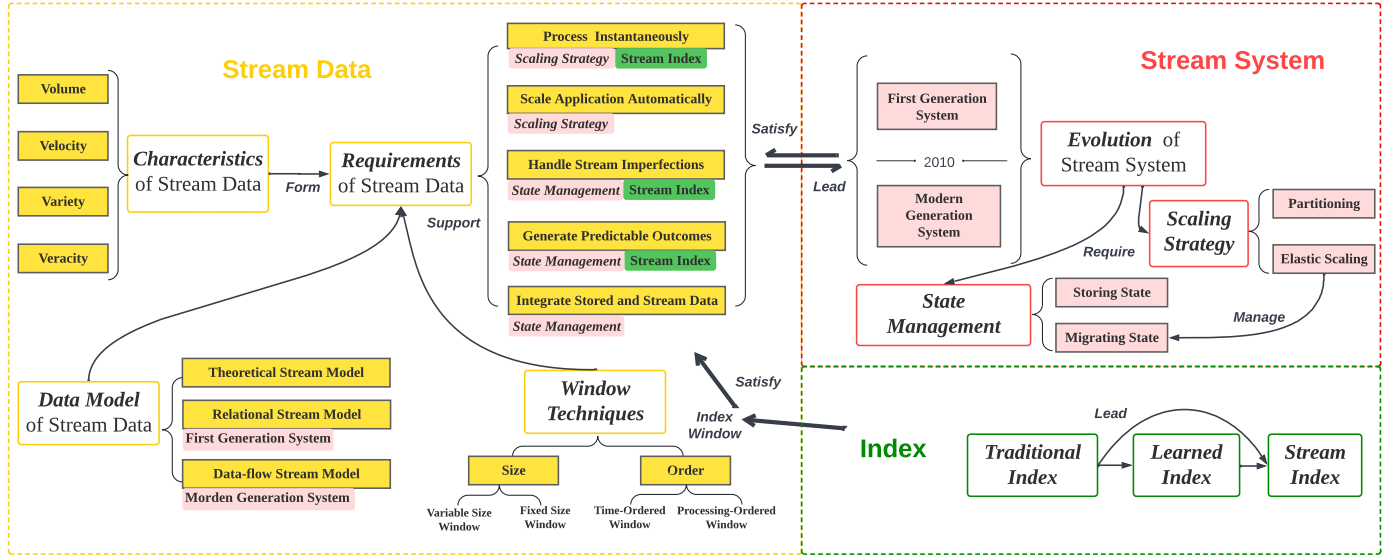


Figure 2.1: Overview of the structure of the background chapter. Yellow highlights topics discussed in Section 2.2. Similarly, Red indicates topics in Section 2.3, and Green describes topics in Section 2.4

2.2 Stream Data

2.2.1 Characteristics of Stream Data

Stream data are the subset of big data, distinguished by its requirement of real-time processing, characteristic of unbounded data size, and representation as the temporal view of a continuous flow of data elements [156]. Stream data shares four critical characteristics with big data, known as the 4Vs [78]:

- **Volume** highlights the massive magnitude of data production and storage in a digital ecosystem. In stream processing, this characteristic corresponds to an unbounded accumulation of data that necessitates systems to dynamically cope with large data loads without losing scalability or responsiveness [71].
- **Velocity** indicates not only the high speed at which data is produced and processed but also the variable rate at which it arrives in the processing system [36]. This introduces complexities to stream processing; for example, when the arrival rate suddenly increases, bursts in data insertions, known as data bursts, need to be effectively handled to avoid

significant latency; conversely, the waste of computing resources caused by low data arrival rate needs to be minimized.

- ***Variety*** and ***Veracity*** refer to the different data formats and the correctness and reliability of the data, respectively [46]. In stream processing, data could come from various sources, such as sensors and transactions, and its format could vary. Beyond the variety of data, incorrect data is also common in practice; for instance, a faulty sensor might report inaccurate data, leading to erroneous computation. Dealing with varying data formats and uncertain data is more complicated than traditional batch processing, necessitating effective metadata, state management, and fault-tolerance mechanisms.

2.2.2 Requirements of Stream Data Processing

The previous section defines the essence of stream data; these characteristics also introduce a level of complication which specialized systems must meet (e.g., performance, reliability and scalability criteria). This section explores five specific operational requirements of interest from [143]. While discussing these requirements, we illustrate how our optimization work satisfies them.

- ***Process and Respond Instantaneously***: To achieve low latency when handling high-volume and high-velocity stream data, stream processing needs to minimize the length of the execution path and maximise the computational power and the use of the data management techniques. Although the execution path varies across different stream processing tasks, making instantaneous optimization challenging, scaling up computational resources and applying data management techniques could be realistic optimizations. Chapter 3 introduces the dynamic scaling strategy to improve the scalability; stream learned indexes are presented in Chapter 4 and 5 to accelerate data retrieval. These works contribute to achieving instantaneous processing and responses in streaming systems.
- ***Scale Applications Automatically***: We discussed that scalability is crucial for improving performance with high-velocity data streams, but "how to scale" remains a chal-

lenge, especially when the data velocity changes. This challenge can be resolved by deploying an automatic scaling strategy. This strategy dynamically adjusts resource allocation based on real-time workload; when more computing resources are needed to process burst workload, more computing resources will be active, and vice-versa. We introduce the auto-scaling technique in Chapter 3, which ensures cost-efficiency and maintains performance across distributed settings.

- ***Handle Stream Imperfections***: Due to the variety and veracity of stream data, stream processing systems need to take account of imperfections such as late data or missing data. The late data could lead to out-of-sequence processing, potentially introducing incorrect results. Furthermore, out-of-sequence not only occurs before the data arrives, but also during processing since data may pass through different processing units. Similarly, having missing data will also introduce inaccuracies. To deal with the stream imperfections, we introduce state management in Chapter 3 to avoid erroneous results; we also introduce SWIX in Chapter 5 to handle out-of-sequence scenarios.
- ***Generate Predictable Outcomes***: Due to imperfections in streaming processing mentioned above, achieving repeatable and predictable outcomes is crucial. Our work to mitigate imperfections can also support this requirement, where the index structures introduced in Chapter 4 and 5 can contribute to detecting unexpected outcomes.
- ***Integrate Stored and Stream Data***: This requirement is essential for stream-table mix queries relying on real-time and static data inputs. An extension of this requirement is that the system needs to interact with stream data and stored states for low-latency computing and imperfection detection. In-memory and lightweight state storage is necessary to retrieve the states, which has been introduced into dispel4py in Chapter 3.

Apart from these five requirements, [143] proposed three additional requirements, namely ***Keep the Data Moving***, ***Query using SQL on Streams***, and ***Guaranteed Data Safety and Availability***. These three requirements are closely related to stream processing engine (SPE) design, query language and fault-tolerance, which are outside the scope of this thesis. We recommend this paper to readers for further information.



Figure 2.2: Theoretical stream model where data is indicated in green, deleted tuples are outlined in red, and inserted tuples are outlined in green.

2.2.3 Stream Data Model

In this section, we first define the general theoretical stream model before introducing the two specific variations. The theoretical stream model studies the stream algorithm-based minimal storage and computational requirements necessary to perform operations on S under real-time constraints. It usually views these data streams as dynamic vectors, as shown in Figure 2.2 [63]. Mathematically, it represents stream data as a sequence of updated data points:

$$S(t) = \{x_i \mid i = 1, 2, \dots, t\}$$

where $S(t)$ denotes the state of the stream at time t , and x_i represents a data element arriving at that specific time. An example is illustrated in Figure 2.2, where deletion is in red and insertion is in green. Note that deletion is not a necessity; the stream data could grow infinitely.

The variants aim to model the unbounded and continuous nature of stream data, but they are designed with a different focus and purpose.

Relational stream model

The relational stream model focuses on the changing view of a fixed schema [12]. This model is commonly used in the database community since one of the sources of the stream could be the result of continuous queries [59]. The different systems use various data models but commonly

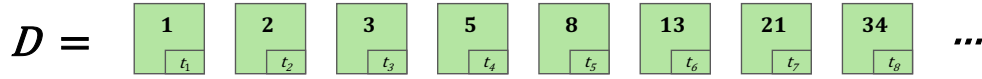


Figure 2.3: Data-flow stream model, in which each tuple includes **data** in bold and *timestamp* in italics.

focus on tracking changes. We abstract the model as follows:

$$\text{Stream} = \{(t_i, \delta_i) \mid i = 1, 2, \dots, n\}$$

This model includes a sequence of timestamp tuples associated with the corresponding state changes. Specifically, t_i is the timestamp, and δ_i denotes the change at that timestamp, for example, insertion ($\delta_i = +$) and (deletion $\delta_i = -$).

Data-flow stream model

The data flow model adds flexibility to the relational model, which is not based on a fixed schema. The only constraint in this model type is the emphasis on the timestamp [6]. The data-flow stream model can be represented as:

$$D(t) = \{(data, t) \mid data \in Data, t = timestamp(data)\}$$

Each piece of data in the stream carries a timestamp t . This timestamp is usually ordered with discrete non-negative integer values, which can be calculated by the user or a system-defined function *timestamp*, such as event time and processing time. This model enables versatile stream data management based on their temporal characteristics. Figure 2.3 illustrates the data-flow stream model, where we use the same example in Figure 2.2.

The data-flow stream model can indicate monotonicity by preserving the chronological order in which events occur [59]. This is important for processing correctness, where the execution order is related to the event timestamp. Monotonicity can be expressed using the existing model's notation as follows:

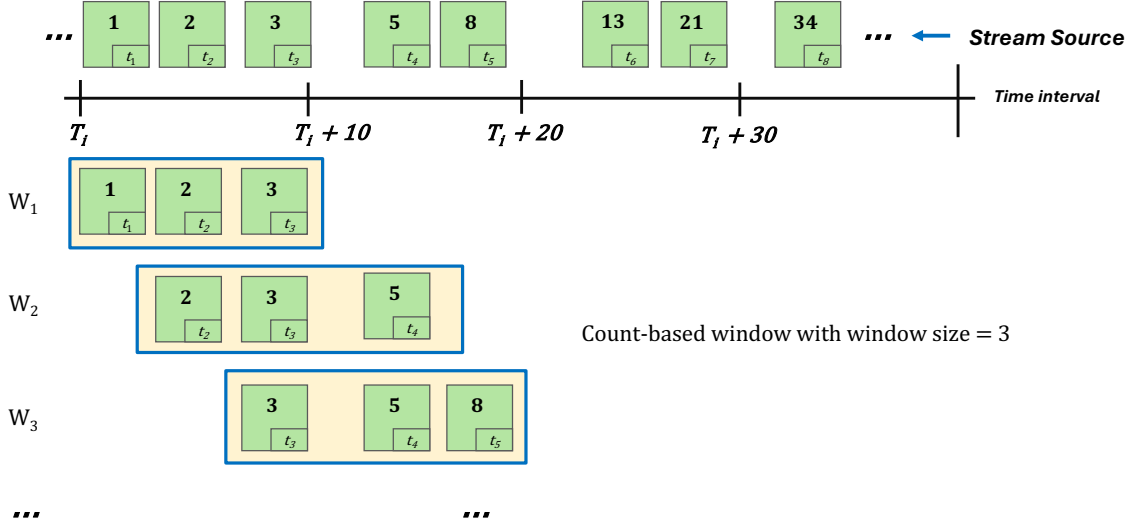


Figure 2.4: Demonstration of a continuous fixed-size window, specifically a count-based window with a size of 3.

Given any two elements $(t_i, data_i)$ and $(t_j, data_j)$ in the stream $D(t)$, if t_i and t_j are their respective timestamps and $i < j$, then it must follow $t_i \leq t_j$.

There are mechanisms like watermarks to handle out-of-order data at the system level [119], but that is beyond the scope of this thesis. We propose a learned index for stream windows focusing on out-of-order data arrivals, which will be discussed in detail in Chapter 5. In this thesis, unless specifically stated, we adopt the data-flow stream model due to its flexibility with the diverse temporal aspects of the stream data.

2.2.4 Stream Window Techniques

The stream window is a technique for handling unbounded stream data by segmenting the stream into manageable chunks with a limited size in which stream data can be effectively executed [6]. There are different types of windows; in this thesis, we categorize them based on two criteria: size and ordering.

Size-based classification classifies stream windows into either fixed-sized windows or variable-sized windows.

- Fixed-size window: A typical fixed-size window is a count-based window that maintains

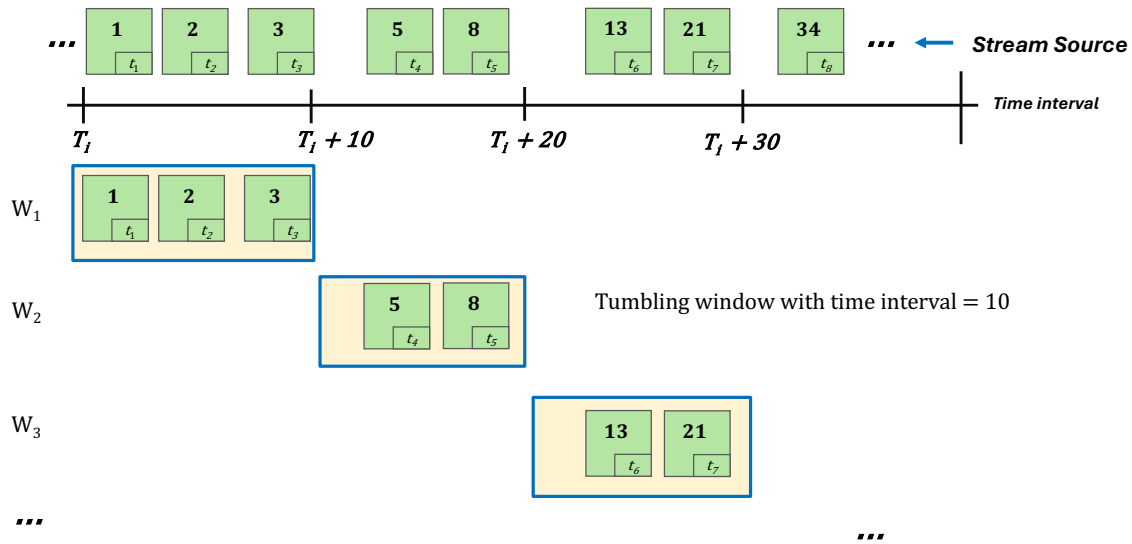


Figure 2.5: Demonstration of a continuous variable-size window, where the time interval is set to 10 (time) units.

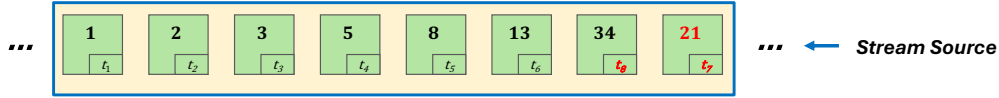


Figure 2.6: Illustration of late data for an event time window, where the data of 21 at t_7 arrives after t_8 .

constant data by continuously expiring the oldest data when new data arrives, as shown in Figure 2.4.

- Variable-size window: This type of window usually is time-based. An example is a tumbling window, depicted in Figure 2.5, which resets after a fixed time interval to process non-overlapping data [122]. When the arrival rate of incoming data is consistent, this window will maintain the fixed size, which applies to all windows with temporal boundaries. Another widely used variable-size window is the sliding window, which dynamically adjusts itself while maintaining a constant duration [18]. In contrast to the tumbling window, it overlaps data since it continuously slides over the stream. Session Windows, as well as variables in size, are triggered by user activities and terminate due to the inactivity [7].

Ordering-based windows organize data within the window based on temporal characteristics.

- Time-ordered window: The event time window is the most commonly used time-ordered

window, which restricts the chronological order of data based on the timestamp of when the event originally occurred (event time). This type of window can ensure processing integrity but necessitates the provision to deal with late data, such as the data with payload as 21 in Figure 2.6.

- Processing-ordered window: This window ignores event time and prioritizes the processing time at which events are observed in the system or window [7]. The order of the incoming data within the window is incremental. It is usually used in event-time agnostic applications aiming for high throughput.

Each type of windowing technique poses its own set of challenges and opportunities with respect to indexing strategies. In fixed-size windows, predictable memory usage enables resource pre-allocation to optimise space use. Variable-size windows naturally introduce more difficulties to indexing methods since the window size changes quickly.

Similarly, for ordering-based windows, maintaining the timestamps in the processing-ordered window would be much simpler than that in the event-ordered window. Since the processing-ordered window has incremental timestamps, the index only needs an index to support appending. However, in the event-time-ordered window, the index has to be designed to accommodate random insertions from potentially late data.

2.3 Stream System

2.3.1 Evolution of Stream System

This section provides a brief overview of the historical development and architectural evolution of stream processing systems, building on top of basic insights of stream data as given in the previous section. According to the classification in recent studies [24, 59], these stream systems can be differentiated into two generations: the first generation, from 2000 to 2010, and the second (modern) generation, from 2011 to the present. Such differentiation intends to empha-

size that stream processing has adapted to and evolved with changing demand, algorithmic developments, and hardware innovations.

First Generation of Stream Systems

Stream processing began to take shape in the early 1990s with the introduction of the Tapestry system [147]. Entering 2000, stream systems experienced their Cambrian explosion, during which many first-generation stream systems emerged. These stream systems, also called data stream management systems and active databases, originated from the databases community and were deeply influenced by contemporary DBMS [70, 67]. This is best illustrated by the fact that they utilized the relational stream model, aiming to solve continuous queries and CEP [41].

NiagaraCQ [37] is the pioneering stream system that was the first to address large-scale continuous queries in active environments with dynamic optimization. This idea inspired subsequent streaming data platforms. This work, along with contemporary stream systems such as TelegraphCQ [35], STREAM [12], Aurora/Borealis [1], and Gigascope [40], which were based on different emphasises on query semantics [13, 25], pushed the understanding of stream processing systems and introduced various challenges, such as sliding window aggregation [15], load balancing [146], and fault tolerance [20]. These research-driven systems also inspired industries to implement commercial systems such as IBM S System S [65] and Oracle CQL/CEP.

These first-generation stream systems mostly share common characteristics such as window query processing, CEP, centralized architecture, and ordered event stream [59]. Their commonality also manifests in how they satisfy processing requirements described in Section 2.2.2, as listed in Table 2.1. First-generation systems applied basic techniques for event-driven models and manual scaling, so scalability was limited with high response latencies (in seconds or milliseconds) [14]. Therefore, they use approximate results to improve latency. Systems adopt simple mechanisms for handling imperfections and erroneous results, yielding limited performance [59].

Table 2.1: First and modern generation stream systems addressing key requirements.

Requirements	First Generation System		Modern Generation System	
	<i>Solutions</i>	<i>Results</i>	<i>Solutions</i>	<i>Results</i>
<i>Process Instantaneously</i>	Simple event-driven models, Scale-up, Load-aware scheduling Using approximate results	Seconds or milliseconds response latency	Highly optimized execution paths, Advanced algorithms, In-memory processing, Elastic load management, Scale-out	Microsecond to millisecond response latency
<i>Scale Applications Automatically</i>	Manual scaling, Standalone, Simple clustered setups	Limited scalability	Auto-scaling techniques, Cloud integration, Container orchestration	Dynamic scalability, Efficient resource utilization
<i>Handle Stream Imperfections</i>	Simple error handling: Timeouts, and retries Event order provisions: Slack, Heartbeat, Punctuations, Replay past data	Limited correctness guarantees	Watermarking, Advanced state management, Event-time processing	Robust handling of out-of-order and late data
<i>Generate Predictable Outcomes</i>	Simpler deterministic models, Transaction logs	Predictable, Limited by state management	Exactly-once processing, Error recovery	Reliable Consistent outcomes
<i>Integrate Stored and Stream Data</i>	Loose coupling with databases, Materialized views, Batch integration	Feasible but with high latency	Unified processing engine, DStreams and structured streaming	Seamless, Real-time data integration

Second (Modern) Generation of Stream Systems

The second-generation stream systems shift their focus to distributed computing, data-parallel engines, and shared-nothing architectures [24]. This shift indicates that the stream systems are embracing principles from the distributed systems community, even though they were initially derived from the database community. This transition is due to the influence of MapReduce, and the rise of cloud computing [116]. In the 2010s, the transition has given rise to well-known stream systems, including Twitter Storm [11], Spark Streaming [10, 162], Samza [118], Flink/Beam [30], Millwheel/Dataflow [5], and Kafka Streams [115]. Modern stream systems adopt a data-flow model, and modular and scalable design. They include features such as ordering provision, state management, load management, and sophisticated scaling techniques. These designs and features demonstrate high performance, process integrity, and fault-tolerance capabilities [88].

To understand modern stream systems, we can review how the second-generation stream systems meet processing requirements in Table 2.1. Modern stream systems leverage optimized execution paths, advanced algorithms, and in-memory processing, bringing response latencies down to the microsecond-to-millisecond range [88]. Innovations of modern systems include auto-scaling techniques, cloud integration, and container orchestration for dynamic scaling and efficient resource utilization [151]. They are also equipped with watermarking, complex state management, processing order provision, exactly-once processing, and error recoveries to deal with processing imperfections and produce predictable results [60]. A unified processing engine and novel data type also facilitate seamless and real-time integration of stored and stream data [16].

dispel4py is a second-generation stream system targeting scalability, flexibility, and real-time data processing [58]. It is tailored for scientific computing applications and has been developed to allow the creation of complex data processing applications from simple, reusable, and interconnected components. dispel4py supports diverse computing environments, ranging from single machines to HPC clusters. To efficiently process data-intensive tasks with the skewed workload in parallel computing, dispel4py can dynamically allocate computing resources, thus

minimising the latency [99].

This thesis aims to extend `dispel4py` functionalities to cope with modern requirements in stream processing and state-of-the-art features. Particular attention is given to state management and scaling strategies.

2.3.2 State Management

In stream processing, the state is an intermediate computation value that can be repeatedly utilized [148]. The processing could be either stateless or stateful. In stateless stream processing, each event is executed independently to any system state. Conversely, stateful applications need the state to produce the correct results [29, 106]. An example is counting occurrences of a specific task executed across different nodes; the state must be preserved, involved in the counting and updated by different nodes. Stateful processing is prevalent because states can effectively capture internal side effects in continuous execution scenarios, such as active windows, incremental aggregations, and stream pipelines that contain user- or system-defined state variables [59]. Those side effects can contribute to computation's performance and correctness. Therefore, effectively managing states is crucial in stream processing systems, but it also introduces technical challenges [148].

Storing states

One of the challenges is how to store states. The chosen solution usually depends on the size of the state. In-memory storage is used for small states, which allows fast access and updates [130, 163]. However, it will be problematic for recovery from machine failures. Therefore, state replication is a backup solution for fault-source when managing states. For applications with large states exceeding memory limits, persistent storage is deployed. While this approach enhances recovery capabilities, it introduces I/O overhead and can increase latency [95, 105]. To improve operations with a large state, load shedding, state spilling, and state cleanup delay were proposed. However, there is an inherent trade-off between accuracy and performance.

Load shedding only preserves part of the states, such as samples, which accelerates the state access and computation but lowers the accuracy [150]. Load spilling is designed for accuracy-sensitive use cases, which utilise both memory and disk storage by temporarily moving states stored in memory into the disk. State cleanup delay is another solution prioritizing accuracy, which postpones the deletion of states. This is useful for scenarios where late-arriving data might still need to be considered [148]. These two approaches improve accuracy at the expense of performance. There are also well-known state storage strategies for fault-tolerance purposes, such as check-pointing, which are beyond the focus of this thesis.

Migrating states

Migrating states is another challenge in state management. It is particularly important in stream processing systems, in which states need to transit efficiently between nodes. [48] describes the objective in state migration: both selecting states to migrate and the migration process itself need to minimize synchronisation overhead and reduce delays. There are diverse approaches to handling this challenge with the objective in mind.

[165] presents a dynamic migration strategy focused on the first-generation stream system, specifically continuous queries. This strategy includes two tactics: moving state and parallel track. The moving state suspends the current execution to directly transfer state data into a new query plan to ensure a quick transition but at the expense of high resource costs. On the other hand, the parallel track strategy conducts gradual migration by running both the old and new query plans together to continuously produce outputs and maintain performance with low computing resources.

For modern stream systems in the distributed environment, the network overhead during the state migration becomes an issue. [123] aims to determine the placement location by adopting an optimizer to evaluate network usage. If the calculated savings reach the predefined minimum migration threshold (MMT), migration will be triggered to improve the overall performance. Furthermore, this work proposes a stream-based overlay network (SBON) to effectively manage the migration. [120] also focuses on minimising network usage by introducing a proactive

approach called MigCEP that generates the migration plan in advance.

[48] addresses the problem of intra-operator state migration among nodes. They present advanced algorithms for both live and incremental state migration with minimal migration delay. This work also introduces the migration cost prediction based on past workloads, which can be used to minimise the overhead further. There are also solutions leveraging index structures, such as hashing, to address state migration under high computational workloads. For example, [54] presents a novel replication representation and scheme to facilitate replicating and backing up state in the stream system.

The migration in dynamic environments also needs to consider the node scaling (addition or removal) caused by the workload, data characteristics, and resource availability. This will be discussed along with scaling strategies in the next section (Section 2.3.3).

2.3.3 Scaling Strategy

To leverage distributed/parallel computing resources, stream systems need to partition stream data or tasks into multiple processing nodes, which can increase throughput [89].

The objective of partition is to achieve load balance, minimal data movement and communication overhead [42]. In practice, the partition strategies usually prioritize one or more of these objectives. For example, a key-based partition could reduce communication overhead by keeping relevant keys within the same partition but could lead to workload imbalance. Conversely, round-robin partitioning prioritizes load-balance instead of data movement and communication overhead. Other advanced, customized partition methods based on machine learning or cost models (that utilize prior knowledge) may balance these objectives to achieve better performance [154].

After partitioning, managing scaling to adapt to changing data volumes and processing demands became challenging. Modern stream systems are equipped with dynamic scaling or elasticity, which can vary resource utilization to efficiently handle workload variations [75]. The scaling can scale-up or scale-out more processes for high performance or scale down to reduce processes

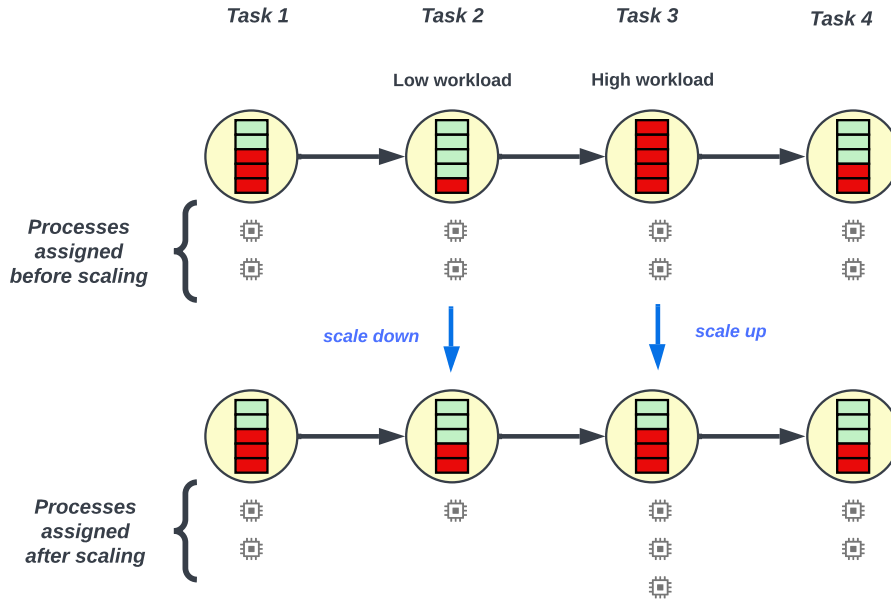


Figure 2.7: Demonstration of scaling strategies in parallel stream systems.

for cost efficiency, as shown in Figure 2.7. Achieving elastic scaling in stream systems involves designing scaling policies and mechanisms. The policy answers when and how to scale, while the mechanism manages task/resource relocation and state migrations [59].

Scaling policies can be classified into heuristic and predictive; the former is based on empirical rules consisting of predefined thresholds such as system metrics (for example, CPU and memory utilization, queue size, monitored throughput, and latency). The latter relies on the predictive model to forecast the system’s behaviour and decide when to scale. Predictive policy may be driven by system theory, such as queueing theory [61, 108] and control theory [8, 91], or machine learning models based on historical data, such as linear regression and reinforcement learning [87, 31].

Scaling mechanisms implement decisions made by policies, including strategies for state redistribution and reconfiguration. States play an essential role in scaling. They need to be correctly relocated and accessed to ensure the correctness of execution. Redistribution uses the hashing technique to allocate states correctly, which can minimise state movement across nodes [29]. More sophisticated methods, such as hybrid partitioning, are also preferred, as they leverage hashing and mapping to handle the skewed state distribution [64].

Reconfiguration transfers states within a single node or across multiple nodes. The mechanism strategies include stop-and-restart, partial pause-and-restart, and proactive replication. The stop-and-restart strategy interrupts the processing, snapshots the state, and restarts with the new configuration. Unlike stop-and-restart, partial pause and restart can only block affected execution, such as downstream operations, which will increase overall performance but introduce implementation complicity [136]. Proactive replication enables real-time reconfiguration by replicating states in multiple nodes [59]. This approach ensures that state consistency is preserved without interrupting ongoing processes at the cost of increased complexity and storage overhead.

2.4 Index

This section introduces index technologies that facilitate processes and are conducive to providing predictable outcomes in stream processing. First, we will introduce index concepts before moving on to traditional indexes, followed by learned indexes and stream-specific indexes.

2.4.1 Index Concepts

Indexes are data structures deployed in a database to accelerate data retrieval [96]. They provide direct pointers to the small range of locations that contain the target data, therefore avoiding full table scans [139]. The index can be classified into:

- Primary or secondary index: This classification depends on the attributes they are built on: primary indexes are constructed on the attribute uniquely identifying each entry in a table, while secondary indexes are built on non-primary attributes that cannot identify the entire table but are commonly used in queries [39]. For example, a query is interested in specific enrollment years in the student table identified by student ID, a primary index, as shown in Query 2.1. In this example, the enroll year attribute, which is likely not the primary key of the student table, is used in the query condition. This attribute is

commonly used in queries but does not uniquely identify records in the table. Here, a secondary index on enroll year would optimize the retrieval process, making the query more efficient by reducing the number of entries the database must scan to find students enrolled in 2023 or 2024.

- **Clustered or non-clustered index:** Clustered indexes physically sort the data entries in the storage media, aligning with the index order [139]. For example, the clustered index stores the entries in the student table in ascending order of student IDs. In contrast, non-clustered indexes do not alter the physical order of the data; instead, they maintain a separate structure to store pointers that reference the actual data locations.
- **Dense or sparse index:** Indexes differ based on density. A dense index includes all entries in the table, while a sparse index only stores some entries [139]. As a result, the dense index improves access speed at the expense of storage overhead, and the sparse index is more storage-efficient but slower in lookups.

Beyond these fundamental classifications, incorporating machine learning techniques also differentiates the index from the index using traditional algorithmic components. The following sections will showcase representative indexes in their corresponding class.

```
SELECT student_id, student_name
FROM student
WHERE enroll_year = 2023 OR enroll_year = 2024;
```

Query 2.1: Query to retrieve students enrolled in 2023 or 2024

2.4.2 Traditional Index

There are several traditional indexes designed for different use cases.

Bitmap Indexes: Bitmap indexes indicate whether a value is present in the table using bit array; each possible entry corresponds to one bit [34]. It is suitable for low-cardinality

attributes, excelling in queries involving categorical data and Boolean logic operations such as AND, OR and NOT. The Query 2.2 selects students enrolled in the ‘Databases’ course who are not simultaneously enrolled in the ‘Introduction to Machine Learning’ course. The query utilizes Boolean logic to exclude certain records based on multiple attribute conditions. In such case, a bitmap index on the course attribute would significantly improve query performance, as it allows the database to quickly determine which rows in the student table correspond to each course by checking bits in the bitmap array.

```
SELECT student_id, student_name
FROM student
WHERE course = 'Databases' AND NOT EXISTS (
    SELECT 1
    FROM student s
    WHERE s.student_id = student.student_id
    AND s.course = 'Introduction to Machine Learning'
);
```

Query 2.2: SQL example using boolean logic

Point Lookup Indexes (Hash Indexes): The hash index is a typical non-clustered index as it does not pay attention to the order of the data [153]. It provides direct data retrieval through a hash function to optimize point lookup queries such as Query 2.3. The average time complexity of point lookup is $O(1)$. However, the choice of the hash function is the key to the performance since an inappropriate hash function will cause a significant collision, meaning that many keys are mapped into the same bucket; therefore, the local refine needs to be conducted, increasing the overhead. Additionally, due to their unordered nature, hash indexes do not support range queries that require traversal among logically ordered contiguous keys.

```
SELECT *
FROM student
WHERE student_id = 2023000000;
```

Query 2.3: Lookup query example
Range Indexes (B-Trees and B+Tree):

B-Tree [38] is designed to efficiently process range queries (see example in Query 2.4), by maintaining the data order to achieve efficient traversal. B-tree consists of two types of nodes: internal and leaf nodes, both of which may contain data (keys). Internal nodes also contain pointers to child nodes. The capacity of each internal node is constrained between $[\lceil \frac{f}{2} \rceil - 1, f - 1]$, where f is the fan-out or order of the tree. The f is an important parameter in B-trees; a higher f results in a larger node at each level, thus lowering the overall height; lower f increases the tree height but reduces the node size. The leaf nodes appear at the bottom of the tree without child nodes. B-Tree self-balances the tree during construction and modification to maintain as low tree height as possible. The height of the tree will be $\log_f N$, in which N is the number of data. B-Tree supports efficient search, insertion, and deletion operations with a logarithmic complexity in time.

```
SELECT *
FROM student
WHERE student_id BETWEEN 2023000000 AND 2024999999;
```

Query 2.4: Range query example

B+Tree

B+Tree is the most commonly used variant of B-tree [74]. Its internal nodes do not store data; they are only used to locate the leaf node, as shown in Figure 2.8. Data is stored in connected leaf nodes, ensuring the data locality and enhancing scanning performance.

The shape of a B+Tree is determined by the fan-out (f) and the leaf node capacity (m). The f is the same as in a B-tree; it defines the tree's breadth and height. The m refers to the number of data in each leaf node, affecting the leaf data density.

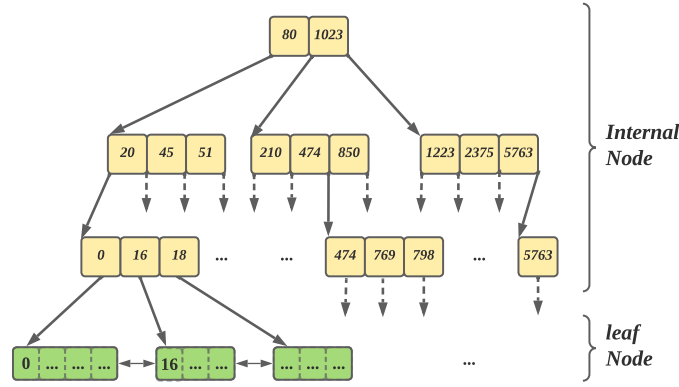


Figure 2.8: Illustration of B+tree.

Lookup of B+Tree

The lookup is the fundamental operation in B+Tree since other operations, such as range query, insert, delete, or update, require a located position.

We use $K = \{k_0, k_1, \dots, k_{n-1}\}$, where $k_0 < k_1 < \dots < k_{n-1}$ and $k_i \in \mathbb{R}$ to represent the data with size n indexed by B+Tree, and we use k for lookup key. The lookup essentially is to locate the position i_k of the key k .

The search starts with internal node traversal, which iteratively progresses to a lower level of internal nodes based on key comparison until reaching the leaf node. The complexity of internal traversal is $O(\log_f(n))$. We can consider a set of internal nodes as a model I , such as a decision tree. It maps a search key $k \in \mathbb{R}$ to a search bound based on index positions, $[i_l, i_h]$, where $i \in \mathbb{Z}_{\geq 0}$.

Once the leaf node is identified, the search bound $[i_l, i_h]$ can be narrowed to $[i_s, i_s + m]$, in which i_s is the position of the first key in the leaf node. The local search in this bound can be performed by binary search since data order is maintained in the leaf node, resulting in the $O(\log m)$ complexity. Therefore, the overall lookup complexity in a B+Tree is $O(\log_f(n) + \log m)$.

Modifications of B+Tree

As we mentioned earlier, the capacity of the internal node is bounded by $[\lceil \frac{f}{2} \rceil - 1, f - 1]$, and the leaf node is bounded by the $[\lceil \frac{m}{2} \rceil - 1, m - 1]$. When inserting a new key, if a node

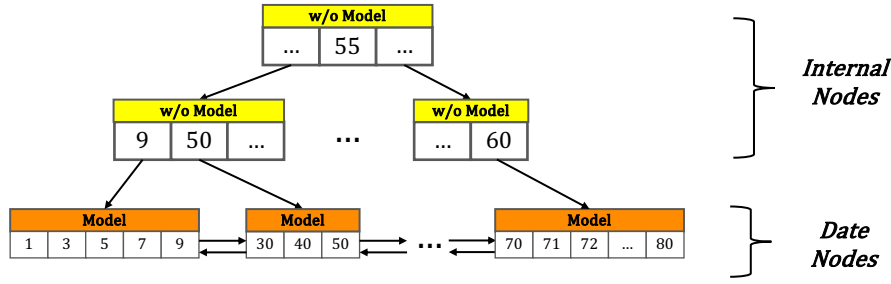


Figure 2.9: Demonstration of common learned index structure.

exceeds its full capacity, the tree needs to split the node. The node-splitting re-partitions data and creates new nodes, which will be inserted into the parent node. If it causes parent nodes to exceed their capacity, the split can propagate changes upward until the root, which costs $O(\log n)$. Conversely, deletion could cause the capacity to fall below the half-full threshold, leading to merging among sibling nodes. Similar to splitting, the merging could propagate upward changes and achieve a $O(\log n)$ complexity.

2.4.3 Learned Index

The learned index originates from a joint effort of machine learning research and database research. In 2018, the first learned index structure, recursive model index (RMI), was proposed for range query, which predicts data locations based on a machine learning model instead of relying on predefined algorithmic paths, as shown in Figure 2.9 [94]. This transformative work treats the index as a predictive model, which provides the framework for our discussion on the B+Tree, in which B+Tree is treated as a model that maps the lookup key to a search bound $[i_l, i_h]$.

Unlike B+Tree or other traditional range indexes, which depend on algorithmic comparison within a hierarchical structure without considering data distribution, the learned indexes employ machine learning models to approximate the distribution of keys based on their position [138]. Specifically, the approximation model learns the cumulative distribution function (CDF) of indexed keys to provide accurate predictions about the search bound. Due to the inherent errors in the models, the search bound will transform from predefined parameter-based search bound $[i_s, i_s + m]$ in a B+Tree to error-based search bound $[i_k - \epsilon, i_k + \epsilon]$ in which ϵ represents

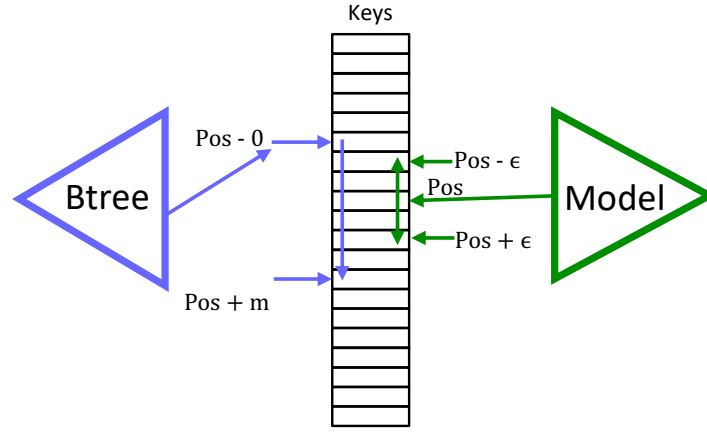


Figure 2.10: Comparison between the difference in search bound of algorithmic and learned models.

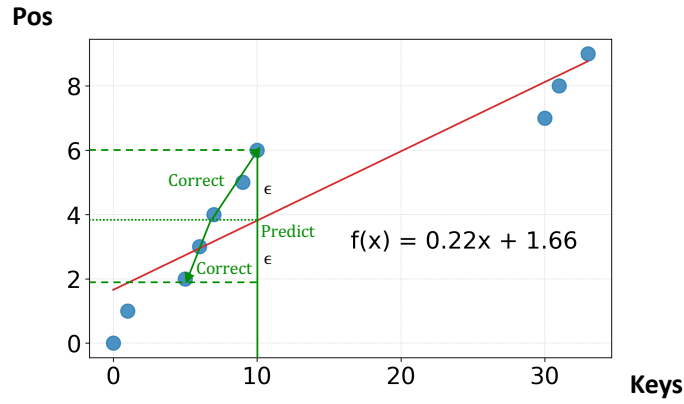


Figure 2.11: Illustration of the predict and correct process in a learned index.

the possible deviation from the predicted key location, as shown in Figure 2.10. The CDF (F) also implies both data and model monotonicity [158], which ensures that if k_i and k_j are keys such that $i < j$, then $k_i \leq k_j$ and $F(k_i) \leq F(k_j)$. The order preserved by monotonicity is crucial for the “last-mile search” or correction phase, which accurately pinpoints data within the predicted error bounds, as shown in Figure 2.11.

The error-based search bound indicates that an accurate model enhances performance. This aspect turns the typical disadvantage of machine learning, over-fitting, into a benefit in static use cases. Therefore, complex models like neural networks, which have high training costs but offer excellent accuracy, are advantageous. However, the considerations will be different for updatable learned indexes.

The first consideration is the trade-off between the search performance and update performance.

In updatable indexes, insertion and deletion could shift the indexed data distribution, increasing the prediction error, thereby necessitating retraining [101]. In this case, the complex model with high retraining cost leads to huge tail latency. Therefore, simpler models are preferable to balance update efficiency with search accuracy. Another consideration involves the trade-off between time and space efficiency [111]. Static learned indexes are space-efficient as they can only store models. However, updatable learned indexes usually leverage buffer techniques to accommodate updates, which requires more space. Creating an efficient learned index for stream data with dynamic characteristics demands a careful balance between those trade-offs.

In this study, we aim to design and implement ideal stream indexes, which should be optimized for both trade-offs. In the next sections, we will examine characteristics of the updatable learned index and the baseline learned indexes involved in our study.

Characteristics of Updatable Learned Index

Influenced by traditional indexes and RMI, updatable learned indexes usually maintain a hierarchical structure of internal and leaf nodes. Figure 2.9 shows that internal nodes store models that are used to quickly locate leaf nodes. The leaf nodes are also known as data segments in which models and data are stored.

There are two approaches to constructing an updatable learned index, namely top-down and bottom-up [157]. The top-down method starts with creating the root, partitioning the data among child nodes and processing these divisions recursively. The top-down approach supports the direct subspace lookup since models in each internal node can directly locate the corresponding child node based on the same method used in the partition. The bottom-up method begins by segmenting the data, using information from the child segment, such as the first key, to construct the internal nodes upwards toward the root [144]. Direct subspace lookup does not work effectively with bottom-up construction since the model trained on the child's information cannot predict the exact position due to the existence of the model error. Therefore, an extra search is necessary for the correct traversal.

Table 2.2: Baseline updatable learned indexes

Index	Bulkload	Structure Modification	Lookup Model	Update Mitigation	Concurrency
Dynamic PGM	Bottom-up	Error-based	Linear Model	Index-level Buffer	No
ALEX	Top-down	Cost-based	Linear Model	Spare Structure	No
Xindex	Bottom-up	Error-based	Linear Model	Node-level Buffer	Yes
FINEdex	Bottom-up	Fullness-based	Linear Model	Data-level Buffer	Yes

PGM

The PGM index [56] is based on a tree structure with error-bounded linear regression at each node. It uses bottom-up bulk loads and constructs nodes through an optimal Piecewise Linear Approximation (PLA) algorithm with user-defined error. As suggested by its theoretical work, PGM can guarantee reliable performance. To cope with updates, a dynamic PGM, which incorporates a log-structured merge tree (LSM-tree) structure, is proposed. An LSM-tree periodically merges multiple levels of trees to handle update operations efficiently. In dynamic PGM, multiple smaller trees serve as delta buffers, temporarily storing new entries until they reach capacity. This buffer technique facilitates the updates' performance but necessitates searching across multiple buffer trees.

ALEX

ALEX [49] is another advanced updatable learned index based on a hierarchical structure. ALEX adopts a top-down approach for the construction, in which a fanout tree is used to determine how to split downwards recursively. This design prioritizes data distribution over tree balance. To efficiently ingest newly inserted keys, each node uses a spare structure, a gapped array. The distinctive feature of ALEX is its cost models; most operations and mechanisms within ALEX are driven by cost considerations, allowing ALEX to adapt dynamically to dynamic data distributions and workloads.

Xindex

Xindex [145] is the first concurrent learned index, which features bottom-up construction and error-based modification. Xindex adopts the RMI framework, allowing an internal node to quickly locate the leaf node. In the leaf node, it introduces concepts of groups to incorporate its buffer system. Each group includes a data array and delta index for inserted data; this delta index acts as a node-level buffer with concurrency support. Furthermore, Xindex implements concurrency controls such as writer-writer coordination and writer-reader coordination to improve performance in high-concurrency environments.

FINEdex

Like Xindex, FINEdex [98] is tailored for high-concurrency environments, sharing a similar structure and bottom-up construction. However, FINEdex focuses on the data-level buffer instead of the node level in Xindex, which enables more granular control for concurrent read and write at the same node. The data-level buffer also helps maintain model accuracy, minimising extensive data shifts. However, it introduces storage inefficiency and complex micro-management such as fullness-based merge and data-level rebalance.

2.4.4 Stream Index

As mentioned in Section 2.3.1, the early systems often prioritized response speed over accuracy. Thus, during that development period, there was minimal focus on stream indexes that accurately retrieve data with a certain overhead. Despite this, there was still notable research on index techniques tailored to support continuous queries under the influence of database research. One significant advancement during the first-generation stream systems period was the development of indexing methods for sliding windows over online data streams. [69] employs and evaluates data structures for accelerating query processing within the sliding window. Those data structures include queues, linked lists, trees, and hash tables.

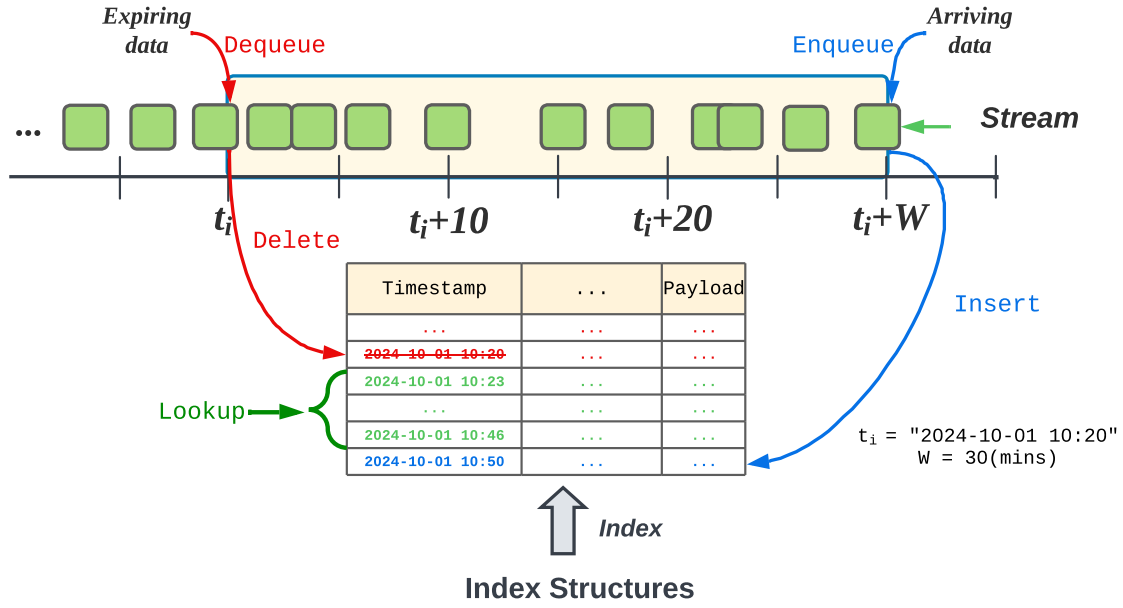


Figure 2.12: Demonstration of an Index-based Window Processing (IBWP) workload.

With advancements in stream systems, modern stream processing applications introduce growing complexity and requirements. One requirement is the need for fast and exact query results, which necessitates stream indexes to handle stream window processing, known as index-based window processing (IBWP). IBWP can be implemented using various types of windows as introduced in Section 2.2.4. Figure 2.12 shows an example of a sliding window where the system receives records from a stream and stores the most recent ones within a specified time frame, W (e.g., seconds). The window indexes the latest records from the past W time units in this diagram. The index needs to effectively manage stream windows to accelerate stream analytics and operations such as stream aggregation and stream joins [159, 137]. Several indexes have been implemented under the scope of IBWP, including SWST [140] and d Trails-tree [109, 110]; this research will focus on the Queue and Tree-based index structures.

Queue

The queue is a simple yet effective index structure for stream data. It follows the FIFO (first in, first out) ordering principle, making it ideal for stream processing in which the oldest data is typically the first to expire. Queue's update operation includes enqueue and dequeue. Enqueue inserts new data into the tail of the queue, while dequeue occurs only at the front of the queue.

A circular queue is an optimized version of the queue which can maximize the reuse of space but requires the prior of the maximum queue size. In practice, while queues maintain the order of data arrival, they do not guarantee the order within the data itself, rendering them unsuitable for search methods necessitating data monotonicity, such as binary search, since the keys may be out-of-order.

IMTree

```
SELECT *  
FROM R, S  
WHERE ABS ( R.x - S.x ) <= 10
```

Query 2.5: Stream join query

IMTree (in-memory merge-tree) is designed to accelerate stream joins, exemplified by Query 2.5 [137]. In this example, two indexes maintain two streams; keys from stream S are inserted into the index upon arrival, and stream R is searched to satisfy the join conditions. The same process applies to new keys arriving from stream R.

IMTree consists of a large search tree and a small insertion tree. The large search tree adopts an immutable B+Tree structure, organized in a breadth-first manner. This design reduces the tree heights and speeds up the lookups frequently triggered in this tree. Additionally, expired data is implicitly deleted by flagging. The small insertion tree is a standard B+Tree that acts as a buffer for frequent insertion. Once the insertion tree fills up, both trees are merged, and flagged data will be physically removed during the merge. This dual-component design effectively separates tasks, allowing each component to be optimized for specific functions.

Chapter 3

Optimization of Stream-based Workflow System

3.1 Overview of dispel4py

This chapter addresses the first research question by implementing system-level optimizations, including scaling techniques and supporting stateful operation specifically for dispel4py. dispel4py is a stream system written in Python for scientific workflow computation, which can handle unlimited data flows with lower latency [58, 45]. It offers different mappings for several widely-used enactment engines, allowing deployment across diverse computing platforms, including HPC and Cloud Computing, as depicted in Figure 3.1.

Generally, to leverage dispel4py, users simply need to construct workflows (abstract workflow) with different processing elements (PE). Then, the system automatically maps the abstract workflow to a concrete workflow that the computing infrastructures can directly execute. Once the concrete workflow is constructed, stream data is passed through instances of PEs and executed correspondingly.

In detail, the dispel4py incorporates DISPEL (Data-Intensive Systems Process Engineering Language) [112] concepts into the Python framework:

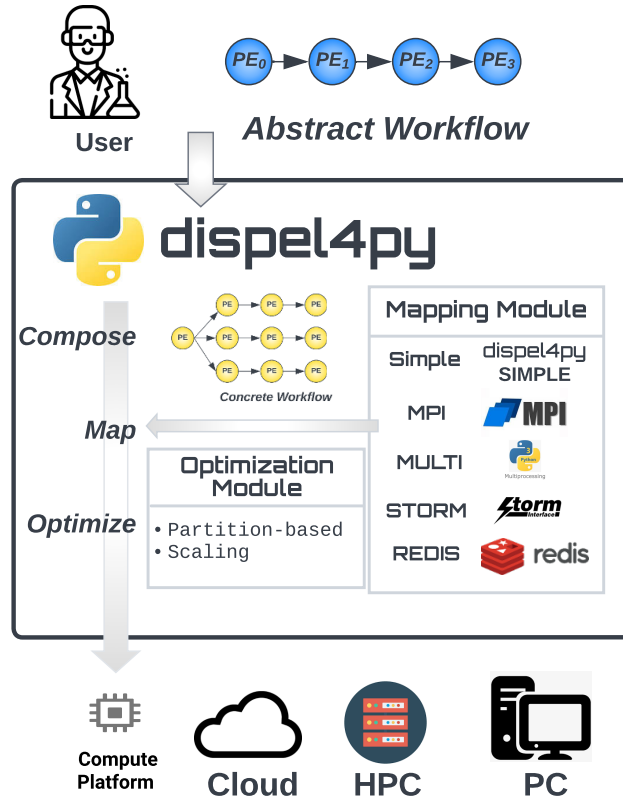


Figure 3.1: dispel4py overview.

- *Processing elements (PEs)*: are the fundamental components of dispel4py, representing computational entities responsible for task processing or data transformation within the workflow graph. In dispel4py, stream data is seamlessly exchanged between connected PEs. There are various types of PEs, which are out of the scope of this work; for details, please refer to [58].
- *Instance*: is an executable copy of a PE. A single PE can have one or more instances depending on the resource allocation.
- *Connection*: indicates how stream data flows from one PE instance's output port to one or more input ports of other instances. The data consumption and production rates could be misaligned depending on the source and destination PEs, necessitating buffers for unprocessed data.
- *Abstract workflow*: is a directed acyclic graph (DAG) consisting of PEs defined by users for their tasks. Similarly, in Apache Spark, users also define tasks or transformations that form a DAG to manage and execute workflows [135].

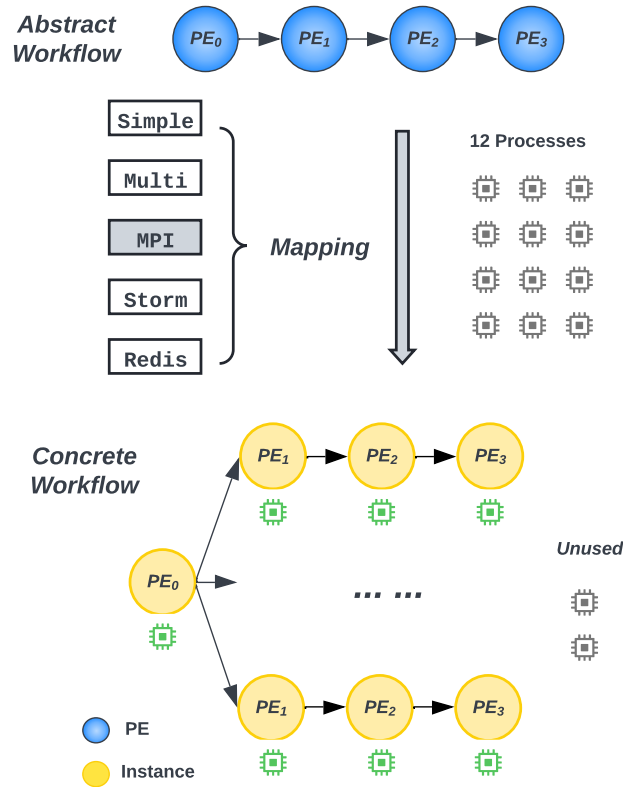


Figure 3.2: Example of the default workload allocation allocation using MPI with 12 processes.

- *Concrete workflows*: is an executable workflow in the form of DAG, which is the actual workflow executed by the compute infrastructure.
- *Mapping*: translates abstract workflow (that the user inputs) to a concrete workflow and then allocates computing resources to this concrete workflow. Mapping supports several enactment engines, including *Simple*, *MPI*, *Storm* and *Multiprocessing*. Except for the *Simple* mapping, where the concrete workflows will be executed serially, all other mapping support parallelism. *MPI* is mainly used in supercomputing environments, while *Storm* plays an important role in distributed computing. *Multiprocessing* is based on Python multiprocessing, which allows concrete workflows to be executed on the shared-memory infrastructures. To enrich the mappings, we have extended execution engines with Redis with additional optimization techniques, which will be discussed in Section 3.4.

3.2 Motivation

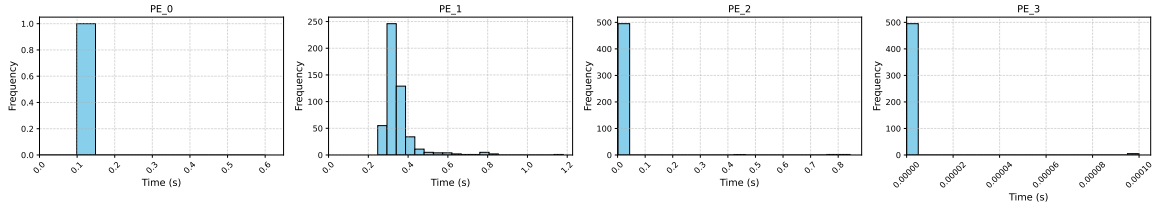
3.2.1 Motivation for dynamic allocation

The default mapping of dispel4py from abstract to concrete workflows is **rigid**. The mapping process is simply determined by the number of PEs in the abstract workflows and the number of available processes. The rules for process assignment are as follows: available sources are equally distributed among all PEs except the first. The first PE (PE_0) is only assigned to one instance and usually acts as a stream data dispatcher with a relatively low computing workload. The single instance can ensure an effective data integrity check. Specifically, as shown in Figure 3.2, the abstract workflow has 5 PEs, and there are 12 available processes; one process is assigned to PE_0 and $3 (\lfloor \frac{12-1}{4-1} \rfloor)$ processes are assigned to other PEs , leaving two processes unused. Such resource allocation does not account for data workload and communication overhead, leading to inefficiencies in performance and resource utilization.

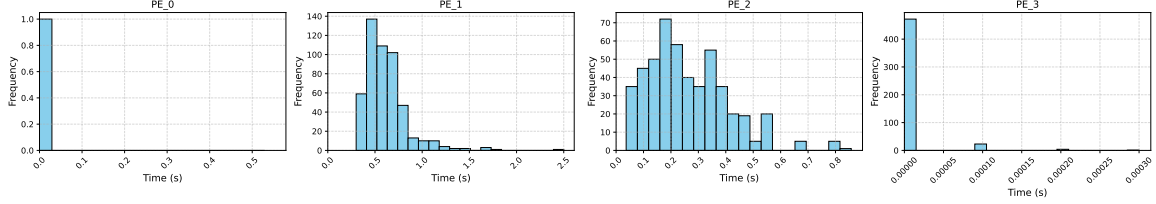
Moreover, the native allocation in dispel4py is **static**, meaning that once PEs are assigned to resources, their execution is fixed; a particular PE can only be executed by the number of assigned resources. However, workload skewness and variance are common phenomena in stream processing engines [52]. When massive stream data flood into a system for processing and analysis, even slight changes in the distribution of the incoming data stream may significantly affect the system performance. Our previous static mapping do not take into account changes in the data stream, potentially generating suboptimal performance when the evolving workload deviates from the expectation.

One way to measure the potential suboptimal performance of static allocation is to measure idle time under different workloads. We used the same example abstract workflow in Figure 3.2, consisting of 4 PEs, and applied different workload types to this workflow. We measured 500 inputs for each workload, observing that PEs with different workloads exhibited varying execution times.

In the uniform workload scenario, the PE receives a similar workload. As shown in the histogram



(a) Workflow illustrating a more uniform distribution of tasks for PE.



(b) Workflow demonstrating a more skewed distribution of tasks for PE.

Figure 3.3: Comparison of Workflow Distributions in Processing Elements (PEs): (a) **Uniform Distribution** and (b) **Skewed Distribution**. In both distributions, the first PE (PE_0) is assigned a consistently low workload and is executed only once, resulting in similar behavior across both figures. The last PE (PE_3) also maintains a relatively low and consistent workload in both distributions. The primary difference lies in PE_1 and PE_2 .

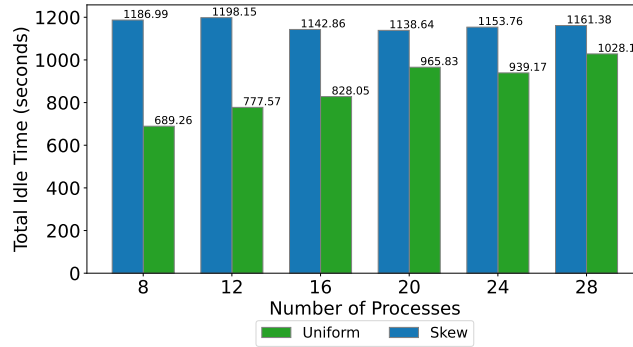


Figure 3.4: Total idle time for different workload types.

in Figure 3.3a, the execution times for PE_1 are concentrated within the range of 0.2 to 0.5 seconds. This uniformity is even more pronounced in PE_2 compared to the skewed workload scenario depicted in Figure 3.3b. In the skewed workload, PE_1 and PE_2 exhibit a broader range of execution times, indicating greater unevenness and potential performance degradation due to tail latency.

Directly measuring the total execution time for workflows with different workloads can be unfair and misleading. This is because, under different workloads, the PEs inherently require varying amounts of processing time to execute their tasks. Consequently, the total execution time is

directly influenced by the nature of the workloads assigned. To ensure a fair performance comparison between different workload types, we focus on measuring another critical contributor to total execution time: the idle time of all processes. Idle time reflects periods when PEs are not actively processing tasks, indicating inefficiencies in resource utilization. From Figure 3.4, the static mapping approach in dispel4py shows that idle time in skewed workloads is consistently larger than in uniform workloads. The reason is that fixed allocation of PEs to computing resources does not consider real-time execution demands. Consequently, when a PE instance has a heavier execution load, subsequent instances must wait instead of dynamically reallocating resources or performing work stealing. An interesting finding is that as the number of processes increases, the uniform workload exhibits a clear increasing trend in idle time, indicating that fixed allocation performs poorly when more resources are provided.

To address this issue, we propose the dynamic scheduling allocation called *dynamic allocation*. In this case, processes are not locked to specific PEs, scheduling PE instances on-the-fly, meaning that if a PE needs more or less “resources”, this technique dynamically scale up or down, re-balancing automatically graph without stopping the workflow execution. This technique allows users to design workflows without any restriction of resources such as the maximum number of threads supported by a single node, and it can deal with workload skewness.

3.2.2 Motivation for auto-scaling allocation

The dispel4py is particularly important for monitoring applications related to environmental hazards, such as earthquakes, flooding, and fires [58]. In these applications, data is continuously collected from sensors. Most of the time, nothing significant occurs, requiring minimal processing power. However, when sensors detect events (e.g., seismic activity, heavy rainfall, fires), the system needs to process a large influx of data, necessitating additional computing power. After the events subside, processing demands decrease again. Auto-scaling in dispel4py becomes essential in dynamically adjusting resources to handle these fluctuations efficiently.

For applications that consistently require a certain number of processing resources, we have observed that efficiency can decay when allocating more resources. Efficiency evaluates how

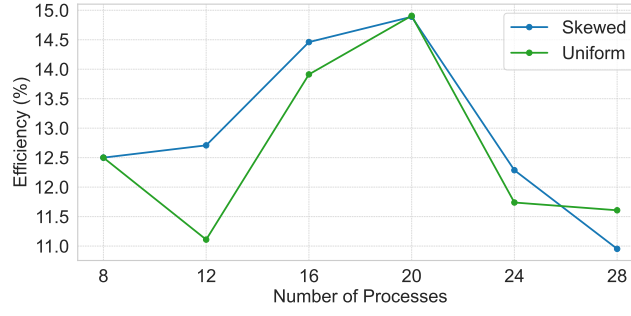


Figure 3.5: Efficiency against number of processes for different workload types.

well parallel processes utilize computational resources. It is defined as the ratio of speedup to the number of processes, indicating the percentage of the theoretical maximum speedup achieved. Speedup (S) is calculated as:

$$S = \frac{T_1}{T_N} \quad (3.1)$$

where T_1 is the execution time with one process and T_N is the execution time with N processes. Efficiency (E) is then:

$$E = \frac{S}{N} = \frac{T_1}{N \times T_N} \quad (3.2)$$

As shown in Figure 3.5, as the number of processes increases from 8 to 28, both different workload types achieve the highest efficiency at around 20 processes. Allocating more resources beyond this point results in a decay of efficiency.

Both application requirements and performance requirements necessitate extending dynamic scaling by enabling auto-scaling in a workflow-wise manner instead of PE-wise (*dynamic allocation*). Therefore, we implement *dynamic auto-scaling*, which aims to avoid underutilization or over-provisioning during dynamic mapping, achieving cost-effectiveness through considerations of monetary and energy costs.

The *dynamic auto-scaling* uniquely extends dispel4py’s mapping mechanisms to handle stream-data driven workloads, differing from existing auto-scaling techniques used in cloud computing. Traditional auto-scaling methods in cloud environments primarily focus on adjusting virtual machines or containers, based on generalized performance metrics like memory consumption or network I/O. Unlike traditional methods that operate at a high abstraction level, *dynamic auto-scaling* delves into the granularity of PEs within the dataflow graph, enabling dynamic and intelligent mapping of tasks based on real-time data characteristics.

3.2.3 Motivation to restore the supporting stateful computing in dynamic dispel4py

PEs in dispel4py can exhibit either stateless or stateful behaviour. A stateless PE relies solely on its current input for execution, while a stateful PE requires information from previous execution and influences subsequent processing. A workflow is considered stateful if it includes at least one stateful PE; it is regarded as stateless if all PEs are stateless. In dispel4py, stateful applications often involve a combination of both stateful and stateless PEs, introducing complexities due to the need to ensure state consistency across stateful PEs.

In native dispel4py, statefulness is guaranteed by the exact routing of data among stateful PEs. States flow with the data and contribute to execution as they pass through different processing units in a predefined order along a static path. This static assignment poses significant challenges when introducing dynamic allocation mechanisms such as *dynamic allocation* and *dynamic auto-scaling*. Specifically, the fixed execution paths no longer exist, disrupting the state flow and consistency. As a result, the dynamic allocation leads potential inconsistencies and state corruption.

State management in dispel4py is unique compared to most existing state management approaches in other stream processing systems such as Spark Streaming and Flink. These systems utilize checkpointing mechanisms, including snapshotting, primarily designed for fault tolerance and state recovery [86]. They manage state per task, allow querying at different time intervals,

and importantly, assume ordered data execution, aided by mechanisms like watermarking to handle late-arriving data.

In contrast, in dynamic dispel4py, state management is solely concerned with state migration, specifically directing states to the correct destinations, rather than supporting fault tolerance or state querying. Furthermore, since the order of execution no longer exists in dynamic dispel4py, traditional checkpointing becomes ineffective for maintaining state consistency.

To restore support for statefulness in dynamic allocation scenarios, we introduce *hybrid*, a hybrid approach that addresses the limitations of native dispel4py’s state management. The *hybrid* ensures state consistency by firstly fixing stateful PEs to specific processes, and then states required by stateful PEs are directly associated with the data being processed, ensuring that state is correctly maintained when PEs are dynamically allocated. By ensuring state consistency and supporting both stateful and stateless PEs, *hybrid* enhances dispel4py’s flexibility and performance.

In essence, this chapter aims to enhance stream processing through system-level optimization in dispel4py, including scaling and state management techniques which are integrated with Multiprocessing and Redis Streams. The project’s source code is available at https://github.com/NoPuzzle/dispel4py_autoscaling.

3.3 Background

3.3.1 Redis

Redis [51] is as an open-source, in-memory data structure store, renowned for its efficiency in managing a diverse array of data tasks. Supporting various data structures such as strings, lists, sets, and hashes, Redis has found widespread application in caching, real-time analytics, and messaging systems due to its rapid data access and low latency [128]. Its minimalist yet high-performance design philosophy enables it to handle substantial data volumes seamlessly. Notably, Redis offers advanced functionalities like replication, clustering, and pub/sub

messaging, bolstering its adaptability and robustness for modern data-intensive applications.

Redis Stream, introduced in Redis 5.0 in 2018, represents a significant advancement in managing real-time data streams [127]. This new data structure is specifically designed to handle streams of data by storing a series of messages, each associated with a unique ID typically based on the message's timestamp. This approach not only allows for precise time-based data retrieval but also optimizes the handling of high-throughput streams. A key feature of Redis Stream is its utilization of a trie-based indexing system for these timestamps, which significantly enhances the efficiency of data operations such as inserts, searches, and deletions.

Moreover, Redis Stream supports consumer groups, an essential feature for real-time data processing applications [129]. Consumer groups allow multiple consumers to participate in the processing of the same stream concurrently, where each consumer can track its progress independently. This feature is particularly beneficial for distributed applications, where workload distribution and fault tolerance are critical. Each message in a stream can be processed by any consumer in the group, and the state of message processing is maintained within Redis. One of the states is the tracking of idle time, which measures how long a consumer or a message has been inactive.

Redis Stream excels in scenarios requiring reliable and scalable event logging, message brokering, and stream processing. It can efficiently handle append operations, range queries, and potential data recovery needs, making it highly suitable for applications that demand robustness and high availability. In practice, Redis Stream showcases high performance in real-time processing [155]. Integrating Redis Stream with dispel4py could be conducive to improving the scalability and statefulness of dispel4py.

3.3.2 Related Work

Within this subsection, we explore the relevant literature with a particular focus on two key dimensions of optimization: scaling approaches and the management of stateful applications. This discussion complements and extends the foundational concepts introduced in Chapter 2,

delving deeper into the complexities and specific strategies that enhance both scalability and state management in dynamic environments.

Dynamic allocation can rebalance the workload of scientific workflow on-the-fly, meaning that if a task needs more or fewer resources, it dynamically scales up or down without stopping the workflow execution. [141] offers a comprehensive taxonomy of dynamic scheduling policies, categorizing them according to available resource and task information: status, processing speed, task length and communication data size, acquired a priori, at run-time or remaining task. This taxonomy helps to identify the type of information that can enhance scheduling decisions in different scenarios. However, there is no “silver bullet”; different systems necessitate different dynamic scheduling approaches, tailored to their specific operational needs and the nature of their workflows.

Auto-scaling mechanisms find application in various domains: (1) cloud service providers [126] like AWS, Google Cloud, and Azure offer auto scaling for efficient resource allocation of VM instances while ensuring optimal user experience; (2) Databases [68]: Amazon Aurora and Azure Cosmos DB leverage auto-scaling to maintain peak performance amidst varying workloads; (3) Workflow systems: Tools like Celery [33] possess auto-scaling capabilities, which can be extended with external tools. Although dispel4py and Celery share the domain of workflow systems, their distinct objectives lead to different considerations when integrating autoscaling. dispel4py targets stream data flow applications, while Celery caters to task-based ones.

Stateful computing in distributed computing remains a popular topic. There are multiple areas that concern statefulness: (1) Traditional relational databases such as MySQL and PostgreSQL support states across transactions. (2) NoSQL databases such as MongoDB [77] support states across nodes, ubiquitous in modern web and mobile application demands. (3) Middleware and Message Brokers: systems such as RabbitMQ maintain stateful information about messages, ensuring reliable and ordered message delivery. (4) Stream processing frameworks: Checkpointing is a prevalent strategy for state management. Global snapshot is a state-of-the-art periodic checkpointing solution which captures the holistic state of execution. For instance, Apache Flink has introduced a sophisticated distributed asynchronous snapshot mechanism

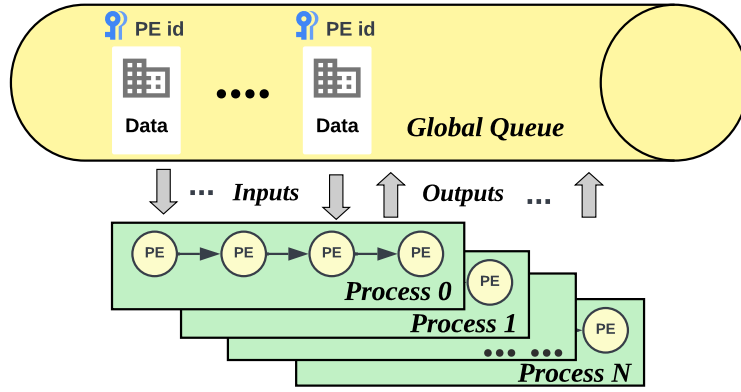


Figure 3.6: The *dynamic allocation* optimization. Each process can freely access the global queue to update data and tasks.

[43]. This ensures a low-cost state management mechanism; however, it depends on Apache Flink’s distinct data flow, which guarantees the ordering. Another check-pointing method for state management is the localized state used by Apache Storm Trident [85]. Both checkpointing methods serve different goals than state management in dispel4py. This distinction is evident in that the checkpoint mechanism, including snapshotting, cannot solve the problem of statefulness in dynamic allocation in dispel4py since existing mechanisms require ordering of data to maintain correctness. However, dispel4py cannot guarantee such ordering in dynamic scaling.

3.4 Optimization towards Scalability and Statefulness

This section introduces our efforts towards two primary objectives: enhancing the scalability of dispel4py by implementing *dynamic allocation* and *dynamic auto-scaling* and supporting statefulness within these optimizations.

3.4.1 The *dynamic allocation* Optimization

We delve into the details of dynamic allocation for two mappings: one implemented with Python’s Multiprocessing for handling concurrent processes in a shared-memory environment, and the other using Redis for achieving high-performance, distributed caching and messaging in a networked setting. Both implementations follow the same architectural pattern. When

a dispel4py workflow is executed using the *dynamic allocation*, all processes receive a copy of the abstract workflow at the beginning of the workflow execution, as shown in Figure 3.6. The *dynamic allocation* uses a global queue as a buffer to store PEs and upcoming stream data, which is available to all processes. The main idea is that each process goes to the global queue as soon as it is free to execute the next PE along with the necessary data. After finishing the execution of a PE, it returns the output to the global queue. In Python Multiprocessing, this global buffer is represented by a queue [125], while in Redis, it is implemented through Redis Stream. The communication pattern, condition, and process behaviour are consistent across both implementations.

Algorithm 1 demonstrates the *dynamic allocation* process within the `DynamicAllocator` class, which initializes with predefined attributes like the workflow graph, task queue, and processor count. The `InitiateProcessing` procedure activates upon receiving input task consisting of PE and data, processing task and adding resulting data with PE to the task queue. The core of the operation, `WorkerProcess`, perpetually scans this queue, pulling tasks for processing via the `HandleCommunication` method. This method processes each task based on the current PE's function, checks for subsequent destinations using `DetermineDestinations`, and re-queues the output if further processing is required by connected PEs in the workflow graph. The processing loop persists until the queue empties and 'poison pills' are broadcast, signaling no further tasks are pending.

Termination Strategy

Ideally, stream data is unbounded. However, we still need to design a termination strategy in practice. Initially, dispel4py employs the "*poison pills*" termination method, in which the instance of source PE signal the end of the input to all subsequent. Following the fixed execution path, all instances will exit accordingly. However, with the dynamic allocation, the processing orders are based on the availability of processes instead of specific orders in the abstract workflow. Therefore, there is no guarantee on task processing order, which makes the "*poison pills*" method ineffective, as it can unexpectedly halt processes and leave tasks unfinished in

Algorithm 1 *dynamic allocation*

```

1: Class DynamicAllocator:
2:   Attributes : Workflow  $G$ , Task Queue  $Q$ , Processor Count  $size$ 
3:
4:   procedure CONSTRUCTOR( $G, Q, size$ )
5:     Initialize member parameters  $G, self.size$ 
6:     Assign task queue to  $self.Q$ 
7:   end procedure
8:
9:   procedure DETERMINEDESTINATIONS( $process$ )
10:    Initialize destinations list  $dest\_list$ 
11:    Search  $self.G$  to identify potential destinations for  $process$ 
12:    if  $dest\_list$  is empty then
13:      return NULL
14:    else
15:      return  $dest\_list$  first element
16:    end if
17:  end procedure
18:
19:  procedure HANDLECOMMUNICATION( $process, PE, data$ )
20:     $task \leftarrow PE.process(data)$ 
21:     $dest \leftarrow self.DetermineDestinations(process)$ 
22:    if  $PE$  has  $dest$  then
23:      Requeue  $task$  into  $self.Q$ 
24:    end if
25:  end procedure
26:
27:  procedure WORKERPROCESS
28:    while True do
29:      if  $self.Q$  is not empty then
30:        Extract a task  $task$  from  $self.Q$ 
31:        Invoke  $self.HandleCommunication(task.process, task.PE, task.data)$ 
32:      else
33:        Broadcast "poison pills" ▷ Exit if CheckTermination == True
34:      end if
35:    end while
36:  end procedure
37:
38:  procedure INITIATEPROCESSING( $task$ )
39:    for each processing element  $task.PE$  in  $self.G$  do
40:       $processedData \leftarrow process(task.data)$ 
41:      if  $processedData$  is not NONE then
42:        Insert  $processedData$  into  $self.Q$ 
43:      end if
44:    end for
45:    Activate  $self.WorkerProcess$  for each processor in the pool
46:  end procedure
47: end Class

```

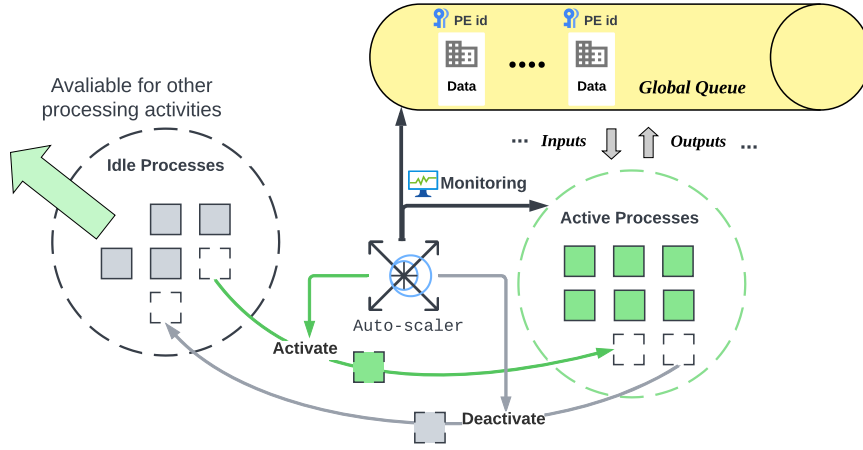


Figure 3.7: The *dynamic auto-scaling* optimization. The auto-scaler can dynamically activate or deactivate processes based on the monitoring framework.

the queue.

To address this, we use a retry mechanism along with “*poison pills*”. If the queue appears empty, processes wait for a configurable threshold duration and retry a specified number of times before deciding to terminate. If the waiting threshold is reached, the process broadcasts the “*poison pills*” to other processes, speeding up their termination check and thereby reducing overall waiting time.

3.4.2 The *dynamic auto-scaling* Optimization

While *dynamic allocation* maximizes resource utilization by keeping processes actively communicating with the buffer or handling tasks, it does not optimize for energy efficiency. To improve cost efficiency, *dynamic auto-scaling* is proposed to enable dispel4py to be responsible for workload spikes and reduce resource wastage during low workload periods.

There are two process statuses in *dynamic auto-scaling*: active and idle, as illustrated in Figure 3.7. Active processes are assigned with abstract workflow and participate in task execution, which involves retrieving data from the queue, executing tasks, and returning results to the queue in the same manner as the *dynamic allocation* optimization. Unlike *dynamic allocation*, during periods of low work demand (e.g., when the queue has a low task workload), active processes transition to an idle state instead of continuously accessing the queue. The idle pro-

cesses operate in a standby mode, offering the potential for reallocating those idle processes elsewhere (e.g., another workflow utilizing auto-scaling optimization). When the workload increases, *dynamic auto-scaling* reactivates idle processes to facilitate execution. The auto-scaler orchestrates this dynamic activation and deactivation process with a monitoring framework and various auto-scaling strategies.

Algorithm 2 shows how the auto-scaler works. It sets parameters like the *max_pool_size* and *workload_threshold* upon initialisation; by default, the *active_size* is half of the maximum number of processes (*max_pool_size*). The configurable *workload_threshold* is used in the auto-scaling strategies. The **shrink** and **grow** methods decrease and increase the active processes, respectively, and are controlled by the **auto_scale** method. **auto_scale** implements the auto-scaling strategies (introduced in Section 3.4.2), which monitor the system state and control the resource adjustments. **start** and **done** dispatch tasks and update active counts (*active_count*) to avoid over-use of processes. The entry method **process** continually evaluates and scales resources before activating processes by calling **start**.

Auto-scaling and Monitoring Strategy

Auto-scaling strategies are crucial for balancing between performance and efficiency, which need to decide: ‘when to scale’ and ‘how to scale’. The former decides if there is a workload change which requires metrics from the monitoring framework. The latter is about the magnitude of scaling. In this work, we adopt a simple incremental approach: incrementing the active size by 1 or -1.

As for when to scale, since we implement *dynamic auto-scaling* in both *Multiprocessing* and *Redis*, similar to *dynamic allocation*, we adopted two different heuristic strategies:

- In the *Multiprocessing* mapping of the *dynamic auto-scaling* strategy, the key performance metric used is the queue size, measured using the `qsize()` function. This metric reflects the number of tasks pending in the queue, providing a direct indicator of the current workload. An increase in queue size suggests a rising workload, necessitating an augmen-

Algorithm 2 AutoScaler

```

1: Class AutoScaler:
2:   Attributes : Maximum pool size max_pool_size, Pool pool, Threshold threshold, Task queue
   queue, Active size active_size, Active count active_count
3:
4:   procedure CONSTRUCTOR(max_pool_size, pool, threshold, queue)
5:     Initialize parameters with provided values
6:     Set active_size to half of max_pool_size by default
7:   end procedure
8:
9:   procedure REDUCEACTIVESIZE(size_to_shrink)
10:    Decrease active_size by size_to_shrink, ensuring it does not fall below 1
11:   end procedure
12:
13:   procedure INCREASEACTIVESIZE(size_to_grow)
14:    Increase active_size by size_to_grow, capped at max_pool_size
15:   end procedure
16:
17:   procedure CHECKTERMINATION
18:    Return a Boolean value based on termination conditions
19:   end procedure
20:
21:   procedure AUTOSCALE
22:    Monitor current state and store in curr_states
23:    if curr_states exceeds threshold then
24:      INCREASEACTIVESIZE(1)
25:    else
26:      REDUCEACTIVESIZE(1)
27:    end if
28:   end procedure
29:
30:   procedure STARTTASK(func, args)
31:    while active_count is greater than or equal to active_size do
32:      Wait until resources are available
33:    end while
34:    Increment active_count
35:    return pool.apply_async(func, args, callback=TASKCOMPLETED())
36:   end procedure
37:
38:   procedure TASKCOMPLETED(result)
39:    Decrement active_count
40:   end procedure
41:
42:   procedure PROCESSTASKS(graph)
43:    while True do
44:      AUTOSCALE
45:      if CHECKTERMINATION then ▷ Check if termination criteria are met
46:        Collect and return results
47:      else
48:        Copy the graph to cp_graph
49:        Set up arguments for the worker
50:        STARTTASK(worker.process, args)
51:      end if
52:    end while
53:   end procedure
54: End Class

```

tation of resources to maintain system performance. Conversely, a reduction in queue size indicates a decrease in workload, allowing for a decrease in the number of active resources. A minimum threshold (5 by default) for `qsize()` is also set to prevent over-allocating processing resources during periods of low demand.

- For the *Redis* mapping, the *dynamic auto-scaling* strategy utilizes the average idle time of the processes, accessed via `xinfo_consumers()['idle']`. Unlike queue size, idle time provides a direct measure of process utilization, reflecting the amount of time processing units remain inactive. An established threshold for idle time helps determine when resources are underutilized and could be scaled down to optimize efficiency and reduce costs. This threshold is based on the average reactivation time of processes, which depends on the computational resources and the specific demands of the workflow. Therefore, the configuration of this threshold requires expert selection or fine-tuning.

The *dynamic allocation* and the *dynamic auto-scaling* can enhance performance and cost efficiency by scaling tasks with resources based on the task's need. However, they cannot be adapted naively to the current dispel4py's stateful workflow because the path for state migration is not determined during process reallocation on the fly. Therefore, we propose a novel approach in the next section to tackle this challenge.

3.4.3 The *hybrid* Optimization for Statefulness

To manage states, dispel4py natively stores the state locally within the process, which can avoid state migration overheads. This works perfectly for fixed and static resource allocation, in which each process is tightly coupled with a specific instance of PE, allowing states to be managed correctly during execution. However, introducing dynamic optimization, in which the processes and PE instances are decoupled, requires advanced state management to handle stateful applications.

We introduce the *hybrid* optimization implemented with *Redis* mapping to support both stateful and stateless applications with *dynamic allocation*. Unlike Spark and Flink, which are designed

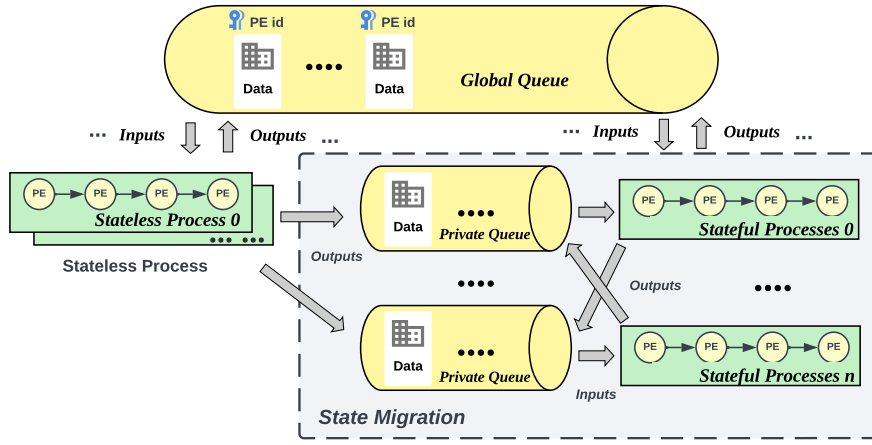


Figure 3.8: Hybrid optimization for stateful operation. Processes categorized into stateless and stateful manage corresponding PEs, optimizing workflow execution. Stateful processes maintain dedicated private queues.

with built-in fault tolerance mechanisms, dispel4py offers a more lightweight and flexible approach. This approach allows for quicker state migration and more granular control, specifically catering to the needs of hybrid workflows that require both rapid stateless processing and robust stateful operations.

The main idea of the *hybrid* optimization is directly mapping stateful PE instances to dedicated (stateful) processes. For state migration, we deploy private task input queues (called private queues in Figure 3.8) for each stateful process. Meanwhile, outputs from these stateful PE instances can be routed to different queues based on connections, reducing state synchronization and enhancing performance compared to traditional global state management approaches.

In a stateful workflow, some PE can be stateless. To maximize the advantages of the dynamic allocation strategy, the *hybrid* optimization introduces a mechanism where stateless PE instances are assigned to the available processes that are not dedicated to stateful tasks. This allocation is determined by calculating $N - \text{number of stateful PE instances}$, where N represents the total number of available processes. These stateless processes function similarly to the conventional dynamic methods, acquiring inputs from the ‘Global queue’ and returning completed tasks to it. However, a key distinction is that these stateless tasks have the capability to deposit their outputs into private queues designated for stateful tasks. These private queues facilitate the migration and buffering of stateful PEs and data, thereby ensuring the integrity

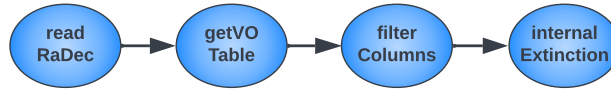


Figure 3.9: Internal Extinction of Galaxies workflow.

of computations and enhancing the efficiency of overall workflow execution.

3.5 Usecases

We introduce three workflows as use cases of dispel4py, which are used in our experimental evaluation.

3.5.1 Internal Extinction of Galaxies Workflow

The Internal Extinction of Galaxies workflow is designed to calculate the extinction metric within galaxies. The metric measures optical luminosity, reflecting the dust extinction of the internal galaxies [82]. The workflow contains four PEs, as shown in Figure 3.9. The first PE (**read RaDec**) is the source PE, which reads the coordinator data of galaxies. Then, these coordinates are dispatched to **getVO Table** to download the corresponding VOTables [84]. **filter Columns** then filters the VOTables based on specified requirements in the computation, followed by **internal Extinction**, which performs the calculation for extinction metric. All PEs in this workflow are stateless, rendering the stateless workflow.

To introduce variability in the workload, we adjust the **read RaDec** PE. For a standard workload (denoted as 1X), it reads data for 100 galaxies. This reading scales to 300 galaxies (3X), 500 (5X), and 1000 (10X). In addition to varying the stream length, we also vary the PE's workload. By using a random sleep time sampled from a $beta(2, 5)$ distribution, we add delays ranging from 0 to 1 second within the **getVO Table** PE and **filter Columns** PE. This modification is labelled as 'heavy'. Therefore, the experiment based on the Internal Extinction of Galaxies workflow now has standard and heavy workloads varying from 1X to 10X.

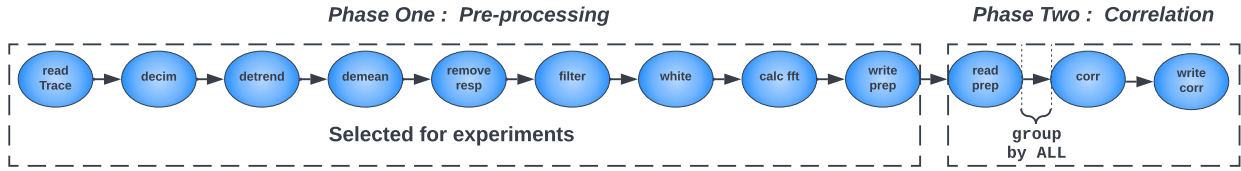


Figure 3.10: Seismic Cross-Correlation workflow.

3.5.2 Seismic Cross-Correlation Workflow

The Seismic Cross-Correlation workflow is engineered to analyze the vast real-time geological waveform data gathered from FDSN [53] stations. It aims to assess the likelihood of volcanic eruptions and earthquakes. The original workflow consists of two phases (in Figure 3.10): the first involves pre-processing data collected from stations, followed by a second phase that conducts cross-correlation analyses based on pre-processed data.

In this study, we focus on the first phase of the workflow, which comprises nine interconnected PEs: the initial PE reads the data, the intermediate PEs process the raw data, and the final PE writes the data to disk. In this experiment, there is a more imbalanced workload among PEs; for example, the intermediate PEs only do calculations in main memory, but the last PE writes data into the disk, which involves IO operations. A detailed setup description can be found in [100, 99].

3.5.3 Sentiment Analyses for News Articles Workflow

The Sentiment Analyses for News Articles workflow is based on newspaper articles [17], which applies two distinct sentiment analyses on articles, then aggregates these sentiment scores based on the published location. This workflow is more complex with various types of *grouping* compared to the first two workflows.

Figure 3.11 illustrates the workflows in which the source PE (**read Articles**) sequentially reads and parses articles from input files and outputs into two downstream PEs. The **sentiment AFINN** PE computes sentiment scores using the AFINN lexicon [117], meanwhile **tokenize WD** and **sentiment SWN3** PEs perform tokenization and sentiment analysis using the SWN3 lexicon

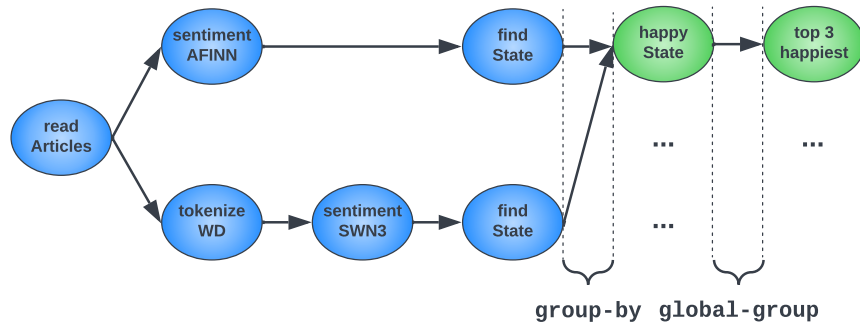


Figure 3.11: Sentiment Analyses for News Articles Workflow.

Table 3.1: Abbreviations of optimization techniques.

Optimization \ Mapping	<i>Multiprocessing</i>	<i>Redis</i>
<i>no optimization</i>	<i>multi</i>	-
<i>dynamic allocation</i>	<i>dyn_multi</i>	<i>dyn_redis, hybrid_redis</i>
<i>dynamic auto-scaling</i>	<i>dyn_auto_multi</i>	<i>dyn_auto_redis</i>
<i>stateful optimization</i>	-	<i>hybrid_redis</i>

[2]. Data from these two pathways convene within their respective `find State` - `happy State` - `top 3 happiest` sequence. These three PEs identify the top three happiest locations with their scores. The workflow is complex due to ‘group-by’ and ‘global-group’ groupings, which render it a stateful workflow. We use this workflow to experiment with the *hybrid* optimization.

3.6 Evaluation

The experimental evaluation assesses the performance and efficiency of the optimization techniques proposed in Section 3.4 utilizing the workflows outlined in Section 3.5.

3.6.1 Experiment Setup

We begin by introducing the abbreviations and key insights into these evaluated techniques shown in Table 3.1:

- *multi*: It stands for the native *Multiprocessing* mappings, which assigns PE instances to processes as detailed in Section 3.1 and 3.3. The *multi* serves as the baseline for both

stateless and stateful workflows since *multi* uses the static resource allocation, which can be effectively handled by the native state management of dispel4py.

- *dyn_multi*: It denotes the *Multiprocessing* mappings enhanced with *dynamic allocation* optimization (in Section 3.4.1). It enables processes to dynamically share the workload on the fly. However, it does not support stateful workload based on inherent dispel4py state management.
- *dyn_auto_multi*: It involves the *dynamic auto-scaling* optimization introduced in Section 3.4.2. Both dynamic scaling optimizations, *dyn_multi* and *dyn_auto_multi*, build upon *Multiprocessing* mappings, which are compared primarily in the stateless workflow to evaluate efficiency gains from the *dynamic auto-scaling* technique.
- *dyn_redis*: It represents the *Redis* mapping with *dynamic allocation* optimization discussed in Section 3.4.1. It is compared horizontally with *dyn_multi* across different mappings.
- *dyn_auto_redis*: It is the *Redis* mapping equipped with *dynamic auto-scaling*. It is compared with *dyn_redis* for the efficient gain in *Redis* mapping and compared with *dyn_auto_multi* to examine how the different implementations of the global queue and scaling strategies affect the performance.
- *hybrid_redis*: It denotes the *Redis* mapping that supports both stateful and stateless workflows, explained in Section 3.4.3. It is compared against the baseline *multi* in stateful workflows.

Platform

Experiments are conducted across multiple infrastructures/platforms to evaluate the portability and robustness of the optimization techniques. The platforms are configured as follows:

- *server*: It is a virtual server for research groups supported by the Department of Computing, Imperial College London [81]. It consists of a 16-core Intel E5-2690@2.60GHz

processor paired with 64GB of RAM and runs on Ubuntu 14.04. We denote it as *server* for short. It runs all three workflows on various processes: 4, 8, 12, and 16.

- *cloud*: It is a server provided by Google Cloud Platform [73], which has 4 Intel(R) Xeon(R) CPUs @ 2.20GHz (8vCPUs), and 16GB of RAM running Ubuntu 20.04. Its configuration for running the experiment is identical to *server*.
- *HPC*: The HPC cluster we used is provided by HPC at Imperial [80]. The HPC servers provide multi-job classes, and we use the *short* class in which the CPU and OS mode are Intel E5-2680 v3 @ 2.50GHz and Centos 8, respectively. We request up to 64 CPUs and 64 GB memory. Since Redis cannot be deployed on the HPC, no techniques based on *Redis* mapping are running on HPC. We employed 4, 8, 16, 32, and 64 cores for experiments.

Metrics

Two primary metrics are used for evaluation, namely runtime (*runtime*) and total process time (*process time*). The *runtime* represents the execution time in the real world, while *process time* reflects the efficiency by accounting for all active process duration. A workflow running on fewer processes could produce a higher *runtime*, but potentially lower *process time* from reduced synchronization overhead among fewer processes. We present pairs of *runtime* and *process time* for each experiment to provide an intuitive comparison between performance and efficiency.

In this experiment, we do not use idle time as a metric because *dynamic auto-scaling* is designed to minimize idle time by deactivating processes when they are not actively processing tasks. As a result, under the same workload, *process time* serves as a fairer comparison metric, accurately reflecting the efficiency of resource utilization without being influenced by artificially minimized idle periods.

Furthermore, we calculate the *runtime* ratio and *process time* ratio between *dynamic allocation* and *dynamic auto-scaling* to quantify the performance gains. For example, if the *runtime* ratio between *dyn_auto_multi* and *dyn_multi* is below 1, it indicates that *dyn_auto_multi* consumes less

time compared to *dyn_multi*. Similarly, we use the ratio between *hybrid_redis* and *multi* to show the performance gain in stateful workflows. To ensure consistent and intuitive interpretation, we consistently put more advanced optimization such as *dynamic auto-scaling* and *hybrid* in the numerator. If both ratios fall below 1, the advanced approach excels in both performance and efficiency, although practical trade-offs may affect ideal outcomes.

3.6.2 Evaluation on the Internal Extinction of Galaxies workflow

We use various workloads for different platforms based on Internal Extinction of Galaxies workflow, tailored to the capabilities of each platform. On both *server* and *cloud*, we use three relatively lightweight workloads: the 1X standard workload (100 galaxies as input), the 5X standard workload (500 galaxies), and the 1X heavy workload (100 galaxies with sleep synthetically added to the PEs). On *HPC*, which offers more cores, we deploy a heavier workload than *server* and *cloud*. *HPC* is experimented with 5X, 10X standard and 5X heavy workload.

Figure 3.12 demonstrates the *runtime* (left) and *process time* (right) for all six techniques running on *server*. All techniques show a decreasing trend for *runtime* with an increase in the number of processes, indicating they share good scalability. The *dynamic auto-scaling* technique with lower workloads exhibits advantages in *runtime* since it can reduce synchronisation costs by promptly deactivating unnecessary processes. However, in heavy-workload scenarios, *dynamic auto-scaling* technique lags slightly behind pure *dynamic allocation* optimizations. This occurs because all processes are committed to execution, yet the auto-scaling strategy of *dynamic auto-scaling* may attempt to deactivate some processes when there are slight decreases in workload, even though the total amount remains high. This situation could be refined with a more sophisticated scaling strategy. The *process time* rises with the increase of the number of processes, indicating that more activated processes can accumulate synchronization overhead. Unsurprisingly, both *process time* of *dyn_auto_multi* and *dyn_auto_redis* increase slightly, yet they still excel over their corresponding *dynamic allocation* competitors.

Although *hybrid_redis* is designed to support statefulness alongside *dynamic allocation* opti-

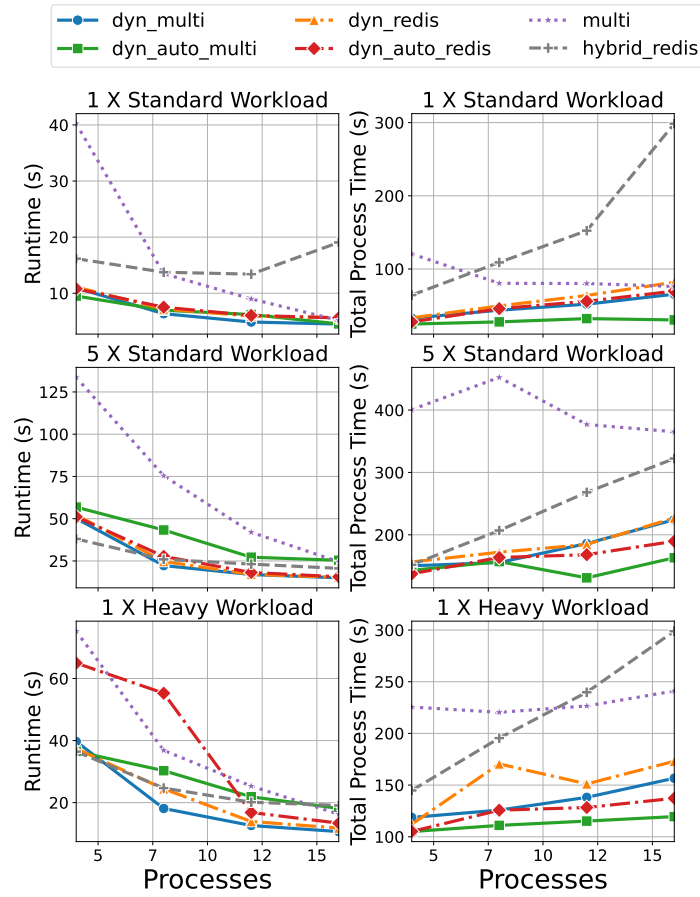


Figure 3.12: Workload Performance for Internal Extinction of Galaxies using the *server* with up to 16 processes.

mization, it achieves competitive performance in stateless workflows. The *hybrid_redis* presents strong *runtime* performance; however, it does not show a similar achievement in efficiency since *dynamic auto-scaling* was not enabled.

In terms of horizontal comparison among different mappings, we notice that there is a common pattern across various experimental setups: both *runtime* and *process time* for the optimization based on *Redis* mapping are larger than those on the *Multiprocessing* mapping. The differences in performance can be attributed to the inherent characteristics of the two mappings. *Multiprocessing* is designed to be lightweight; however, *Redis* supports more features regarding monitoring, reliable messaging, and robust data persistence, which render *Redis* more resource-intensive.

Figure 3.13 shows that performance trends on the *cloud* are similar to those observed on the *server*. As for the experiments running on *HPC* in Figure 3.14, it shows the performance (both

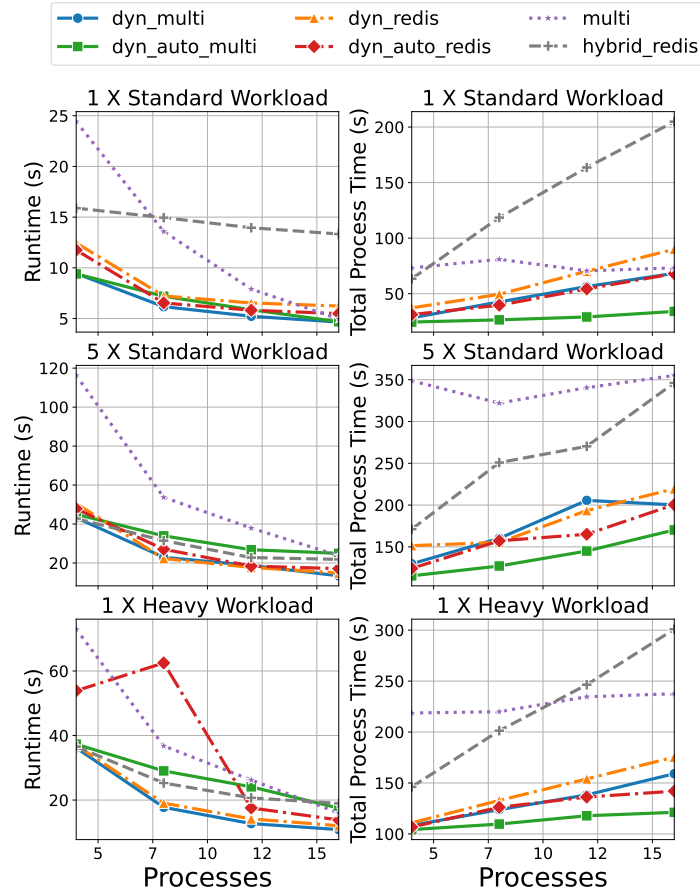


Figure 3.13: Workload Performance for Internal Extinction of Galaxies using the *cloud* with up to 16 processes.

process time and *runtime*) in a larger scale in processes. The *runtime* of all three methods based on the *multi* mapping show a quick drop when increasing the number of processes from 4 to 16, then becoming steady with a slight decrease in runtime. However, the *process time* of *dyn_multi* and *dyn_auto_multi* show a linear increase with the number of processes, with a noticeable acceleration at 16 processes, indicating overallocation.

Table 3.2 displays *runtime* and *process time* ratios between optimizations across various platforms. In the best *runtime* case, *dyn_auto_multi* requires only 87% *runtime* and 76% *process time* of *dyn_multi*. As for prioritizing *process time*, the most efficient ratio we can achieve is 0.46, with a 1.01 *runtime* ratio. This observation suggests that *dynamic auto-scaling* techniques demonstrate efficiency by slightly extending *runtime* while reducing *process time*.

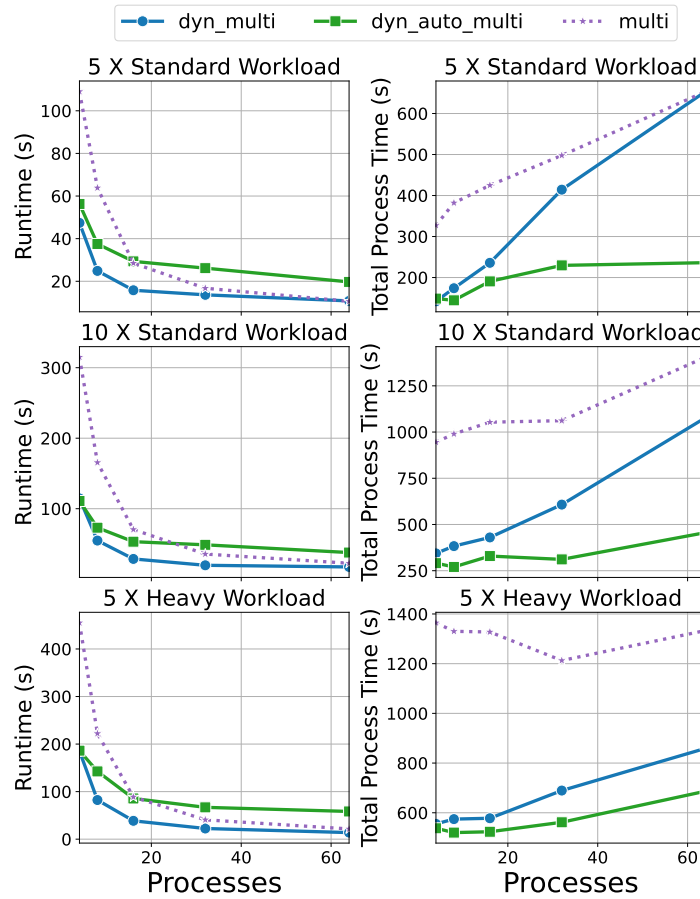


Figure 3.14: Workload Performance for Internal Extinction of Galaxies using the *HPC* with up to 64 processes.

3.6.3 Evaluation on the Seismic Cross-Correlation workflow

The workload of Seismic Cross-Correlation workflow, which involves processing input from 50 stations, is consistent across all platforms. Seismic Cross-Correlation consists of a large number of PEs and features a heterogeneously distributed workload among PEs. It is worth pointing out that *multi* need to initiate at least 9 processes for this workflow since, given the fixed allocation of *multi*, the number of PE is the minimum requirement of processes.

Figures 3.15 show a downward trend for *runtime* across all techniques and an upward trend for *process time* as the number of processes increases for all platforms. Compared with the same number of processes, Seismic Cross-Correlation's performance in *server* is better than *cloud*. These trends are similar to those observed in the Internal Extinction of Galaxies workflow. However, two differences emerge. Firstly, for *runtime*, the optimizations on *Redis* mapping

Table 3.2: Performance comparison based on the Internal Extinction of Galaxies workflow between techniques (A and B). The *runtime* ratio is the *runtime* of mapping A over the *runtime* of mapping B. Similarly, the *process time* ratio is the total process time elapsed ratio between A and B. The ratios are prioritized by the metric in the "Prioritized By" column, and the [Mean, Std] shows the average and standard deviation for all *runtime* and *process time* ratios.

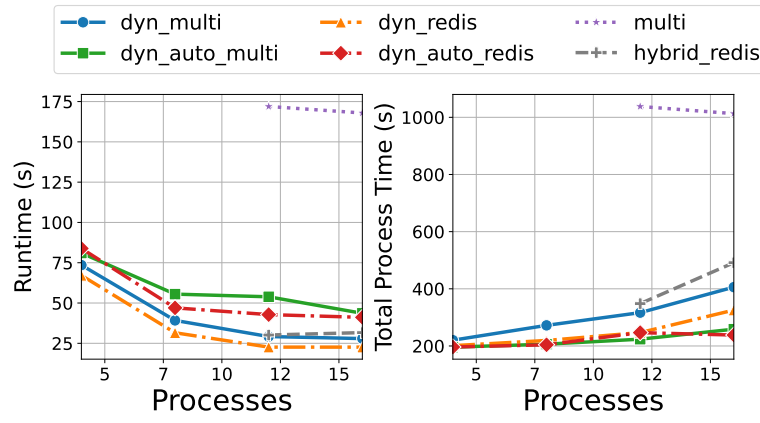
Platform	Comparison between A/B	Prioritized By	Runtime Ratio	Process Time Ratio
<i>server</i>	$\frac{dyn_auto_multi}{dyn_multi}$	<i>runtime</i>	0.87	0.76
		<i>process time</i>	1.01	0.46
		[Mean, Std]	[1.39, 0.37]	[0.77, 0.15]
	$\frac{dyn_auto_redis}{dyn_redis}$	<i>runtime</i>	0.97	0.83
		<i>process time</i>	2.26	0.74
		[Mean, Std]	[1.21, 0.39]	[0.86, 0.06]
<i>cloud</i>	$\frac{dyn_auto_multi}{dyn_multi}$	<i>runtime</i>	1.00	0.87
		<i>process time</i>	1.01	0.50
		[Mean, Std]	[1.36, 0.34]	[0.77, 0.15]
	$\frac{dyn_auto_redis}{dyn_redis}$	<i>runtime</i>	0.88	0.76
		<i>process time</i>	0.88	0.76
		[Mean, Std]	[1.26, 0.66]	[0.86, 0.08]
<i>HPC</i>	$\frac{dyn_auto_multi}{dyn_multi}$	<i>runtime</i>	0.97	0.85
		<i>process time</i>	1.83	0.36
		[Mean, Std]	[1.95, 0.83]	[0.75, 0.20]

outperform *multi*. This advantage is likely due to the *Redis* stream structure, which can effectively manage the heterogeneous workload and frequent access to the global queue. Another difference is that this workflow requires a higher number of processes to stabilize *runtime*, especially for the baseline *multi* due to its strict mapping mechanism.

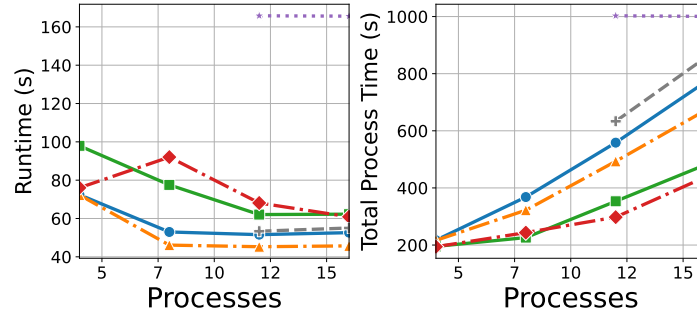
Statistics from Table 3.3 reveal that advanced optimizations do not outperform their baseline in terms of *runtime*, as the ratio generally exceeds 1 in this case. This indicates that the *dynamic auto-scaling* optimization faces challenges in complex workflows, likely due to the limitation of naive auto-scaling strategies in accurately responding to complex workflows. However, all *process time* ratios are below 1, meaning they can still achieve cost efficiency despite the challenges.

3.6.4 Evaluation on Sentiment Analyses for News Articles workflow

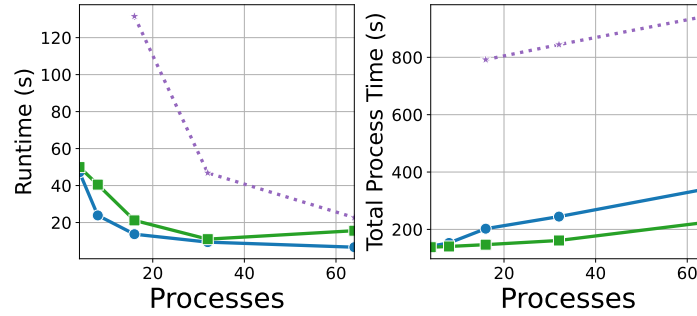
The goal of this workflow evaluation is to assess *hybrid.redis*'s performance in stateful applications compared to the baseline *multi*. The `happy state` and `top 3 happiest` stateful PEs



(a) Workload Performance for Seismic Cross-Correlation using the *server* with up to 16 processes.



(b) Workload Performance for Seismic Cross-Correlation using the *cloud* with up to 16 processes.



(c) Workload Performance for Seismic Cross-Correlation using the *HPC* with up to 64 processes.

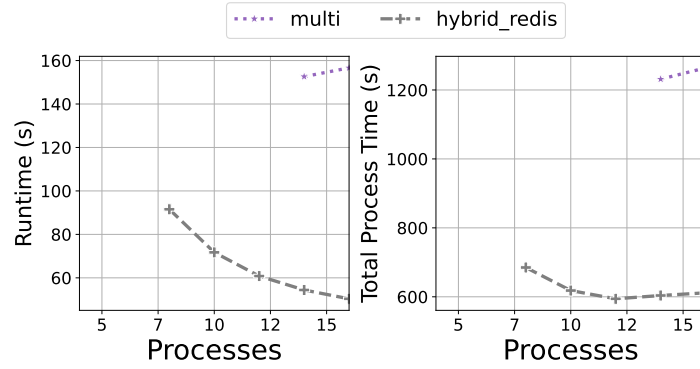
Figure 3.15: Experiments on Seismic Cross-Correlation over different platforms.

are assigned with 4 and 2 instances, respectively. With 1 process reserved for stateless PEs, *hybrid_redis* starts with 8 instances in the experiment. In contrast, *multi* demands a minimum of 14 processes due to its one-to-one instance-to-process mapping. As the upper process limit is 16, we use finer increments of 8, 10, 12, 14, and 16 processes.

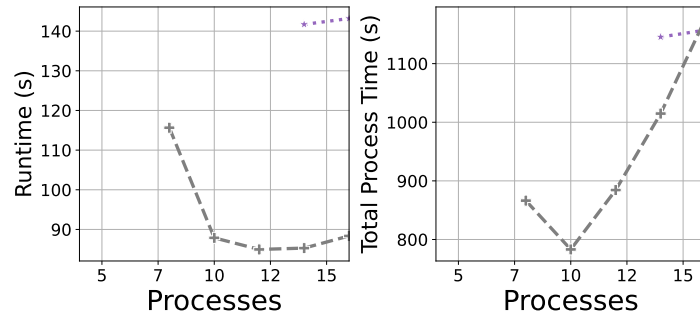
The results on the *server*, as shown in Figure 3.16a, demonstrate that *hybrid_redis* significantly outperforms the *multi* in terms of *runtime*. Additionally, *hybrid_redis* exhibits a speed-up as the number of processes increases. This can be attributed to the fact that as more processes are added, the stateless workload can be shared by more processes, thereby reducing the overall

Table 3.3: Performance comparison based Seismic Cross-Correlation workflow between *dynamic auto-scaling* techniques and their baselines.

Platform	Comparison between A/B	Prioritized By	Runtime Ratio	Process Time Ratio
server	$\frac{dyn_auto_multi}{dyn_multi}$	<i>runtime</i>	1.10	0.89
		<i>process time</i>	1.56	0.64
		[Mean, Std]	[1.48, 0.31]	[0.75, 0.11]
	$\frac{dyn_auto_redis}{dyn_redis}$	<i>runtime</i>	1.25	0.98
		<i>process time</i>	1.82	0.73
		[Mean, Std]	[1.61, 0.30]	[0.91, 0.12]
cloud	$\frac{dyn_auto_multi}{dyn_multi}$	<i>runtime</i>	1.18	0.62
		<i>process time</i>	1.46	0.61
		[Mean, Std]	[1.30, 0.13]	[0.69, 0.14]
	$\frac{dyn_auto_redis}{dyn_redis}$	<i>runtime</i>	1.05	0.90
		<i>process time</i>	1.50	0.60
		[Mean, Std]	[1.47, 0.40]	[0.73, 0.13]
HPC	$\frac{dyn_auto_multi}{dyn_multi}$	<i>runtime</i>	1.06	0.98
		<i>process time</i>	2.34	0.66
		[Mean, Std]	[1.56, 0.51]	[0.79, 0.15]



(a) Workload Performance for Sentiment Analyses for News Articles using the *server* with up to 16 processes.



(b) Workload Performance for Sentiment Analyses for News Articles using the *cloud* with up to 16 processes.

Figure 3.16: Experiments on Sentiment Analyses for News Articles over different platforms.

execution time. Unexpectedly, the trend of *process time* is similar to *runtime*. Given that *dynamic auto-scaling* is not applied into *hybrid_redis*, these efficiency gains likely come from

Table 3.4: Performance comparison based Sentiment Analyses for News Articles workflow between stateful techniques (*hybrid_redis* and *multi*).

Platform	Comparison between A/B	Prioritized By	Runtime Ratio	Process Time Ratio
<i>server</i>	$\frac{\text{hybrid_redis}}{\text{multi}}$	<i>runtime</i>	0.32	0.48
		<i>process time</i>	0.32	0.48
		[Mean, Std]	[0.34, 0.02]	[0.49, 0.01]
<i>cloud</i>	$\frac{\text{hybrid_redis}}{\text{multi}}$	<i>runtime</i>	0.60	0.89
		<i>process time</i>	0.60	0.89
		[Mean, Std]	[0.61, 0.01]	[0.95, 0.09]

more efficient processing of stateless tasks, which reduces the idle time for stateful processing awaiting outputs from stateless PEs. Thus, even with the additional synchronization overhead caused by the increasing number of processes, this reduction in idle time still results in a net efficiency gain. It is worth mentioning that, in *cloud*, *runtime* of *hybrid_redis* shows an increase as the number of processes increases. This trend could be attributed to the limitations of virtual hyperthreading when most processes are fully engaged with tasks.

For the *runtime* and *process time* ratios from various platforms and priorities, all ratios are smaller than 1, as shown in Table 3.4. This indicates that, with *dynamic allocation* optimization, *hybrid_redis* outperforms *multi*. This is especially noteworthy, based on the observation that the *Redis* mapping is overall slower than *Multiprocessing* with the same settings.

3.6.5 Analysis of *dynamic auto-scaling*

We experiment with the *auto-scaler* mechanism using the Internal Extinction of Galaxies and Seismic Cross-Correlation workflows on both *server* and *HPC*. In Figure 3.17, the left y-axis depicts the active process count throughout the runtime. The right y-axis represents the queue size for *dyn_auto_multi* and the average idle time for *dyn_auto_redis*. The x-axis indicates iteration counts, which are recorded when the monitored metrics (right y-axis) change. Please note that these iterations are not uniformly spaced regarding time intervals.

For *dyn_auto_multi* in Figure 3.17a, 3.17c, 3.17d and 3.17f, there is noticeable positive correlation between the number of active processes and queue size. It aligns with our expectations: a

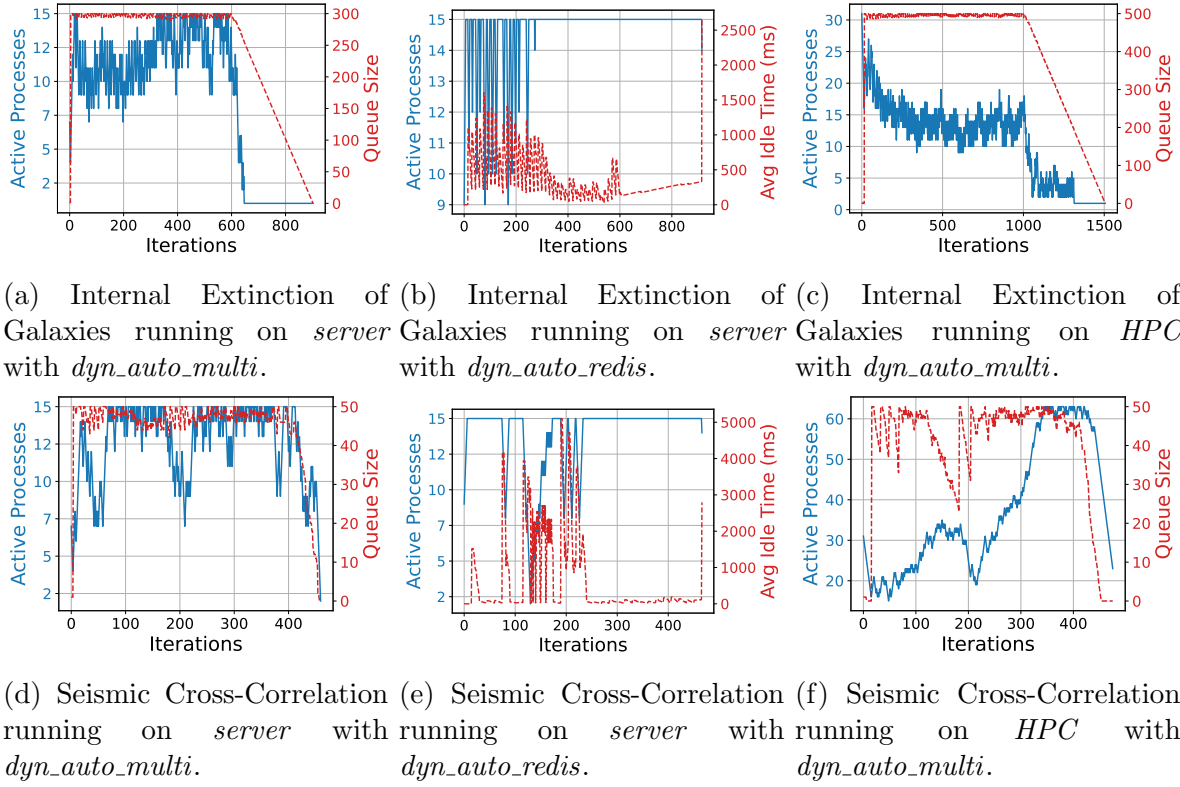


Figure 3.17: Active Size and the monitoring metrics (Queue Size and Avg Idle Time) running with Internal Extinction of Galaxies workflow and Seismic Cross-Correlation workflow.

larger workload in the queue needs more active processes. However, an interesting observation in the *HPC* settings, particularly with simpler workflows, is that the active size rarely reaches the maximum capacity of 64, even when the queue size remains consistently high. This pattern suggests that the *auto-scaler* in *dyn_auto_multi* primarily adjusts the active size by responding to changes in the queue size rather than considering the absolute workload, which might not accurately reflect the actual processing needs. This reactive scaling strategy may lead to a decrease in active processes, as seen in Figure 3.17c, even when the queue size is substantial. Such fluctuations in queue size can mislead the *auto-scaler*, resulting in suboptimal resource utilization and potential delays in processing due to an insufficient number of active processes being maintained.

For *dyn_auto_redis* in Figure 3.17b and Figure 3.17e, there is an inverse relationship between the number of active processes and the average idle time. This means that the *auto-scaler* reduces the number of active processes when active processes have larger idle periods, indicating a reduced workload. The sub-figures reveal a consistent trend: active process numbers lag behind

metric changes due to inertia in the naive *dynamic auto-scaling* strategy. This can result in mismatches between actual needs and active process count. We recognize the need to optimize the auto-scaling strategy, enhancing its ability to accurately capture real requirements and predict workload changes.

3.6.6 Key Insights

Summarizing our extensive experiments, key findings include:

- **Consistent *dynamic auto-scaling* Efficiency:** The *dynamic auto-scaling* consistently demonstrates efficiency across diverse platforms and workflows. It achieves 87% *runtime* and 76% *process time* of *dynamic allocation*'s performance in optimal cases.
- **Complex Workflow Challenges:** While generally effective, *dynamic auto-scaling* faces challenges with complex workflows, in which the *auto-scaler* struggles to predict needs accurately, increasing *runtime*. However, *process time* efficiency remains strong.
- **Stateful Mapping Superiority:** In the context of workflows involving stateful applications, *hybrid-redis* outperform *multi* by achieving as low as 32% of *runtime* compared to *multi*.
- ***Multiprocessing* mapping vs. *Redis* mapping:** The performance achieved with the *Multiprocessing* mapping based optimizations (*dyn-multi* and *dyn-auto-multi*) generally outperforms those of based on *Redis* mapping (*dyn-redis* and *dyn-auto-redis*). The advantage is primarily attributed to the lightweight nature of *Multiprocessing*. However, in scenarios with more heterogeneous workloads and frequent access to the global queue, *Redis* can achieve better performance due to its robust synchronization mechanisms.

3.7 Conclusion

In this chapter, we enhance the scientific computing-specific stream system, dispel4py, by deploying scaling techniques, including *dynamic allocation* and *dynamic auto-scaling*. Furthermore, we enable advanced state management along with a new Redis mapping for more robust and adaptable stream data processing. Extensive experiments across diverse workflows and platforms demonstrate that both *dynamic allocation* and *dynamic auto-scaling* can achieve excellent performance in most cases. *dynamic auto-scaling* is also shown to excel in cost efficiency. Notably, the stateful optimization (*hybrid-redis*) outperforms native mappings. While initial *dynamic auto-scaling* strategies have limitations, our study lays a foundation for refining and enhancing the scalability of dispel4py. These works could inspire further optimizations of related stream systems.

Chapter 4

An Efficient Learned Index for Sequential Stream Processing

4.1 Motivation

This chapter addresses the research question regarding optimising stream processing at the algorithmic level. Effective management of stream data, characterized by high column and velocity, is crucial in stream systems [47]. Managing unbounded stream data is particularly challenging; thus, applying window concepts to bound the stream data becomes essential. As we discussed in Section 2.4.4, indexing stream data within a window can expedite stream processing by allowing fast retrieval, thereby efficiently facilitating streaming operations within specific time frames, such as stream join and stream aggregations [97, 66].

A common index-based window processing (IBWP) type is when the data (which we also refer to as a ‘key’ when it needs to be indexed, updated, and searched) is in sequential order [101]. For example, Figure 4.1 shows the indexed sequence is incremented when new timestamps arrive, which is common in stream processing. In the sequential IBWP, the order of the key aligns with the order of arrival. Thus, the newly indexed key will be larger than all previously indexed keys in the window, which allows insertion only to occur at the end of the index, facilitating the use of a queue-like structure.

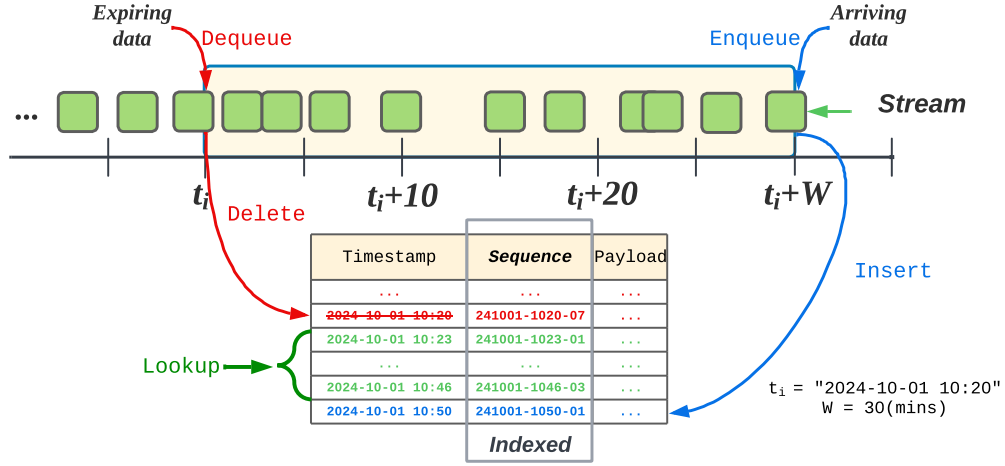


Figure 4.1: Illustration of the sequential IBWP workload.

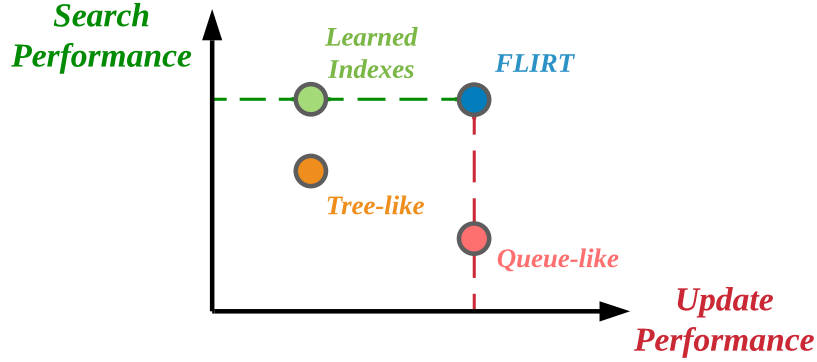


Figure 4.2: Trade-off between search and update in sequential IBWP for algorithmic and learned indexes.

Recently, learned indexes have presented innovative alternatives using a machine learning model to accelerate lookup performance [55]. To capitalize on this advancement, we aim to integrate learned indexes into stream windows. However, there is a significant challenge: learned indexes are not inherently designed for handling updates, as updates could alter the data distribution, resulting in the overhead from the model retraining [157].

Although there are updatable learned indexes that accommodate updates, most are primarily designed just to enable updates rather than optimizing for high-performance updates. Despite this, they provide valuable insights that can inspire the design of a learned index to handle the trade-off between search and update as mentioned in Section 2.4.3. To mitigate performance degradation during updates, updatable learned indexes prefer employing insertion buffers and lazy deletion strategies, which can delay the immediate impact of continuous and frequent up-

dates, allowing for efficient batch processing and thus reducing the amortized costs [49, 56]. Additionally, updatable learned indexes typically adopt hierarchical structure with models to manage large-volume data. The hierarchical structures can facilitate data retrieval but often necessitate multilevel model retraining or structural rebalancing during updates, thereby introducing significant overheads. In contrast, stream indexes use flat structures, such as queues, which are more effective in ingesting high-velocity stream data. Therefore, leveraging the advantages of both types of indexes could help us to design and implement a stream-learned index that combines the search efficiency of learned indexes with the update efficiency of stream indexes, as shown in Figure 4.2.

To attain this goal, we introduce FLIRT, accessible at <https://github.com/SWIXProject/SWIX/tree/main/lib/flirt>. FLIRT is a fast learned index tailored for rolling time frames, adept at handling updates efficiently and maintaining robust range query capabilities. FLIRT caters to index sequences in sliding windows in which each key arrives in a monotonic order, which aligns with the monotonicity in learned indexes (explained in Section 2.4.3). Given this arrival order, FLIRT integrates queue structure with learned index techniques to achieve excellent performance in both update and search. We provide a design overview of FLIRT in Section 4.1, followed by the structure and operation of FLIRT in Section 4.3. Two concurrent FLIRTs are introduced to further enhance the search performance in Section 4.4, and experiments in Section 4.5 show FLIRT and its parallel variants achieve competitive performance. Finally, Section 4.6 concludes this work and introduces future improvements.

4.2 Design Overview

The design of FLIRT is centred around balancing the trade-offs between search- and update-efficiency. The approach to optimizing both performances involves capitalizing on the advantages of various index types (e.g., some indexes excel in updates, and some are optimal for search) and tailoring their advantage to suit the specific application needs we aim to address. This approach informs the architectural choices made in FLIRT, as illustrated in Figure 4.3,

FLIRT structure

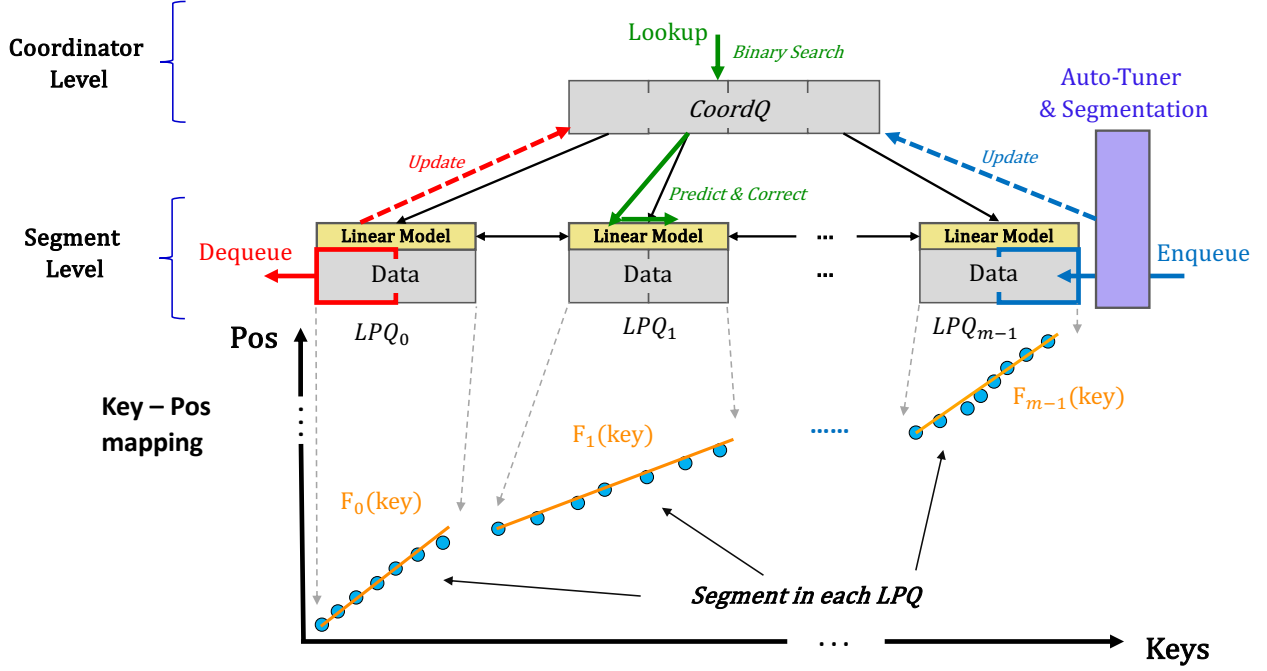


Figure 4.3: FLIRT overview, illustrating the system’s hierarchical design. The Coordinator Level is the *CoordQ*. Below this, the Segment Level is comprised of multiple Linear Prediction Queues (*LPQ*), each equipped with a linear model $F_i(key)$. Key operations include ‘Enqueue’ for data input and ‘Dequeue’ for output. The ‘Auto-Tuner & Segmentation’ dynamically refines these models and segmentation to optimize system responsiveness to changes in data distribution.

which include:

- *FLIRT handle update operation as a queue.* As we introduced, FLIRT is designed for sequential IBWP, in which the arrival order of the keys and the keys themselves are monotonic. This design allows FLIRT to only insert the key into the tail of the index, a process we refer to as *enqueue*, and to remove keys from the front of the index, known as *dequeue*. Consequently, FLIRT adheres to the FIFO principle, which allows FLIRT to handle update operations as a queue. The detailed mechanics of dequeue and enqueue in FLIRT will be discussed in Section 4.3.
- *FLIRT is a two-level index structure.* We partition the data window into multiple segments named *LPQ* to utilise divide-and-conquer strategies common in traditional indexes. These segments are conducive to improving search by localizing the scan range and also

enhancing the update performance by isolating the update effects. To oversee these segments, we pioneer the use of a flat structure named *CoordQ* in the upper level, inspired by the queue structure. This flat structure avoids the costs associated with re-balancing in a hierarchical structure. Consequently, FLIRT showcases a distinctive structural design that diverges from conventional tree-style or single-level queue architectures, which will be explored in greater detail in Section 4.3.

- *FLIRT acts as a piecewise linear function to map the key to its position.* FLIRT adopts principles from learned indexes, utilizing approximation functions to map keys to physical storage, thus accelerating search operations. Yet, predictions may not be precise, introducing potential errors. Models with higher complexity can reduce errors and improve search performance; however, such models could increase the overhead due to the need for frequent retraining to accommodate changing data distribution in stream processing. However, simple model could underfit the data distribution, but achieve consistent performance due to the low retraining costs. The model complexity in the learned index is an aspect of the trade-off between update and search efficiency.

To address this challenge, FLIRT adopts a linear approximation function for each segment, aligning with the divide-and-conquer philosophy. We use model-based segmentation approaches named shrinking cone to generate *LPQ*. The shrinking cone approach co-locates data that can be modelled by linear functions within certain predefined error thresholds (*Err*), therefore guaranteeing accuracy even with a simple linear model. Combining multiple linear functions across different segments renders FLIRT as a piecewise linear function from a mathematical perspective, which can optimize both search and update efficiencies.

- *FLIRT auto-tunes the Err.* As mentioned above, the *Err* determines the segmentation in FLIRT. Specifically, larger *Err* degrades the accuracy in each *LPQ*, thereby decreasing the search performance in *LPQ*. Meanwhile, larger *Err* compacts the data, generating fewer segments, therefore reducing the size of *CoordQ* to improve the search performance of *CoordQ*. Conversely, smaller *Err* improves the search performance in *LPQ* but impedes

the search in *CoordQ*.

The optimal error threshold, Err , is dependent on the data distribution. Determining optimal Err becomes particularly challenging in stream processing environments where data distributions can shift dynamically. To balance this trade-off between search performance at different levels of the index in a dynamic setting, FLIRT incorporates an auto-tuner that adaptively adjusts Err , making it efficient and parameter-free. The details of this mechanism will be discussed in Section 4.3.

- *FLIRT is parallelizable.* The design, particularly its queue-style update mechanism and two-level structure, makes FLIRT suitable for parallel processing. To take this advantage, we introduce two parallel variations: Partition-based FLIRT (PPFLIRT) and Shared FLIRT (PSFLIRT). Both of them are designed to enhance search performance by leveraging multiprocessing capabilities. PSFLIRT is designed for skewed workloads, which are common in stream processing. Further details about PPFLIRT and PSFLIRT will be provided in Section 4.4.

To summarize, FLIRT effectivity supports update patterns from sequential IBWP while achieving excellent search performance by adopting a two-level structure in which a piece-wise linear model is deployed. To achieve optimal performance at both levels of FLIRT, it is crucial to identify the optimal threshold, referred to as Err . We develop an auto-tuner to dynamically and adaptively adjust the Err on the fly. Furthermore, we introduce two parallel variants of FLIRT to further improve search performance.

4.3 FLIRT

4.3.1 Structures

FLIRT consists of two levels: the top level is the coordinator level, and the bottom is the segment level. Its construction is bottom-up, in which segments are first constructed, and then the coordinator level is constructed based on information from the segment level.

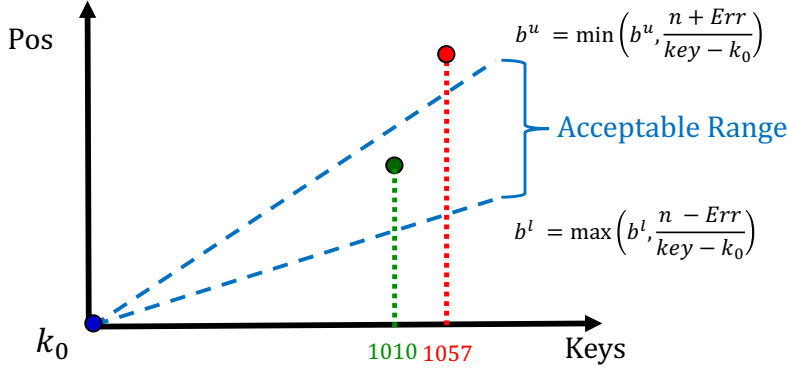


Figure 4.4: Shrinking cone algorithm for segmentation.

Segment Level

The segment level comprises disjoint segments where data is stored and managed, akin to leaf nodes in a B+Tree. Each segment supports queue-based update operations, enqueue and dequeue, and enables fast retrieval by adopting a linear model, making it a Linear Predictable Queue (*LPQ*). The linear model in *LPQ* is in the form of $F(key) = a + b \times key$ in which a is the intercept, and b is the slope. To achieve constant computation cost $O(1)$, we use interpolation approximation [90] to directly compute the slope by using first and last keys, ($b = \frac{n}{k_{n-1}-k_0}$). After obtaining the slope, we use the first key (k_0) and its position (0) with slope b to calculate intercept $a = -b \times k_0$. Therefore, we can have the equation in the form of $F(key) = b \times (key - k_0)$, which normalizes all keys within a segment relative to the first key.

To generate segments, we use the shrinking cone algorithm proposed in FITing-Tree [62]¹. It is a greedy stream algorithm similar to stream data segmentation algorithms such as FSW (Feasible Space Window) [107] and SWAB (Sliding Window and Bottom-up) [90], but focuses on monotonic data segmentation. Algorithm 3 shows how the shrinking cone algorithm works with predefined *Err*. It co-locates keys in the same *LPQ* if the slope of the key (calculated as $\frac{n}{key-k_0}$ where n represents the number of existing keys in the *LPQ* and k_0 is the first key of the *LPQ*) is between lower bound b^l and upper bound b^u , for example, the green point in Figure

¹FITing-Tree is an updatable learned index designed which bones of B+Tree only change the leaf node of B+Tree to model-based, and it use sgemeent level buffer to accelarate the random insertions, such design can leverage the benfit of B+tree, but the combusome of Tree structure is not ideal for high frequency updates in stream setting. Besides, it is not designe for stream updates and it didn't release it C++ implemtatation therefore excluyed in the evaluation

Algorithm 3 Enqueue of FLIRT based on shrinking cone

```

1: function ENQUEUE(key)
2:    $LPQ_i \leftarrow$  Perform binary search in CoordQ to get corresponding segment
3:   data  $\leftarrow$  Data stored in  $LPQ_i$ 
4:    $n \leftarrow$  Number of keys in the  $LPQ_i$ 
5:    $k_0 \leftarrow$  First key in the  $LPQ_i$ 
6:    $b^l \leftarrow$  Lower bound in the  $LPQ_i$ 
7:    $b^u \leftarrow$  Upper bound in the  $LPQ_i$ 
8:
9:   if  $key < data[n - 1].k$  then
10:    throw invalid_argument('Key must be larger than the last key to be appended')
11:  end if
12:
13:   $temp\_b \leftarrow \frac{n}{key - k_0}$ 
14:  if  $b^l \leq temp\_b \leq b^u$  then
15:     $temp\_b^u \leftarrow \frac{n + Err}{key - k_0}$ 
16:     $temp\_b^l \leftarrow \frac{n - Err}{key - k_0}$ 
17:     $b^u \leftarrow \min(b^u, temp\_b^u)$ 
18:     $b^l \leftarrow \max(b^l, temp\_b^l)$ 
19:     $b \leftarrow \frac{b^u + b^l}{2}$ 
20:    data.push_back(key)
21:     $n \leftarrow n + 1$ 
22:  else
23:    segPtr  $\leftarrow$  new Segment(key)
24:    rightSibling  $\leftarrow$  segPtr
25:    segPtr.leftSibling  $\leftarrow$  this
26:    Appending key and segPtr into CoordQ
27:  end if
28:  return
29: end function

```

4.4. Then, both bounds b^l and b^u are dynamically adjusted by interpolating between the first key and the newly enqueued key based on predefined *Err*. As a result, b^l may incrementally increase, and b^u may decrease, thereby gradually narrowing the cone. As this shrinking cone narrows, the next key is more likely to fall outside these bounds; then, it will be the first key of the new *LPQ*, spawning new segments.

Apart from data and model, each *LPQ* also bookkeeps essential states, the number of keys (n), and the number of flagged as deleted keys (n^d). These are used to track the current capacity and manage deletions within the segment efficiently. Given the dequeue is from the beginning to the end, n^d indicates that the first n^d keys are flagged as deleted. Such a flag mechanism

not only avoids the key’s shift cost during actual deletion, but also saves memory from storing a flag in each key. Furthermore, *LPQ* also stores the pointer to its sibling, which can enhance scan across multiple segments when performance range query with large scan range.

Coordinator Level

At the coordinator level, FLIRT deploys a circular queue structure named *CoordQ* to manage the *LPQ*s by storing their first keys. The first key is directly appended to the *CoordQ* when the key exceeds the shrinking cone boundaries of the preceding *LPQ*. The method of storing initial keys, coupled with the one-pass shrinking cone segmentation, ensures that the construction of the coordinator level is simultaneous with the creation of the segmentation level. This simultaneous development enhances the efficiency of FLIRT’s construction, making it different from hierarchical structures like B+Tree, where layers are built sequentially.

The *CoordQ* acts similarly to the internal nodes in B+Trees, guiding the search towards the correct leaf node (*LPQ*). Specifically, *CoordQ* uses the traditional binary search instead of machine learning models since we want to simplify the trade-off between search performance across levels. Using models would bind the search cost to model errors rather than size, complicating the fine-tuning of errors across both levels. This design prioritized size-bounded performance can contribute to leveraging hardware efficiencies such as cache optimization (e.g., sizing the *CoordQ* to be a multiple of the cache line size). This aligns with our design choice of an array-based circular queue over a traditional queue or linked list. This design can reduce memory access and cache misses by reducing the data movement within the queue caused by shifting and pointer traversal during searching.

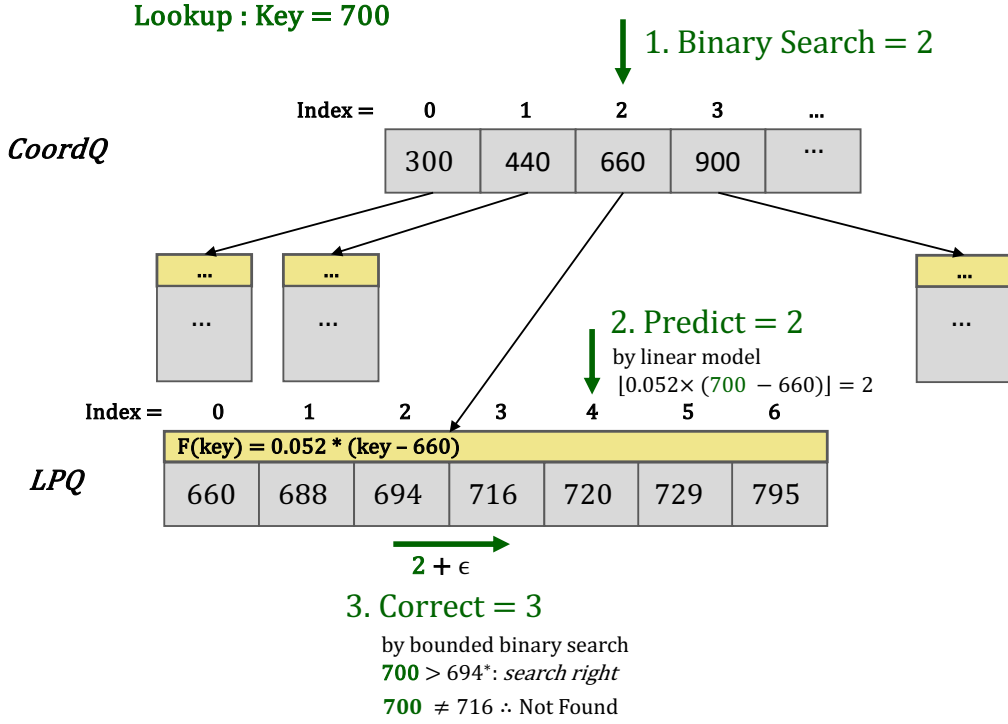


Figure 4.5: Steps of the search operation in FLIRT.

4.3.2 Operations

Search

Search in FLIRT involves both coordinator and segment levels. Utilizing methodologies discussed in Sections 2.4.2 and 2.4.3, we treat index as a model to refine the search range defined by $[i_l, i_h]$ (the lower and upper bounds of the index of target lookup key), where the initial range spans the entire stream window with size W .

The search starts at the coordinator level, where a binary search is performed to shrink the range from the window size to the size of the corresponding *LPQ* ($[i_s, i_s + n]$, where i_s is the starting index of the *LPQ*, and n is the size of *LPQ*). Specifically, this step finds the last *LPQ* whose initial key (k^0) stored in *CoordQ* is less than the lookup key. As an example shown in Figure 4.5, the look-up key is 700, and it finds *LPQ* with starting key 660 since it is the last key in *CoordQ* smaller than 700. This narrowing of the search range by the binary search has a complexity of $O(\log m)$, where m is the number of segments.

Once the *LPQ* is located, FLIRT predicts the lookup key's position using the linear function

$F(key)$ built upon the LPQ , as shown in step 2 in Figure 4.5. This prediction further refines the search range from the segment size to the bounds defined by Err . If the key in the predicate position is smaller than the lookup key, then the bound will be $[i_p, i_p + Err]$ in which i_p is the index of the predicted key. If it is larger, the range is $[i_p - Err, i_p]$. This prediction operates with constant complexity $O(1)$. To pinpoint the exact position, a corrective binary search is performed, as depicted in step 3 of Figure 4.5, in which the key at the predicted position (694) is less than 700, therefore a rightward binary scan within $[i_p, i_p + Err]$ is processed to find the exact location at the cost of $O(\log(Err))$.

The general procedures described above outline the search process in FLIRT. However, special consideration is required when the located LPQ is the first in the segment level in which keys may be implicitly deleted already. In such cases, we further shrink the range from $[i_s, i_s + n]$ to $[i_s + n^d, i_s + n^d + n]$, where n^d is the number of keys flagged as deleted. This adjustment focuses the search on valid data in the window, guaranteeing correctness and enhancing efficiency.

Furthermore, to perform a range query, the initial step involves locating the starting position for the range scan using the general procedures outlined previously. Then, a linear scan is performed based on the specific conditions of the query. If the scan extends across multiple LPQ , FLIRT can directly proceed to adjacent segments without reverting to the $CoordQ$, which facilitates the scan performance.

Enqueue

Unlike the search operation, which requires navigation through the coordinator level to locate the corresponding LPQ , the enqueue operation is directly performed at the segment level since it only occurs in the last LPQ with constant cost $O(1)$. There are two cases for this operation as illustrated in Algorithm 3 and Figure 4.6:

- Within cone bounds: If the key falls within the boundaries of the shrinking cone, it is enqueued in the current LPQ , as exemplified in the bottom left of Figure 4.6. Subsequently, these two boundaries and the slope of the linear model will be updated based on

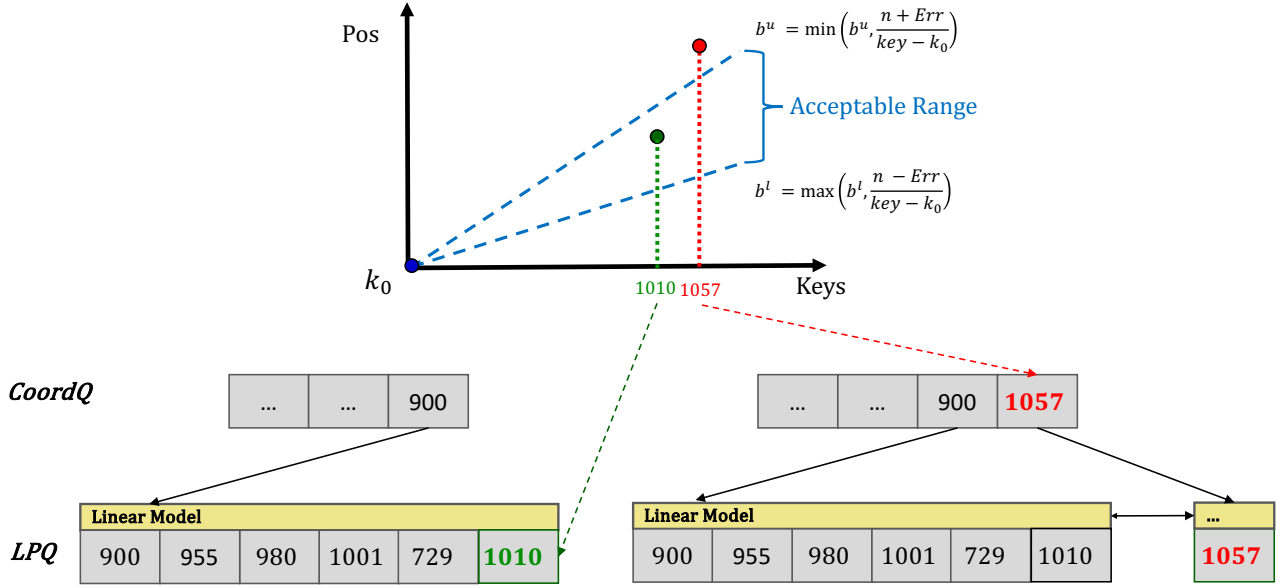


Figure 4.6: Steps for enqueueing into FLIRT.

equations in Lines 15 - 19 in Algorithm 3.

- Outside cone bounds: If the new key falls outside the acceptable range, as shown in the bottom right of Figure 4.6, FLIRT does not permit it to enqueue to the current *LPQ*. This policy guarantees that the maximum distance between the predicted and actual positions does not exceed Err . Therefore, a new *LPQ* with the new key as the starting key is created. The siblings of this *LPQ* and previous *LPQ* are updated accordingly, and the starting key is appended to *CoordQ* (Line 23 - 26 in Algorithm 3). If appending the key exceeds the current capacity of *CoordQ*, the array is dynamically resized to double its capacity, similar to how `std::vector` in C++ behaves. Furthermore, when only one key is in the new *LPQ*, the upper and lower boundaries are initially set to infinitely large and zero, which allows the enqueue of any subsequent key larger than the first key.

Dequeue

Similar to Enqueue, dequeue begins with the segment level with $O(1)$ complexity, but at the first *LPQ*. There are also two cases, as shown in Figure 4.7, depending on whether the *LPQ* remains non-empty post-deletion:

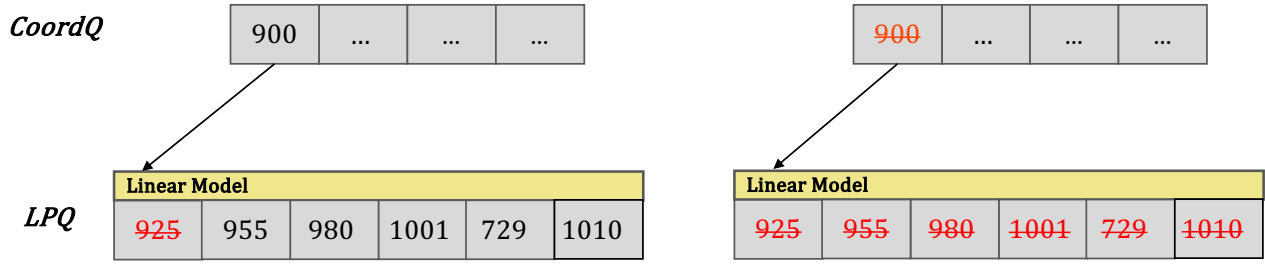


Figure 4.7: Implicit deletion during dequeuing of FLIRT. *Left*: Non-empty segment flagging expired keys. *Right*: Erasing the entire segment.

- **Non-empty segment:** When a key is dequeued, but does not empty the *LPQ*, as shown in the left side of Figure 4.7, the key is flagged as deleted by incrementing the number of keys deleted (n^d) stored in *LPQ*. This implicit deletion strategy leaves space unused within the *LPQ*, but avoids the computational overhead associated with data shifting, which could significantly impact CPU load and memory movement. Notably, the dequeue does not affect the linear model used within the *LPQ*; the first key continues to be used in the model even if it is flagged as removed.
- **Empty segment:** the right side of Figure 4.7 shows that all keys in *LPQ* are flagged as deleted, which is indicated by $n^d = n$. The *LPQ* is discarded, the sibling of the next *LPQ* is updated, and the first key in *LPQ* is dequeued. Since *CoordQ* is implemented based on a circular queue, the queue is achieved by updating the **front** indicator without resulting shifts.

Auto-tune

According to our previous discussions on the operational costs, both enqueue and dequeue operations generally incur a constant cost. However, overall performance is primarily determined by the search cost. This cost consists of two main components: the *Err* and the number of segments (*LPQ*) denoted as m (Equation 4.1). A larger *Err* typically reduces m , which decreases the search cost in the root segment, but increases the correction cost in subsequent segments. Conversely, a smaller *Err* leads to higher prediction accuracy, but results in a greater number of segments, thereby increasing the workload at the coordinator level and potentially elevating

the overall search cost.

The logarithmic functions applied to both m and Err in the cost model are integral to representing the algorithm's performance characteristics. The term $\log(m)$ reflects the time complexity of performing a binary search within the segments, as binary search operates by repeatedly dividing the search interval in half, resulting in a time complexity of $O(\log m)$. Similarly, $\log(Err)$ accounts for the impact of the error threshold on the search process.

$$Cost = \log(m) + \log(Err) \quad (4.1)$$

To optimize performance across both the coordinator and segment levels, we can employ tuning techniques such as grid search to determine the optimal Err statically. However, as data distributions can change over time, a static Err can become suboptimal. To cope with stream data, we propose a dynamic auto-tuner to fine-tune Err based on the search cost model on the fly, as shown in Algorithm 4.

Auto-tune in FLIRT dynamically minimizes the search operation cost by adjusting the prediction error (Err) based on its impact on derived costs. The adjustment strategy involves observing the cost changes when Err is modified; if the cost decreases, the current tuning direction (either increasing or decreasing Err) is maintained. If not, the direction is reversed.

Initially, Err is set to 256, a value chosen based on microbenchmarks that suggested values ranging from 128 to 1024 generally yield good performance for most scenarios. However, performance can vary significantly when Err is set below 128 or above 1024, particularly when data or window sizes change.

To support the tuning of Err , additional parameters include a modification factor of 2, which adjusts the error threshold, and a boolean indicator that determines whether to increase (`true`) or decrease (`false`) the error threshold. The system also tracks the number of directional changes in tuning (`bouncesCount`) and the consistency of these changes (`consistentDirectionCount`), with a set threshold of 5 for determining significant directional shifts. Among these parameters, only the bounce threshold impacts performance. A lower threshold makes the algorithm

Algorithm 4 FLIRT Auto-Tune of FLIRT

```

1: class Autotuner:
2:   private:
3:     error, initialError, direction, adjustment
4:     bouncesCount, consistentDirectionCount
5:   public:
6:     constructor():
7:       initialError  $\leftarrow$  256 ▷ Initial error threshold
8:       error  $\leftarrow$  initialError
9:       adjustment  $\leftarrow$  2 ▷ Initial adjustment factor
10:      direction  $\leftarrow$  true ▷ True for increase, False for decrease
11:      bouncesCount  $\leftarrow$  0
12:      consistentDirectionCount  $\leftarrow$  0
13:      threshold  $\leftarrow$  5 ▷ Threshold for number of direction changes
14: procedure AUTOTUNE(N, Nnew)
15:   Cinitial  $\leftarrow$   $\log(N) + \log(\text{error})$ 
16:   adjustedError  $\leftarrow$  direction?error  $\times$  adjustment : error/adjustment
17:   Ccurrent  $\leftarrow$   $\log(N_{\text{new}}) + \log(\text{adjustedError})$ 
18:   if Ccurrent < Cinitial then
19:     error  $\leftarrow$  adjustedError ▷ Adopt the new error threshold
20:     consistentDirectionCount  $\leftarrow$  consistentDirectionCount + 1
21:   else
22:     direction  $\leftarrow$   $\neg$ direction ▷ Reverse the direction
23:     bouncesCount  $\leftarrow$  bouncesCount + 1
24:   end if
25:   ADJUSTTUNINGSTEP
26: end procedure
27: procedure ADJUSTTUNINGSTEP
28:   if bouncesCount  $\geq$  threshold then
29:     adjustment  $\leftarrow$  adjustment/2 ▷ Halve the tuning step
30:     bouncesCount  $\leftarrow$  0
31:   else if consistentDirectionCount  $\geq$  threshold then
32:     adjustment  $\leftarrow$  adjustment  $\times$  2 ▷ Double the tuning step
33:     consistentDirectionCount  $\leftarrow$  0
34:   end if
35: end procedure

```

more adaptive to changes, allowing for quicker responsiveness but increasing the likelihood of frequent adjustments and potential instability. In contrast, a higher threshold provides a buffer against transient fluctuations, promoting smoother and more consistent tuning behavior but reducing responsiveness to sudden changes.

The search cost is then evaluated using the cost model described in Line 14 of Algorithm 4. Once the tuning rate is reached, *Err* is modified, and the resulting cost is recorded (Lines 15 - 16).

Subsequent adjustments in Err are guided by comparing these recorded costs to previous values. This method resembles the approach used in gradient descent algorithms, where adjustments are based on the direction of the cost gradient, thus ensuring that the algorithm remains sensitive to parameter variations and adapts to maintain optimal performance.

The auto-tuning process in FLIRT is governed by two critical parameters: the tuning rate and the tuning step. The tuning rate determines the frequency of adjustments to the Err . The tuning step, on the other hand, dictates the magnitude of each adjustment. While a smaller tuning step ensures a gradual convergence towards an optimal Err , potentially leading to precise tuning, it may also result in slower responses to changing conditions. Conversely, a larger tuning step, although speeding up the convergence process, might cause the tuning to overshoot the optimal value, missing it entirely.

To mitigate these issues and manage potential oscillations effectively, FLIRT’s auto-tuning mechanism dynamically adjusts the tuning step based on a pre-configured direction change threshold, such as 5 consecutive reversals. Frequent reversals in the adjustment direction of Err , or fluctuations, suggest that the current tuning step is excessively large. This observation prompts the system to halve the tuning step, aiming to enhance stability and prevent overshooting. If, however, the direction of adjustment remains consistent over several cycles without reaching the reversal threshold, it indicates that the tuning step may be too conservative. In such cases, the step is doubled to expedite the search for an optimal Err , thereby adjusting more aggressively to meet performance targets more quickly.

4.4 Parallel FLIRTs

As mentioned in Section 4.2, the design of FLIRT inherently supports parallelization. Particularly, the queue-style update mechanism reduces the synchronization between operations by localizing updates to specific LPQ s. The flat structure reduces the synchronization across levels compared to multi-level structures like trees. Capitalizing on these advantages, we propose two parallel variants of FLIRT to improve the search performance: Partition-based FLIRT

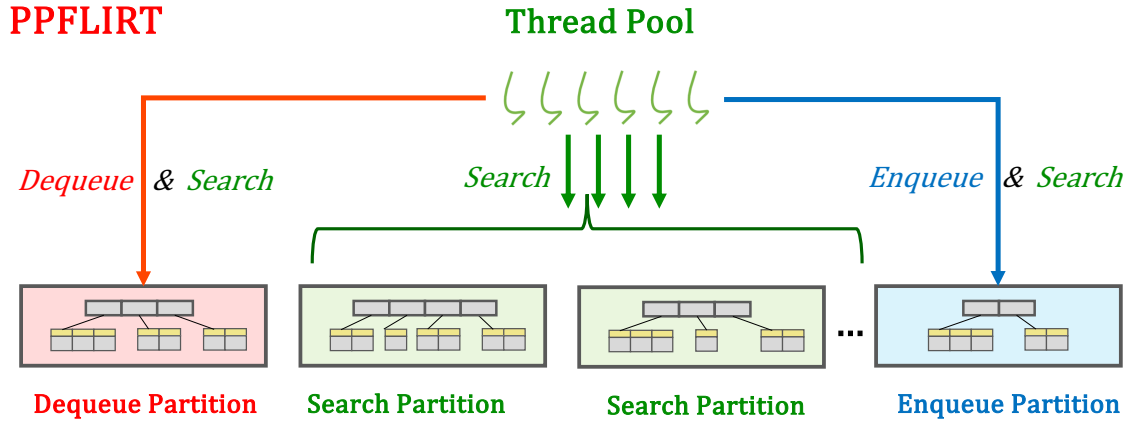


Figure 4.8: Overview of the structure of PPFLIRT.

(PPFLIRT) and Shared FLIRT (PSFLIRT).

4.4.1 Partition-based FLIRT

Partition-based FLIRT (PPFLIRT), which is tailored for a fixed window, optimizes performance by balancing load distribution across the window. PPFLIRT equally partitions the window into subsets; each subset is managed locally by its own instance of FLIRT. PPFLIRT is in the form of a chained index commonly used in parallel IBWP [137]. Each partition maintains and handles a window segment with a size of the $\frac{W}{t-1}$, where W is window size, and t is the total number of threads. The subtraction of one account for a dedicated thread for an empty partition specifically reserved for new enqueues.

As shown in Figure 4.8, PPFLIRT organizes partitions into three types: one for enqueues, one for dequeues, and $t - 2$ for searches. Each partition is indexed by local FLIRT and is maintained by distinct threads. While the workload across all search segments remains consistent, the structure of the FLIRT may differ due to variations in data distribution within each segment. Additionally, these partitions are logically arranged in a circular queue configuration, in which enqueue and dequeue partitions are always adjacent to each other, resembling the front and tail in a full circular queue. Partitions change their tasks as the window advances, and they manage specific state information to optimize updates and searches.

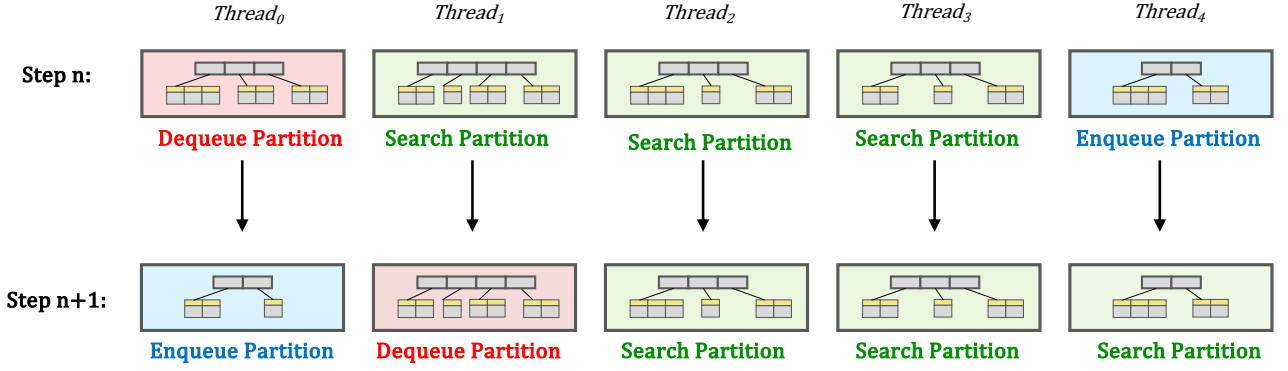


Figure 4.9: Illustration of thread transition in PPFLIRT when the enqueue thread becomes full and the dequeue thread becomes empty.

Enqueue and Dequeue

PPFLIRT is designed for a fixed window, such as a count-based window and sliding window with a uniform data arrival rate. Therefore, we can assume that the W remains unchanged during execution. This fixed size facilitates two key operational insights for enqueue and dequeue operations: First, the enqueue operation is designed to accept a specific number of keys before it considers the partition full and ceases to accept more. This number is calculated as $\frac{W}{t-1}$, where t is the total number of threads (partitions). This ensures exact and balanced workflow for all search partitions. Secondly, enqueue and dequeue operations are always paired, which allows them to process the transition to their subsequent operational states simultaneously. But, the paired enqueue and dequeue necessitate synchronization between these two threads.

The actual enqueue and dequeue are carried out by local FLIRT, which operates just as it does in the serial version without extra overheads. However, updates in PPFLIRT introduce synchronization overheads. The synchronization occurs with each enqueue and dequeue thread, ensuring that each operation must be completed before the next beginning. Another synchronization comes from rolling the tasks of threads, which involves one more thread. When the enqueue partition fills up (reaching the size of $\frac{W}{t-1}$) and the dequeue partition empties, these respective threads inform the thread maintaining the partition containing the oldest data (this thread is another neighbour of the dequeue thread) to transition tasks together. Specifically, as illustrated in Figure 4.9, the thread that empties its partition transitions from managing dequeues to handling enqueues. Simultaneously, the previous enqueue thread shifts to managing

searches, and the informed search thread takes over the dequeue responsibilities.

Search

Given the monotonicity in sequential Index-Based Window Processing (IBWP), each partition maintains a distinct, non-overlapping range of the data sequence. Therefore, the point lookup is only carried out in one partition. First, queries are dispatched to all threads, including those handling enqueues and dequeues. Each thread decides whether to conduct a local search by comparing the lookup key and the minimum and maximum keys with constant complexity. Bookkeeping of the minimum and maximum keys is overhead-free for all search threads, as the change only occurs in enqueue and dequeue threads. Specifically, the enqueue thread needs to update the maximum key, while the dequeue thread updates the minimum key. If the lookup key falls between the minimum and the maximum, local FLIRT is searched.

Range queries may involve more partitions, so each thread needs to check the range query conditions against its maintained status to determine whether to participate in the query execution. Given this design, PPFLIRT is expected to achieve excellent search performance when queries are uniformly distributed across the data distribution in the window.

4.4.2 Shared FLIRT

Shared FLIRT (PSFLIRT) prioritizes load balancing in the queries rather than data. It keeps data in one global index, allowing threads to access different parts of the index concurrently, as shown in Figure 4.10. PSFLIRT is expected to outperform PPFLIRT in a skewed query workload in which query distribution does not align with data distribution. For example, if queries are heavily concentrated in a small range, PPFLIRT might limit query execution to one thread, leaving others idle. In contrast, PSFLIRT allows all threads to engage in search activities concurrently as long as there are no updates. To avoid race conditions during updates, FLIRT uses read-write locks to synchronize these operations.

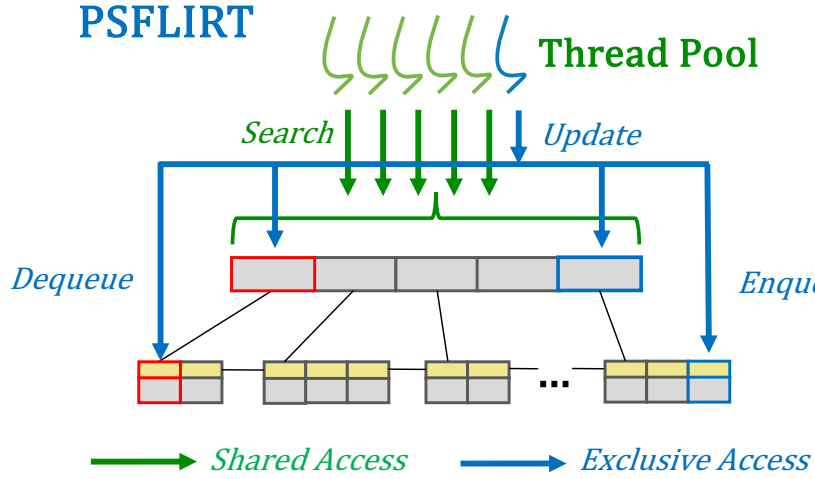


Figure 4.10: Overview of the structure of PSFLIRT.

Enqueue and Dequeue

Updates in PSFLIRT follow the same procedures as those in serial FLIRT, which is carried out by one thread. Since we do not assume the fixed window size, enqueue and dequeue operations are not necessarily paired; each thread can freely enqueue and dequeue keys as they become available or expire. PPFLIRT handles the synchronization among partitions. In contrast, PSFLIRT needs to manage synchronization within the FLIRT structure, including the first and the last LPQ and $CoordQ$.

In the first and last LPQ , the update thread deploys a read-write lock to synchronize enqueue and dequeue. Additionally, to mimic the procedure of serial FLIRT, PSFLIRT also maintains status indicators, such as the number of records in each segment n and the number of keys flagged as deleted n^d . The read-write locks in LPQ are also used for updating n^d in the first LPQ and n in the last LPQ . The synchronization in $CoordQ$ is triggered when dequeuing or enqueueing. To maximize concurrency, the read-write locks are just applied to the first and last entry of $CoordQ$, and other search threads are still permitted to perform lookups simultaneously.

Search

Contrary to PPFLIRT, where all threads receive a replicated query, PSFLIRT distributes queries uniquely across different threads, ensuring no threads receive the same query. Multiple threads can conduct different local searches simultaneously.

Each search thread will have its own search queue with different lookup keys and uses a reader lock to freely traverse PSFLIRT until an update is triggered. All threads can access the first LPQ ; therefore the search operation needs to check if the key has expired. The sequential ID of a key in LPQ_i is the position of the key in LPQ_i plus the sequential ID of the first key $k_{i,0}$. The key has expired if the sequential ID is less than the update counter. The search operations, including range search, are the same as FLIRT with the addition of reader locks for synchronization.

4.5 Evaluation

We conduct comprehensive experiments to evaluate FLIRT and its two parallel variants. For FLIRT, we first perform a self-analysis to assess the effectiveness of its auto-tuning mechanism. Then, FLIRT is benchmarked against several baseline indexes across various scenarios. For parallel FLIRTs, we first evaluate the search performance of PPFLIRT and PSFLIRT compared to FLIRT to assess improvements in search efficiency with increasing numbers of threads. Then, we conduct a performance comparison regarding updates. Additionally, we test the parallel FLIRTs in skewed query workloads to analyze the adaptability and performance of PSFLIRT.

4.5.1 Experiment Setup

Baselines

Several baseline indexes are utilized in this experiment, and some of them are introduced in Section 2.4.

- **B+Tree**: It is the most widely used traditional index and is a baseline for many learned index studies. We utilize the STX implementation [22] in this study. As mentioned in Section 2.4, the fanout influences the performance of B+Tree. We conduct experiments with varying fanout from 4 to 4096 using a grid search approach to select the optimal results for comparison.
- **Queue**: Known for its outstanding update performance, the queue serves as a baseline for comparing the enqueue and dequeue performance of FLIRT. We implement it using a circular queue mechanism with `std::array` [27] in C++.
- **LIST**: LIST proposed by [69] is a linked list data structure used for managing stream windows, which is a representative index in early stream systems. Since the source code is not available, we implement the LIST based on `std::list` [28] in C++.
- **ALEX**: ALEX [113] introduced in Section 2.4 is an updatable learned index with specific considerations for appending data at the end of an index. It represents a major competitor from learned indexes for FLIRT in both update and search performance.
- **PGM**: PGM [152] also discussed in Section 2.4 is based on LSM-tree-like structure. In this study, we use dynamic PGM and refer to it as PGM for simplicity. Together with ALEX, they are used to evaluate different learned index structure designs against FLIRT. PGM is not parameter-free; it requires user input for the error parameter. We use the same approach for B+Tree, varying the error from 4 to 4096 and selecting the best-performing setting for each dataset.

FITing-Tree is an updatable learned index. The only two common points between FITing-Tree and FLIRT are that FLIRT uses a similar segmentation method and that they both have model-based leaf nodes. However, FITing-Tree is a B+Tree-based updatable learned index. It modifies the leaf nodes of a B+Tree to be model-based with a segment-level buffer to accelerate random insertions. Its design leverages the benefits of a B+Tree, but the cumbersome tree structure is not ideal for high-frequency updates in stream settings. Since FITing-Tree has not released their C++ implementation, we have chosen to exclude it from our experiments. Other indexes

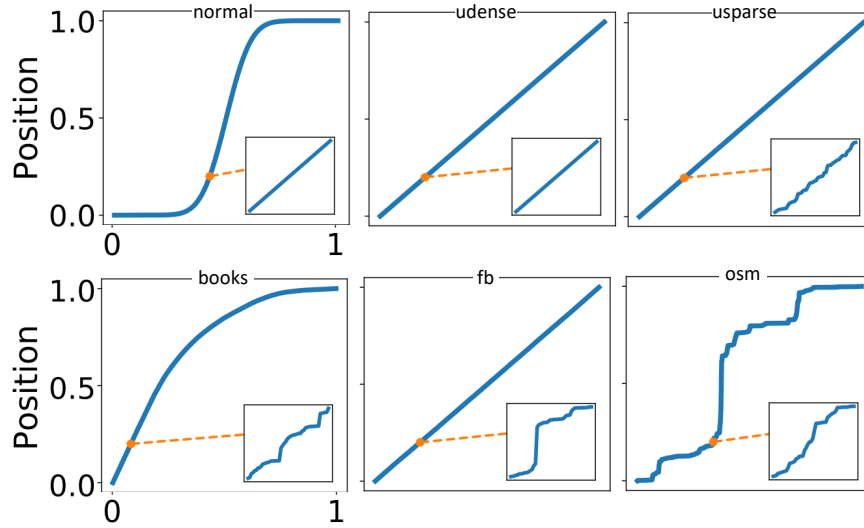


Figure 4.11: Distribution of datasets in the SOSD benchmark [93].

support updates and range queries, including traditional index, learned index and stream-specific index. For traditional indexes, such as LSM-tree [121] and FAST [92], some of them are similar in design to the baselines used in this study, and some of them do not efficiently support deletions. Therefore, they are not included in this experiment. Similarly, learned indexes, especially Xindex and FIndex, are not included due to their lack of specific optimizations for sequential IBWP. However, they will be evaluated in Chapter 5 when experimenting with SWIX in more generic window scenarios. The same rationale applies to the stream-specific index, IMTree.

Platforms

All implementations, including the baselines, FLIRT and parallel FLIRT, are written in C++. They are compiled with the gcc on optimization O3 for the experiments, which are conducted on a machine equipped with an Intel E5-2680 v3 @ 2.50GHz CPU with 64GB of RAM running Centos 8.

Benchmark Datasets

For reproducibility and comparability, we adopt the SOSD benchmark [93], which is widely used for evaluating learned indexes. We select six datasets from this benchmark, including three synthetic datasets (*normal*, *udense*, *uspase*) and three real-world datasets (*books*, *fb* and *osm*). Each dataset exhibits distinct distribution characteristics. Generally, the distributions in synthetic datasets are more linear, making them easier to capture with linear models commonly used in learned indexes. Conversely, real-world datasets often contain irregularities and noise that challenge these models, impeding the performance of learned indexes.

In these datasets, distributions can be classified as micro and macro, as shown in Figure 4.11. Synthetic data often shows linearity at least on one of these scales. For example, *udense* shows both micro and macro linearity. However, real-world datasets often contain noise, especially *osm*, which is notoriously difficult for linear approximation [49].

Workload and Scenarios

The experiments simulate a tuple-based sliding window by pairing the enqueue and dequeue operations to maintain a constant window. The window size varies across different experiments, with a maximum of $100M$ keys. In the SOSD benchmark, each key is represented as an unsigned 64-bit integer (`uint64_t` in C++); therefore, the window takes up about $800M$ bytes. This size can be fitted in memory in the platform and exceeds its cache size. This setup provides a more realistic test scenario for evaluating in-memory index performance.

The default workload initializes the window with $100M$ keys and then executes a standard operation set. The standard operation set includes one enqueue, one dequeue, and one search. Keys are uniformly randomly selected from within the window. For skewed query workloads, we use a Zipfian distribution with different degrees of skewness θ from 0.1 to 0.9. Basically, lower values of θ result in a more balanced selection of keys across the window, while higher values select keys from a more centralized area, known as hot spots. The number of queries Q in each process i is given by:

$$|Q_i| = \frac{|Q|}{i^\theta \times \sum_{j=1}^{thd} \frac{1}{j^\theta}} \quad (4.2)$$

Where, $|Q|$ is the number of queries and thd is the number of threads. For PPFLIRT, $|Q_i|$ is the number of queries in each thread. More queries are given to the first threads as θ increases. For PSFLIRT, $|Q_i|$ is the number of keys in each different part of the data. Similarly, more data is sampled from the front of the data as $theta$ increases. We then distribute the skewed queries evenly across search threads.

Scenarios used in the experimental study vary key factors affecting the query performance of IBWP, including *Window Size*, *Number of Executions*, *Scan Range* and *Read-write Ratio*. By default, a balanced workload is used for experiments with the following configuration: *Window Size* set to 100M, *Number of Executions* at 1M, *Scan Range* at 100, and *Read-write Ratio* at 1, unless specified otherwise.

Metrics

We employ two metrics, execution time and throughput, for the experimental evaluation. Execution time is measured using the `std::chrono` [26] in C++ for each operation. Throughput is used to quantify the number of operations executed per second in the parallel evaluation.

4.5.2 Self Analysis

This section aims to verify the importance of *Err* and the performance of the auto-tuner of FLIRT. Results are displayed in Figure 4.12. There are 12 sets of experiments for each dataset; 11 of them are running on FLIRT with a fixed *Err* from 4 to 4096, where *Err* is not modified during execution, and another one is FLIRT employing dynamic auto-tuning of *Err* with a default tuning rate of 0.1%. These configurations are referred to as static FLIRT and auto-tune FLIRT, respectively, for simplicity.

In Figure 4.12, there are two y-axes: the left y-axis represents the cost calculated by the cost

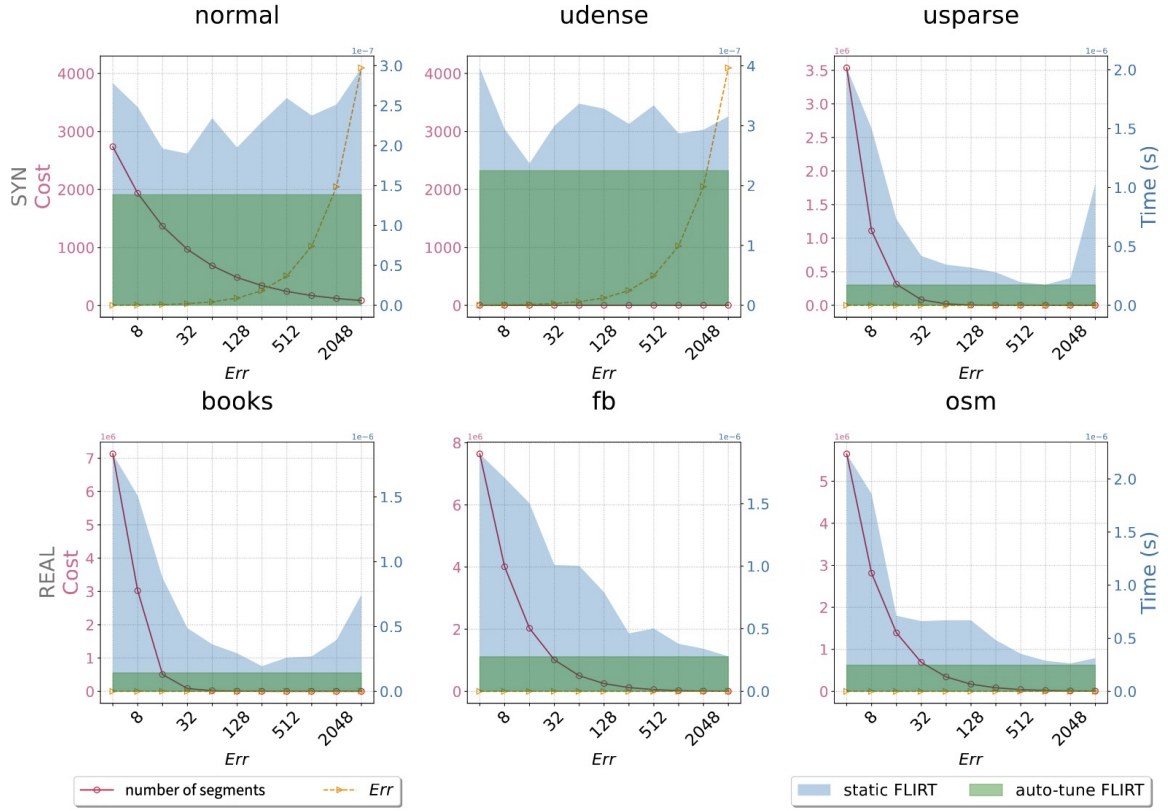


Figure 4.12: Evaluation of the effectiveness of FLIRT's auto-tuner.

model, which is referenced by the number of segments (red line) and *Err* (orange line). The right y-axis represents actual latency in seconds, which is used for reference to the performance of static FLIRT (blue area) and auto-tuned FLIRT (green area). For the x-axis, it represents the different *Err* used for static FLIRT. With the increase of the *Err*, static FLIRT demonstrates a decrease in the number of segments (red line) and an increase in error (orange line). Such changes reflect the performance trade-off discussed in Sections 4.2 and 4.3. Meanwhile, the cost curve of static FLIRT (blue area) shows a convex shape, indicating there is an optimal cost, notably in the *normal*, *books*, and *uspase* datasets.

In contrast, since the *Err* of auto-tune FLIRT dynamically changes during execution, its performance is consistent with the x-axis, represented by the green area. Across all datasets, the performance of auto-tune FLIRT (green area) is always close to or better than that of static FLIRT (blue area). This indicates that the auto-tuning mechanism is effective, and the tuning rate ratio is good. Therefore, the FLIRT used in subsequent experiments is the auto-tune FLIRT with the default tuning rate.

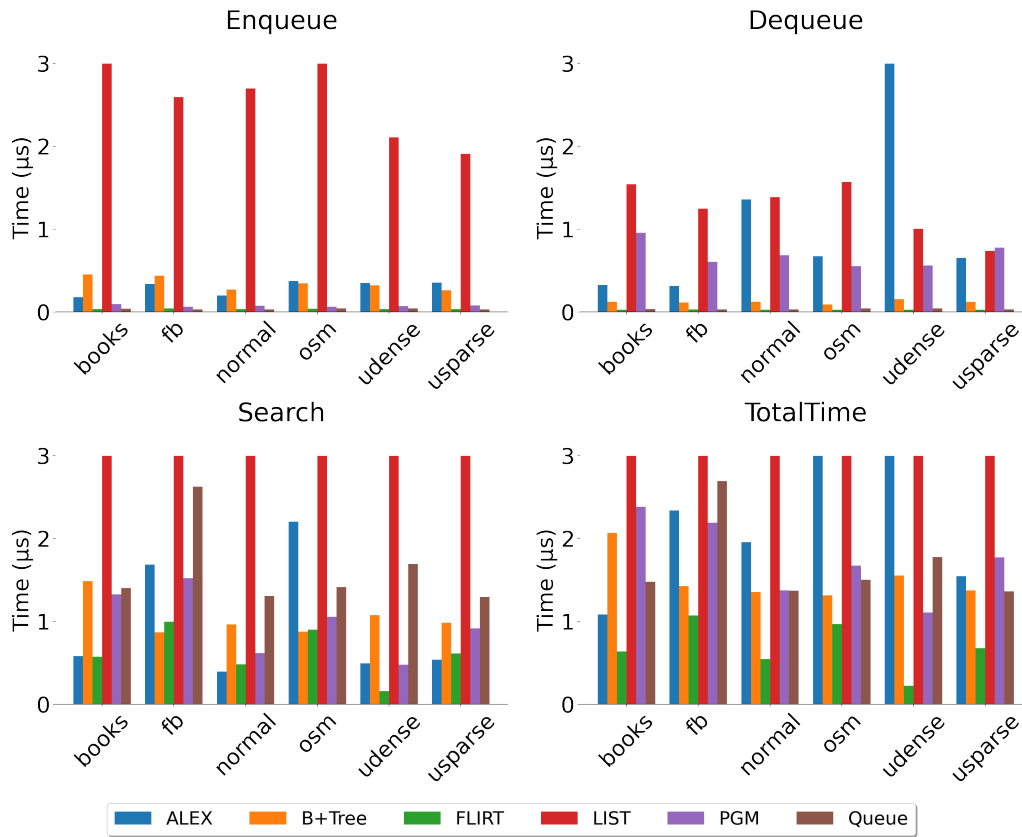


Figure 4.13: Overall performance of FLIRT.

4.5.3 Comparative Analysis

We conduct a comparative analysis of FLIRT against the aforementioned baselines. Figure 4.13 summarizes the total latency performance and its breakdown into different operations: enqueue, dequeue, and search. For the enqueue and dequeue operations, FLIRT is not only consistently faster than its competitors, but also approximately 10 times better than its own search performance. The search cost of FLIRT dominates the performance for all datasets, as evident from the similar patterns and heights from the search and the total sub-figures in Figure 4.13. Nevertheless, FLIRT demonstrates competitive performance in search among baselines, only slightly slower than ALEX in *fb* and *uspase* datasets. However, considering FLIRT's good performance in update operations, its overall performance remains the best. Furthermore, we also analyze the memory overhead, which is calculated as in Equation 4.3. Memory overhead is defined as the percentage difference between the total memory used for the index and the actual data size, providing a measure of the indexing method's efficiency. Specifically, it quantifies how much additional memory the index consumes relative to the data it manages. FLIRT

only requires about 0.04% of the window size. It is significantly less than rival indexes, which consume between 5.5% and 14% of the window size. The lightweight footprint is one of the key contributors to FLIRT’s superior performance.

$$\text{Memory Overhead} = \left(\frac{\text{Total Memory Used for Index} - \text{Data Size}}{\text{Data Size}} \right) \times 100\% \quad (4.3)$$

LIST, designed for stream windows, shows unexpected inefficiency for all operations. It is possibly due to our implementation based on a doubly linked list in C++; the linked list requires traversal through pointers for each record. This structure likely hinders cache efficiency. The queue is constantly good at enqueue and dequeue, as expected. Yet, it is slightly behind other index structures for search in some datasets, which aligns with the expectation. B+Tree performs moderately for all operations across most datasets, but offers more consistency across different data distributions. This is because of its tree structure and distribution-agnostic internal node design.

For ALEX, its design specifically tailored for appending data contributes to its efficient handling of enqueues. However, it underperforms in dequeue, because ALEX requires reconstructing segments when reaching a predetermined emptiness ratio. This results in significant data movement costs. This is especially problematic for the *udense* where data tends to be co-located in single segments, exacerbating the costs during reconstruction. PGM performs well in enqueue since it leverages index-level buffers to accommodate incoming keys. However, its dequeue performance suffers because deletions necessitate data shifting to compact the space. PGM shows a suboptimal search performance since the index and all buffers need to be traversed to retrieve a result.

Window Size

We evaluate FLIRT and other baseline indexes across various window sizes ranging from 10K to 100M keys. As depicted in Figure 4.14, except LIST, which shows subpar performance with larger window size, others are within the same magnitude of costs. FLIRT consistently

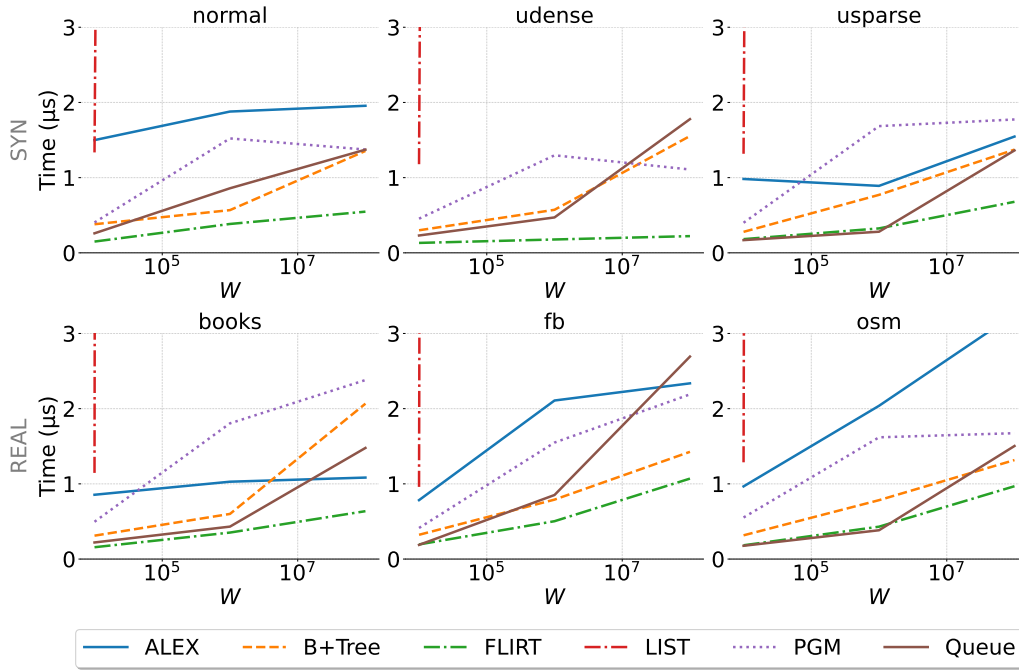


Figure 4.14: Performance of indexes under varying window sizes.

outperforms other indexes across all window sizes. Its execution time increases linearly with the increasing window size, especially for synthetic datasets or model-friendly real-world datasets (*books*). Notably, Queue exhibits competitive performance when the window size is small, but its performance decreases as the window size increases. This is because it lacks an internal structure that could speed up the search for the large dataset.

Number of Executions

The index has to be constructed based on current datasets, and as distribution changes with window advances, the current construction may no longer be effective. Therefore, we vary the number of operations from $1K$ to $1M$ to assess the durability of index performance over time.

Figure 4.15 shows the performance of indexes across a varying number of operations for different datasets. LIST is excluded from this comparison due to its average latency of $2.5 \times 10^4 \mu s$, which significantly exceeds that of other indexes and could obscure their comparative performance in the visualization.

Most indexes, including FLIRT, exhibit consistent performance for all datasets, regardless of

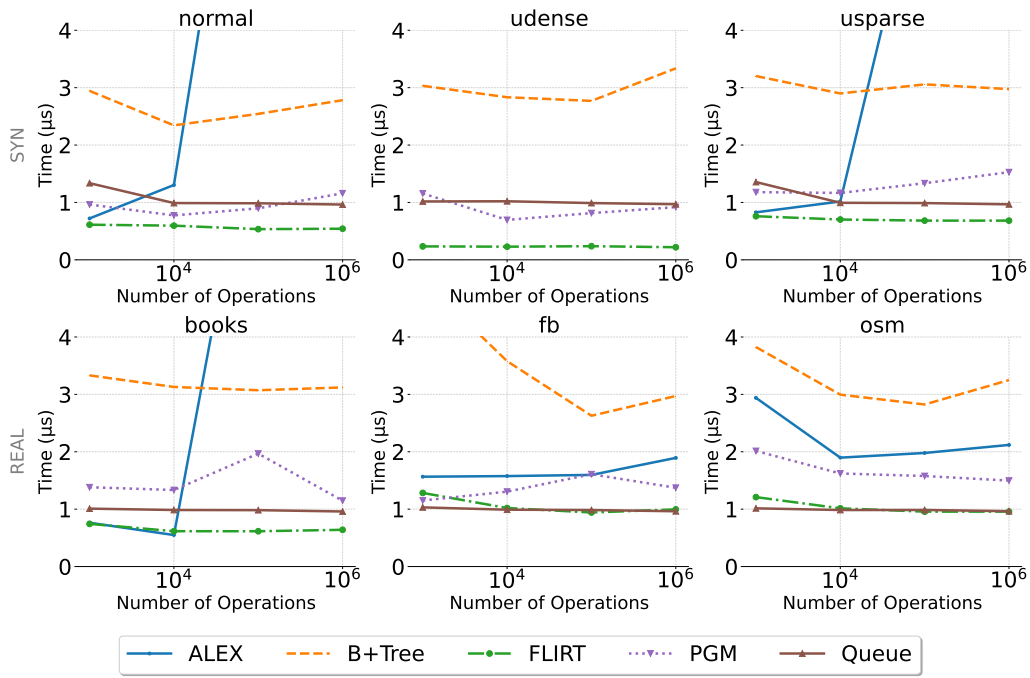


Figure 4.15: Performance of indexes under varying number of executions.

changes in the number of operations. ALEX is an exception; its bulk load might over-fit the current micro distribution for simpler datasets. When facing macro distribution changes, ALEX needs to be restructured to accommodate the new distribution, thereby introducing significant costs. However, for data with hard-to-learn micro distribution, for example, *fb* and *osm*, ALEX can avoid over-fitting, introducing more tolerance and surprisingly resulting in a good performance.

Scan Range

The scan range is important in evaluating index performance for range queries. We conducted experiments with scan ranges varying from 1 to 1000. At lower scan ranges, the lookup cost dominates the performance; while at higher rates, the scan cost becomes more significant.

Figure 4.16 shows that the performance of all indexes increases as the scan range increases. FLIRT exhibits the gentlest slope of the increase due to its compact data structure and the connectivity of its segments. In contrast, PGM and ALEX underperform with a large scan range, as PGM needs to scan multiple trees (buffers), and ALEX frequently jumps between segments to perform scans.

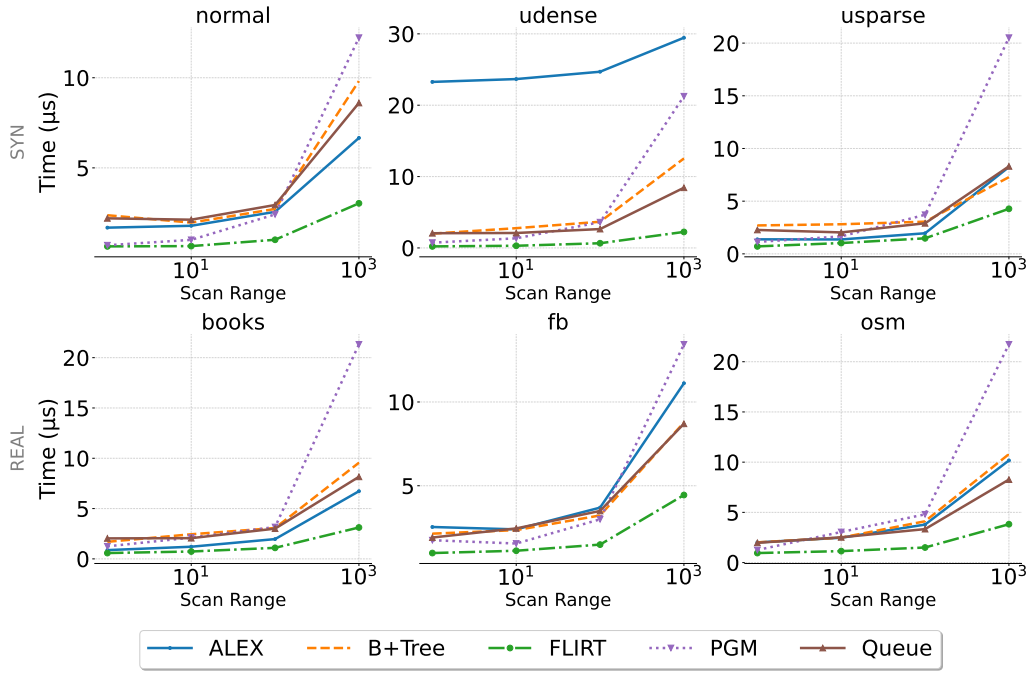


Figure 4.16: Performance of indexes under varying scan ranges.

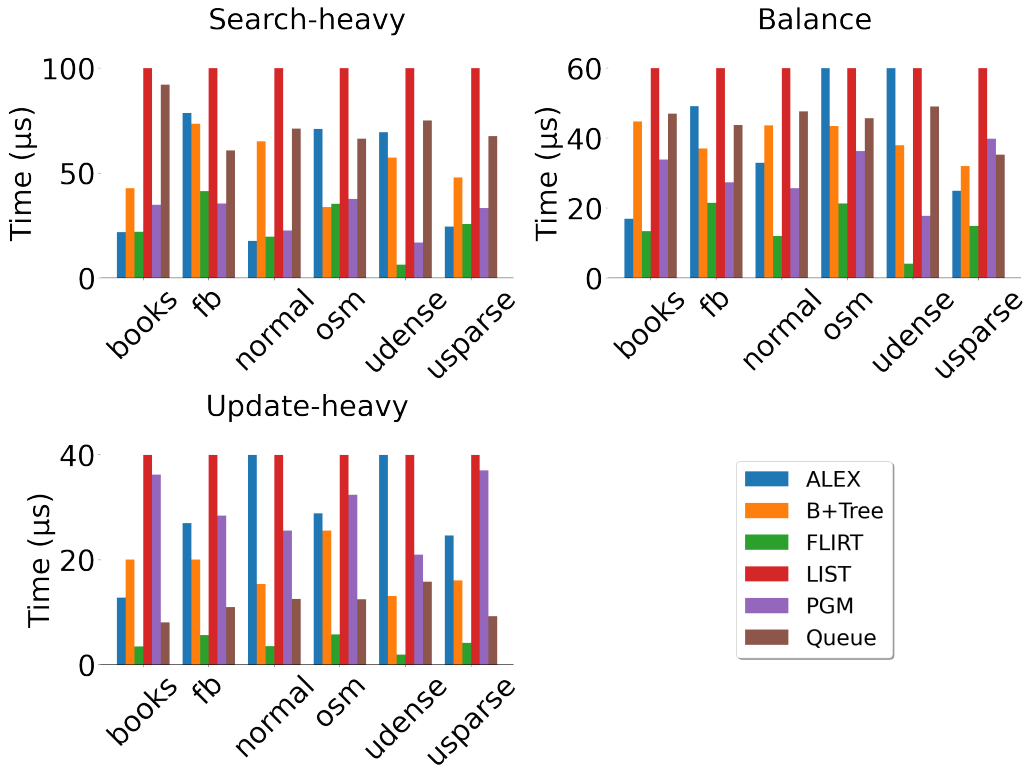


Figure 4.17: Performance of indexes under three sets of read-write ratios.

Read-Write Ratio

A standard operation consists of one enqueue, one dequeue and one search; therefore, its read-write ratio is 1:1. To explore performance under different workloads, we also test search-heavy

scenarios, where 10 search queries are performed for each update (read-write ratio = 10), and update-heavy with one search query for every 10 updates (read-write ratio = 0.1).

As displayed in Figure 4.17, tree-like structures, including B+Tree, ALEX and PGM, exhibit better performance in search-heavy scenarios. Conversely, the queue structure shows superior performance in update-heavy scenarios. FLIRT is consistently good for all workloads, benefiting from its hybrid design that combines queue efficiency for updates and a two-level structure with predictive search capabilities.

4.5.4 Parallel FLIRTs Analysis

In this section, we first evaluate the scalability of PPFLIRT and PSFLIRT regarding search performance and update performance against FLIRT. The evaluation is based on a balanced workload with a varied number of threads from 4 to 24. Then, we further evaluate PPFLIRT and PSFLIRT on the skewed workload outlined in the previous section. For all experiments in parallel environments, the window size and number of operations are configured as $50M$.

Search Performance

From Figure 4.18, the search performance of both parallel FLIRTs increases as involving more threads, particularly evident in PPFLIRT. In PPFLIRT, threads concurrently search their own partition without synchronization. Therefore, PPFLIRT can achieve up to $8.0\times$ increase in throughput compared to FLIRT when scaling to 24 threads.

PSFLIRT achieves a $2.7\times$ increase in search throughput. It is not as scalable as PPFLIRT, probably due to synchronization with the two read-write locks. This effect is especially notable in scenarios like *udesne*, where PSFLIRT tends to cluster data into one *LPQ*. Therefore, search and update are more likely to happen in the *LPQ*, reducing concurrency and increasing contention.

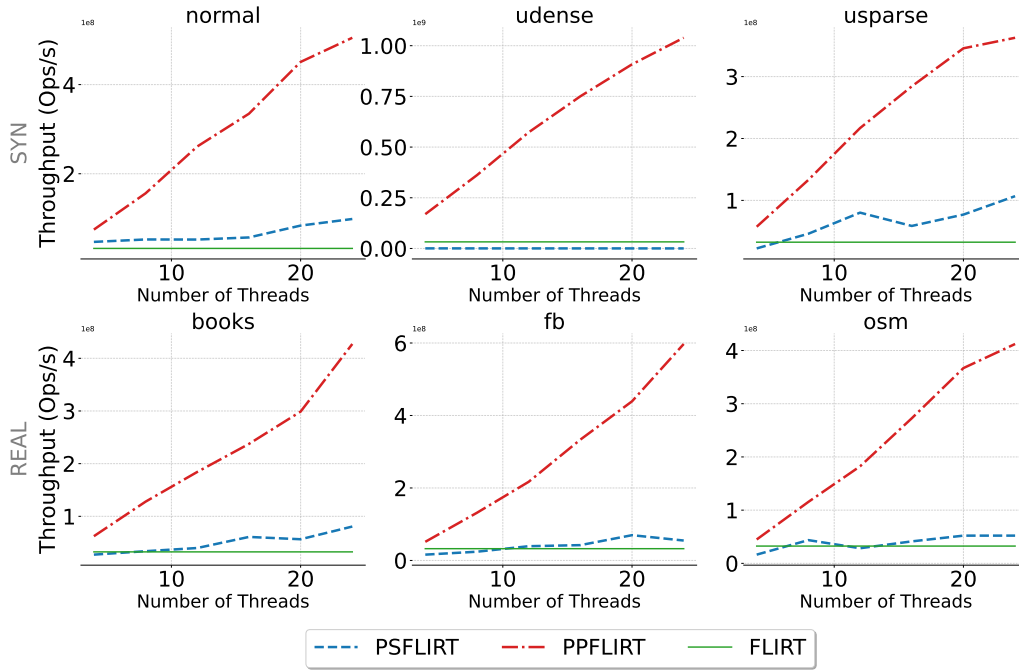


Figure 4.18: Performance of parallel search between parallel and serial FLIRTs.

Update Performance

Figure 4.19 illustrates that both parallel FLIRTs show lower update throughput compared to FLIRT. For PPFLIRT, its updates are continuously synchronized by enqueue and dequeue threads. Additionally, when these update threads complete their tasks, they need to inform another relevant thread to shift the work accordingly. Shifting is more prevalent as the number of threads increases due to the reduced workload per thread. Consequently, PPFLIRT’s update throughput decreases with more threads. PSFLIRT faces the same synchronization issue from the read-write lock to avoid the race condition.

Skewed Query Workloads

To test parallel FLIRT under skewed query workloads, we experiment with skewness degree θ varying from 0.1 to 0.9, running with 24 processes.

As shown in Figure 4.20, PSFLIRT maintains the constant performance when varying θ , while the throughput of PPFLIRT decreases significantly. PPFLIRT’s performance drops because higher θ values cause query keys to concentrate within specific partitions, leading to only one

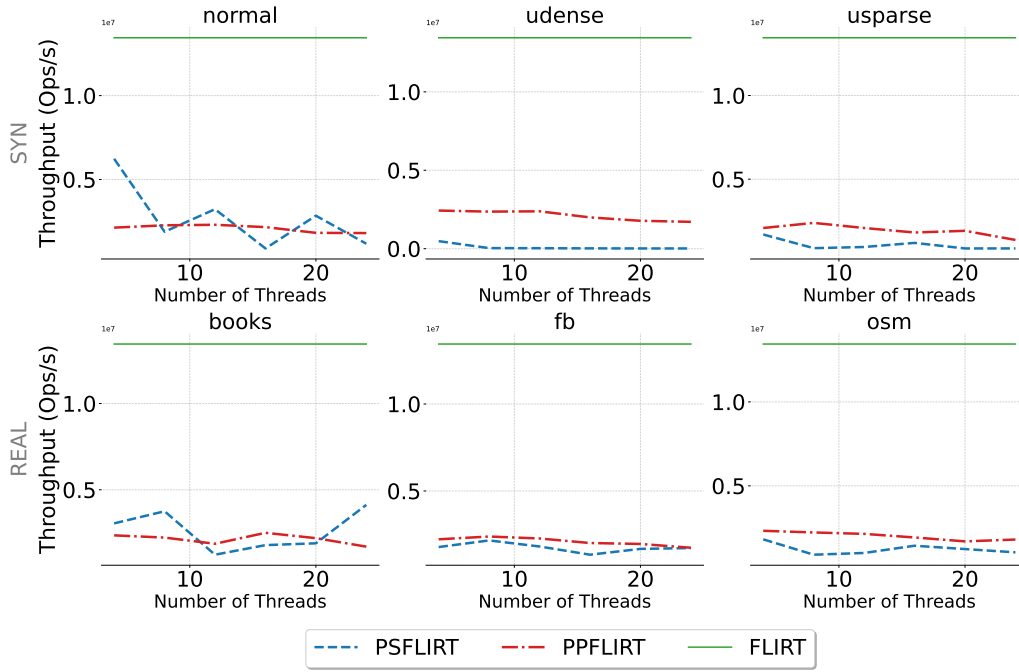


Figure 4.19: Performance of parallel updates between parallel and serial FLIRTs.

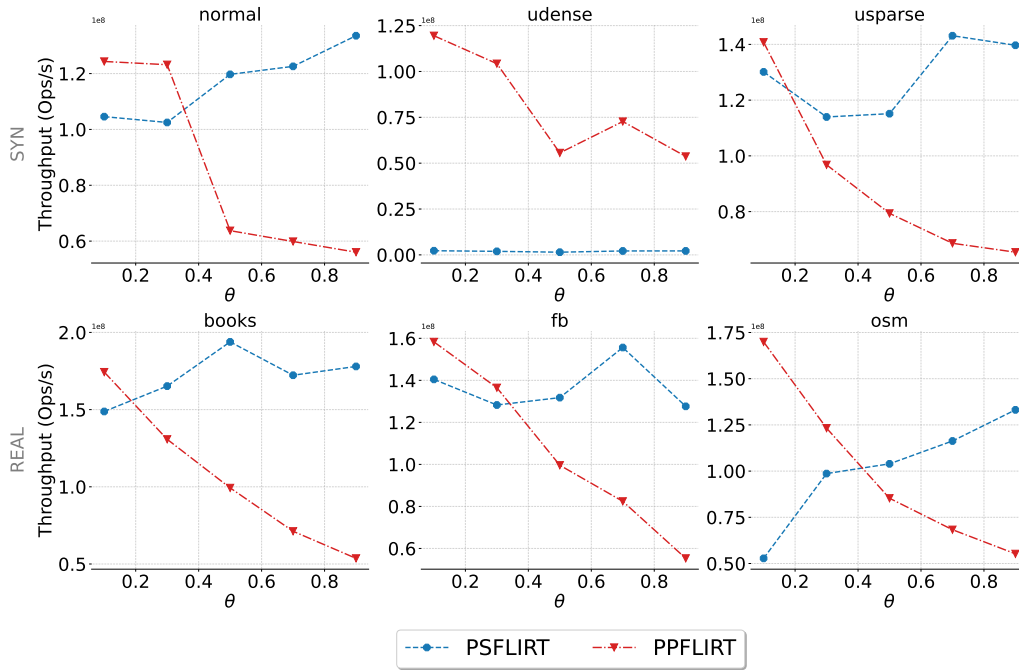


Figure 4.20: Performance under various degrees of query skewness.

thread being actively engaged while the others remain idle.

PSFLIRT still shows the worst performance in *udense* for the same reason in balanced query distribution: there is one *LPQ*. Therefore, frequent locking for synchronization across different operations could happen in the *LPQ*.

4.5.5 Key Insights

We summarize experiments as following key insights:

- **Effective Auto-tuner:** FLIRT’s auto-tuner can effectively and dynamically obtain optimal *Err* on the fly, rendering FLIRT an excellent performance in the stream window.
- **Superior Performance of FLIRT:** FLIRT can consistently outperform competitors, by achieving up to $6.9\times$ speed-up over B+Tree, $8.0\times$ over queue, $108.5\times$ over ALEX, $5.0\times$ over PGM and $10^5\times$ over LIST.
- **Enhanced Search Performance with Parallel FLIRTs:** PPFLIRT and PSFLIRT improve the search throughput of serial FLIRT by providing $32.0\times$ and $4.1\times$ improvement, respectively. However, both variants struggle with update operations due to synchronization overhead.
- **PSFLIRT’s Effective Handling of Skewed Workloads:** PSFLIRT effectively handles skewed workloads and achieves $3.3\times$ improvement compared to PPFLIRT.

4.6 Conclusion

In this chapter, we introduce FLIRT, a novel, parameter-free, two-level learned index designed for sequential IBWP. Blending the efficiency of queue structures with the learned indexes, FLIRT effectively achieves high performance in both updates and searches. Furthermore, we propose two parallel variants, PPFLIRT and PSFLIRT, further enhancing search performance. Our comprehensive experiments endorse the superior performance of FLIRT and its parallel variants. However, FLIRT only supports sequential IBWP update operation, which introduces limitations to its applicability. Therefore, we address this limitation in the next chapter by introducing SWIX.

Chapter 5

A Versatile Learned Index for Stream Windows

5.1 Motivation

In this chapter, we continue our focus on optimizing stream processing at the algorithmic level. Chapter 4 introduces FLIRT, designed for sequential IBWP, in which the order in which keys arrive aligns with the keys themselves. However, there are other cases in practice, for example, in event-time-based windows (introduced in Section 2.2.4), the data arrival time (processing time) may not match with the actual event times due to external reasons (e.g., different network latency) and internal factors (e.g., different scheduling mechanism). In these cases, data can arrive at the window out-of-order; for example, smaller keys might arrive before the larger ones [149]. Furthermore, indexing keys without order guarantees itself is also important for IBWP; for example, applications such as join keys in time- or sequence-agnostic windows do not guarantee any order [104]. Given these complexities, FLIRT’s application is limited. Therefore, in this chapter, we aim to extend learned indexes to support generic IBWP.

To support generic IBWP in which keys are queried but expired based on a different non-indexed timestamp sequence, as depicted in Figure 5.1, indexes need to maintain both key and timestamp to yield accurate query results. Currently, non-stream methods often require

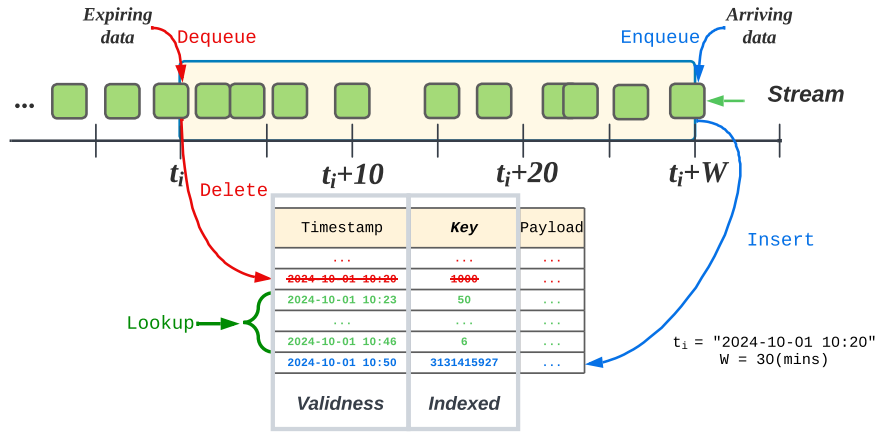


Figure 5.1: Demonstration of a generic IBWP workload in which keys are indexed for querying, and keys' (validness) time-to-live in the index (window) depends on the non-indexed timestamp

additional queries based on timestamps to update the index or necessitate a secondary index for timestamp management. As illustrated in Figure 5.2, traditional and learned non-stream indexes either suffer from increased time complexity due to extra queries or space inefficiencies caused by the secondary index. Therefore, although these non-stream indexes utilize internal nodes (sometimes incorporating models) to enhance lookup speeds and use buffer techniques to accommodate updates, they still struggle to deliver optimal performance under IBWP workloads.

On the contrary, stream-specific indexes exhibit strong performance in at least one metric (such as space efficiency or latency), as shown in Figure 5.2. The queue offers space efficiency due to its flat structure, and the FIFO principle requires no extra effort to maintain time validity. However, queues fall short in terms of search performance. Another stream index, the IMTree, introduced in Section 2.4.4, addresses the validness issue by periodically expiring keys in a bulk-load manner, which not only reduces the cost of validness check but also minimizes the space waste.

Drawing from the strengths of these indexes and inspired by FLIRT's effective performance with its flat structure, we introduce SWIX (Sliding Window Learned Index). SWIX is a versatile learned index structure for a broader spectrum of IBWP scenarios, yielding memory-efficient performance while maintaining low latency in both update and search. SWIX is accessible at <https://github.com/SWIXProject/SWIX/tree/main/src>.

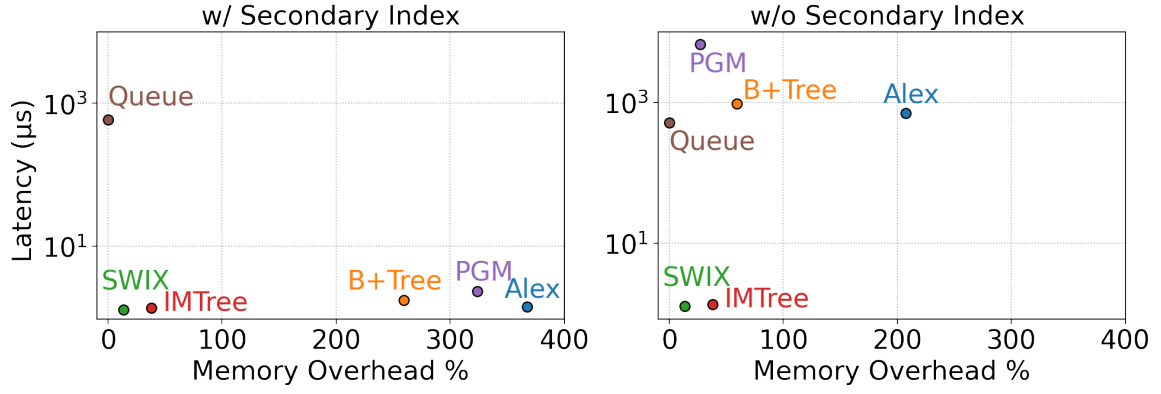


Figure 5.2: Illustration of how different indexes support a generic IBWP workload, where non-stream methods require additional queries or secondary indexes. Note that IMTree and SWIX do not require these mechanisms; hence, they stay in the same location in both figures.

The detailed design consideration will be introduced in Section 5.2, and details of structure and operations are in Section 5.3. We introduce parallel SWIX in Section 5.4 to take advantage of parallelism, followed by evaluation in Section 5.5. Section 5.6 concluded this chapter.

5.2 Design Overview

To design a space- and time-efficient updatable learned index for a generic IBWP workload, we integrate the following features into SWIX:

- *Dynamic Structure*: SWIX uses *DParr*, a novel base structure that combines a linear prediction model for efficient querying and a dynamic expanding/shrinking mechanism (see Section 5.3) for efficient updates while remaining lightweight and accurate. *DParr* uses a continuous memory layout to ensure data locality and scan performance. Furthermore, *DParr* implicitly discards expired data without requiring a secondary index.
- *Flat Structure*: SWIX uses a flat structure with two levels rather than a hierarchical tree structure for space efficiency. In hierarchical structures such as B+Trees, a higher tree causes more redundant keys, but it can reduce the leaf node size. Therefore, the depth of the tree is a trade-off between internal and leaf node performance. However, adding more levels in learned indexes does not necessarily result in lower search times. This is

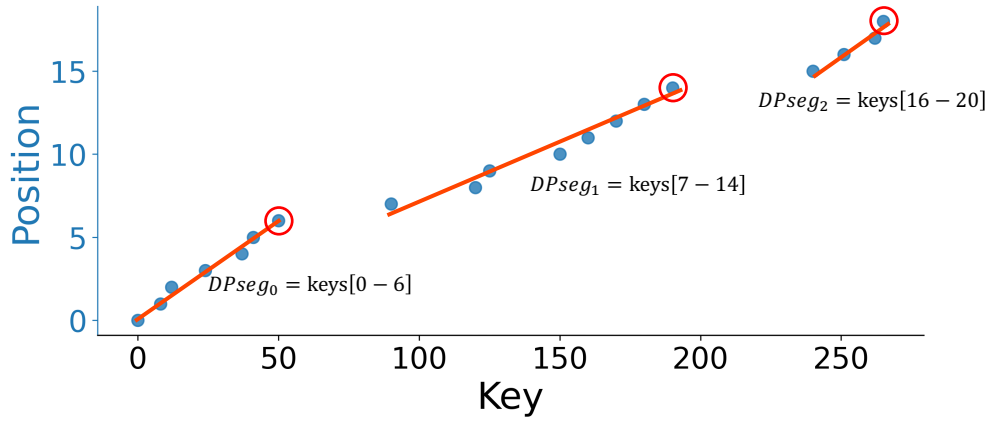


Figure 5.3: Illustration of the piecewise linear functions used in SWIX for segmenting keys.

because local search performance is often bounded by error rather than the leaf node size. Therefore, we prioritize the memory benefits of a flat structure over the uncertain latency improvements that additional levels might offer.

- *Segmentation Mechanism for Bulk-load*: SWIX uses a novel segmentation algorithm called *SD Algorithm* that uses second derivatives to identify points of sudden slope changes as potential split points. These points are marked by red circles in Figure 5.3, which guides data segmentation. The algorithm ensures segments are linearly interpolated and load-balanced without pre-runs, making it ideal for initialization. The detail of the *SD Algorithm* will be introduced in Section 5.3.
- *Piecewise Linear Function*: Similar to other updatable learned indexes, including FLIRT, SWIX acts as a piecewise linear function, mapping the key to its position. However, SWIX employs a nested linear model across two levels, details of which will be discussed in Section 5.3.
- *Auto-tune*: To dynamically adapt the SWIX structure to the changing data distribution, we use an auto-tune algorithm based on a threshold called *tol* to adjust SWIX based on current performance and data distribution, thereby optimising performance on the fly. The *tol* is integrated into the design of different operations of SWIX, such as online segmentation, insertion strategies, search, and retraining. It acts as a central control of the overall performance.

- *Parallelism*: We propose parallel SWIX (PSWIX) to support parallel updates and search. PSWIX differs from existing learned indexes, such as Xindex, FINedex, and PSFLIRT, based on a concurrent design that allows processes to access different parts of data but requires significant synchronization. It also diverges from PPFLIRT, which employs a chained index to partition data and improve load balance but introduces overheads to maintain and synchronize multiple indexes. Instead, PSWIX only maintains a single index and dynamically partitions the lower level to allow different parts of SWIX to update and retrain simultaneously without locking. The detail of PSWIX will be introduced in Section 5.4.

To summarize SWIX from a high-level perspective, SWIX employs a two-level index structure, where each level utilizes a dynamic data structure called *DParr*. This structure combines the location prediction of learned indexes with dynamic restructuring mechanisms to adjust the index during updates. To optimize the performance of SWIX under changing data distributions in the stream window, an online auto-tune algorithm based on *tol* is proposed. Additionally, to handle cold starts where the data distribution is unknown, we introduce a segmentation algorithm called *SD Algorithm* to locate split points and then bulk-load SWIX based on these segments. Like FLIRT, SWIX also incorporates parallel processing to speed up queries.

5.3 SWIX

5.3.1 Structure of SWIX

The basic component in SWIX is *DParr* which is designed to accommodate out-of-order insertions in a generic IBWP workload. *DParr* builds a linear model on maintained keys to facilitate lookup. It utilizes a continuous memory layout to enhance data locality and scan performance and allows dynamic restructuring during updates. Specifically, *DParr* expands itself by incorporating gaps to accommodate new keys when it appears to be full. This mechanism resembles the node capacity management in B+Trees, which fluctuates between half-full ($\frac{f}{2}$) and fully

loaded (f) states. However, B+Tree’s ‘gap’ is only at the end of the node, while *DParr* linearly distributed gaps based on the distribution of current keys. The expansion does not require retraining the model; instead, it proportionally increases the slope and generates gaps. This results in gaps being more likely to occur over a wider key range. In the range, new keys are more likely to be inserted compared to regions beyond the last key in B+Tree. Therefore, this approach minimizes shifts during the insertion of new keys.

Gaps are created not only through extension but also by implicitly flagging expired data. More gaps allow for more efficient insertions but result in space inefficiency; therefore, *DParr* shrinks when gaps make up 50% of its composition. During shrinkage, the model of *DParr* is retrained and valid keys are reallocated based on the model. This retrain introduces overheads but re-calibrates the model for better search performance.

Both levels of SWIX use *DParr*, as shown in Figure 5.4, but each level has a specifically tailored design to meet different operational demands. The data is stored at the lower level segments called *DPseg*. The upper level is called *DPmeta* and manages these *DPsegs*. SWIX is constructed bottom-up; the data is first segmented and organized into *DPsegs*, then SWIX forms *DPmeta* based on *DPsegs*:

- *DPseg*: *DPseg* is used at the lower level to handle keys that are frequently updated; therefore, *DPseg* needs to respond to the update swiftly. Beyond the expansion mechanism in *DParr* that creates in-place gaps, a segment-level buffer is also implemented to accommodate insertions. We limit the buffer size and instantly remove any expired key from the buffer to ensure memory efficiency. The buffer can reduce the error caused by data shifts and minimize the cost of the shifts themselves. However, the buffer compromises monotonicity and data locality within *DPseg*, requiring an extra scan for the buffer for complete and correct query results. Furthermore, retraining in *DPseg* is required to re-calibrate its precision to improve the search performance. Due to the existence of the buffer, the retraining also merges the buffer, which introduces all extra overhead. To mitigate this overhead, we configured the buffer to align with the cache size.

To effectively manage the validity of each key, *DPseg* stores a timestamp sequence along-

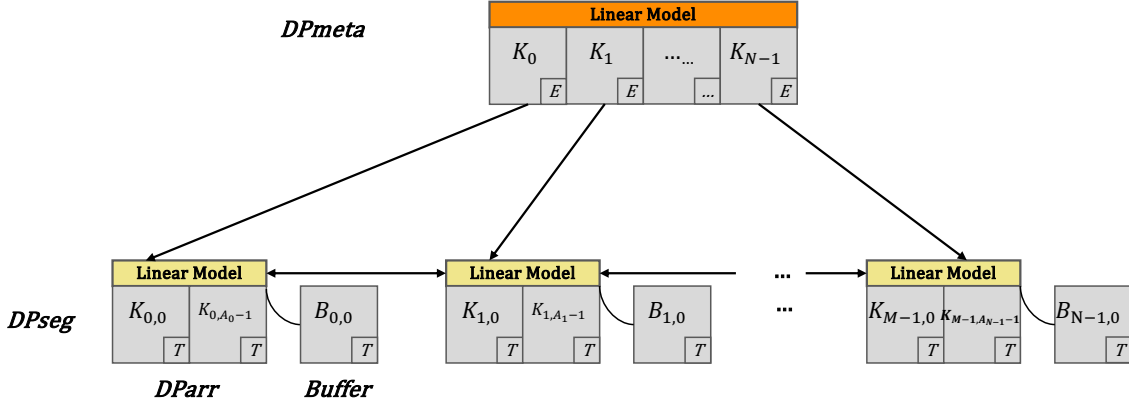


Figure 5.4: SWIX Structure Overview: The top-level *DPmeta* coordinates the *DPsegs*, which manage data segments using linear models at the lower level. Both *DPmeta* and *DPseg* are constructed on the *DParr*. Additionally, *DPseg* integrates a buffer to enhance update performance.

side the keys. The timestamp entry is flagged as a gap (-1) when the key expires. Additionally, *DPsegs* are connected to improve range scan performance by allowing direct traversal in the lower level.

- *DPmeta*: *DPmeta* manages the *DPseg* at the upper level by storing the first keys of *DPsegs*. In *DPmeta*, updates are relatively infrequent, only triggered by retraining or deleting *DPsegs*. Therefore, *DPmeta* only enables *DParr* to retrain based on a higher threshold, up to 20%, as opposed to the 50% gap rule used in *DPseg*. This design ensures compactness to enhance search performance, given that searches are more frequent than updates at this level.

In *DPmeta*, we use a bitmap to indicate where the gap is. The bitmap is lightweight with a size of $\text{ceil}(\frac{M}{64})$ in which M is the total number of entries in *DPmeta* including both keys and gaps. The bitmap facilitates fast queries to find continuous gaps, which can be used in the retrain.

5.3.2 Search

SWIX performs two *predict-correct* operations to look up a key as shown in Figure 5.5. The first *predict-correct* takes place in *DPmeta* to refine the search range to a specific *DPseg*. This

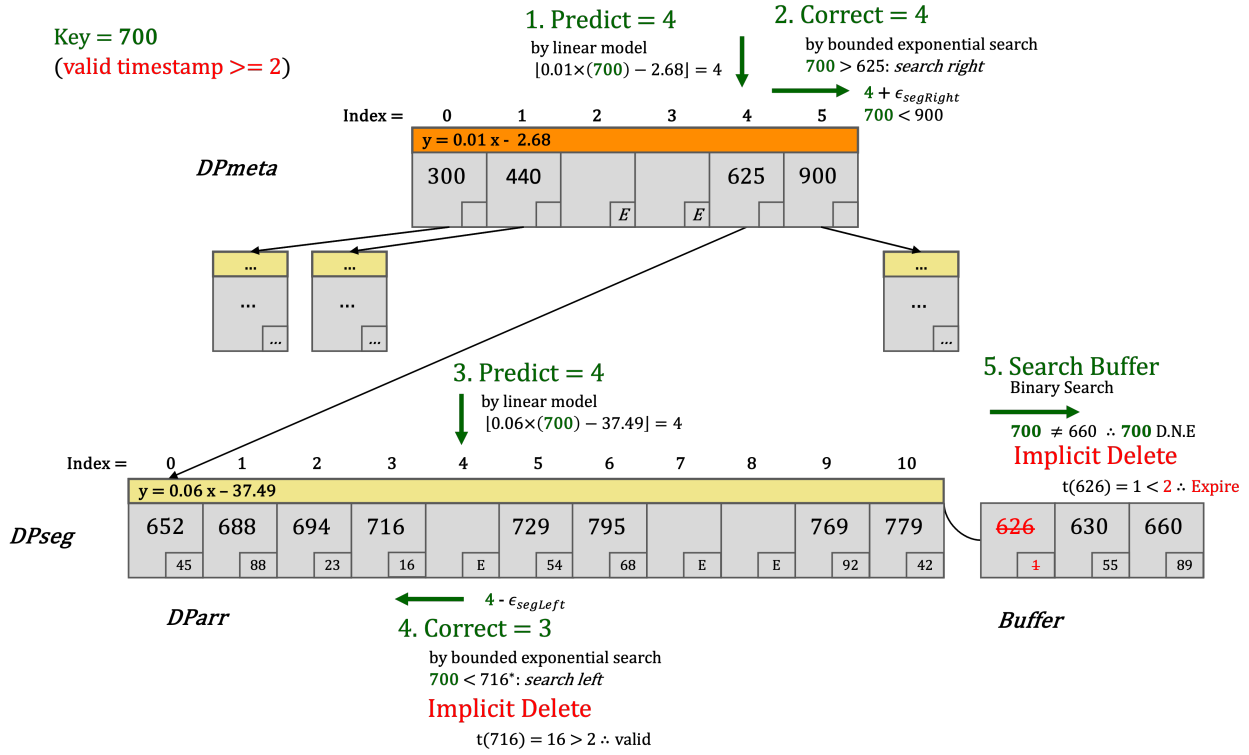


Figure 5.5: Steps for searching a key of 700 in SWIX, where red indicates implicit delete operations.

operation uses machine learning models (linear model in SWIX) to predict the approximate location of the key (*predict*), followed by a local correction mechanism (*correct*) to pinpoint the exact key within a narrowed search interval. Using search bounds $[i_l, i_h]$ introduced in Section 2.4 and Section 4.3, this process narrows the search bound into $[i_s, i_s + n] \cup [0, B]$ where $[i_s, i_s + n]$ is the size of *DParr* in *DPseg*, and B is the buffer size. To further refine the search range, we conduct the second *predict-correct*; the *predict* narrows the range to $[i_s, i_s \pm \text{error}]$, and *correct* attempts to find the exact location within the range. If the key is not found in *DParr*, the search extends to the buffer. Since the buffer maintains the keys' order, the binary search can be used to accelerate the process.

During the search procedure in *DPseg*, the validity of the key must be verified by checking its associated timestamp. If the key is invalid, it is implicitly deleted. For example, in Figure 5.5, the earliest valid timestamp is 2; key 626 with a timestamp of 1 is implicitly deleted during the verification process.

We formulate the search cost of SWIX in the worst case as follows:

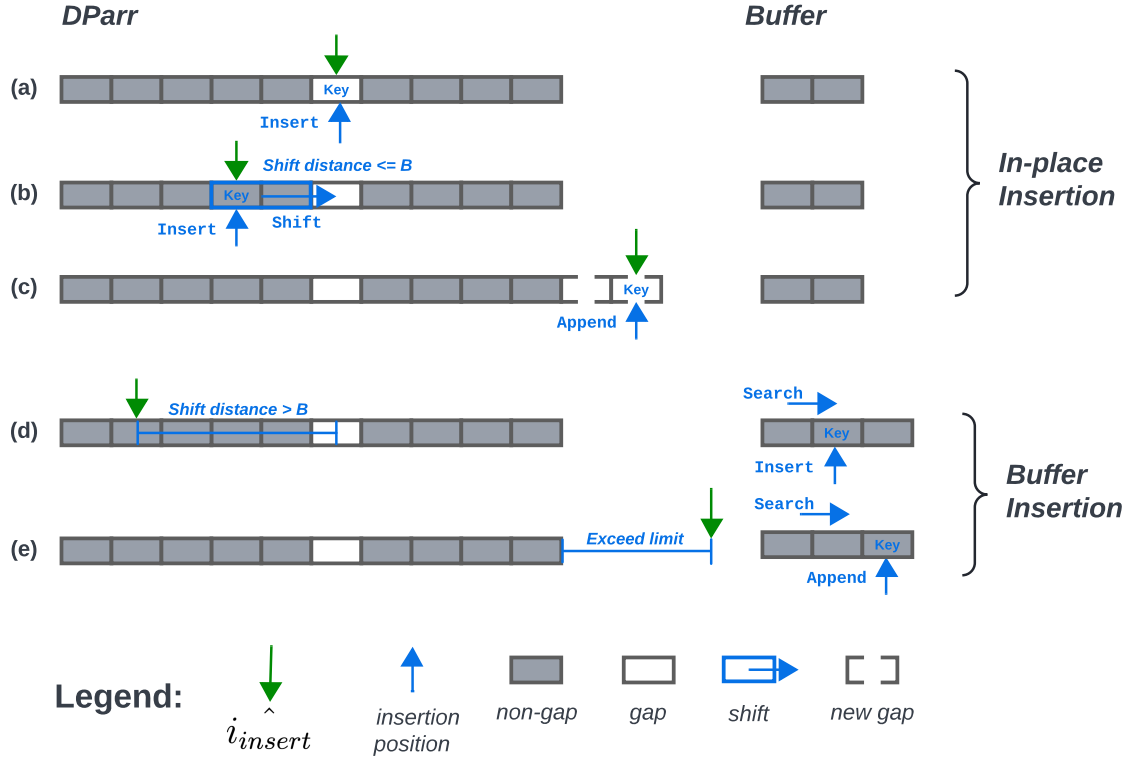


Figure 5.6: Types of insertions after finding the insertion position in *DPseg*.

$$C^{search} = \log \epsilon_{meta} + \log \epsilon_{seg} + \log B \quad (5.1)$$

In which ϵ_{meta} is an error in *DPmeta* in and ϵ_{seg} is an error in *DPseg*. These two errors are negatively correlated and influenced by various factors, including *tol*. A larger *tol* usually results in a larger ϵ_{seg} and a smaller ϵ_{meta} . Conversely, a smaller *tol* tends to produce a smaller ϵ_{seg} and a larger ϵ_{meta} . The cost model also includes $\log B$ because a binary search within the buffer may be required.

5.3.3 Insertion

For generic IBWP workloads, insertions can randomly occur at any position within the index. To maintain the monotonicity, the index needs to shift data to ingest the random insertion. The shift introduces overheads and affects model accuracy. To mitigate these challenges, we incorporate a spare structure design in *DParr*, complemented by a buffer.

The insertion in SIWX requires a lookup to find the insertion position (denoted as $i_{insert}^{\wedge}$) for a new key. Depending on $i_{insert}^{\wedge}$ and the status of $DParr$, SWIX applies one of two insertion strategies: in-place insertion directly into $DParr$ or buffer insertion. The in-place insertion involves three cases:

- *Gap insertion*: If $i_{insert}^{\wedge}$ is a gap, the key is directly inserted into the gap. The insertion location is the predicted location; therefore, the model accuracy remains at the same level after insertion. Furthermore, the gap insertion incurs no additional overheads, making this insertion an ideal scenario.
- *Shift insertion*: If $i_{insert}^{\wedge}$ is occupied, a bidirectional linear scan searches for a nearby gap within a buffer-defined range (B). If a gap within the range is found, keys between $i_{insert}^{\wedge}$ and the gap are shifted towards the gap to make room for the new key. The shift insertion incurs a scan cost up to $2 \times B$ (buffer size), and the shifts cost up to B . The data movement introduces prediction errors. If data shifts rightward, it increases $error_{right}$; if leftward, it increases $error_{left}$. Distinguishing these two errors can reduce the correction range during the search by only applying one appropriate error bound rather than combining both.
- *Extrapolation insertion*: When $i_{insert}^{\wedge}$ is a position beyond the end of $DParr$, we consider the necessity of adding gaps to extend the $DParr$ to reach $i_{insert}^{\wedge}$. We assess if adding the key and filling the resultant gaps would overly ‘dilute’ the $DParr$. If adding these gaps does not trigger shrinkage of $DParr$ (50% of the initial segment size), the key is appended to the $DParr$.

The buffer insertion is utilized to manage exceptions of in-place insertion, especially for shift insertion and extrapolation insertion.

- *Buffer for shift insertion*: If a gap is not found within a buffer-defined distance of B from $i_{insert}^{\wedge}$ in either direction, the key is directly inserted into the buffer. A binary search is performed to locate the insertion position within the buffer. To maintain the order of the buffer, if keys in the buffer are larger than the insertion key in the buffer, they are moved towards the end.

- *Buffer for extrapolation insertion*: If adding gaps directly triggers the shrinkage mechanism, the key is placed directly into the buffer.

When new segments are created through retraining, they must be updated in *DPmeta*. Insertion in *DPmeta* is relatively straightforward, only following the first two in-place insertions strategies. As for extrapolation insertion in *DPmeta*, the key is appended directly to the end without creating gaps for compactness.

The worst-case cost model of insertion can be formulated as:

$$C^{insert} = C^{search} + B \quad (5.2)$$

Besides the cost of searching for the insertion position, the main insertion cost is derived from the maximum shift distance encountered when gaps are not available at the place of prediction, which is bounded by B by design. The ratio of the number of keys that fail to be inserted into the predicted location is influenced by tol . For example, with a larger tol , *DParr* is more linear and has more gaps, reducing the failure rate and likelihood of shifts.

5.3.4 Deletion

Invalid data is implicitly erased during query operations such as search, scan, insertion, and adjustments of *DParr* in *DPseg* such as expansion and shrinkage. SWIX uses the current timestamps, adjusted by a window threshold, to compare timestamps associated with keys to determine if keys have expired during the above processes.

The implicit deletion minimizes the deletion cost and eliminates the need for external structures to assist the validation check. Once invalid data (e.g., expired keys) are found, we convert them to gaps in *DParr* and erase them in the buffer. To ensure memory efficiency, we compact the segment (via retraining) when they reach the maximum number of gaps unique to each segment. If the entire segment consists of invalid data, *DPmeta* flags the segment as a gap.

Deletion in $DPmeta$ could also occur during the retraining of $DPseg$, as multiple segments are condensed into fewer segments. This requires corresponding deletions in $DPmeta$, where reduced $DPseg$ are simply flagged as gaps.

5.3.5 Bulk-load

Algorithm 5 Segmentation Algorithm for Bulk-load

```

1: procedure SPLIT_SECOND_DERIVATIVE( $K$ )
2:   Input:  $K$ : collection of keys, where  $|K|$  is the number of keys.
3:   Output:  $Split$ : collection of index in  $K$  to split.
4:    $\frac{d^2(pos)}{d(k)^2} \leftarrow$  second derivative for  $x = [k_0, k_{|K|-1}]$ ,  $y = [0, |K| - 1]$ 
5:   while  $N < N_{threshold}$  do
6:      $pos = \max(\frac{d^2(pos)}{d(k)^2})$ 
7:     if  $pos = 0$  then
8:       break
9:     end if
10:     $Split.push\_back(pos)$ 
11:     $\frac{d^2(pos)}{d(k)^2}[pos] = 0$ 
12:     $\frac{d^2(pos)}{d(k)^2}.penalize\_neighbors(pos)$ 
13:  end while
14: end procedure

```

The rationale of the bulk-load algorithm is to segment the data such that the error is minimized when the data distribution is unknown (*cold starts*). Since we use linear models, the segmentation should try to make the data distribution as linear as possible. We propose a second derivative segmentation algorithm using a top-down approach, shown in Algorithm 5. The objective is to find optimal split points such that each segment is linearly predictable while minimizing the number of segments to reduce the memory overhead of $DPmeta$. The insight is that second derivatives capture abrupt distribution changes in a function. Splitting segments at maximums can effectively reduce the errors in each segment.

The algorithm begins with a collection of keys representing data points, where $|K|$ denotes the total number of keys. An initially empty collection, $Split$, is prepared to store the indices at which the dataset will be segmented. The algorithm computes the second derivative of the data points with respect to their positions, mathematically represented as $\frac{d^2(pos)}{d(k)^2}$, where $x = [k_0, k_{|K|-1}]$ corresponds to the range of keys and $y = [0, |K| - 1]$ represents their positions.

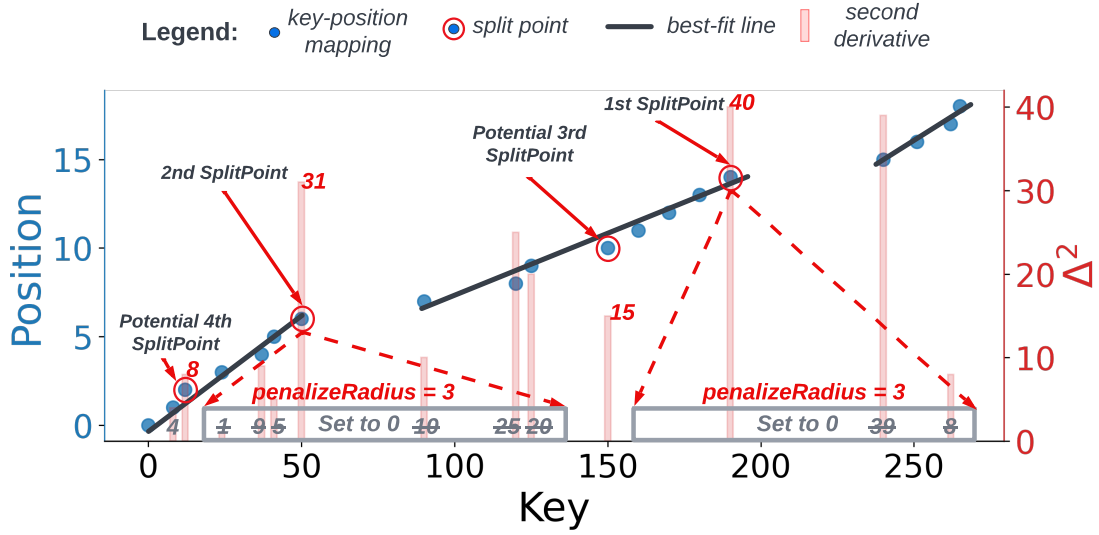


Figure 5.7: *SD Algorithm* for bulk-loading during cold-starts.

The algorithm then enters a loop that continues until the number of splits reaches a specified threshold ($N_{threshold}$). In each iteration of the loop, it identifies the position (pos) with the maximum second derivative value, indicating the most significant change in the data distribution and marking it as an optimal candidate for segmentation. To prevent the selection of split points that are too close to each other, thereby avoiding the creation of excessively small segments, the algorithm applies a penalty to its neighboring positions by setting their second derivative values to zero.

As illustrated with an example in Figure 5.7, the algorithm first locates positions with the highest second derivatives (shown by the red y-axis). We neglect neighbouring points to prevent the segments from being too short. For instance, we ignore the point at position 39, which has the second-highest second derivative but is adjacent to the highest one. This process is repeated until the stop conditions are met. One of the stop conditions is the maximum segments ($0.01 \times W$ as default). We also set an early-stop condition if all remaining second derivatives have similar values. A stop condition with larger second derivatives can result in fewer segments with larger sizes, which may prompt adjustments to the auto-tuner to reduce tol and create more segments to balance performance across different levels.

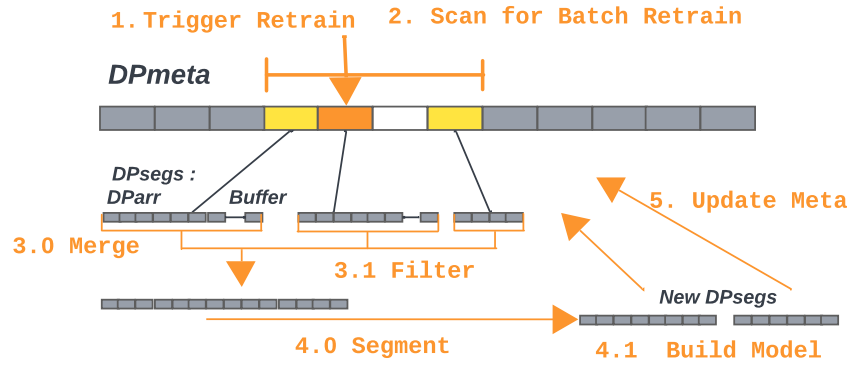


Figure 5.8: Illustration of the rendezvous batch retrain process when the orange segment triggers a retrain.

5.3.6 Retrain

When the buffer is full or *DParr* has excessive gaps (e.g., having 50% gaps), the *DPseg* is retrained. Both buffer fullness and emptiness of *DParr* suggest that the current model of *DParr* cannot capture the new data distribution. Specifically, the two conditions are met when new data is either inserted into the buffer - leading to its fullness, or redirected to a different *DPseg* - causing *DParr* too empty. Therefore, retraining is required to adjust the model to fit the new distribution better and to keep *DPseg* lightweight.

For training in *DPseg*, we propose rendezvous batch retraining, which retrains multiple *DPsegs* collectively, as illustrated in Figure 5.8. This approach minimises frequent updates in *DPmeta* and prevents fragmenting the segments during retraining. When the retraining procedure is triggered, *DPsegs* that require retraining are marked in a bitmap maintained by *DPmeta*. Then, we use the bitmap to identify all neighbouring segments flagged for retraining. Their segments are merged and filtered before applying a one-pass segmentation algorithm to find positions with abrupt changes. Segments are re-split according to a specified *tol* parameter, which controls the size and number of gaps in each segment. Finally, the new segments are updated to *DPmeta*, which can potentially trigger *DPmeta* retraining.

The retraining condition for *DPmeta* is determined by its maximum of gaps (20%). When the retrain is triggered, *DPmeta* is re-fitted into a new linear model followed by integrating an additional 5% of gaps to accommodate future insertions. The fullness in *DPmeta* does not

directly necessitate a retrain; exceeding a search error threshold greater than tol also triggers a retrain to re-calibrate the model for search performance.

The cost model of retraining can be formulated as:

$$C^{retrain} = \underbrace{|DPseg| + M}_{DPseg \text{ retrain}} + \underbrace{M}_{DPmeta \text{ retrain}} \quad (5.3)$$

In which $|DPseg|$ represents the size of the retrained segment, which includes both the $DParr$ and buffer sizes. Although batch retraining encompasses multiple segments, the amortized cost for each can be considered as about one segment. M , the capacity of $DPmeta$, which limits the update cost of $DPmeta$, potentially triggered by $DPseg$ retraining. The cost model also involves the cost of $DPmeta$ retraining (M), since the updates in $DPmeta$ could trigger its retraining in the worst case. The retraining cost is also influenced by the tol : a larger tol increases $|DPseg|$ but decreases M . Furthermore, a larger tol could reduce the frequency of training in both levels but degrade search performance. Conversely, a smaller tol would have the opposite effect.

5.3.7 Auto-tuning

The previous discussions based on the cost model of each operation show that the tuning tol is crucial as it affects all operation performance. To capture how different internal states affect the performance of the index in terms of tol , we derive a heuristic model denoted as Equation 5.4, where each element represents the cost of an operation. Specifically, we conduct multiple experiments with various datasets and index configurations to explore the relationship between tol and the actual performance (time cost). We monitor the actual states from different operations, such as search, scan, and shift distances. We use the AIC [4] as a metric to determine the significance of states through step-wise selection. The process involves iteratively adding and removing states for comparison. We selected primary states including average search distances ($\overline{SearchDPmeta}$ and $\overline{SearchDPseg}$), as both *correct* operations involve exponential searches, making the costs logarithmic. Additionally, we consider neighbor traversals ($\overline{Ptr}$), the

length of merges ($Length_{merge}$), and the length of retraining ($Length_{retrain}$) for our analysis, as outlined in Equation 5.4.

$$\begin{aligned}
C = & w_{s1}(\log(\overline{Search}_{DPmeta}) + \log(\overline{Search}_{DPseg})) \\
& + w_{s2}tol \\
& + w_{sc}\overline{Ptr} \\
& + w_r(Length_{merge} + Length_{retrain})
\end{aligned} \tag{5.4}$$

Where, w_{s1} , w_{s2} , w_{sc} , w_r are the weights for search ($s1$ & $s2$), scan (sc), and retrain (r) operations respectively. The weights are determined based on actual performance measurements. These states with these weights serve as the bridge between tol and performance so that we can adjust the tol to influence these states and consequently optimize performance during runtime. Based on monitoring the states and estimating the time cost, the auto-tuner adjusts the tol when reaching the tuning rate by adding/subtracting tol , similar to the tuning step in FLIRT.

5.4 Parallel SWIX (PSWIX)

5.4.1 Overview of PSWIX

In parallel SWIX (PSWIX), each thread performs operations simultaneously on a subset of data. As Figure 5.9 shows, the meta thread ($thread_0$) maintains the entire $DPmeta$, and worker threads maintain the associated segments and their corresponding part of the $DPmeta$. The meta thread handles all cross-partition modifications and orchestrates the worker threads during updating, retraining, and re-partitioning of $DPmeta$.

The number of segments in each partition is static; however, the number of data in each partition varies as the segment size is non-static. Since SWIX's performance is error-dependent

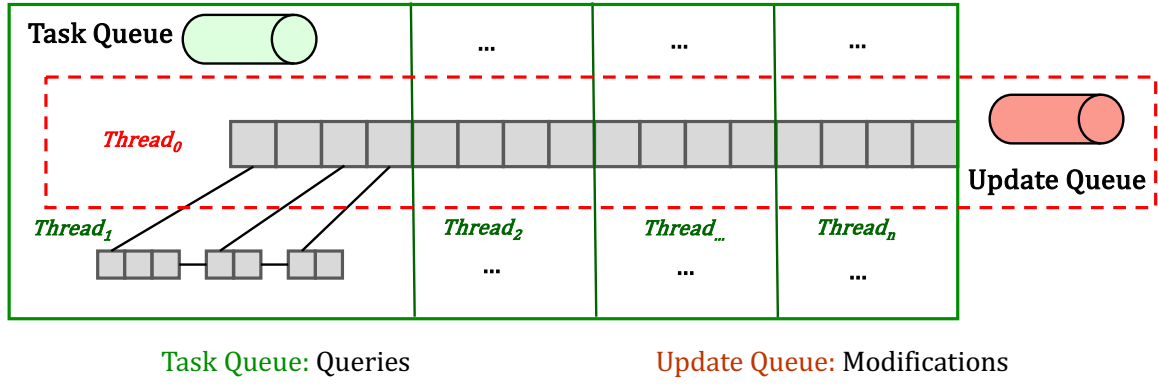


Figure 5.9: Structure of parallel SWIX.

rather than size-dependent, load balancing is ensured by having errors instead of sizes in each partition at the same level.

The $thread_0$ handles all cross-partition modifications and orchestrates the worker threads during updating, retraining, and re-partitioning of $DPmeta$. PSWIX partitions the data for different threads. We add concurrency control measures to avoid race conditions while minimizing the synchronization cost. Segments are connected within each partition and disconnected between threads to prevent concurrent data access. Each thread except $thread_0$ has a task queue from the dispatcher and works asynchronously to increase throughput. The $thread_0$ uses a queue to receive update requests from the rest of the threads.

5.4.2 Operations of PSWIX

Search

The PSWIX search procedure follows the *prediction-correct*. All threads receive the same search query, each checking if the error bound overlaps with its partition in $DPmeta$. If an intersection occurs, the thread engages in a correction process within its segment. If multiple threads overlap with the error bound, the search space is efficiently divided among the threads. And the *predict-correct* operation for identifying the search key in $DPseg$ is executed independently in each thread.

When it comes to range queries, several threads may be involved since the result may span

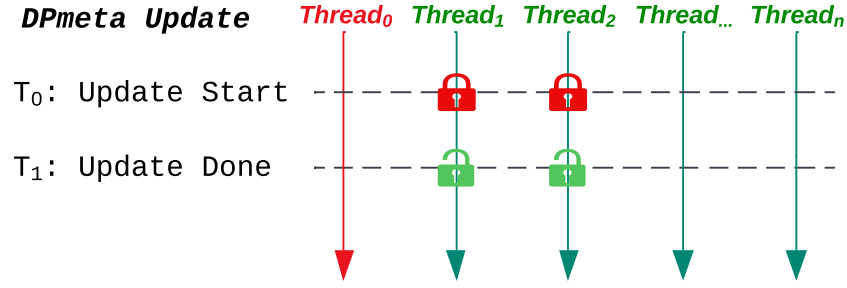


Figure 5.10: Synchronization between threads when *DPmeta* is updated.

across multiple partitions. This case requires a coordinated search in which each thread refines the lower and upper bounds of the range query based on its bookkept min and max values, then performs a localized range query simultaneously. Results are buffered until all relevant threads complete the query.

Update

The update in *DPseg*, including insertion and deletion, is straightforward since it only occurs within a single partition, which mirrors the serial update process.

Updates in *DPseg* can sometimes trigger retraining, which may lead to the changes of *DPmeta*. The insertion into *DPmeta* may require additional synchronization depending on whether there is an available gap in a local partition. If a gap is found in the local partition, then the local thread updates its part of *DPmeta* without synchronization. If no gap is found in the local partition, the local thread informs *thread₀* to take over the update via the update queue. After receiving the request, *thread₀* performs a scan to locate the gap and then locks the corresponding partition to carry out the update. The corresponding thread may continue to execute the query at *DPsegs* level but cannot access *DPmeta* during the update to prevent race conditions, as shown in Figure 5.10.

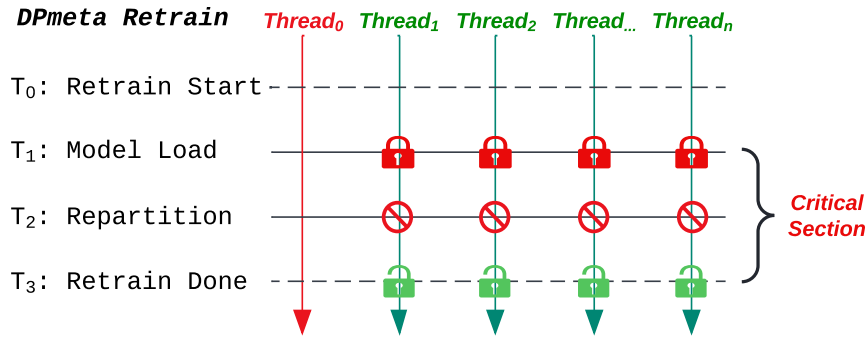


Figure 5.11: Synchronization required when *DPmeta* triggers a retrain.

Retrain

Retraining *DPseg* in PSWIX closely resembles the serial process, but it only applies rendezvous batch retraining within its own partition to minimise the need for synchronization across partitions.

The synchronisation is inevitable for retraining *DPmeta*, but we can reduce the critical section. As shown in Figure 5.11, during retraining, *thread₀* first retrain the model. Meanwhile, other threads continue accessing the existing *DPmeta* without interruption. The critical section begins once *thread₀* starts loading the new model. The *thread₀* then repartitions for each thread. After completing these updates, *thread₀* releases the new *DPmeta* to all worker threads.

5.5 Evaluation

The experiments evaluate SWIX's auto-tuner and error distribution and compare SWIX's performance against the competitors under various streaming parameters and workloads.

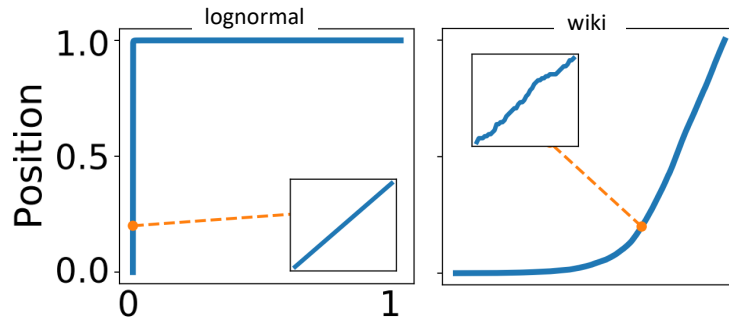


Figure 5.12: Key distribution of *lognormal* and *wiki* datasets from the SOSD benchmark[93].

5.5.1 Experiment Setup

Baselines

The experiments incorporate a range of baseline algorithms previously discussed in Chapter 4, including B+Tree, Queue, ALEX, and PGM. For non-parameter-free indexes such as B+Tree and PGM, various configurations are tested, and the ones yielding the best performance are selected for comparison.

This experiment also includes new competitors relevant to IBWP, including IMTree and FLIRT. IMTree can handle generic IBWP, while FLIRT is limited to sequential IBWP, which can only be used as a baseline in queue-like update environments.

To comparatively evaluate PSWIX, we include PPFLIRT, Xindex, and FINEdex. They employ distinct parallelism designs: PPFLIRT utilizes a chained index in different partitions, while Xindex and FINEdex adopt more concurrent approaches. Comparing them can examine the impact of different parallel designs on performance.

We add a secondary index to non-stream indexes such as B+Tree, PGM-Index, ALEX, XIndex, and FINEdex to handle IBWP workloads. The secondary index includes a mapping from timestamp to key in a sorted queue. To be fair to the baselines, we did not include the memory and search cost of the secondary index in the experiment.

Platform and Benchmark Datasets

We implement SWIX, PSWIX, and baselines such as IMTree and Queue in C++, and other indexes have C++ implementation versions. Therefore, all indexes are compiled using gcc with O3 optimization. All reported results are running on the hardware used in previous FLIRT evaluations, a machine equipped with an Intel E5-2680 v3 @ 2.50GHz CPU with 64GB of RAM running Centos 8.

The experiment is based on all 8 datasets from the SOSD dataset, extending beyond the six datasets previously employed in the FLIRT evaluation to include an additional synthetic dataset (*lognormal*) and a real-world dataset (*wiki*). The distribution of these two datasets is shown in Figure 5.12. We conduct all experiments based on all datasets, but we selected the most representative results to display, primarily focusing on one synthetic dataset *udense* and one real-world dataset *osm*; their distribution can be found in Section 4.5.

Workload and Scenarios

We simulate an IBWP workload with a sliding time window of size W . By default, the window size is constant at $10M$ (10 million keys) for the fixed window scenario. The window size may vary during execution if the arrival/departure rates differ. To simulate that, we generate timestamps from a normal distribution $N(\mu, \sigma^2)$ with varying values of σ^2 . Smaller variances result in data bursts, while larger variances spread the timestamps more evenly. To avoid the generation of negative timestamps, which are not feasible in real-world scenarios, all sampled values are shifted by adding the absolute value of the smallest negative sample to each timestamp. It is important to note that this adjustment increases the mean of the timestamps from the original μ to $\mu + c$, where c is the absolute value of the smallest negative sample.

The default execution is read-write balanced, with each insertion paired with a search. The default number of operations is $40M$, which allows the window to be updated four times. By default, the insertion keys are shuffled to ensure random updates. For sequential IBWP baseline evaluations, the keys are sorted before being appended to the window. Additionally, we test

different read-write ratios by adjusting the number of searches per update. The insertions solely determine a set of operations; therefore, for read-heavy workloads, the set of operations includes a higher number of query operations (more searches) compared to other workload configurations.

Metrics

The latency measures the average execution time for each set of operations and is timed using the Time Stamp Counter (TSC) [83], chosen for its low measurement overhead [164]. The throughput measures the number of operations per second. Memory usage includes the index and data sizes and is reported in megabytes (MB). Memory overhead is also utilized to measure index performance, calculated by the formula $\frac{totalsize - datasize}{data\ size}$. This metric indicates the additional memory required by the index relative to the size of the stored data.

5.5.2 Self Analysis

Auto-tuner

To verify the effectiveness of the auto-tuner, we run SWIX with static values of *tol* varying from 16 to 4096, and compare it with auto-tuned *tol*. In Figure 5.13, the blue line is the latency referencing the left *y*-axis, and the orange line indicates the memory usage using the right *y*-axis. The auto-tuner results are shown as horizontal lines. Memory usage and latency are convex when varying *tol*, which suggests there is a minimum. The near-optimal latency performance validates the ability of the heuristic model to capture time cost and supports the use of the auto-tuner.

However, the auto-tuner could yield a high index size for datasets with strong nonlinearities, such as *osm*, since the heuristic model does not explicitly minimize the memory cost. For *osm*, in particular, the auto-tuner tends to balance space against time by generating many smaller segments to reduce the accuracy penalties.

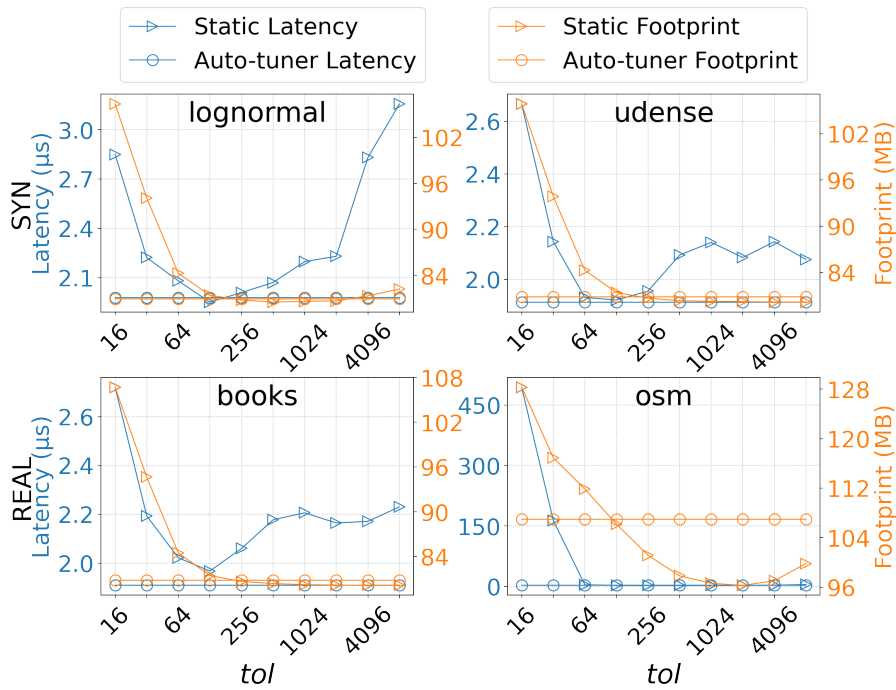


Figure 5.13: Performance of the auto-tuner in terms of latency and memory usage compared to static values of tol .

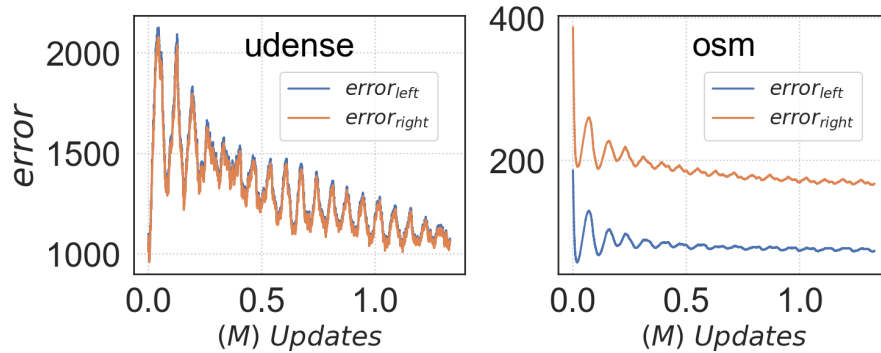


Figure 5.14: The changes in error with the number of updates when applying the auto-tuner.

Segment Error Evolution

To evaluate how the auto-tuner affects the performance in DP_{seg} , we show the change in average segment error during updates in Figure 5.14. The error trends fluctuate from larger ranges to a stable and lower level, indicating that the auto-tuner can stabilize errors, thereby enhancing performance. For data with linear distribution (*udense*), the right error $error_{right}$ and left error $error_{left}$ are mostly balanced. However, for local non-linear CDFs, segments need to be expanded to generate more gaps for insertions. These gaps are located more towards the right end of the segments, which results in a tendency for right shifts.

Index	Metrics	Synthetic Data				Real-world Data			
		lognormal	normal	udense	uspase	books	fb	osm	wiki
Queue	Overhead (%)	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	Latency (μ s)	2857.8	2794.4	2809.8	2793.1	2790.6	2817.2	2807.4	2840.7
B+Tree	Overhead (%)	72.2	67.1	66.9	72.3	72.2	66.9	67.1	66.9
	Latency (μ s)	27.1	26.3	27.7	27.0	27.1	26.5	26.2	26.9
IMTree	Overhead (%)	35.2	35.2	35.2	35.2	35.2	35.2	35.2	35.2
	Latency (μ s)	25.3	24.9	24.7	24.6	24.6	24.4	25.7	24.5
PGM	Overhead (%)	124.9	124.8	124.4	124.5	125.0	125.4	125.3	125.0
	Latency (μ s)	32.6	33.1	32.3	32.4	32.6	33.3	33.5	32.7
ALEX	Overhead (%)	45.6	43.4	55.9	55.9	56.0	50.0	102.8	40.5
	Latency (μ s)	19.4	18.6	28.0	20.1	24.7	32.2	31.4	24.7
SWIX	Overhead (%)	7.0	7.2	6.8	7.0	6.8	32.4	39.1	14.3
	Latency (μ s)	19.2	19.6	18.8	18.5	18.3	25.0	26.0	22.8

Figure 5.15: Summary of the average memory overhead and latency for all indexes and datasets.

5.5.3 Comparative Analysis

We evaluate SWIX against baselines under various workloads (stream characteristics). Figure 5.15 summarizes all comparative experiments. We highlight the best-performing index in blue for latency and yellow for overhead for each dataset. In terms of latency, SWIX is shown to have the overall best performance across datasets only to be beaten by ALEX for *normal* and IMTree for *fb* & *osm*. Regarding memory overheads, SWIX shows the lowest overhead, except for the *osm* dataset.

Regarding memory overhead, tree-like structures are generally shown to have high memory usage, as evidenced by the B+Tree. Learned models can compact the data and significantly reduce the memory overhead of tree structures, as shown by ALEX. Interestingly, the IMTree uses two trees, which significantly lowers the memory overhead compared to the B+Tree by lowering the height of the trees. The search tree of IMTree does not receive any updates and can be fully compacted (average occupancy of 99%). The smaller insertion tree has an average occupancy of 71%, using less space than a single B+tree with a 63% average occupancy. However, utilizing multiple trees does not necessarily decrease overhead; for example, PGM's

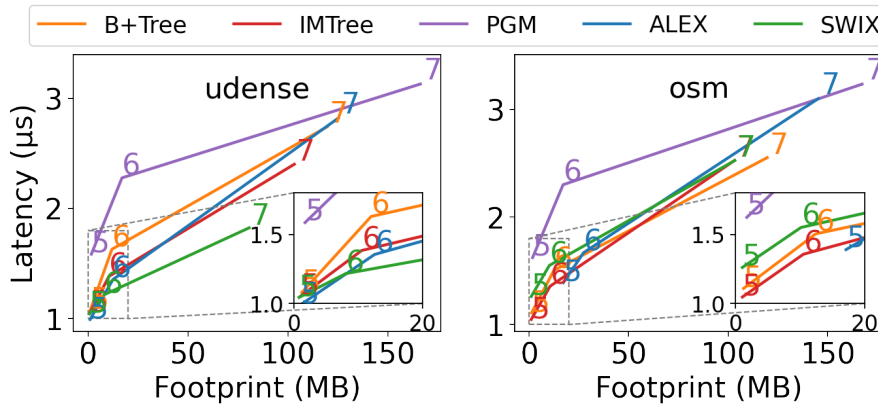


Figure 5.16: Speed vs size performance of indexes under varying window sizes.

use of multiple sub-indexes increases memory consumption because half of the space serves as a buffer for insertions. The key is to minimize unused spaces, which is why SWIX employs a shrinkage mechanism in *DParr*. Combined with its two-level single index design, SWIX achieves efficient memory management.

Like the memory findings, learned indexes cannot achieve low latency with most real-world datasets, especially *osm* and *fb*, since linear models in learned indexes cannot easily capture their distribution. To maintain the performance, the learned index has to compromise in having a deeper structure. For example, ALEX frequently splits during retraining when dealing with non-linear data. This frequent splitting is necessary to accommodate non-linear distributions but significantly increases the costs associated with tree traversal. In contrast, SWIX utilizes a flat structure and batch retraining to prevent the structure from becoming too deep or wide. Therefore, SWIX performs relatively better on non-linear data. Surprisingly, the PGM is shown to be distribution-independent, suggesting that its log-merge tree structure can effectively buffer distribution changes.

Window Size

We vary the window size from 10^5 to 10^7 to examine the scalability of indexes. In Figure 5.16, the number indicates the exponent values of each window, the x-axis is the footprint, and the y-axis represents the latency.

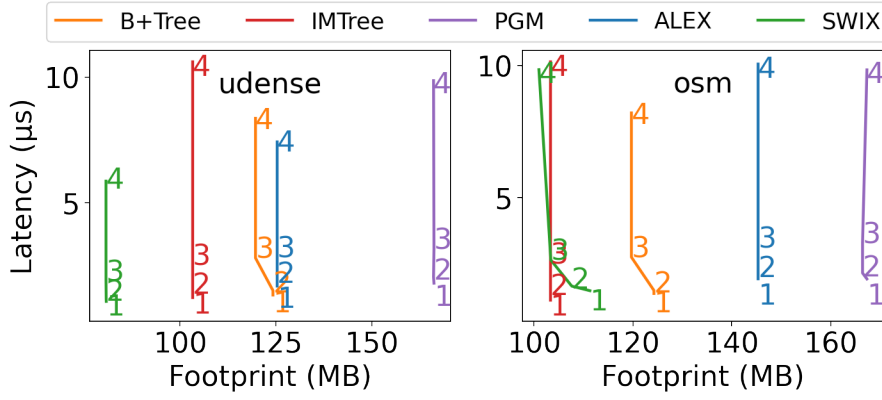


Figure 5.17: Speed vs size performance of indexes under varying scan ranges.

Closer to the left-bottom corner indicates better time and memory performance, as SWIX demonstrated in the *udense* dataset. To further evaluate the performance, we monitor stats including average search distances ($\overline{Search}_{DP_{meta}}$ and $\overline{Search}_{DP_{seg}}$) and number of segments when scaling window size. The search distance increased by 1.36x, and the number of segments increased by 4.85x, considerably smaller than the 100x increase in window size. This indicates the auto-tuner can keep SWIX compact and concise for easy-to-learn distributions. However, for hard-to-learn distribution, the number of segments increases 63X, and retraining length also increases by 24x, indicating that the auto-tuner has to create more segments to maintain a low *tol*. As a result, the performance shows a slight decrease but remains competitive with other baselines.

Non-learned indexes show a consistent pattern for different datasets, where latency increases with window size across different datasets. Baseline learned indexes perform poorly in real-world datasets. The performance of PGM is as expected. ALEX faces challenges with *osm*, since it generates more than 600 internal nodes at a larger window size, which is 3x more than the number of B+Tree internal nodes. This is why ALEX appears in the top left corner of the *osm* figure.

Scan Range

We vary the scan range from 10^1 to 10^4 and demote the exponent in Figure 5.17 to label the change. The change in scan range directly impacts the latency; therefore, all indexes exhibit a

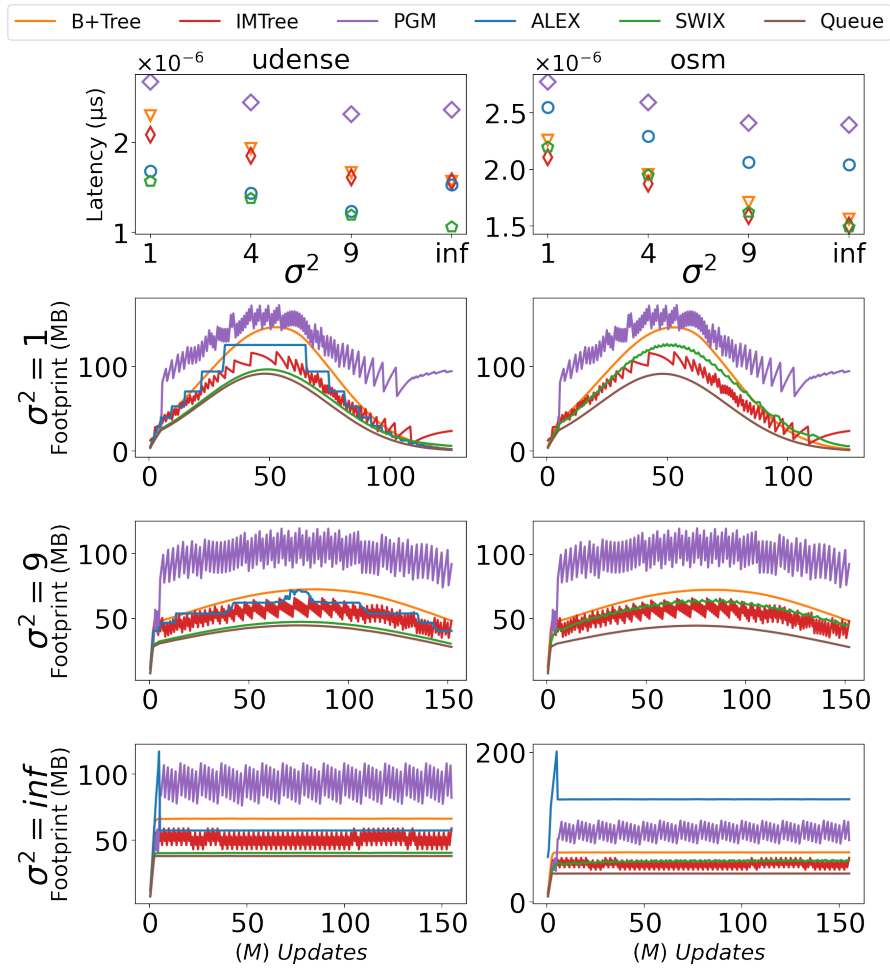


Figure 5.18: Evaluation of indexes under varying degrees of skewness. The top row shows the performance in terms of latency. The bottom three rows display the memory usage under continuous updates.

trend similar to a vertical line. Only two indexes show changes in memory overhead due to the varying scan range.

For B+Tree, the size varies because different fan-outs are selected to optimize performance under varying experimental parameters. As for SWIX, the change in memory overhead is due to the flagging of expired keys during scanning. A large scan range is more likely to trigger shrinkage due to excess gaps in *DParr*; This mechanism allows SWIX to become more memory efficient with a larger scan range.

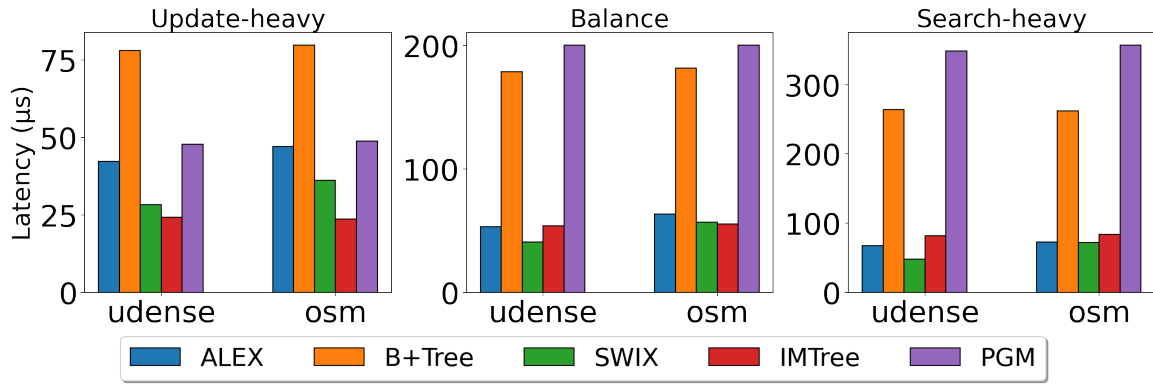


Figure 5.19: Latency performance for three read-to-write ratios.

Skewness in the Arrival Rate of Keys

The velocity of data arrival can vary in the stream window. When the arrival rate is higher than the expiration rate, the window expands, known as a data burst. Conversely, the window size could shrink with a low arrival rate. We conduct experiments with varying arrival rates to assess the adaptability of indexes. As previously introduced, we use the σ^2 to control the skewness of the distribution. In Figure 5.18, the top row illustrates the time efficiency against σ^2 , and the bottom three rows provide a temporal view of the memory footprint over updates from bursty to even distributions ($\sigma^2 = 1, 9$, and inf).

In the top row of Figure 5.18, both ALEX and SWIX show excellent performance in *udense*. However, ALEX cannot maintain this level of performance with the *osm* dataset. The performance of other baselines is consistent across different datasets. As for memory footprint, the queue serves as a baseline to reflect the actual window size since queues have minimal overhead that can be ignored. SWIX remains close to the queue in *udense* but deviates slightly in *osm*. Nevertheless, the overall memory profile of SWIX is smooth, credited to its effective shrinkage mechanism and auto-tuner. However, memory profiles of ALEX, IMTree, and PGM show either a zigzag or saw-like spiky pattern due to their rigorous extension, batch deletion, or merging mechanisms.

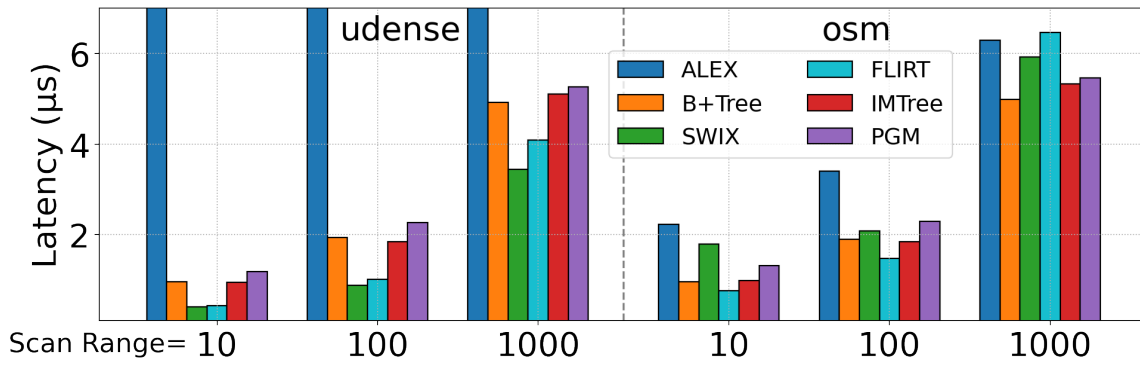


Figure 5.20: Performance of indexes under varying scan ranges in a sequential IBWP workload.

Read-to-write Ratio

We vary the read-to-write ratio from 1:10 (update-heavy workload) to 10:1 (read-heavy workload). Results are shown in Figure 5.19. The memory usage does not vary with the workload; hence, we do not present the results here. We kept the update length constant at 400M to simulate these workloads, each with differing numbers of searches. There are 10x more search operations in the read-heavy case compared to the number of searches in the balance case. Therefore, the different workloads cannot be directly compared; we cannot conclude indexes are better at updates. In reality, for most indexes, updates consume more than 40% of the latency compared to point lookups.

In update-heavy scenarios, which more closely resemble the high-velocity dynamics of data streams, SWIX consistently achieves the best time efficiency compared to other non-stream indexes. The lightweight design of the structure and dynamic *DParrs* significantly enhance its update performance. However, it does not surpass IMTree, which benefits from an efficient tree-level buffer design for insertions. On the other hand, in read-heavy scenarios, which approximate general index settings, SWIX also demonstrates commendable performance. The predictive model primarily drives its search efficiency, explaining why indexes like ALEX may outperform others in specific instances due to their optimization for query-heavy environments.

Sequential IBWP

We simulate a sequential IBWP workload where data is appended to the index. We include FLIRT in this experiment as shown in Figure 5.20. FLIRT outperforms other indexes in both datasets as expected, especially with a small match rate. FLIRT ensures an error threshold by creating small segments, leading to an excellent point lookup performance, even in *osm*. However, as the scan range increases, FLIRT has to scan more segments, decreasing the performance by increasing pointer traversals.

SWIX also displays competitive overall performance since it allows *DParr* to extend without affecting accuracy. However, SWIX shows performance degradation when there are sudden macro-level distribution changes over a short period. This is due to the auto-tuner struggling to respond to sudden distribution changes in the last segment as it tries to balance the performance of the entire index. Although ALEX has consideration for handling out-of-bounds inserts, it still encounters challenges in the *udense* workload. We suspect that ALEX places many keys in a single segment since the micro-level distribution is nearly linear. This results in a single larger segment that not only heavily degrades update performance but also puts a heavy workload on its expansion mechanism. SWIX, however, avoids this issue through its auto-tuner, which penalizes overloading a segment to maintain load balance. This allows SWIX to achieve a $57.95\times$ speed up over ALEX.

5.5.4 Evaluation on Parallel Learned Index

The parallel evaluation involves the generic and sequential IBWP workloads.

Generic IBWP

Figure 5.21 reports the throughput for different workloads, from update-heavy to search-heavy, running on various numbers of threads. The bottom-right figure in Figure 5.21 also displays the memory usage across varying threads.

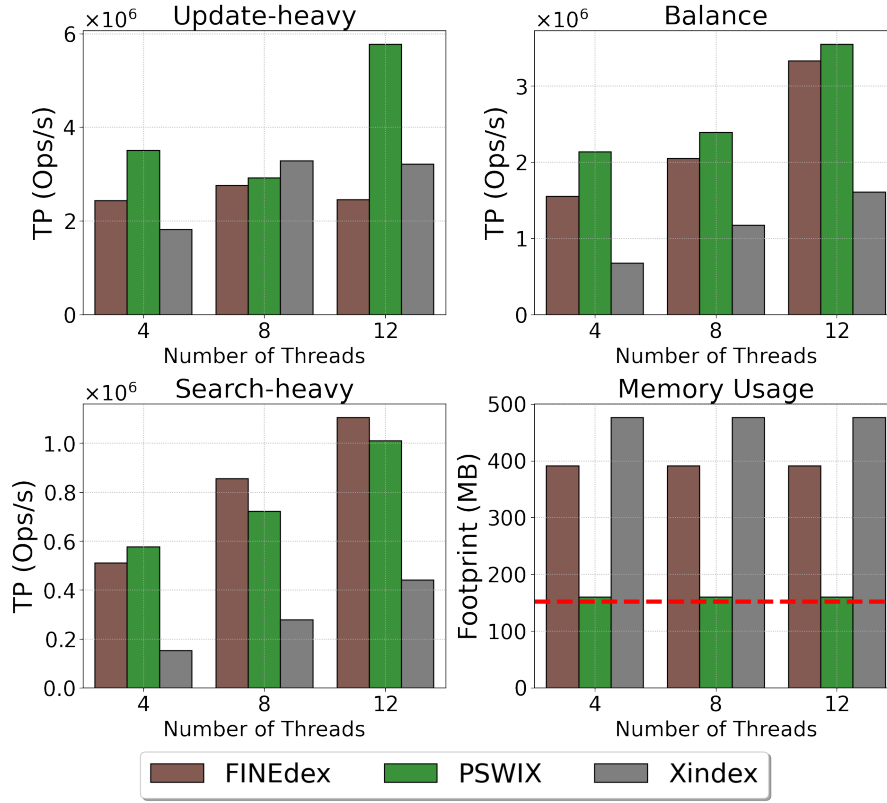


Figure 5.21: Performance of parallel indexes for three read-to-write ratios.

As previously mentioned, PSWIX adopts a distinct parallelism approach compared to XIndex and FINEdex. PSWIX uses a divide-and-conquer approach, distributing portions of $DPmeta$ and corresponding $DPsegs$ across different threads. Each thread only handles queries within its designated segments. The design leads to PSWIX scaling well with an increased number of threads in update-heavy workloads due to its point-lookup query nature, where PSWIX ensures performance by not involving other partitions. However, PSWIX struggles to maintain performance in search-heavy scenarios where scan ranges cover multiple partitions. In contrast, XIndex and FINEdex opt for concurrent designs in which the threads are allowed to access the index freely. The main difference is that FINEdex adopts a finer-grained locking mechanism and efficient in-place buffer, which enables it to scale better than others under search-heavy workloads.

The bottom-right sub-figure shows a constant pattern of memory usage across different numbers of threads. PSWIX achieves memory efficiency through its lightweight structure. In contrast, FINEdex and Xindex use a tree structure, which introduces redundancy. Additionally, Xindex

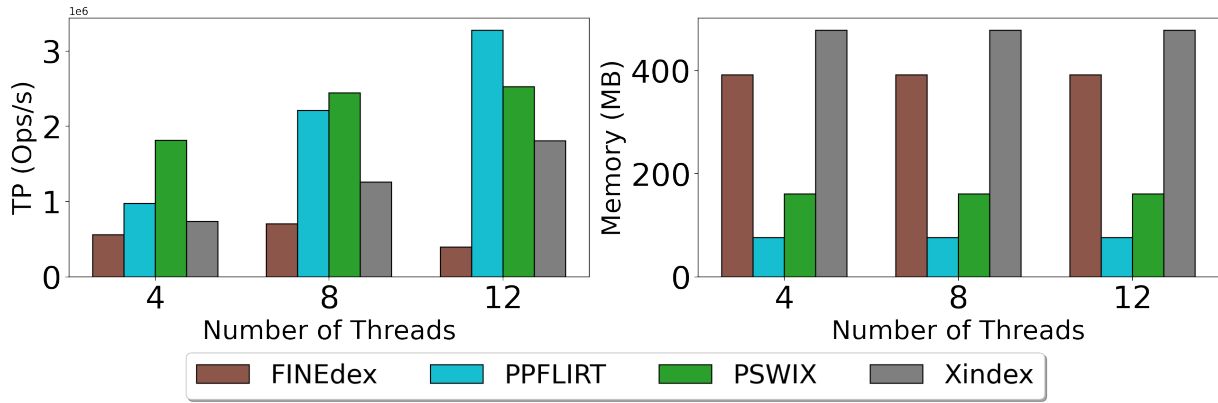


Figure 5.22: Parallel evaluation under a sequential IBWP workload.

consumes the most memory due to its complex buffer groups.

Sequential IBWP

Figure 5.22 presents the throughput and memory usage of parallel learned indexes. FINEdex and Xindex struggle with queue-like updates, as their concurrent design results in contention among threads when only updating the last segments, thereby increasing the synchronization overhead. In contrast, PSWIX allows only the thread managing the last partition to handle updates, effectively avoiding the synchronization of the enqueue operations. PPFLIRT utilizes a chained index design that significantly minimizes the synchronization requirements. Moreover, PPFLIRTs do not use a buffering technique to ingest random insertion, enhancing its compactness and efficiency. Therefore, PPFLIRT achieves both time- and memory- efficiency.

5.5.5 Key Insights

We summarize the experiment with the following key insights:

- **Effective Design:** The design features of SWIX, including its flat structure, dynamic *DParr*, and auto-tuner, have proven effective in optimizing both time efficiency and memory usage across various experimental setups.

- **Consistent Low Latency Performance:** SWIX consistently shows competitive performance across various experimental setups. On average, SWIX can achieve $1.59\times$ speed up over learned indexes, $1.30\times$ over traditional indexes, and $1.20\times$ over stream-specific tree-based indexes.
- **Consistent Lightweight:** Like time efficiency, SWIX consistently demonstrates low memory usage across different workloads. It, on average, only takes up 25.91% of learned indexes, 22.22% of traditional indexes, and 42.79% of stream-specific tree-based indexes.
- **Performance Scales with Parallelism.** For a generic IBWP workload, PSWIX can achieve up to $1.90\times$ throughput over FINEdex and $3.45\times$ over Xindex while only consuming 41% and 34% of the memory used by these indexes, respectively.

5.6 Conclusion

In this chapter, we present SWIX and its parallel version, PSWIX, which are designed to handle the generic IBWP. For a generic IBWP workload, the order of data arrivals does not necessarily align with their key order. This misalignment requires the index to effectively handle random insertions in high-speed stream data while maintaining the timestamp for validity checks. SWIX borrows the flat structure from FLIRT and incorporates a dynamic *DParr* and an auto-tuner to adapt to changing data distributions. Comprehensive experiments show that both SWIX and PSWIX can effectively handle generic IBWP under varying settings and workloads.

Chapter 6

Conclusion

6.1 Summary of Thesis Achievements

In this thesis, we present optimizations at both the system and algorithm levels to meet modern stream processing requirements, such as explicit support for state management, scalability, maintaining accuracy in query results, and providing instantaneous responses.

At the system level, Chapter 3 answers the research question, "*How can state management optimizations be designed and implemented in a stream workflow system, and how do we achieve scaling for a diverse range of platforms?*". In this Chapter, we optimize a stream workflow system, dispel4py, in terms of scalability and state management. Specifically, we first introduce scaling techniques, *dynamic allocation* and *dynamic auto-scaling*. The *dynamic allocation* enables dynamic resource allocation by scaling resources for each task, while the *dynamic auto-scaling* enhances efficiency by automatically adjusting the number of processes deployed in the workflow. However, these scaling techniques add complexity to state management, leading to difficulties in maintaining state consistency. To address this, we introduce *hybrid*, which applies a sophisticated state migration mechanism to support state management during scaling. The comprehensive experiments based on different platforms and use cases show that the three innovative optimizations enhance the performance, efficiency, and state management capabilities of dispel4py, respectively. We believe these optimizations also pave the path for the further

optimization of related stream systems.

At the algorithm level, Chapter 4 and 5 present new learned indexes for different types of IBWP to answer the question "*How can learned indexes be designed to enhance stream processing, particularly in addressing challenges including updatability and temporal ordering, and what performance gains can learned indexes bring to stream window processing?*".

In Chapter 4, we introduce FLIRT, designed to effectively support sequential IBWP, in which the key arrival order aligns with the key's sequence. Sequential IBWP is common in the processing time windows. FLIRT employs a queue structure for efficient updates (enqueue and dequeue) and innovatively adopts a two-level structure combined with a machine learning model to efficiently support searches. To handle the ever-changing data distribution in a stream window, FLIRT is equipped with an auto-tuner to adjust its structure during execution. The auto-tuner functions like a gradient descent algorithm to dynamically minimise the cost model, thereby maximising FLIRT's performance on the fly. To further improve the search performance of FLIRT, we propose two parallel variants, PPFLIRT and PSFLIRT. PPFLIRT employs partitioned parallelism, in which each partition owns a local FLIRT. This design efficiently handles queries with uneven data distributions among different partitions but struggles with skewed query distributions. Therefore, PSFLIRT is proposed, which maintains a single FLIRT and allows different threads to access the index freely. Our experiments show that FLIRT outperforms all other baselines for IBWP, achieving more than a $100\times$ speedup. The experiments also demonstrate that its parallel variants achieve their design objectives.

In Chapter 5, we focus on generic IBWP, in which the key arrival order may not necessarily align with the key's sequence. Such scenarios are common in time-based windows and for indexed keys lacking temporal order. To efficiently handle generic IBWP, we introduce SWIX. For search efficiency, SWIX leverages a two-level structure; each level is equipped with a machine learning model to swiftly predict key locations. For update efficiency, SWIX adopts a dynamic structure, which can be extended to ingest incoming data alongside a segment-level buffer to manage edge cases. The dynamic data structure is also designed to shrink to maintain compactness for space efficiency. An auto-tuner with a heuristic cost model dynamically adapts to changes in the

data stream. The parallel version of SWIX, PSWIX, is introduced to further enhance update and search performance. Comprehensive experiments show that SIWX consistently achieves superior performance in both time and space. Specifically, it can achieve a $1.59\times$ speed up and only take 25.19% memory compared to its learned index competitors.

6.2 Future Work

Building on our work to optimize stream processing at both system and algorithm levels, our future work continues to enhance these two-level optimizations.

6.2.1 System-level Future Work

At the system level, we plan to integrate stream systems with machine learning.

Predictive Scaling Strategies

The first system-level future work is about ML4Sys, in which a machine learning model is used to speed up the system's performance. We plan to extend scaling strategies (in Chapter 3) of dispel4py to include a predictive model. The predictive model can learn the periodic data distribution in the stream window, which enables a scaling mechanism to predict data distribution changes. By predicting these changes, the model-based scaling strategies can proactively allocate resources tailored to expected data loads.

Stream Systems for PEFT

Another direction for future work is implementing a system to support machine learning tasks, known as Sys4ML. Specifically, we plan to design a stream system to support the parameter efficient fine-tuning (PEFT) of a large language model (LLM). PEFT can modify a minimal set of parameters in LLM based on new data, adapting the model to specific tasks [50]. Integrating

PEFT into stream systems is motivated by two reasons. First, online training environments have become increasingly important since users focus more on current activities and low-latency responses. A stream system supporting online PEFT can meet the demand. Secondly, continuous updates is inherent in stream processing, allowing models to stay relevant as the data evolves. Integrating PEFT into stream systems can make incremental updates to LLM in real-time with minimal training cost and low latency, compared with traditional full retraining in batch processing systems.

6.2.2 Algorithm-level Future Work

Our future work at the algorithmic level will continue to focus on the learned index in stream data processing.

Online Learning for Learned Indexes

Retraining learned indexes is crucial since the model becomes inaccurate when data distributions shift. This is particularly common in stream processing environments. Traditional retraining methods for learned indexes (e.g., least squares) do not leverage prior data when only a subset of the data changes. This usually leads to an inefficient retraining process. We plan to investigate online learning methods, such as recursive least squares (RLS) and Bayesian approaches, to update the model in stream data contexts incrementally. These online learning strategies could provide consistently accurate models with lower latency, thereby improving the overall performance of learned indexes.

Learned Index for Stream Joins

[133] suggests that learned indexes can significantly optimize static joins. We believe the learned index can also optimize stream joins in IBWP. In stream joins, there are at least two streams R and S (except self-stream join). Once a key arrives at stream window R , it needs to be inserted into R , while stream window S carries out a search based on the key and the join condition. A

stream join involves both insertion and search, which stream-learned indexes can optimize, as proven in Chapter 4 and Chapter 5.

Bibliography

- [1] ABADI, D. J., CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Aurora: a new model and architecture for data stream management. *the VLDB Journal* 12 (2003), 120–139.
- [2] ABDUL-MAGEED, M., AND DIAB, M. Toward building a large-scale arabic sentiment lexicon. In *Proceedings of the 6th international global WordNet conference* (2012), pp. 18–22.
- [3] ACHARYA, A., AND SIDNAL, N. S. High frequency trading with complex event processing. In *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)* (2016), IEEE, pp. 39–42.
- [4] AKAIKE, H. A new look at the statistical model identification. *IEEE transactions on automatic control* 19, 6 (1974), 716–723.
- [5] AKIDAU, T., BALIKOV, A., BEKIROĞLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., AND WHITTLE, S. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.
- [6] AKIDAU, T., BRADSHAW, R., CHAMBERS, C., CHERNYAK, S., FERNÁNDEZ-MOCTEZUMA, R. J., LAX, R., MCVEETY, S., MILLS, D., PERRY, F., SCHMIDT, E., ET AL. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

- [7] AKIDAU, T., CHERNYAK, S., AND LAX, R. *Streaming systems: the what, where, when, and how of large-scale data processing.* " O'Reilly Media, Inc.", 2018.
- [8] AMINI, L., JAIN, N., SEHGAL, A., SILBER, J., AND VERSCHEURE, O. Adaptive control of extreme-scale stream processing systems. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)* (2006), IEEE, pp. 71–71.
- [9] APACHE SOFTWARE FOUNDATION. Apache flink project. <http://flink.apache.org/>. Accessed: 2024-08-12.
- [10] APACHE SOFTWARE FOUNDATION. Apache spark project. <http://spark.apache.org/>. Accessed: 2024-08-12.
- [11] APACHE SOFTWARE FOUNDATION. Apache storm project. <http://storm.apache.org/>. Accessed: 2024-08-12.
- [12] ARASU, A., BABCOCK, B., BABU, S., DATAR, M., ITO, K., NISHIZAWA, I., ROSENSTEIN, J., AND WIDOM, J. Stream: the stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), pp. 665–665.
- [13] ARASU, A., BABU, S., AND WIDOM, J. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal* 15 (2006), 121–142.
- [14] ARASU, A., CHERNIACK, M., GALVEZ, E., MAIER, D., MASKEY, A. S., RYVKINA, E., STONEBRAKER, M., AND TIBBETTS, R. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30* (2004), pp. 480–491.
- [15] ARASU, A., AND WIDOM, J. Resource sharing in continuous sliding-window aggregates. In *VLDB* (2004), pp. 336–347.
- [16] ARMBRUST, M., DAS, T., TORRES, J., YAVUZ, B., ZHU, S., XIN, R., GHODSI, A., STOICA, I., AND ZAHARIA, M. Structured streaming: A declarative api for real-time

- applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 601–613.
- [17] ASAD MEMON. News articles dataset. <https://www.kaggle.com/datasets/asad1m9a9h6mood/news-articles>. Accessed: 2024-08-18.
- [18] BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. 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.
- [19] BADIOZAMANY, S., AND RISCH, T. Scalable ordered indexing of streaming data. In *ADMS@ VLDB* (2012), pp. 57–67.
- [20] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S., AND STONEBRAKER, M. Fault-tolerance in the borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (2005), pp. 13–24.
- [21] BARKER, A., AND VAN HEMERT, J. Scientific workflow: a survey and research directions. In *International Conference on Parallel Processing and Applied Mathematics* (2007), Springer, pp. 746–753.
- [22] BINGMANN, T. Stx b+tree. <https://github.com/bingmann/stx-btree>. Accessed: 2024-08-30.
- [23] BONIFATI, A., AND TOMMASINI, R. An overview of continuous querying in (modern) data systems. In *Companion of the 2024 International Conference on Management of Data* (2024), pp. 605–612.
- [24] BORDIN, M. V., GRIEBLER, D., MENCAGLI, G., GEYER, C. F., AND FERNANDES, L. G. L. Dspbench: A suite of benchmark applications for distributed data stream processing systems. *IEEE Access* 8 (2020), 222900–222917.
- [25] BOTAN, I., DERA KHSHAN, R., DINDAR, N., HAAS, L., MILLER, R. J., AND TATBUL, N. Secret: A model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 232–243.

- [26] C++ REFERENCE. C++ chrono library. <https://en.cppreference.com/w/cpp/header/chrono>. Accessed: 2024-08-31.
- [27] C++ REFERENCE. std::array - c++ reference. <https://en.cppreference.com/w/cpp/container/array>. Accessed: 2024-08-30.
- [28] C++ REFERENCE. std::list - c++ reference. <https://en.cppreference.com/w/cpp/container/list>. Accessed: 2024-08-30.
- [29] CARBONE, P., EWEN, S., FÓRA, G., HARIDI, S., RICHTER, S., AND TZOUMAS, K. State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.
- [30] CARBONE, P., KATSIFODIMOS, A., EWEN, S., MARKL, V., HARIDI, S., AND TZOUMAS, K. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).
- [31] CARDELLINI, V., LO PRESTI, F., NARDELLI, M., AND RUSSO RUSSO, G. Auto-scaling in data stream processing applications: A model-based reinforcement learning approach. In *New Frontiers in Quantitative Methods in Informatics: 7th Workshop, InfQ 2017, Venice, Italy, December 4, 2017, Revised Selected Papers* 7 (2018), Springer, pp. 97–110.
- [32] CASTRO FERNANDEZ, R., MIGLIAVACCA, M., KALYVIANAKI, E., AND PIETZUCH, P. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data* (2013), pp. 725–736.
- [33] CELERY PROJECT CONTRIBUTORS. Celery: Distributed task queue. <https://docs.celeryq.dev/en/stable/>. Accessed: 2024-12-12.
- [34] CHAN, C.-Y., AND IOANNIDIS, Y. E. Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data* (1998), pp. 355–366.

- [35] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., REISS, F., AND SHAH, M. A. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), pp. 668–668.
- [36] CHARDONNENS, T., CUDRE-MAUROUX, P., GRUND, M., AND PERROUD, B. Big data analytics on high velocity streams: A case study. In *2013 IEEE International Conference on Big Data* (2013), IEEE, pp. 784–787.
- [37] CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. Niagaraqc: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data* (2000), pp. 379–390.
- [38] COMER, D. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.
- [39] CONNOLLY, T. M., AND BEGG, C. E. *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005.
- [40] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), pp. 647–651.
- [41] CUGOLA, G., AND MARGARA, A. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–62.
- [42] DE ASSUNCAO, M. D., DA SILVA VEITH, A., AND BUYYA, R. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications* 103 (2018), 1–17.
- [43] DE HEUS, M., PSARAKIS, K., FRAGKOULIS, M., AND KATSIFODIMOS, A. Distributed transactions on serverless stateful functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems* (2021), pp. 31–42.
- [44] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 51, 1 (2008), 107–113.

- [45] DEELMAN, E., PETERKA, T., ALTINTAS, I., CAROTHERS, C. D., VAN DAM, K. K., MORELAND, K., PARASHAR, M., RAMAKRISHNAN, L., TAUFER, M., AND VETTER, J. The future of scientific workflows. *The International Journal of High Performance Computing Applications* 32, 1 (2018), 159–175.
- [46] DELLA VALLE, E., DELL’AGLIO, D., AND MARGARA, A. Taming velocity and variety simultaneously in big data with stream reasoning: tutorial. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems* (2016), pp. 394–401.
- [47] DIAO, Y., LI, B., LIU, A., PENG, L., SUTTON, C., TRAN, T., AND ZINK, M. Capturing data uncertainty in high-volume stream processing. *arXiv preprint arXiv:0909.1777* (2009).
- [48] DING, J., FU, T. Z., MA, R. T., WINSLETT, M., YANG, Y., ZHANG, Z., AND CHAO, H. Optimal operator state migration for elastic data stream processing. *arXiv preprint arXiv:1501.03619* (2015).
- [49] DING, J., MINHAS, U. F., YU, J., WANG, C., DO, J., LI, Y., ZHANG, H., CHANDRAMOULI, B., GEHRKE, J., KOSSMANN, D., ET AL. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 969–984.
- [50] DING, N., QIN, Y., YANG, G., WEI, F., YANG, Z., SU, Y., HU, S., CHEN, Y., CHAN, C.-M., CHEN, W., ET AL. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence* 5, 3 (2023), 220–235.
- [51] EDDERBUETTEL, D. A brief introduction to redis, 2022.
- [52] FANG, J., ZHANG, R., FU, T. Z., ZHANG, Z., ZHOU, A., AND ZHU, J. Parallel stream processing against workload skewness and variance. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* (2017), pp. 15–26.

- [53] FEDERATION OF DIGITAL SEISMOGRAPH NETWORKS. Fdsn homepage. <https://www.fdsn.org>. Accessed: 2024-08-27.
- [54] FENG, Y.-H., HUANG, N.-F., AND WU, Y.-M. Efficient and adaptive stateful replication for stream processing engines in high-availability cluster. *IEEE Transactions on Parallel and Distributed Systems* 22, 11 (2011), 1788–1796.
- [55] FERRAGINA, P., LILLO, F., AND VINCIGUERRA, G. Why are learned indexes so effective? In *International Conference on Machine Learning* (2020), PMLR, pp. 3123–3132.
- [56] FERRAGINA, P., AND VINCIGUERRA, G. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.
- [57] FILGUEIRA, R., KRAUSE, A., ATKINSON, M., KLAMPANOS, I., SPINUSO, A., AND SANCHEZ-EXPOSITO, S. dispel4py: An agile framework for data-intensive escience. In *2015 IEEE 11th International Conference on e-Science* (2015), IEEE, pp. 454–464.
- [58] FILGUIERA, R., KRAUSE, A., ATKINSON, M., KLAMPANOS, I., AND MORENO, A. dispel4py: A python framework for data-intensive scientific computing. *The International Journal of High Performance Computing Applications* 31, 4 (2017), 316–334.
- [59] FRAGKOULIS, M., CARBONE, P., KALAVRI, V., AND KATSIFODIMOS, A. A survey on the evolution of stream processing systems. *The VLDB Journal* 33, 2 (2024), 507–541.
- [60] FRIEDMAN, E., AND TZOUMAS, K. *Introduction to Apache Flink: stream processing for real time and beyond.* ” O’Reilly Media, Inc.”, 2016.
- [61] FU, T. Z., DING, J., MA, R. T., WINSLETT, M., YANG, Y., AND ZHANG, Z. Drs: Auto-scaling for real-time stream analytics. *IEEE/ACM Transactions on networking* 25, 6 (2017), 3338–3352.
- [62] GALAKATOS, A., MARKOVITCH, M., BINNIG, C., FONSECA, R., AND KRASKA, T. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 international conference on management of data* (2019), pp. 1189–1206.

- [63] GAROFALAKIS, M., GEHRKE, J., AND RASTOGI, R. *Data stream management: processing high-speed data streams*. Springer, 2016.
- [64] GEDIK, B. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal* 23, 4 (2014), 517–539.
- [65] GEDIK, B., ANDRADE, H., WU, K.-L., YU, P. S., AND DOO, M. Spade: The system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), pp. 1123–1134.
- [66] GEDIK, B., WU, K.-L., YU, P. S., AND LIU, L. Adaptive load shedding for windowed stream joins. In *Proceedings of the 14th ACM international conference on Information and knowledge management* (2005), pp. 171–178.
- [67] GEISLER, S. Data stream management systems. In *Dagstuhl Follow-Ups* (2013), vol. 5, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- [68] GEORGIU, M. A., PAPHITIS, A., SIRIVIANOS, M., AND HERODOTOU, H. Towards auto-scaling existing transactional databases with strong consistency. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)* (2019), IEEE, pp. 107–112.
- [69] GOLAB, L., GARG, S., AND ÖZSU, M. T. On indexing sliding windows over online data streams. In *International Conference on Extending Database Technology* (2004), Springer, pp. 712–729.
- [70] GOLAB, L., AND ÖZSU, M. T. Issues in data stream management. *ACM Sigmod Record* 32, 2 (2003), 5–14.
- [71] GOMES, E. H., PLENTZ, P. D., ROLT, C. R. D., AND DANTAS, M. A. A survey on data stream, big data and real-time. *International Journal of Networking and Virtual Organisations* 20, 2 (2019), 143–167.
- [72] GOOGLE. Google dataflow. <https://cloud.google.com/dataflow>. Accessed: 2024-08-12.

- [73] GOOGLE LLC. Google cloud homepage. <https://cloud.google.com/>. Accessed: 2024-08-24.
- [74] GRAEFE, G., ET AL. Modern b-tree techniques. *Foundations and Trends® in Databases* 3, 4 (2011), 203–402.
- [75] GULISANO, V., JIMENEZ-PERIS, R., PATINO-MARTINEZ, M., SORIENTE, C., AND VALDURIEZ, P. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems* 23, 12 (2012), 2351–2365.
- [76] GULISANO, V., JIMENEZ-PERIS, R., PATINO-MARTINEZ, M., AND VALDURIEZ, P. Streamcloud: A large scale data streaming system. In *2010 IEEE 30th International Conference on Distributed Computing Systems* (2010), IEEE, pp. 126–137.
- [77] GUPTA, A., AND JAIN, S. Optimizing performance of real-time big data stateful streaming applications on cloud. In *2022 IEEE International Conference on Big Data and Smart Computing (BigComp)* (2022), IEEE, pp. 1–4.
- [78] GÜRCAN, F., AND BERIGEL, M. Real-time processing of big data streams: Lifecycle, tools, tasks, and challenges. In *2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)* (2018), IEEE, pp. 1–6.
- [79] HIDEN, H., WOODMAN, S., WATSON, P., AND CALA, J. Developing cloud applications using the e-science central platform. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 371, 1983 (2013), 20120085.
- [80] IMPERIAL COLLEGE LONDON. High performance computing at imperial college london. <https://www.imperial.ac.uk/computational-methods/hpc/>. Accessed: 2024-08-24.
- [81] IMPERIAL COLLEGE LONDON. Private cloud services for research. <https://www.imperial.ac.uk/computing/people/csg/services/cloud/>. Accessed: 2024-08-24.
- [82] INSTITUTO DE ASTROFÍSICA DE ANDALUCÍA (IAA). Amiga project. <https://amiga.iaa.csic.es/>. Accessed: 2024-08-27.

- [83] INTEL. Read time-stamp counter. https://c9x.me/x86/html/file_module_x86_id_278.html. Accessed: 2024-08-12.
- [84] INTERNATIONAL VIRTUAL OBSERVATORY ALLIANCE. Votable format definition. <https://www.ivoa.net/documents/PR/VOTable/VOTable-20040322.html>. Accessed: 2024-08-27.
- [85] JAIN, A. *Mastering apache storm: Real-time big data streaming using kafka, hbase and redis*. Packt Publishing Ltd, 2017.
- [86] JAYASEKARA, S., KARUNASEKERA, S., AND HARWOOD, A. Optimizing checkpoint-based fault-tolerance in distributed stream processing systems: Theory to practice. *Software: Practice and Experience* 52, 1 (2022), 296–315.
- [87] KALAVRI, V., LIAGOURIS, J., HOFFMANN, M., DIMITROVA, D., FORSHAW, M., AND ROSCOE, T. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 783–798.
- [88] KARIMOV, J., RABL, T., KATSIFODIMOS, A., SAMAREV, R., HEISKANEN, H., AND MARKL, V. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th international conference on data engineering (ICDE)* (2018), IEEE, pp. 1507–1518.
- [89] KATSIPOULAKIS, N. R., LABRINIDIS, A., AND CHRYSANTHIS, P. K. A holistic view of stream partitioning costs. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1286–1297.
- [90] KEOGH, E., CHU, S., HART, D., AND PAZZANI, M. An online algorithm for segmenting time series. In *Proceedings 2001 IEEE international conference on data mining* (2001), IEEE, pp. 289–296.
- [91] KHOSHKBARFOROUSHHA, A., KHOSRAVIAN, A., AND RANJAN, R. Elasticity management of streaming data analytics flows on clouds. *Journal of Computer and System Sciences* 89 (2017), 24–40.

- [92] KIM, C., CHHUGANI, J., SATISH, N., SEDLAR, E., NGUYEN, A. D., KALDEWEY, T., LEE, V. W., BRANDT, S. A., AND DUBEY, P. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), pp. 339–350.
- [93] KIPF, A., MARCUS, R., VAN RENEN, A., STOIAN, M., KEMPER, A., KRASKA, T., AND NEUMANN, T. Sossd: A benchmark for learned indexes. *arXiv preprint arXiv:1911.13014* (2019).
- [94] KRASKA, T., BEUTEL, A., CHI, E. H., DEAN, J., AND POLYZOTIS, N. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data* (2018), pp. 489–504.
- [95] KWON, Y., BALAZINSKA, M., AND GREENBERG, A. Fault-tolerant stream processing using a distributed, replicated file system. *Proceedings of the VLDB Endowment* 1, 1 (2008), 574–585.
- [96] LEHMAN, T. J., AND CAREY, M. J. A study of index structures for main memory database management systems. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 1985.
- [97] LI, J., MAIER, D., TUFTE, K., PAPADIMOS, V., AND TUCKER, P. A. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (2005), pp. 311–322.
- [98] LI, P., HUA, Y., JIA, J., AND ZUO, P. Finedex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.
- [99] LIANG, L., FILGUEIRA, R., YAN, Y., AND HEINIS, T. Scalable adaptive optimizations for stream-based workflows in multi-hpc-clusters and cloud infrastructures. *Future Generation Computer Systems* 128 (2022), 102–116.

- [100] LIANG, L., FILGUIERA, R., AND YAN, Y. Adaptive optimizations for stream-based workflows. In *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)* (2020), IEEE, pp. 33–40.
- [101] LIANG, L., YANG, G., HADIAN, A., CROQUEVIELLE, L. A., AND HEINIS, T. Swix: A memory-efficient sliding window learned index. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.
- [102] LIANG, L., YANG, G., HEINIS, T., AND TANIAR, D. Soja: A memory-efficient small-large outer join for mpi. In *EDBT* (2021), pp. 523–528.
- [103] LIANG, L., ZHANG, H., YANG, G., HEINIS, T., AND FILGUEIRA, R. Optimization towards efficiency and stateful of dispel4py. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (2023), pp. 2021–2032.
- [104] LIN, Q., OOI, B. C., WANG, Z., AND YU, C. Scalable distributed stream join processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), pp. 811–825.
- [105] LIU, B., ZHU, Y., AND RUNDENSTEINER, E. Run-time operator state spilling for memory intensive long-running queries. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (2006), pp. 347–358.
- [106] LIU, P., XU, H., DA SILVA, D., WANG, Q., AHMED, S. T., AND HU, L. Fp4s: Fragment-based parallel state recovery for stateful stream applications. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2020), IEEE, pp. 1102–1111.
- [107] LIU, X., LIN, Z., AND WANG, H. Novel online methods for time series segmentation. *IEEE Transactions on knowledge and data engineering* 20, 12 (2008), 1616–1626.
- [108] LOHRMANN, B., JANACIK, P., AND KAO, O. Elastic stream processing with latency guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems* (2015), IEEE, pp. 399–410.

- [109] MAHMOOD, A. R., ALY, A. M., KUZNETSOVA, T., BASALAMAH, S., AND AREF, W. G. Disk-based indexing of recent trajectories. *ACM Transactions on Spatial Algorithms and Systems (TSAS)* 4, 3 (2018), 1–27.
- [110] MAHMOOD, A. R., AREF, W. G., ALY, A. M., AND BASALAMAH, S. Indexing recent trajectories of moving objects. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2014), pp. 393–396.
- [111] MARCUS, R., ZHANG, E., AND KRASKA, T. Cdfshop: Exploring and optimizing learned index structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 2789–2792.
- [112] MARTIN, P., AND YAIKHOM, G. Definition of the dispel language. *The Data Bonanza: Improving Knowledge Discovery in Science, Engineering, and Business* (2013), 203–236.
- [113] MICROSOFT. Alex: A learned index for range queries. <https://github.com/microsoft/ALEX>. Accessed: 2024-08-30.
- [114] MOHAMED, A., NAJAFABADI, M. K., WAH, Y. B., ZAMAN, E. A. K., AND MASKAT, R. The state of the art and taxonomy of big data analytics: view from new big data framework. *Artificial intelligence review* 53 (2020), 989–1037.
- [115] NARKHEDE, N., SHAPIRA, G., AND PALINO, T. *Kafka: the definitive guide: real-time data and stream processing at scale.* ” O’Reilly Media, Inc.”, 2017.
- [116] NGUYEN, D., LUCKOW, A., DUFFY, E., KENNEDY, K., AND APON, A. Evaluation of highly available cloud streaming systems for performance and price. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2018), IEEE, pp. 360–363.
- [117] NIELSEN, F. Å. A new anew: Evaluation of a word list for sentiment analysis in microblogs. *arXiv preprint arXiv:1103.2903* (2011).

- [118] NOGHABI, S. A., PARAMASIVAM, K., PAN, Y., RAMESH, N., BRINGHURST, J., GUPTA, I., AND CAMPBELL, R. H. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645.
- [119] ONISHI, T., MICHAELIS, J., AND KANEMASA, Y. Recovery-conscious adaptive watermark generation for time-order event stream processing. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)* (2020), IEEE, pp. 66–78.
- [120] OTTENWÄLDER, B., KOLDEHOFE, B., ROTHERMEL, K., AND RAMACHANDRAN, U. Migcep: Operator migration for mobility driven distributed complex event processing. In *Proceedings of the 7th ACM international conference on Distributed event-based systems* (2013), pp. 183–194.
- [121] O’NEIL, P., CHENG, E., GAWLICK, D., AND O’NEIL, E. The log-structured merge-tree (lsm-tree). *Acta Informatica* 33 (1996), 351–385.
- [122] PATROUMPAS, K., AND SELLIS, T. Window specification over data streams. In *International Conference on Extending Database Technology* (2006), Springer, pp. 445–464.
- [123] PIETZUCH, P., LEDLIE, J., SHNEIDMAN, J., ROUSSOPOULOS, M., WELSH, M., AND SELTZER, M. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE’06)* (2006), IEEE, pp. 49–49.
- [124] PYPI. stream-d4py: New dispel4py streaming workflow repository, 2024. Accessed: 2024-08-14.
- [125] PYTHON SOFTWARE FOUNDATION. Multiprocessing - pipes and queues. <https://docs.python.org/3/library/multiprocessing.html#pipes-and-queues>. Accessed: 2024-08-27.
- [126] RADHIKA, E., AND SADASIVAM, G. S. A review on prediction based autoscaling techniques for heterogeneous applications in cloud environment. *Materials Today: Proceedings* 45 (2021), 2793–2800.

- [127] REDIS. Redis data types: Streams. <https://redis.io/docs/latest/develop/data-types/streams/>. Accessed: 2024-08-18.
- [128] REDIS. Redis enterprise data structures. <https://redis.io/redis-enterprise/data-structures/>. Accessed: 2024-08-18.
- [129] REDIS. Redis insight: Stream consumer tutorial. <https://redis.io/docs/latest/develop/connect/insight/tutorials/insight-stream-consumer/>. Accessed: 2024-08-18.
- [130] REN, K., DIAMOND, T., ABADI, D. J., AND THOMSON, A. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 1539–1551.
- [131] ROLOFF, E., DIENER, M., CARISSIMI, A., AND NAVAUX, P. O. High performance computing in the cloud: Deployment, performance and cost efficiency. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings* (2012), IEEE, pp. 371–378.
- [132] RYDNING, D. R.-J. G.-J., REINSEL, J., AND GANTZ, J. The digitization of the world from edge to core. *Framingham: International Data Corporation 16* (2018), 1–28.
- [133] SABEK, I., AND KRASKA, T. The case for learned in-memory joins. *arXiv preprint arXiv:2111.08824* (2021).
- [134] SAFAEI, A. A. Real-time processing of streaming big data. *Real-Time Systems 53* (2017), 1–44.
- [135] SALLOUM, S., DAUTOV, R., CHEN, X., PENG, P. X., AND HUANG, J. Z. Big data analytics on apache spark. *International Journal of Data Science and Analytics 1* (2016), 145–164.
- [136] SHAH, M. A., HELLERSTEIN, J. M., CHANDRASEKARAN, S., AND FRANKLIN, M. J. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings*

- 19th International Conference on Data Engineering (Cat. No. 03CH37405)* (2003), IEEE, pp. 25–36.
- [137] SHAHVARANI, A., AND JACOBSEN, H.-A. Parallel index-based stream join on a multi-core cpu. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 2523–2537.
- [138] SHI, J., CONG, G., AND LI, X.-L. Learned index benefits: Machine learning based index performance estimation. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3950–3962.
- [139] SILBERSCHATZ, A., KORTH, H. F., AND SUDARSHAN, S. Database system concepts.
- [140] SINGH, M., ZHU, Q., AND JAGADISH, H. Swst: A disk based index for sliding window spatio-temporal data. In *2012 IEEE 28th International Conference on Data Engineering* (2012), IEEE, pp. 342–353.
- [141] SONMEZ, O., YIGITBASI, N., ABRISHAMI, S., IOSUP, A., AND EPEMA, D. Performance analysis of dynamic workflow scheduling in multicluster grids. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (2010), pp. 49–60.
- [142] STOCKINGER, K., BUNDI, N., HEITZ, J., AND BREYMAN, W. Scalable architecture for big data financial analytics: user-defined functions vs. sql. *Journal of Big Data* 6, 1 (2019), 46.
- [143] STONEBRAKER, M., ÇETINTEMEL, U., AND ZDONIK, S. The 8 requirements of real-time stream processing. *ACM Sigmod Record* 34, 4 (2005), 42–47.
- [144] SUN, Z., ZHOU, X., AND LI, G. Learned index: A comprehensive experimental evaluation. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1992–2004.
- [145] TANG, C., WANG, Y., DONG, Z., HU, G., WANG, Z., WANG, M., AND CHEN, H. Xindex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming* (2020), pp. 308–320.

- [146] TATBUL, N., ÇETINTEMEL, U., ZDONIK, S., CHERNIACK, M., AND STONEBRAKER, M. Load shedding in a data stream manager. In *Proceedings 2003 vldb conference* (2003), Elsevier, pp. 309–320.
- [147] TERRY, D., GOLDBERG, D., NICHOLS, D., AND OKI, B. Continuous queries over append-only databases. *Acm Sigmod Record* 21, 2 (1992), 321–330.
- [148] TO, Q.-C., SOTO, J., AND MARKL, V. A survey of state management in big data processing systems. *The VLDB Journal* 27, 6 (2018), 847–872.
- [149] TRAUB, J., GRULICH, P. M., CUELLAR, A. R., BRESS, S., KATSIFODIMOS, A., RABL, T., AND MARKL, V. Scotty: Efficient window aggregation for out-of-order stream processing. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), IEEE, pp. 1300–1303.
- [150] TU, Y.-C., LIU, S., PRABHAKAR, S., AND YAO, B. Load shedding in stream databases: a control-based approach.
- [151] VAN DONGEN, G., AND VAN DEN POEL, D. Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020), 1845–1858.
- [152] VINCIGUERRA, G. Pgm-index: Piecewise geometric model index. <https://github.com/gvinciguerra/PGM-index>. Accessed: 2024-08-30.
- [153] WANG, J., LIU, W., KUMAR, S., AND CHANG, S.-F. Learning to hash for indexing big data—a survey. *Proceedings of the IEEE* 104, 1 (2015), 34–57.
- [154] WANG, Z., AND O’BOYLE, M. F. Using machine learning to partition streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 3 (2013), 1–25.
- [155] WEISSBACH, M., AND SPRINGER, T. Performance of databases used in data stream processing environments. In *CLOSER* (2022), pp. 15–26.
- [156] WINGERATH, W., GESSERT, F., FRIEDRICH, S., AND RITTER, N. Real-time stream processing for big data. *it-Information Technology* 58, 4 (2016), 186–194.

- [157] WONGKHAM, C., LU, B., LIU, C., ZHONG, Z., LO, E., AND WANG, T. Are updatable learned indexes ready? *arXiv preprint arXiv:2207.02900* (2022).
- [158] WU, J., ZHANG, Y., CHEN, S., WANG, J., CHEN, Y., AND XING, C. Updatable learned index with precise positions. *arXiv preprint arXiv:2104.05520* (2021).
- [159] YA-XIN, Y., XING-HUA, Y., GE, Y., AND SHAN-SHAN, W. An indexed non-equijoin algorithm based on sliding windows over data streams. *Wuhan University Journal of Natural Sciences* 11 (2006), 294–298.
- [160] YANG, G., GHOSH, A., LIANG, L., AND HEINIS, T. In-network approximate and efficient spatiotemporal range queries on moving objects. In *EDBT* (2024).
- [161] YANG, G., LIANG, L., HADIAN, A., AND HEINIS, T. Flirt: A fast learned index for rolling time frames. In *EDBT* (2023), pp. 234–246.
- [162] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)* (2010).
- [163] ZHANG, H., CHEN, G., OOI, B. C., TAN, K.-L., AND ZHANG, M. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948.
- [164] ZHANG, S., MAO, Y., HE, J., GRULICH, P. M., ZEUCH, S., HE, B., MA, R. T., AND MARKL, V. Parallelizing intra-window join on multicores: An experimental study. In *Proceedings of the 2021 International Conference on Management of Data* (2021), pp. 2089–2101.
- [165] ZHU, Y., RUNDENSTEINER, E. A., AND HEINEMAN, G. T. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (2004), pp. 431–442.