UNIVERSITY OF CALIFORNIA,
IRVINE

# Managing Complex Join Queries in Big Data Management Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Shiva Jahangiri

Dissertation Committee:
Professor Michael J. Carey, Chair
Professor Johann-Christoph Freytag
Professor Sharad Mehrotra

2022

# DEDICATION

To my beloved family, for their unconditional love and support.
Also, to those beautiful souls who supported me in choosing my path and helped me endure the tough moments, but life did not get them the chance to celebrate reaching the finish line with me.

# Contents

# List of Figures

Page

# List of Tables

Page

# ACKNOWLEDGMENTS

From the early days of my undergraduate studies, when I got interested in databases, I never imagined in my wildest dreams that I would have not one but two legends in this field of research as my Ph.D. advisor. Most importantly, I am lucky to have two life mentors who shaped me into a more thoughtful, patient, and generally better person than I was before.

I am forever indebted to my Ph.D. advisor, Professor Michael Carey, for his guidance, support, and patience throughout the past six years of my graduate studies at UC Irvine. I am forever grateful to him for trusting me and allowing me to be his Ph.D. student; it has indeed been an honor. I would also like to express my most profound appreciation to my Ph.D. co-advisor, Professor Johann-Christoph Freytag, for his motivation, enthusiasm, brilliant insights, and his availability and staying up late for our weekly meetings due to the 9-hour time difference between California and Berlin. Because of my advisors' consistent and patient instructions, this work turned into a successful research project.

I would also like to extend my sincere gratitude to the other member of my dissertation committee, Professor Sharad Mehrotra, for generously providing knowledge and expertise.

I have had the opportunity to learn from Professor Mehrotra and Professor Chen Li by being their students in Transaction Management and Principles of Data Management courses. I can't thank them enough for providing such rich and valuable courses, which helped me immensely through my research and Ph.D. studies.

I would also like to thank the faculty members who kindly accepted to be in the candidacy exam and shared their insight and expertise with me: Professor Michael J. Carey, Professor Johann-Christoph Freytag, Professor Soroosh Sorooshian, Professor Chen Li, Professor Sharad Mehrotra, Professor Sandy Irani, and Professor Sang-Woo Jun.

During my graduate studies, I had the opportunity to work and learn from the most knowledgeable yet humble researchers who shaped my thinking and helped me become a better researcher. I am grateful to Professor Craig Knoblock, Professor Jose-Luis Ambite, Professor Karin Becker, and Ippokratis Pandis for sharing their expertise with me.

I am indebted to Professor Shahram Ghandeharizadeh for letting me learn from him by collaborating with DBLab at the University of Southern California. His guidance and teachings in the Advanced Database Systems course amplified my interest in pursuing a Ph.D. in databases.

Special thanks to the Couchbase and AsterixDB team members, especially Ian Maxon, Till Westmann, Dmitry Lychagin, Murtadha Hubail, Ali Alsuliman, Hussain Towaileb, and Murali Mallela Krishna. I would also like to thank my fellow graduate students Taewoo Kim, Phanwadee Sinthong, Chen Luo, Xikui Wang, Wail Alkowaileet, Glenn Galvizo, and Jianfeng Jia. This journey became memorable because of you all.

Most importantly, I would like to thank my family; none of these would have been possible

without their support, care, and love. First, I want to thank my parents, Shirin Khobchehr and Esfandiar Jahangiri for all their help, support, and, most importantly, prayers. I couldn't be more grateful to God for selecting them as my parents. I am thankful to Manouchehr Hakhamaneshi, my beloved husband, whom I had the privilege to meet during my Ph.D. studies and have his company from then. I am most grateful to him for all his sacrifices and understandings, his professional, mental, and emotional support, and for being there for me in all the easy and challenging moments.

Special thanks and gratitudes to my grandparents, Keikhosrow Khobchehr and Homayoun Visprat, Kharman Azadi and Mandegar Jahangiri, who were my first life mentors and helped me grow under their unconditional love and their wisdom.

I want to thank my siblings: Nooshin, Rashin, Vida, and Payman, and their spouses: Khashayar, Rostam, Maziyar, and Katayoun, and my nieces and nephews: Farnaz, Pouya, Rosha, Soren, Arian, and Arshan. This achievement would not have been possible without your unwavering love and care.

Special thanks to my brother-in-law, Bahram Hakhamaneshi, for all his helpful advice and for having our company during the tough times of Covid and lockdowns.

This work has been supported by NSF awards IIS-1838248, CNS- 1925610, and IIS-1954962, along with industrial support from Google and the Donald Bren Foundation (via a Bren Chair).

# VITA

## Shiva Jahangiri

### EDUCATION

**Doctor of Philosophy in Computer Science** **2022**
University of California, Irvine (UCI) *Irvine, California*

**Master of Science in Computer Science** **2016**
University of Southern California (USC) *Los Angeles, California*

**Bachelor of Science in Information Technology Management** **2012**
Shahrood University of Technology (SUT) *Shahrood, Semnan, Iran*

### PUBLICATIONS

**Design Trade-offs for a Robust Dynamic Hybrid Hash Join** **2022**
Proceedings of the VLDB Endowment (PVLDB)

**Wisconsin Benchmark Data Generator: To JSON and Beyond** **2022**
The ACM Special Interest Group on Management of Data (SIGMOD)

**Robust and efficient memory management in Apache AsterixDB** **2020**
Software - Practice and Experience Journal

**Re-evaluating the Performance Trade-offs for Hash-Based Multi-Join Queries** **2020**
The ACM Special Interest Group on Management of Data (SIGMOD)

**The FastMap Algorithm for Shortest Path Computations** **2018**
International Joint Conferences on Artificial Intelligence (IJCIA)

**The FastMap Algorithm for Shortest Path Computations** **2018**
International Symposium on Artificial Intelligence and Mathematics (ISAIM)

**Finding, Assessing, and Integrating Statistical Sources for Data Mining** **2015**
Workshop on Knowledge Discovery and Data Mining Meets Linked Open Data (KNOW@LOD)

**A Quantitative Survey on the Use of the Cube** **2015**
International Workshop on Semantic Statistics co-located with International Semantic Web Conference (SemStats@ISWC)

**A demonstration of KOSAR: an elastic, scalable, highly available SQL middleware** **2014**
Proceedings of the Middleware Posters and Demos Session (Middleware)

# ABSTRACT OF THE DISSERTATION

Managing Complex Join Queries in Big Data Management Systems

By

Shiva Jahangiri

Doctor of Philosophy in Computer Science

University of California, Irvine, 2022

Professor Michael J. Carey, Chair

In addition to storing and managing the data and providing capabilities to query them, a Database Management System (DBMS) tries to achieve performance goals. High resource utilization, high throughput, and low query execution time are a few of the performance goals that are considered for various DBMSs. The system's success in achieving its performance goals highly depends on the performance of queries and their operators. Many factors can impact a query's performance, including how much of its resource requirements are satisfied, when it is scheduled for execution, and which other queries will execute concurrently with it. This thesis is an experimental study focusing on resource management and scheduling techniques to assist a database management system in reaching its performance goals.

We begin this thesis by exploring the design space for a robust dynamic Hybrid Hash Join operator, one of the main and most common types of memory-intensive database operators. Our variant of this operator is specifically designed to perform well even when the required statistics and information for a Hybrid Hash Join operator are unavailable or inaccurate.

Next, we explore various memory management and execution strategies for efficiently executing queries containing multiple join operators. We specifically study variations of Left Deep Trees, Right Deep Trees, and Bushy Trees containing one to eight join operators. We evaluate their performance under different memory availabilities, join and scan selectivities, degrees of parallelism, storage types, and query complexities.

Lastly, we study and evaluate the performance of various schedulers designed to schedule queries with highly different memory requirements and execution times in a concurrent environment. Our performance goal is to design a fair scheduler that keeps different classes of queries in admission and resource control queues in proportion to their execution times.

# Chapter 1

# Introduction

Efficiency in processing queries and managing data is one of the key criteria for choosing a DBMS. Several factors, including the efficiency of query operators in execution, having access to statistics and information about datasets and their attributes, quality of the query optimizer in choosing the best operators, access methods, and operator orderings, significantly impacts the performance of a DBMS.

Memory-intensive operators are database operators whose performance is highly dependent on the amount of memory made available to them. For such operators, careful consideration should be given in their implementation and design to use the available memory properly and therefore execute efficiently. Hybrid Hash Join and hash-based or sort-based Group By operators are examples of memory-intensive operators.

As one of the most common and expensive database operators, the join operator plays an important role in the query response time and/or throughput of a DBMS. Although the study of various join algorithms and their designs has a long history, changes in the variety of data, queries, and workloads, as well as the improvements in the hardware technology and the arrival of new storage types, leave the way open for further improvements in the implementation and design of join operators. Hybrid Hash Join (HHJ) has proven to be one of the most efficient and widely-used join algorithms. While HHJ's performance largely depends on accurate statistics and information about the input relations, it may not always be practical or possible for a system to have such information available. Chapter 3 empirically and analytically studies the trade-offs in designing a robust and dynamic HHJ. We revisit the design and optimization techniques suggested by previous studies by extensive experiments and comparing them with other algorithms designed by us or used in related studies.

Chapter 4 explores the trade-offs involved in executing a multi-join query using various query shapes, memory management schemes, and execution strategies. Through experiments with various memory availabilities, join and scan selectivities, storage types, degrees of parallelism, and query complexities, we compare the performance of query plans, including variations of Right Deep Trees, Left Deep Trees, and Bushy Trees.

Chapter 5 turns to the topic of multi-query workload management. The chapter focuses on designing a fair scheduler to schedule queries with widely varying memory requirements and execution times in a concurrent environment. The main responsibility of each scheduler is to decide on the execution order of incoming queries so that each query waits in the system in proportion to its execution time.

# Chapter 2

# Preliminaries

## 2.1 Apache AsterixDB

Apache AsterixDB is a parallel big data management system (BDMS) for managing and processing large amounts of semi-structured data with a declarative query language [2, 9, 44].

AsterixDB manages a flexible cluster of shared-nothing commodity nodes which may be sized to adjust to the storage and computation needs of an application. Figure 2.1 illustrates the physical architecture of AsterixDB. One of the nodes in the cluster serves as the Cluster Controller (CC), while the rest serve as Node Controllers (NCs). In a nutshell, the Cluster Controller node is the entry point for the user requests and compiles and transforms the requests into executable jobs. The Node Controllers are the worker nodes that execute jobs sent to them by the Cluster Controller. One Node Controller also serves as the Metadata Controller Node, providing access to AsterixDB's metadata. Each Node Controller manages one or more data partitions, and an instance of each query is executed in parallel on each data partition that the query needs to access. We will cover more details of the internal modules of AsterixDB in the next section.

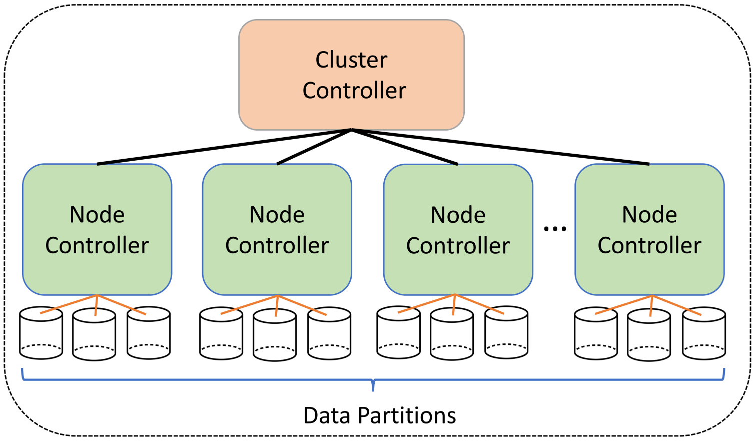


Figure 2.1: AsterixDB Architecture

### 2.1.1 Software Modules

AsterixDB uses SQL++, a declarative language for JSON data [60]. Additionally, other projects and query languages, including Apache VXQuery and JSONiq, have used the lower layers of AsterixDB to execute their parallel queries.

The AsterixDB Data Model (ADM) is a highly flexible data model based on a superset of JSON that supports primitive data types (boolean, integer, string, date, etc.), special types (missing and null), and derived types (arrays, multi-sets, and objects). Data enters AsterixDB in various ways, including loading, insertion, and feeds.

Figure 2.2 shows the internal software layers of AsterixDB. The component at the top level of this figure receives queries using an HTTP-based API and returns the results synchronously or asynchronously. It then creates a logical plan for the received query to be used by Algebricks [16], AsterixDB's rule-based optimizer, to be further optimized before execution. Algebricks applies the optimization rules to generate an optimized logical plan and then generates a physical plan by selecting a physical operator for each logical operator in the optimized logical plan. Next, Algebricks uses the physical plan to generate a Hyracks [17] job which is a directed acyclic graph (DAG) containing physical operators as the nodes and data

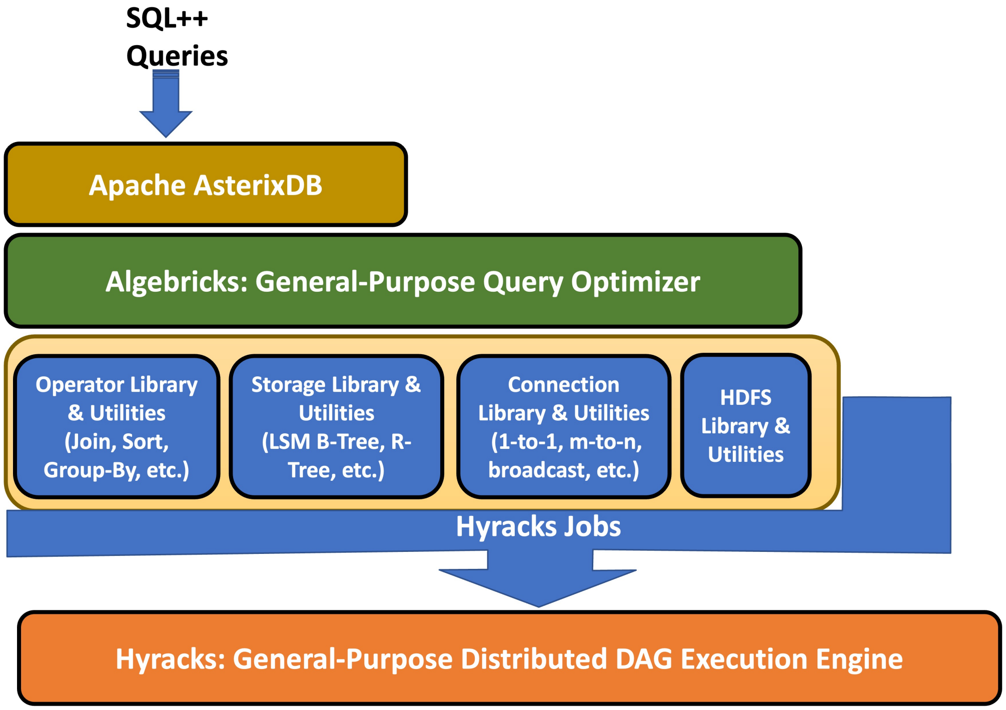

SQL++ Queries
Apache AsterixDB
Algebricks: General-Purpose Query Optimizer
Operator Library & Utilities (Join, Sort, Group-By, etc.)
Storage Library & Utilities (LSM B-Tree, R-Tree, etc.)
Connection Library & Utilities (1-to-1, m-to-n, broadcast, etc.)
HDFS Library & Utilities
Hyracks Jobs
Hyracks: General-Purpose Distributed DAG Execution Engine

Figure 2.2: Software Layers in AsterixDB

connectors as edges. Data is transfered between physical operators through connectors using a fixed-size and configurable set of contiguous bytes called *frame*. Hyracks [17], the lowest layer in the AsterixDB software hierarchy, is the scalable execution engine for AsterixDB. Each NC node uses Hyracks to execute the DAG of a query on each one of its data partitions.

We chose Apache AsterixDB as the platform for implementing and evaluating our proposed techniques for several reasons. First, it is an open-source platform that allows us to share our techniques and their evaluations with the community. Second, AsterixDB is a parallel big data management system for managing and processing large amounts of semi-structured data with a declarative language. Finally, its similarity in structure and design to other NoSQL and NewSQL database systems and query engines makes our results and techniques applicable to other systems as well.

### 2.1.2 Resource Parameters in AsterixDB

Physical operators in AsterixDB are divided into two groups of operators: minimal-memory and memory-intensive operators. The minimal-memory operators only use a few frames as their inputs and outputs. The exact number of these frames depends on the cardinality (1-to-1, 1-to-m, m-to-n, etc.) of their input and output connectors. AsterixDB automatically allocates the required number of frames to the minimal-memory operators. Memory-intensive operators, on the other hand, need much more memory than what is required for input and output buffers. The memory-intensive operators, including Hybrid Hash Join, Group By, and Sort operators, need memory to keep data in memory for further processing. For example, Hybrid Hash Join requires memory to hold a complete hash table with its data in memory.

AsterixDB provides several parameters so that a user may specify how much memory a specific memory-intensive operator can use at maximum. This amount of memory is not pre-allocated; however, it will be reserved as a budget for that operator and cannot be used by other operators or queries. For example, a user can specify a memory budget per data partition for each Group By operator in a query by setting the *compiler.groupmemory* parameter to the desired value. Similarly, the user can set the memory budget per data partition for each join operator in the query by setting the *compiler.joinmemory* parameter.

Additionally, AsterixDB provides a *core-multiplier* parameter to specify how many queries a CPU core can be asked to execute concurrently. For example, a core-multiplier of three indicates that each CPU core in an NC node can be asked to execute three queries concurrently. By default, the core-multiplier is set to three. Also, each query alone uses as many CPU cores as the number of data partitions that it is accessing. For example, consider an AsterixDB cluster consisting of one CC and one NC node where the NC node has four data partitions. In this case, a query that executes on all four data partitions will use four CPU cores, one for each data partition. Thus, if the core-multiplier is set to three, at most three such 4-way parallel queries can run concurrently. We use this parameter extensively in Chapter 5.

### 2.1.3 Joins in Apache AsterixDB

Since in this thesis our focus is on the join queries, we cover the join operators and their execution details in more depth in this section.

AsterixDB supports a number of different join algorithms including Block Nested Loop Join, Dynamic HHJ, Broadcast Join, and Indexed Nested Loop Join. However, due to its superior and robust performance, Dynamic HHJ is the default and primary join type in AsterixDB for processing equi-joins.

Since AsterixDB is not currently benefiting from the availability of statistics, hints can be provided by users to guide AsterixDB at execution time. For example, a user can use a hint for an Indexed Nested Loop Join to request this join algorithm instead of a Hybrid Hash Join. Such a request is respected when possible; otherwise, AsterixDB utilizes Hybrid Hash Join (by default). In addition, a hint to use a Broadcast Join might be advantageous when the build dataset is small enough to be sent to all nodes instead of hash partitioning it.

The current release of AsterixDB follows the specific join order in a query's FROM clause for indicating the query's join order and the build and probe inputs to the joins. The first input in the FROM clause will serve as the query's probe relation, and the rest will be build inputs to the query's joins. This makes the default query plan a Right Deep Tree [66], and all build datasets are scanned in parallel. Users may obtain other kinds of query plan shapes (if desired) by introducing nesting using subqueries in the FROM clause of their query.

## 2.2 Wisconsin Benchmark JSON Data Generator

In this section, we describe a flexible, easy-to-use, and scalable JSON Data Generator [41, 42] that we implemented in Java based on the Wisconsin Benchmark data generator [23] description. This data generator includes more advanced features to provide relations and attributes closer to real-world data. We use this data generator for all of our experiments in this thesis.

### 2.2.1 Wisconsin Benchmark

The Wisconsin Benchmark [23] was one of the first and main benchmarking tools designed and implemented by Dewitt et al. at the University of Wisconsin four decades ago. One of the most powerful features of this benchmark is that its relations are designed so their structure and distribution of attributes is easy to understand and control. While the Wisconsin Benchmark was a very powerful and widely-used benchmark years ago, it is given less attention in current studies. We believe that the Wisconsin Benchmark and its carefully designed relations can still be utilized and provides unique and useful capabilities.

### 2.2.2 JSON Data Generator

One of the challenges of benchmarks that use synthetic data is that they are essentially incapable of generating realistic data. On the other hand, benchmarks that use real data often contain data values that are not flexible and controllable which makes them harder to scale and understand and less capable of providing a range of specific scenarios. To overcome these problems, we created a Wisconsin-inspired JSON Data Generator in Java [42], creating records based on the same logic and attributes as the Wisconsin Data Generator. In addition, we added other features such as attribute distributions to provide attribute skewness, which is a missing but very important and valuable feature. Several other features were added to support semi-structured in addition to structured databases; those are explained below.

### JSON Records

Our data generator provides records in JSON format. JSON is supported as the input format, or even as the data model, in many modern database systems, especially those that manage semi-structured data.

### Nullable and Missing Attributes

One of the features missing from the original Wisconsin Data Generator was the capability to have nullable attributes and a knob to control the distribution of null values in those attributes. Also, in a semi-structured world, a field may appear in some of the records but not in all. Some database management systems that manage semi-structured data, such as AsterixDB, have the option to define a field as optional which may exist in some of the records and be missing in others. In our generator, for each attribute there are options for setting a field as nullable and/or missing. In addition to that, users can specify what percentages of the data they wish to be null and/or missing.

### Variable Record Lengths

In the original Wisconsin Data Generator, strings are generated in a random or cyclic format. In the case of random strings, the string representation of the benchmark's unique1 attribute is used as the prefix (which is unique as well) and enough 'x' characters are appended to the string as padding to reach a desired length. In the case of cyclic strings, string values are generated from four prefix values in a cyclic format. While adding padding to the strings is a simple and useful way of reaching the required length of a string, it does not create any variation in strings' lengths. In order to generate variable-length strings, we added five more properties to the definition of an attribute. The first property is a percentage which, based on the binomial distribution, decides if the string should be long or short. The other four fields are used for specifying the minimum and maximum length of the long and short strings. The length of the long and short strings will be chosen uniformly from these ranges. These knobs provide control of the distribution of short and long strings as well as their length ranges. We also support selecting the length of a string from a defined range using a Zipf distribution.

### Real-Word and HEX Strings

In addition to supporting the aforementioned algorithms, we support strings that are generated by concatenating words from a list made of 10,000 real words. This approach helps with reducing the impact of data compression due to then having less repetitive characters. For this data, an average number of words to be included in the string is provided as an input, and the generator algorithm concatenates as many as asked from the word list based on a specific distribution. (Uniform, Normal, Gamma, and Zipf distributions are supported.) Generating random HEX strings is another way of generating variable length strings with lower impact due to compression.

### Attribute Skewness

One of the missing but important features in a number of current benchmark data generators is the ability to vary attribute skewness, yet data skew is unavoidable in real world. For our purpose, to benchmark the performance of AsterixDB's join algorithms under join attribute skewness, we applied a normal distribution on integer attributes. Users are allowed to specify the standard deviation and the mean of their distribution to get the desired dataset. More information about the generator can be found in [42], and its is freely available at [41].

# Chapter 3

# Robust Design of Dynamic Hybrid Hash Join

Hybrid Hash Join (HHJ) has proven to be one of the most efficient and widely-used join algorithms in DMBSs. While HHJ's performance depends largely on accurate statistics and information about the input relations, it may not always be practical or possible for a DBMS to have such information available.

HHJ's design depends on accurate statistics to perform well. This chapter is an experimental and analytical study of the trade-offs in designing a robust and dynamic HHJ operator. We revisit the design and optimization techniques suggested by previous studies through extensive experiments and compare them with other algorithms designed by us or used in related studies.

We explore the impact of the number of partitions on HHJ's performance and propose a new lower bound for the number of partitions. We design and evaluate different partition insertion techniques to maximize memory utilization with the least CPU cost. Additionally, we consider a comprehensive set of algorithms for dynamically selecting a partition to spill and compare the results against previously published studies. We then present and evaluate two alternative growth policies for spilled partitions.

These algorithms have been implemented in the context of Apache AsterixDB and evaluated under different scenarios such as variable record sizes, different distributions of join attributes, and different storage types, including HDD, SSD, and AWS Elastic Block Store (AWS EBS).

## 3.1 Introduction

As one of the most popular and expensive DBMS operators, the join operator can significantly impact the performance of a DBMS. HHJ [68, 27] has shown superior performance in computing the equijoin of two datasets among other kinds of join operators. In a nutshell, HHJ groups the records of each dataset into disjoint partitions. A hash table is created to hold one of the partitions in memory (memory-resident partition), while the rest will be written (spilled) to disk to be processed in the next rounds of HHJ at a later time. The number of partitions and the selection of the memory-resident partition are static decisions made at compile time for an HHJ operator. While previous studies [68, 37] suggested various cost models and optimization techniques for enhancing such decisions, these studies have two shortcomings: 1. They assume a uniform distribution for join attribute values. 2. Their cost models rely on having accurate statistical information such as input sizes prior to query execution.

Unfortunately, collecting and accessing or predicting such information may not always be feasible. For example:

- Many data management systems process external data that resides outside their storage for which they have little or no information. (Examples include: Apache AsterixDB

[2], Apache Spark [3], and Oracle external tables [7].)

- The accurate sizes of join inputs may not be known if they result from other operators instead of being base relations.
- Newly developed DBMSs may not have statistics available until they become more mature in other dimensions.

Not having sufficient statistics can be detrimental to the performance of operators whose designs depend on such information. In the [59], the authors have proposed *Dynamic HHJ* to address the unbalanced distribution of join attribute values by dynamically destaging the partitions at the runtime of a join query.

Investigating the Dynamic HHJ algorithm reveals several design questions that must be explored carefully, as they may impact the system's overall performance:

- Number of partitions: How many partitions should the records be hashed into if the sizes of inputs are unknown or inaccurate?
- Partition Insertion: How can we find a "good" page (memory frame) within a partition for inserting a new record?
- Victim Selection Policy: How can we select a "good" partition to spill in the case of insufficient memory?
- Growth Policy: How many memory frames should a spilled partition be allowed to occupy?

With this motivation, this chapter is an experimental survey of the trade-offs in designing a robust Dynamic HHJ algorithm. We answer the questions above through a comprehensive evaluation of different design aspects of the Dynamic HHJ algorithm and evaluate the alternative options through extensive experimental and model-based analyses.

First, we propose a new lower bound for the number of partitions for Dynamic HHJ. We show that our proposed lower bound, while simple, can reduce the total amount of I/O by up to a factor of three in some investigated scenarios. Second, we study different partition insertion algorithms to efficiently find a frame with enough space in the target partition. We evaluate the effectiveness of these algorithms on partition compactness (fullness) and total I/O reduction. Additionally, we propose and evaluate two policies for allocating memory frames to spilled partitions. Finally, we propose and implement various dynamic destaging (victim selection) strategies and evaluate them under different scenarios such as different record size distributions, join attribute value distributions, and combinations thereof. The suggested optimization techniques and algorithm variants have been implemented in the Apache AsterixDB system and evaluated on different storage types, including HDD, SSD, and AWS EBS.

The remainder of the chapter is organized as follows: Section 3.2 provides background information on Apache AsterixDB and the workflow of the HHJ and Dynamic HHJ operators. Section 3.3 discusses previous work related to this study. In Section 3.4, we discuss the lower bound on the number of partitions to use in practice. Section 3.5 introduces and evaluates different partition insertion algorithms. In Section 3.6, two policies for the growth of spilled partitions are discussed and evaluated. Section 3.7 discusses and evaluates various destaging partition selection policies. In Section 3.8, we study the performance tradeoffs between single-core and multi-core execution of Dynamic HHJ. Section 4.9 summarizes the chapter.

## 3.2 Background

### 3.2.1 Hybrid Hash Join

Like other hash-based join algorithms, HHJ uses hashing to stage large inputs to reduce record comparisons during the join. HHJ has been shown to outperform other join types in computing equijoins of two datasets. It was designed as a hybrid version of the Grace Hash Join and Simple Hash Join algorithms [68, 27]. All three mentioned hash join algorithms consist of two phases, namely "build" and "probe". During the build phase, they partition the smaller input, which we refer to as "build input", into disjoint subsets. Similarly, the probe phase divides the larger input, which we refer to as "probe input", into the same number of partitions as the build input. While all three algorithms share a similar high-level design, they differ in their details, making each of them suitable for a specific scenario.

Grace Hash Join partitions the build and probe inputs consecutively, writing each partition back to disk into a separate file. This partitioning process continues for each partition until it fits into memory. A hash table is created to process the join once a partition is small enough to fit in memory. Grace Hash Join performs best when the smaller dataset is significantly larger than the main memory.

In Simple Hash Join, records are hashed into two partitions: a memory-resident and a disk (spilled) partition. A portion of memory is used for a hash table to hold the memory-resident partition's records. Simple Hash Join performs well when memory is large enough to hold most of the smaller dataset. In Grace Hash Join, the idea is to use memory to divide a large amount of data into smaller partitions that fit into memory, while Simple Hash Join focuses on the idea of keeping some portion of data in memory to reduce the total amount of I/O, considering that a large amount of memory is available. In the following, we discuss the details of the HHJ operator and compare its design with its parent algorithms.

Like Grace Hash Join, HHJ uses hash partitioning to group each input's records into "joinable" partitions to avoid unnecessary record comparisons. Like Simple Hash Join, HHJ uses a portion of memory to keep one of the partitions and its hash table in memory, while the rest write to disk. Keeping data in memory reduces the total amount of I/O, and utilizing a hash table lowers the number of record comparisons. During the build phase of HHJ, the records of the smaller input are scanned and hash-partitioned based on the values of the join attributes (Figure 3.1-(a)-1). The hash function used for partitioning is called a "split function". The records mapped to the memory-resident partition remain in memory (Figure 3.1-(a)-2), while the rest of the partitions are written (frame by frame) to disk (Figure 3.1-(a)-3). Pointers to the memory-resident partition's records are inserted into a hash table at the end of the build phase (Figure 3.1-(a)-4).

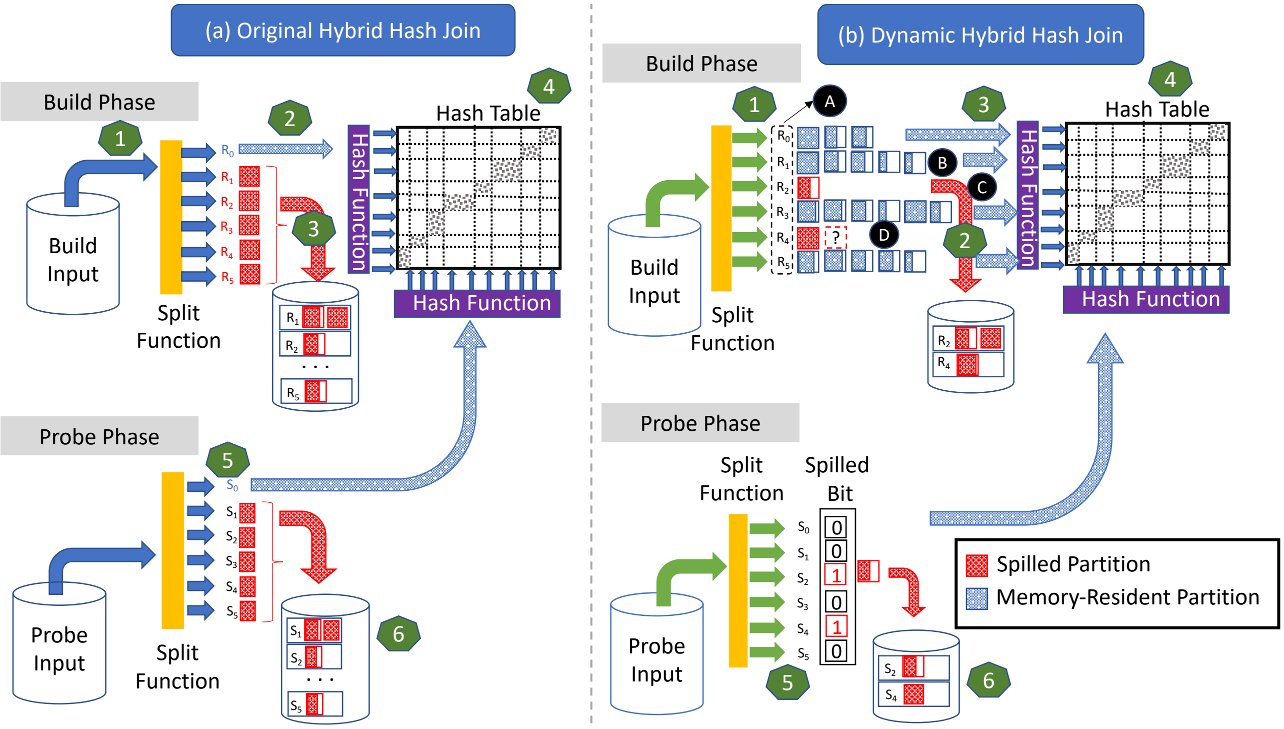


Figure 3.1: Workflow of (a)- Hybrid Hash Join (left) and (b)- Dynamic HHJ (right)

After finishing the build phase, the probe phase starts by scanning and hash-partitioning the records of the larger input. The same split function used during the build phase is used for this step. The records that map to the memory-resident partition are hashed using the same hash function used in the build phase to probe the hash table. All other records are written (frame by frame) to their partition's probe file on disk (Figure 3.1-(a)-5).

After all records of the probe input have been processed, the pairs of spilled partitions from the build phase and probe phase are processed as inputs to the next rounds of HHJ (Figure 3.1-(a)-6). The initial execution of build and probe inputs is considered round 1, and round *n* consists of joining a set of spilled partitions pairs from round*(n-1)* using HHJ recursively.

### 3.2.2 Dynamic Hybrid Hash Join

Dynamic HHJ was first introduced in [59], where the authors used dynamic destaging instead of the static predefined memory-resident partition method. As Figure 3.1-(b)-1 shows, the build phase starts by reading the records of the build input frame by frame into memory. In Dynamic HHJ, as opposed to HHJ, all partitions in the build phase have an equal chance to grow as long as enough memory frames are available. This flexibility in acquiring frames may cause some partitions to receive more frames than others if join attribute values are skewed. Every time that all of the memory frames are allocated, one of the partitions will be dynamically selected to be written to disk (Figure 3.1-(b)-2). This dynamic destaging is especially useful when the build input size or the distributions of join attribute values are unknown or inaccurate.

After partitioning the build dataset's records, pointers to the records of the memory-resident partitions are hashed and inserted into the hash table to be probed (Figure 3.1-(b)-3 and Figure 3.1-(b)-4). Once the build phase is over and the hash table is created, the probe phase starts by reading the probe dataset into memory one frame after another. All of the incoming records will be hashed using the same split function used during the build phase to find out if their corresponding partition from the build phase is a disk or an in-memory partition with the assistance of a bit vector (Figure 3.1-(b)-5). The records mapping to a disk-resident partition will be written to disk using an output frame. The records that belong to an in-memory partition will be hashed using the same hash function used during the build phase in order to find their potential matches by probing the hash table. As the last step (Figure 3.1-(b)-6), once the probe phase is over, the spilled partitions from the build phase along with their corresponding partitions from the probe phase will be processed in a similar way in the next round of the HHJ operator (Steps 1 through 6 in Figure 3.1-(b)).

## 3.3 Related Work

HHJ was first proposed in [27]. The authors showed its superior performance compared to other types of joins using simple cost models, especially if a large amount of memory is available[68]. In [37], the authors provided a more detailed cost model to determine the optimal buffer allocation for various join types.

One of the key problems in configuring HHJ for execution is to choose the number of partitions into which to hash the records. In [68], the author provided an equation for calculating the number of partitions based on the memory and build input size. In [47], the authors derived an upper bound on the number of partitions and then merged smaller partitions to reduce the fragmentation in each partition, which is helpful when the join attribute values are skewed. We introduce a lower bound for the number of partitions and show how it can significantly reduce the total amount of I/O in some cases.

Another challenge for executing HHJ is to efficiently find a frame with sufficient space in the target partition for each incoming record. This problem is similar to the Bin-Packing problem [50, 28]. The problem has also been widely studied in the operating system and the DBMS literature [55, 69] for managing free disk space. In this chapter we will examine those algorithms and a few more for inserting records in partitions during HHJ. The difference between our work and disk-related studies is that in our work records will not reside in the partitions long term, and no deletion apart from partition spilling happens in this case.

The authors of [59] proposed a dynamic destaging scheme where the partition written to disk is selected dynamically during execution. In [33], Graefe et al. detailed the optimization techniques and the design of Dynamic HHJ variant in Microsoft SQL Server. Those two studies are closely related to our work; both choose the largest partition to be written to disk. Despite some reasoning, the authors discuss no other options, nor do they evaluate them. Our study defines 13 different possibilities and evaluates them under various record sizes and join attribute value distributions.

In a concurrent study, the authors in [13] have investigated how and when to use radix join instead of the non-partitioned hash join in a main memory DBMS. Regarding AsterixDB [2, 9, 44], the details of its default Dynamic HHJ can be found in [44].

## 3.4 Number Of Partitions

The first step in configuring the HHJ operator is to determine the number of the partitions for partitioning the input datasets.

There are two main constraints to be considered when choosing the number of partitions: 1. An HHJ operator needs at least two partitions to divide the input dataset into smaller subsets. 2. Each partition needs at least one output frame in order not to spill less than half-full frames to disk.

As such, the number of partitions for an HHJ should be chosen from the range of:

$$Number\, of\, Partitions = [2, \#\, of\, memory\, frames] \tag{3.1}$$

In [68], the author offers the following equation to calculate the number of partitions for an HHJ operator.

$$B = \left\lceil \frac{|R| * F - |M|}{|M| - 1} \right\rceil \tag{3.2}$$

$|R|$ represents the size of the build input in frames, F is a fudge factor, $|M|$ represents the size of the memory in frames available to this join operator, and B is the number of disk-resident partitions. Based on this equation, the HHJ operator will use B+1 partitions (including a memory-resident partition) and finish in B+1 rounds.

While this equation calculates the number of partitions in a way that minimizes the total amount of I/O and rounds in HHJ, any inaccuracy in estimating its input parameter, $|R|$, can introduce fluctuations in the performance of HHJ as the amount of available memory varies. This is especially true when only a few partitions are created (large memory). In this case, data is distributed among just a few partitions, causing a high penalty for spilling a partition as a large amount of data will be written to disk. The purpose of this section is to provide a lower bound on the number of partitions to prevent excessive spilling due to inaccuracy of the provided information.

Figure 3.2 shows the result of a simulation study that explores the impact of the number of partitions on the total amount of I/O during the execution of an HHJ operator. Final result writing is excluded from this measurement. For simplicity, both the build and probe inputs contain the same size of data and the amount of memory is set to 10GB in all cases. In Figure 3.2-(a), a fixed number of partitions have been used for all rounds of HHJ. The black diamonds on each line show the number of partitions suggested by Eq. 3.2 given accurate parameter values. As Figure 3.2-(a) shows, if accurate input values such as input dataset sizes were provided, Eq. 3.2 can accurately calculate the minimum number of partitions that minimizes the total amount of I/O for HHJ. However, if there is no a priori information

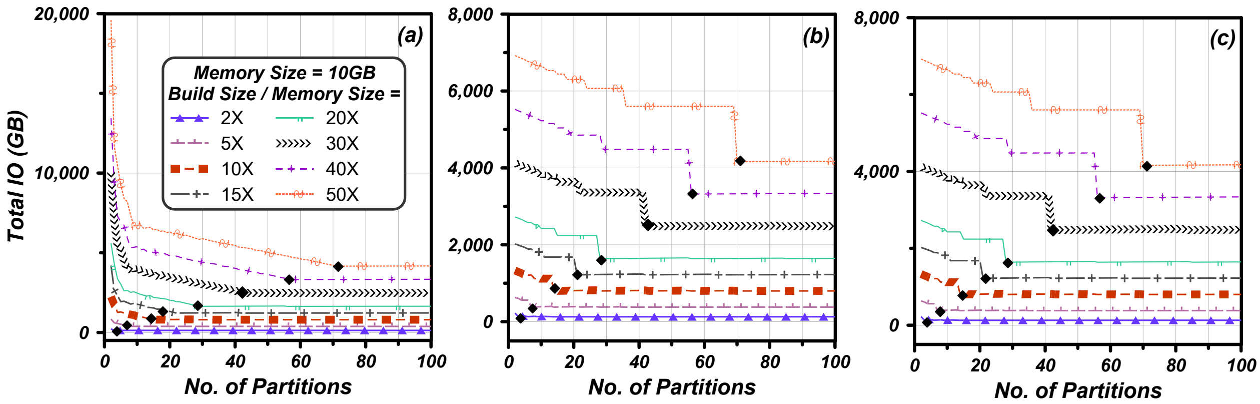


Figure 3.2: Impact of number of partitions on the total amount of I/O in Dynamic HHJ (excluding final result write). (a) Fixed number of partitions used in all rounds of Dynamic HHJ (Frame Size=32KB) - (b,c) Fixed number of partitions used in the first round, Eq. 3.2 used for rounds +2.((b). Frame Size = 32KB, (c). Frame Size = 128KB).

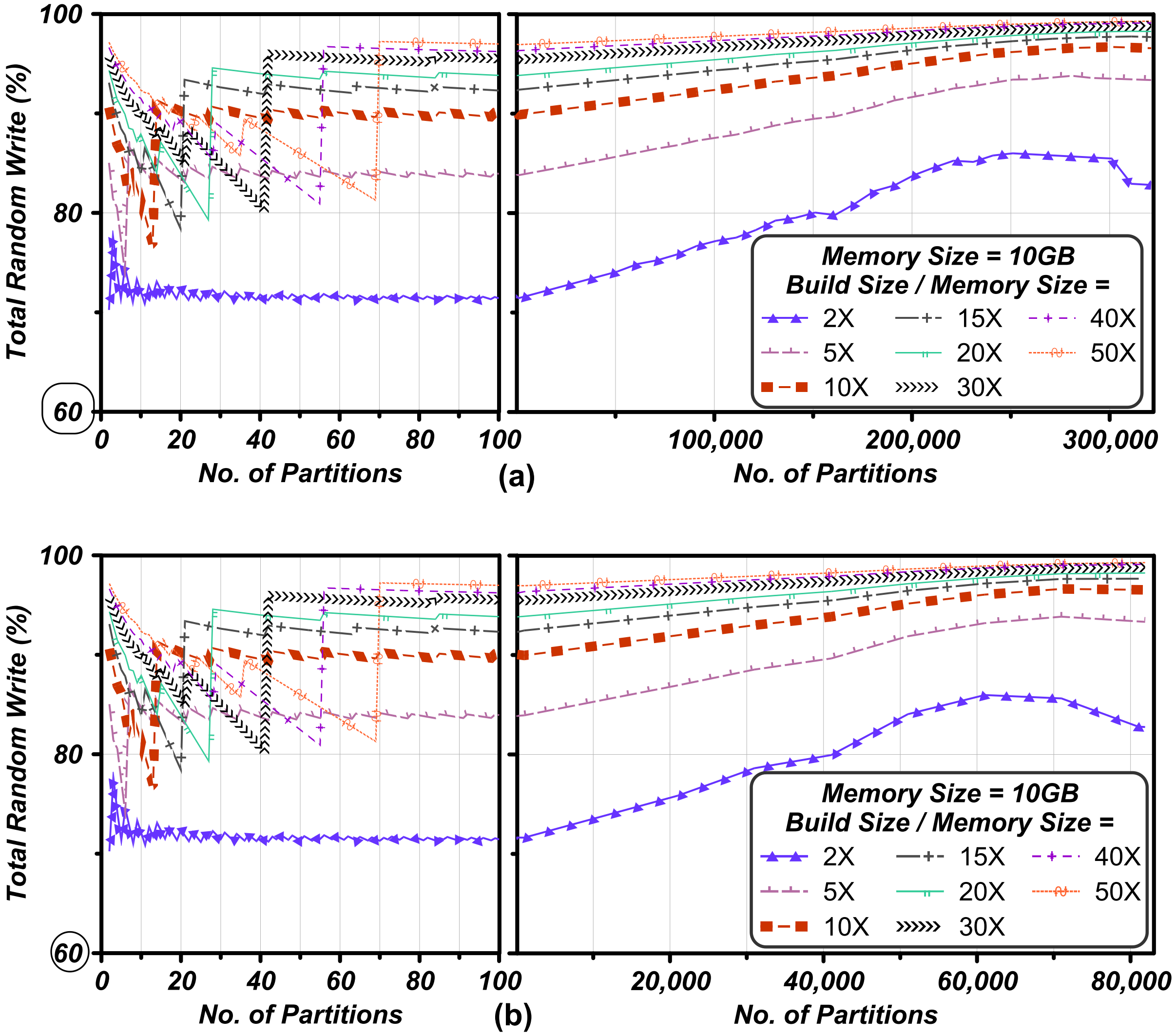


Figure 3.3: Ratio of random writes over total amount of writes reported in percentages (excluding writing the final results)- (a) Frame Size = 32KB, (b) Frame Size = 128KB

or if the provided information is inaccurate and the build input is larger than anticipated, Eq. 3.2 will suggest a smaller number of partitions than needed and cause extra I/O. As Figure 3.2-(a) shows, choosing a small number of partitions can lead to a large amount of unnecessary I/O and degrade the system's performance. We can, however, use Eq. 3.2 to calculate the number of partitions for the subsequent rounds of HHJ as the sizes of spilled partitions are known. Figure 3.2-(b) shows how using the spilled partition sizes to calculate the number of partitions for the next rounds of an HHJ can reduce the total amount of spilling of the HHJ operator.

We recommend using 20 as the minimum number of partitions instead of 2 when accurate a priori information is not available for the HHJ operator. As Figures 3.2-(a) and 3.2-(b) show, the amount of I/O drops dramatically before 20 partitions. By having a lower bound of 20, each spilled partition spills no more than 5% of the data, so the potential for significant "spilling error" is low.

As we saw so far, choosing too few partitions leads to a handful of large-sized partitions causing extra rounds of HHJ and a large amount of spilling to disk. On the other hand, while using a larger number of partitions can reduce the total amount of spilling, it can make the join's I/O pattern more random due to frequent writings of partitions containing just a few frames. Fragmentation within frames is another downside of having a very large number of partitions. In [47], the authors defined an upper bound for the number of partitions in order to reduce fragmentation and random writes due to too many single-frame partitions. However, to the best of our knowledge, no lower bound on the number of partitions has been suggested to improve the performance of the HHJ algorithm.

Additionally, we study the impact of frame size on the amount and pattern of I/Os happening during the execution of the HHJ operator. Figure 3.2-(c) shows the impact of the number of partitions on the amount of I/O when the frame size is set to 128KB. By comparing Figures of 3.2-(b) and 3.2-(c), we can see that changing the size of memory frames from 32KB to

128KB does not change the total amount of I/O occurred during the join execution. Figures 3.3-(a) and 3.3-(b) show the percentage of writes (excluding final result writing) that are conducted randomly when the memory frame size is 32KB and 128KB, respectively. As these figures show, using either 32KB or 128KB leads to a similar I/O pattern since for each spilling the first write is random and the rest of the data is written sequentially regardless of being a large frame or several small frames. Lastly, a lower bound of 20 partitions does not cause too many random I/Os since data will be written to only a few (at most 20) files on the disk. A modest filesystem cache can turn many of these random writes into sequential ones (Elevator Algorithm). As Figure 3.3 shows, choosing a very large number of partitions can cause the majority of the writes to disk to be random.

## 3.5 Partition Insertion

After choosing the number of partitions (P), the build phase starts by reading its input into memory one frame after another. The split function is applied to each incoming record's join attribute(s) to find their destination partition. Once the partition is known, we need to search for a frame with sufficient space within the destination partition to hold the record. If all of the records have the same or similar sizes, all of the previously allocated frames apart from the last frame will be similarly full. In this case, we only need to check if the last frame can hold the record or a new frame should be allocated. However, if records are variable in size, then each allocated frame may have a different amount of leftover space. Thus, for each incoming record, we need to search for a frame with enough space to hold it. Note that leftover space in frames can also happen when records are fixed-size (i.e. when the frame size is not divisible by the record size), but then all of the frames will have the same amount of free space. The search starts from the newest allocated frame and proceeds towards the oldest one. If this search is unsuccessful, a new frame will be allocated and appended to this partition's in-memory frames array if enough memory is available. However, if the available memory is not sufficient for a new frame allocation, one of the memory-resident partitions will be selected for spilling to release some memory space. This choice is called victim selection and will be discussed in Section 3.7.

**Problem Definition.** Our goal for partition insertion is to make each partition as non-fragmented as possible by choosing the destination frame for each incoming record in such a way that minimizes the free space in each frame to avoid unnecessary I/O. On the other hand, searching for a proper frame for each record could be CPU-time-consuming. Our goal is to find a destination frame efficiently while making the partition as compact as possible. Two influential factors should be considered for designing partition insertion algorithms. First, there will be no record deletions to cause fragmentation in this scenario; only a complete partition will be written to disk in case of insufficient memory. Second, records can come in many different sizes. This variation in record sizes adds to the complexity of partition insertion for two reasons. First, the space required for each record is different from other records. Second, the insertion of variable-sized records in fixed-size frames leaves a different amount of free space in each frame. Placing variable-sized objects in a fixed-size space is known as an "online object placement" or "online organization" problem. It is an example of the online bin-packing problem [11], a well-known NP-hard problem. Some object placement strategies have been studied and optimized for free space management on disk for permanent placement of objects [55]; however, they may not exhibit similar performance characteristics when used for memory space management.

In the following, we present and evaluate different algorithms for partition insertion. he algorithms considered here are:

**Append($n$).** Append($n$) performs a search on the last $n$ frames of the target partition in the order of the newest frame to the oldest. The incoming record will be placed in the first frame with enough space. If no such frame is found, a new frame with enough space will be appended to this partition.

**First-Fit.** In the First-Fit algorithm, the search starts from the last (newest) frame towards the first (oldest) frame of the partition and stops as soon as a frame with enough space for the record is found. In First-Fit's worst-case scenario, all frames are searched and a new frame is appended upon an unsuccessful search.

**First-Fit(%$p$).** This is a parameterized and more general version of the First-Fit algorithm in which at most %$p$ of the partition's frames are searched for the record insertion. Similar to the previous algorithms, the search proceeds from the newest frame towards the oldest. It appends a new frame to the array and inserts the record if no frame with enough space is found. In comparison to First-Fit, this algorithm provides a better balance between extensive search and the compactness of the frames in the partition. This algorithm is similar to Append($n$) as they both start searching from the end of the frames array and stop if the stopping criteria are met. The stopping criteria in Append($n$) is $n$ frames, while in First-Fit(%$p$) it is %$p$ of frames.

**Best-Fit.** Best-Fit, a well-known space management algorithm, searches through all of the partition's frames to find the frame with the smallest free space that can accommodate the record. This algorithm tries to maximize frame compactness based on the current state of the frames and the size of the record being inserted.

**Next-Fit.** Next-Fit starts searching from a different location for each record to avoid checking some frames over and over again. In this algorithm, the search is guided based on the size and insertion location of the previous record.

As a modified version of the First-Fit algorithm, Next-Fit initially starts searching from the end of the partition's array. However, after the first record, the search starts from the location where the previous record was inserted. If the size of the current record is larger than the previous record, the search continues toward the newer frames. However, if the current frame is smaller than the previous record, the older frames are searched first. In the latter case, if no frame with enough space is found, then the search continues toward the newer frames. A new frame is appended to the end of the partition's frame array if no frame with enough space is found.

**Random(%$p$)** In this algorithm, for each record, up to %$p$ of the partition's frames are randomly searched. The search stops as soon as a frame with enough space is found. This algorithm avoids searching the same frames extensively and unnecessarily by its random selection of frames. We tried different random number generators such as Java's default random number generator, Mersenne Twister Fast [6], C++ 11 MinSTD, and XorShift 64 bits and compared their performance to choose the least expensive random number generator for our case. Our experiments showed that these random number generators performed very similar to each other. As such, we decided to use Java's default random number generator.

### 3.5.1 Choosing the best parameter values

As we discussed, some of the partition insertion algorithms such as Random(%$p$), Append($n$), and First-Fit(%$p$) have a parameter that needs to be properly set. We compared the performance of these algorithms under different value settings for their parameters using the 1 Large Record Coexist setting whose specification can be found in Table 3.1.

Figures 3.4-a, 3.4-b, and 3.4-c show, on average, how much of the frames are filled with records when 90%, 50%, and 10% of the records are large. As we can see, all of the different parameters lead to a similar frame fullness in the 90% and 50% cases as the majority of the records are large, only one large record can fit in a frame, and there are few small records to fill the holes in frames. However, when 10% of the records are large, these parameters' fullness results slightly differ from one another. Append(8) appears to have a frame fullness close to the frame fullness of Append(9) and Append(10); however, as Figures 3.4-d, 3.4-e, and 3.4-f show, Append(8) checks fewer frames than Append(9) and Append(10). Figure 3.5 and Figure 3.6 show the average frame fullness and the number of searched frames for different

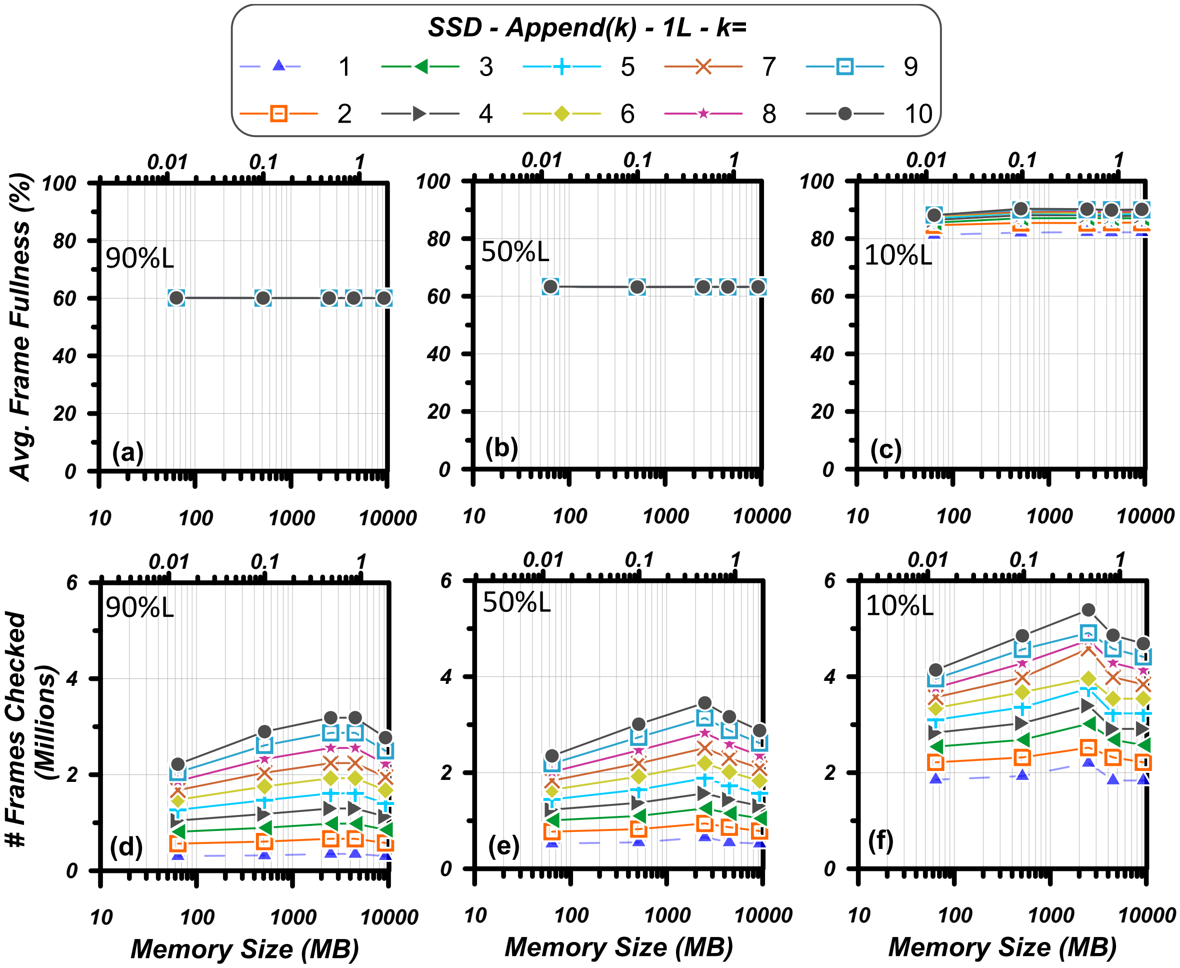


Figure 3.4: Choosing The Best Parameter Value for Append(K) (1 Large Record Coexist). (a) - Average frame fullness when 90% of records are large. (b) - Average frame fullness when 50% of records are large. (c) -Average frame fullness when 10% of records are large. (d) - Total number of searched frames when 90% of records are large. (e) - Total number of searched frames when 50% of records are large. (f) - Total number of searched frames when 10% of records are large.

parameter values for First-Fit(P) and Random(P). In this experiment, enough memory is available to keep all of the joins in memory. As Figure 3.5-a and Figure 3.6-a show, all parameters of First-Fit(P) and Random(P) have a similar average frame fullness; however, they differ in the number of frames that they search. Hence, based on these experiments, Random(%10), Append(8), and First-Fit(%10) were found to achieve the highest degree of frame fullness with the least number of frames being checked. We will therefore study just these settings as we move forward with these policies.

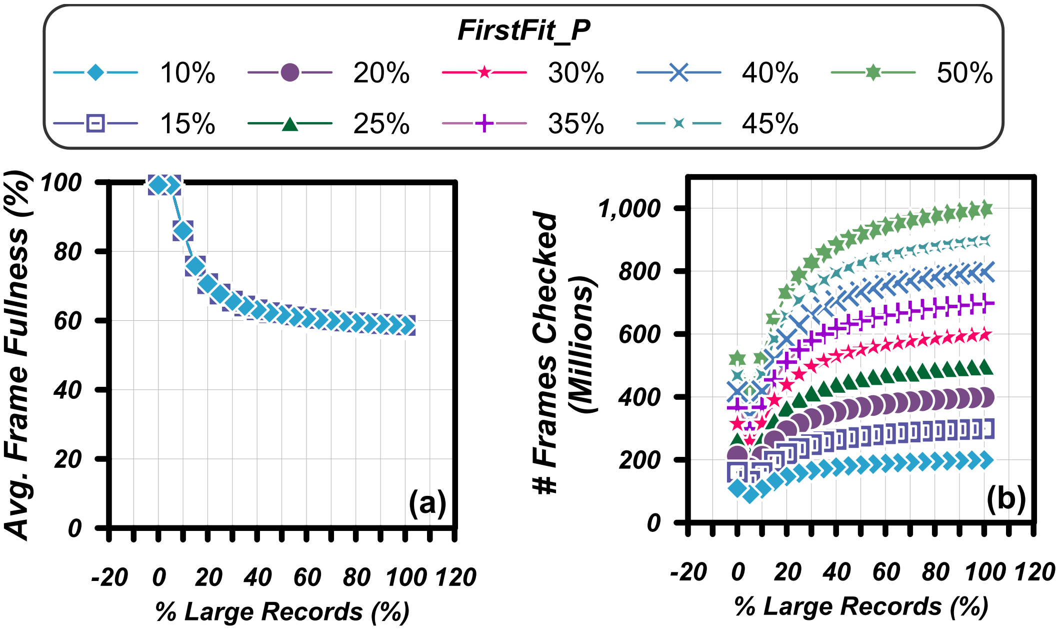


Figure 3.5: Choosing The Best Parameter Value for First-Fit(P) (1 Large Record Coexist). (a) - Average frame fullness by different parameters of First-Fit(P). (b) - Number of searched frames by different parameters of First-Fit(P).

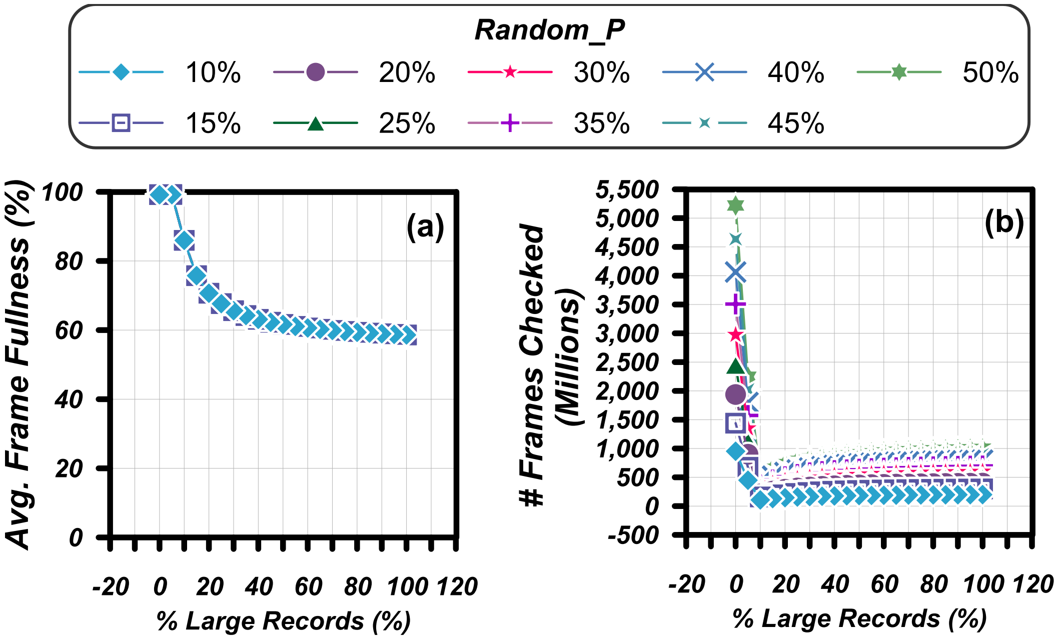


Figure 3.6: Choosing The Best Parameter Value for Random(P) (1 Large Record Coexist). (a) - Average frame fullness by different parameters of Random(P). (b) - Number of searched frames by different parameters of Random(P).

### 3.5.2 Dataset and Experiment Design

We use an updated and modified version of the Wisconsin Benchmark [23] data to evaluate the partition insertion algorithms. Its attributes and datasets' high tunability and selectivity make the Wisconsin Benchmark's dataset a good synthetic benchmark dataset for evaluating and benchmarking join queries.

We use variable-length records, one of the modifications added to the Wisconsin Benchmark Data Generator in [42], to introduce two groups of small-sized and large-sized records with a specific ratio between these two groups. We use what we call the 1-Large Record Coexist, 3-Large Record Coexist, and All Small Records datasets in this study, each of which is 1 GB in size. Each memory frame is 32KB in size. The names of 1-Large Record Coexist and 3-Large Record Coexist come from the number of large records that can fit in one frame. Variable-length records are used for small and large records to represent a more realistic scenario. We consider two specific ranges for large records (1-Large and 3-Large record coexist) to study the impact of semi-large and extra-large record sizes fitting in one frame to cover the two ends of the spectrum of large record sizes. Table 3.1 contains the details of the datasets used.

Table 3.1: Dataset Specifications

| **Dataset** | **Small Records** | **Large Records** |
|---|---|---|
| 1-Large Record Coexist | 700 B - 1500 B | 18 KB - 20 KB |
| 3-Large Records Coexist | 700 B - 1500 B | 8 KB - 10 KB |
| All Small Records | 700 B - 1500 B | None |

Each experiment is conducted using an AsterixDB cluster consisting of a Cluster Controller and a Node Controller with one data partition executing on two different nodes of the same AWS type. Each query runs in isolation and utilizes one CPU core. All instances are chosen from US-West-2 availability zone of AWS and have 4 vCPUs and 30.5GB of RAM. The d2.xlarge instance type was used for the HDD experiments, while i3.xlarge and r4.xlarge were used for the SSD and EBS experiments, respectively.

### 3.5.3 Partition Insertion Algorithms' Evaluation

This section evaluates the performance of the described partition insertion algorithms for fixed- and variable-sized records.

#### Small Records Experiment

In our first experiment, both the build and probe datasets are 1GB in size and follow the All Small Records dataset configuration. In this experiment, we are interested in comparing the partition insertion algorithms with respect to the average frame fullness (compactness) and the query execution time to evaluate the efficiency of each algorithm in reaching this degree of frame fullness. The query execution time is the time that it took for a query to execute, excluding the time for query compilation and result returning. Since queries were running in an isolated setting with no other queries running concurrently, the execution time includes zero wait time. In these experiments, we consider different ratios of record sizes over the memory frame size. Since memory frames and records can come in many different sizes, the *ratio* of their sizes is the important factor here. Similarly, we consider various ratios between the data and memory sizes to study the performance trends of the various algorithms.

Figure 3.7(a) shows the average frame fullness as a function of the ratio of the build dataset size to the amount of available memory. The Y-axis starts from 80% for a better visualization. As this figure shows, all algorithms deliver a high and similar average frame fullness when the records are small. This is because small records can easily fit in most frames and increase the average frame fullness by minimizing the leftover space in each frame.

Next, we analyze the performance of the different partition insertion algorithms in reaching

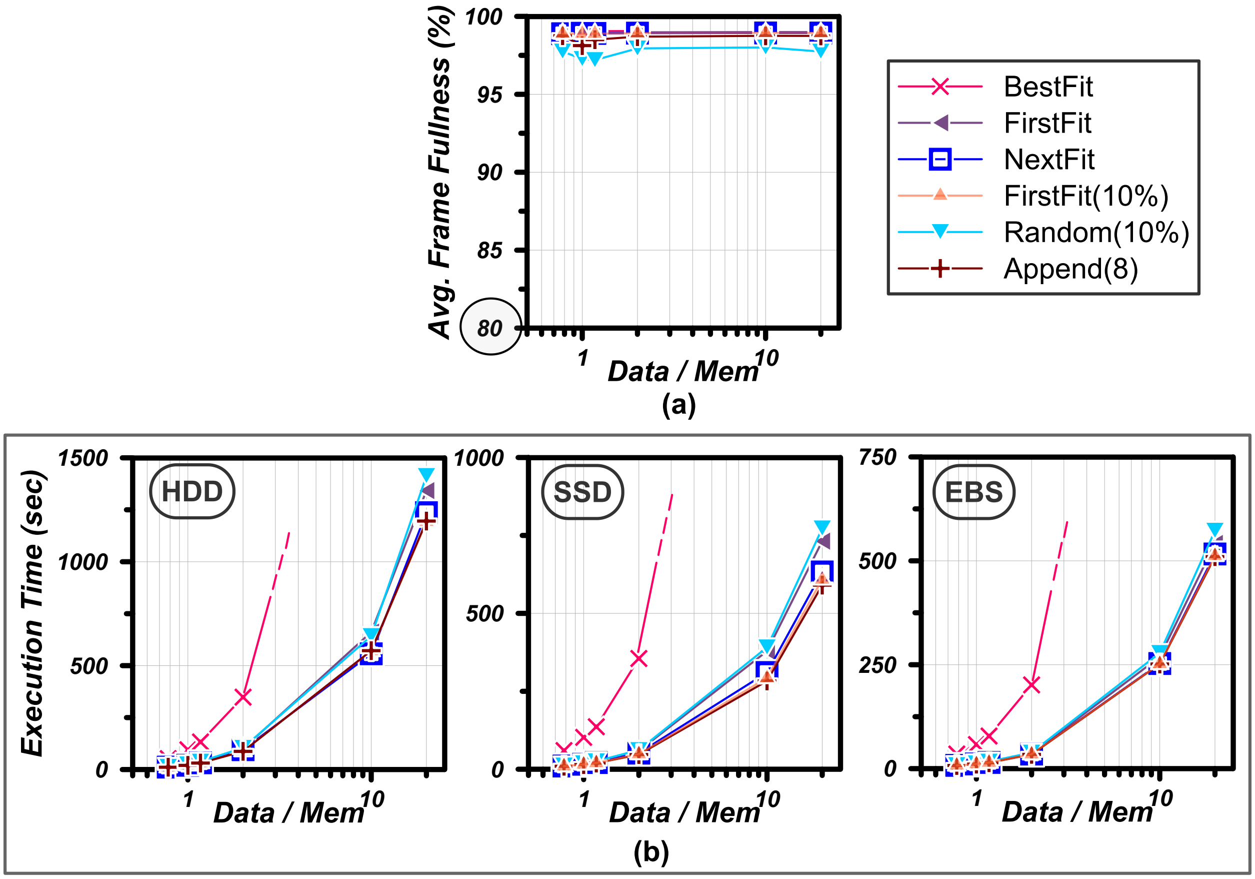


Figure 3.7: Partition Insertion - Small Record Sizes (a) Average frame fullness (b) Execution time on different storage types

their reported frame compactness. Figure 3.7(b) exhibits the execution time of the partition insertion algorithms for three storage types of HDD, SSD, and AWS EBS. We use different storage types to study the impact of the difference in frame compactness of different partition insertion algorithms (which can lead to differences in the amount of disk I/O) on the execution time for each storage type.

The similarity in the size of the records makes the frames, especially the older ones, similarly full. Additionally, suppose a previous record could not find a frame by checking all of the partition's frames due to similarity in record sizes. In that case, it is likely that the next record will not fit in those frames either.

As Figure 3.7(b) shows, the CPU cost due to extensive searching in Best-Fit significantly degrades its performance in all three storage types. Random(10%) is the second-worst algorithm with a slightly higher execution time than the others. Although Random(10%) benefits from the additional stopping criteria, the high time-overhead of the Random function and the high frequency of calling it degrades its performance. First-Fit is the third-worst algorithm in our experiments. First-Fit has a higher execution time than the algorithms with a guided search method (Next-Fit) or additional stopping criteria. This is due to the extensive search of First-Fit. However, the performance of First-Fit is much better than Best-Fit, another extensive search algorithm, as First-Fit stops if it finds a suitable frame. This "first find" strategy has a high impact, especially in this experiment, as all of the records are small and have a good chance to fit in even a relatively full frame.

Next-Fit and First-Fit(10%) perform similarly here with relatively low execution times. Next-Fit's different starting point and its guided search improve its performance. The early termination due to stopping criteria in First-Fit(10%) makes it one of the best-performing algorithms here. Append(8), however, seems to be the best algorithm in this experiment. As Figures 3.7(a) and 3.7(b) show, Append(8) reaches a similar average frame fullness as the other alternatives with the least amount of search effort. (For each record, at most 8 frames are checked.)

### Variable Size Records

This section evaluates the performance of different partition insertion algorithms with input datasets containing records of various sizes.

**3-Large Coexist.** We use the 3-Large Record Coexist dataset for this experiment. The large records versus small records ratio varies between 10%, 50%, and 90%. As Figure 3.8-(a) shows, increasing the percentage of large records lowers the average frame fullness in all algorithms and minimizes their differences in frame compactness. Inserting large records in

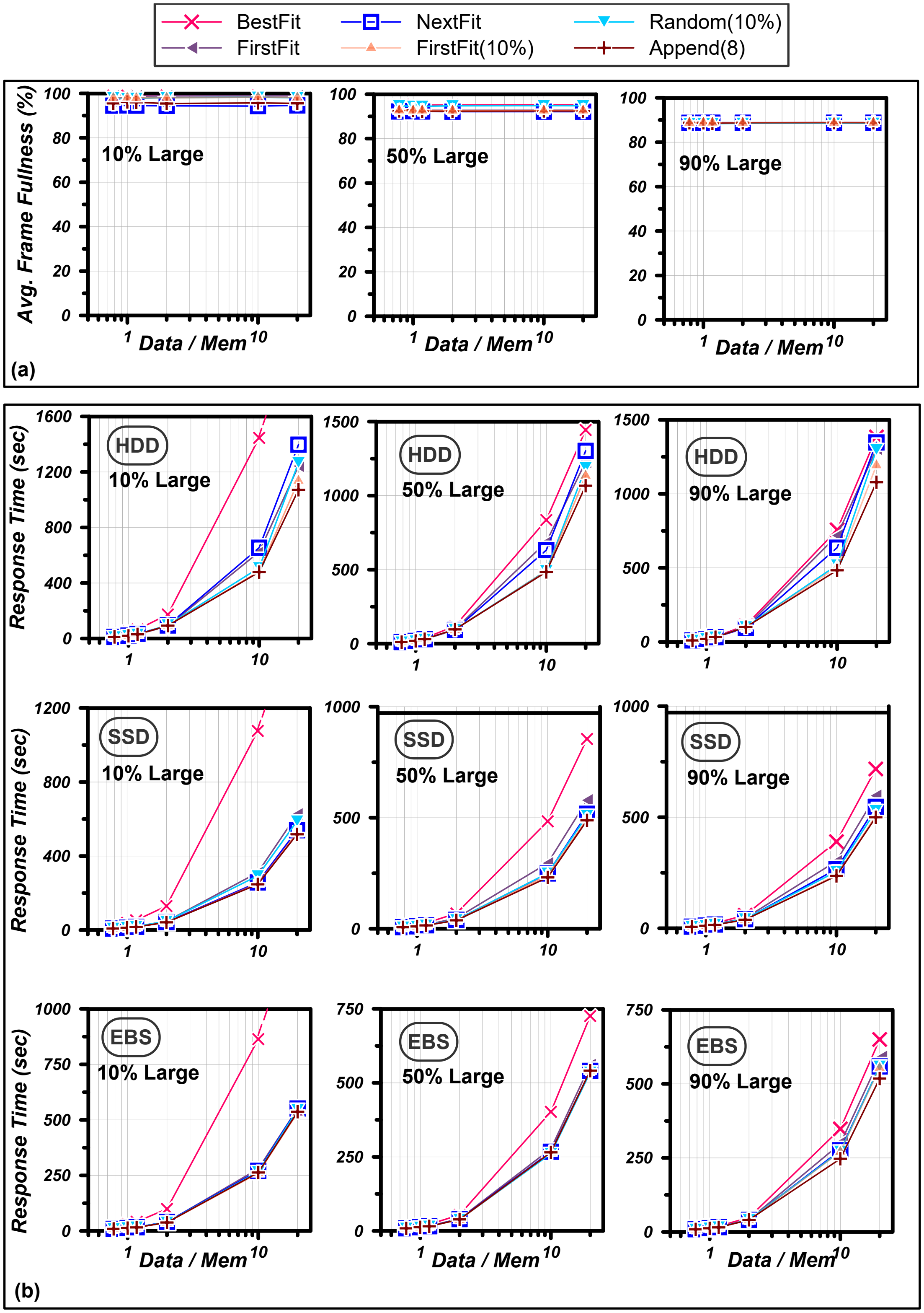


Figure 3.8: Partition Insertion - 3-Large Record Coexist (a) Average frame fullness (b) Execution time on different storage types

a frame may leave a large leftover space that can only be filled with small records. If the small records are limited in number (higher percentage of large records), these leftover spaces remain unfilled and decrease the average fullness. Additionally, the difference between the average frame fullness of the various algorithms diminishes if most of the records are large since only a few frames may have enough space for large records.

As Figure 3.8-(b) shows, Best-Fit again has the highest execution time since for each record insertion it searches all of the in-memory frames of the partition. Furthermore, a higher number of records leads to more searching and thus to a higher execution time for Best-Fit. This rationale is true for the Random algorithm, too, since the random function will be called for 10% of the frames per record insertion. In all of these experiments, Append(8) has the lowest execution time; doing the least amount of work, it still achieves a similar frame fullness to the more intelligent and search-intensive algorithms. While the algorithms other than Append(8) and Best-Fit perform similarly, the algorithms with a stopping criteria perform slightly better. Storage-wise, the overall execution time is higher for HDD than for SSD and AWS EBS due to its longer time for I/O operations. The impact of the difference in the amount of I/O on the execution times of the different algorithms is greater in HDD due to the efficiency of SSD in handling I/O and the high network latency in AWS EBS.

**1-Large Coexist.** In the second variation of our experiments for partition insertion with variable-sized records, we use the 1-Large Record Coexist dataset. As above, both inputs are 1GB. Although all of the datasets are 1GB in this and the previous experiment, the datasets for this experiment have a lower cardinality. This is because the large records in this experiment are approximately 3 times larger than the large records in the previous experiment. We can observe from Figure 3.9(a) that the frame fullness is higher in cases where most of the records are small, especially in the 10% Large case. This is due to several factors:

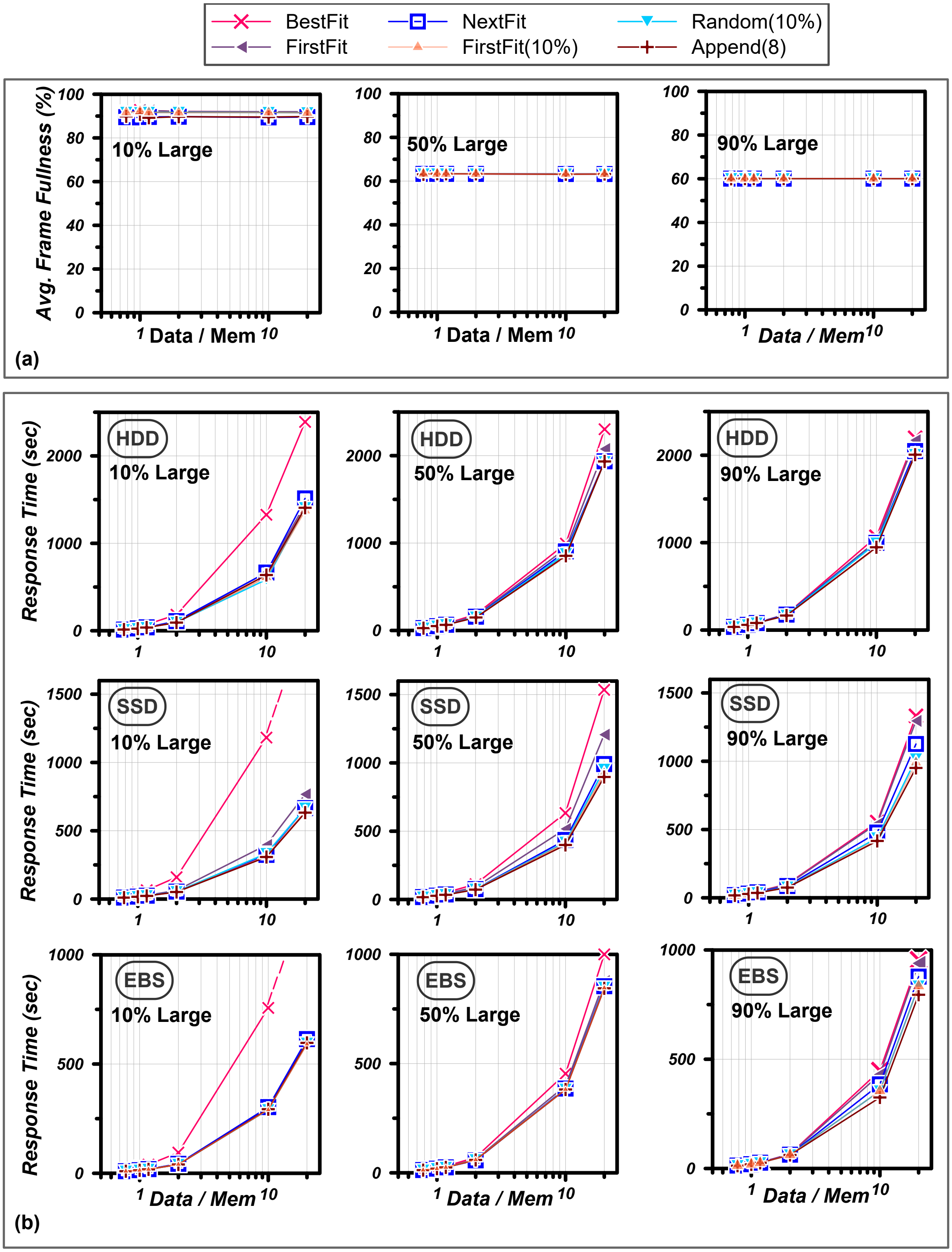


Figure 3.9: Partition Insertion - 1 Large Record Coexist (a) Average frame fullness (b) Response time on different storage types

- The lower percentage of large records means that most of the 1 GB relation is made up of smaller records. Smaller records have a better chance of fitting in partially full frames and thus increasing the frames' compactness.
- The higher number of remaining records, especially when they are small, increases the possibility of making the allocated frames more full.

We see frame fullness drop from 90% to 62% and 60% as we increase the ratio of large records from 10% to 50% and 90%, respectively. This is because each large record requires its own frame, and the small records in the minority can fill the leftover space.

The overall frame fullness in this experiment is lower than in the previous experiment since in this case only one large record can fit in a frame, while in the previous experiment, 3 large records could coexist in one frame and further reduce the leftover space.
As Figure 3.9-(b) shows, similar to the previous experiment, the Best-Fit algorithm has the highest response time and the Append(8) algorithm has the lowest response time in the majority of the cases. However, the difference between Best-Fit and the other algorithms is not as high as in the 3 Large Record Coexist experiment due to the lower cardinality of the inputs reducing the total search costs.

Append(8) had the lowest execution time for both small and variable sized records, so we will use Append(8) as the partition insertion algorithm for the rest of this study.

## 3.6 Spilled Partitions' Growth Policies

In the case of insufficient memory, some of the partitions must be written to disk to open up space for additional incoming records. We will consider several victim selection policies –

policies which select a memory-resident partition to spill – under two variations of how the memory allocation to spilled partitions is managed:

1. **No Grow-No Steal (NG-NS):** There are two main rules for this policy:
   - **No Grow**: A spilled partition can only have one frame to be used as its output buffer once it has spilled.
   - **No Steal**: Only unspilled partitions are selected as victims in case of insufficient memory. A spilled partition writes its output buffer to disk only if the next record hashed to that partition requires more space.
2. **Grow-Steal (G-S):** This growth policy consists of two main rules as well:
   - **Grow**: Spilled partitions may grow as large as the available memory lets them.
   - **Steal**: Spilled partitions have a higher priority to be chosen as a victim partition in cases of insufficient memory.

While more growth policies could be considered for future work (e.g., spill only the full frames of a victim partition), we chose to study these two growth policies as the two ends of the spectrum.

### 3.6.1 Analytical I/O Study for NG-NS and G-S

In this section, we look at the I/O differences between the two growth policies for spilled partitions from an analytical point of view. It is important to realize that both policies perform almost the same amount of I/O; however, they differ from one another in their use of random versus sequential I/O. All of the notations used in this section can be found in Table 3.2.

Table 3.2: Notation used in cost formulas

| Notation | Definition | Example |
|---|---|---|
| R | Size of build relation in frames | 100 |
| M | Size of memory in frames | 50 |
| P | Number of partitions | 20 |
| x | Number of spilled partitions | 5 |

**I/O Analysis for NG-NS.** Let us assume that records are similar in size and that there is no skew in join attribute values. Using this assumption, all partitions are similar in size, in the number of frames, and in the number of records. The following equation calculates the total number of partitions remaining in memory at the end of the build phase:

$$P - x = MAX\left(P, \left\lfloor \frac{M}{\frac{R}{P}} \right\rfloor\right) \tag{3.3}$$

In NG-NS, a memory-resident partition is selected for spilling to disk only when 1. the partitions spilled so far each have a maximum of one frame and, 2. the in-memory partitions ($P - x$ partitions) have used the rest of the frames ($M - x$) and, 3. the next incoming record is hashed into a memory-resident partition.

For our calculation, we can choose any of the partitions to spill as they are all in a similar situation due to the uniformity of data. By spilling the selected partition, $\frac{M-x}{P-x}$ of this partition's data is written to disk sequentially, while the rest of its data ($\frac{R}{P} - \frac{M-x}{P-x}$) will later be written to disk randomly (*i.e.,* one frame at a time).

The following equation calculates the amount of temporary results (build phase only) written to disk in a random and sequential fashion under the NG-NS growth policy:

$$\sum_{i=1}^{x} \left( \frac{R}{P} - \frac{M-i+1}{P-i+1} \text{ Random I/O} \right) + \left( \frac{M-i+1}{P-i+1} \text{ Seq. I/O} \right) \tag{3.4}$$

**I/O Analysis for G-S.** Similar to NG-NS, the next memory-resident partition will spill to disk only if 1. the incoming record is hashed into a memory-resident partition, and 2. each spilled partition has at most 1 frame, and 3. the rest of the memory frames have already been assigned to memory-resident partitions.

Like NG-NS, each spilling partition writes $\frac{M-x}{P-x}$ of its data frames to disk sequentially when it spills for the first time; however, in contrast to NG-NS, G-S writes the rest of the partition's frames in chunks consisting of more than one frame.

The second part of Equation 3.4 holds for G-S as well, as the next victim partition has $\frac{M-x}{P-x}$ frames in memory. The following equation calculates the sizes of the data chunks written to disk by spilled partitions between the $x^{\text{th}}$ and $(x+1)^{\text{st}}$ time that a memory-resident partition was selected as a victim.

$$\frac{1}{P} * \frac{M-x+1}{P-x+1} + \left(\frac{1}{P}\right)^2 * \frac{M-x+1}{P-x+1} + \left(\frac{1}{P}\right)^3 * \frac{M-x+1}{P-x+1} + ... + 1 \tag{3.5}$$

which reduces to:

$$\lim_{P \to \infty} \frac{1}{1-\frac{1}{P}} \left(\frac{M-x+1}{P-x+1}\right) \tag{3.6}$$

Therefore the cost formula in number of I/Os for G-S is:

$$\sum_{i=1}^{x} \left( \lim_{P \to \infty} \frac{1}{1-\frac{1}{P}} \left(\frac{M-i+1}{P-i+1}\right) \text{ Seq. I/O } \right) + \left( \frac{M-i+1}{P-i+1} \text{ Seq. I/O } \right) \tag{3.7}$$

The first term in Equation 3.7 shows that in G-S, each spilled partition writes the rest of its data to disk sequentially. This I/O behavior is different from NG-NS (Equation 3.4), in which the rest of a partition's data is written to disk frame by frame once it first spills.

### 3.6.2 Experimental Analysis of Growth Policies

Based on the cost functions we developed in the previous subsection, we showed that the NG-NS policy leads to more random writes due to using one output buffer allocation per spilled partition. On the other hand, G-S allows the spilled partitions to acquire more than one frame, so its I/O pattern becomes more sequential. Turning random writes into sequential ones can improve performance, especially in systems utilizing HDD. This section compares these two algorithms empirically to verify our expectations from the cost analysis. We used a single join query for which the build and probe datasets contain identical data generated based on the All Small Record dataset configuration. In this experiment, the available memory for the join is a fixed value of 1024MB, while the size of the build and probe inputs varies from 1.2GB, 2GB, 10GB, 20GB, to 100GB. A hard disk is used as the storage device in this experiment. This experiment compares the two growth policies for

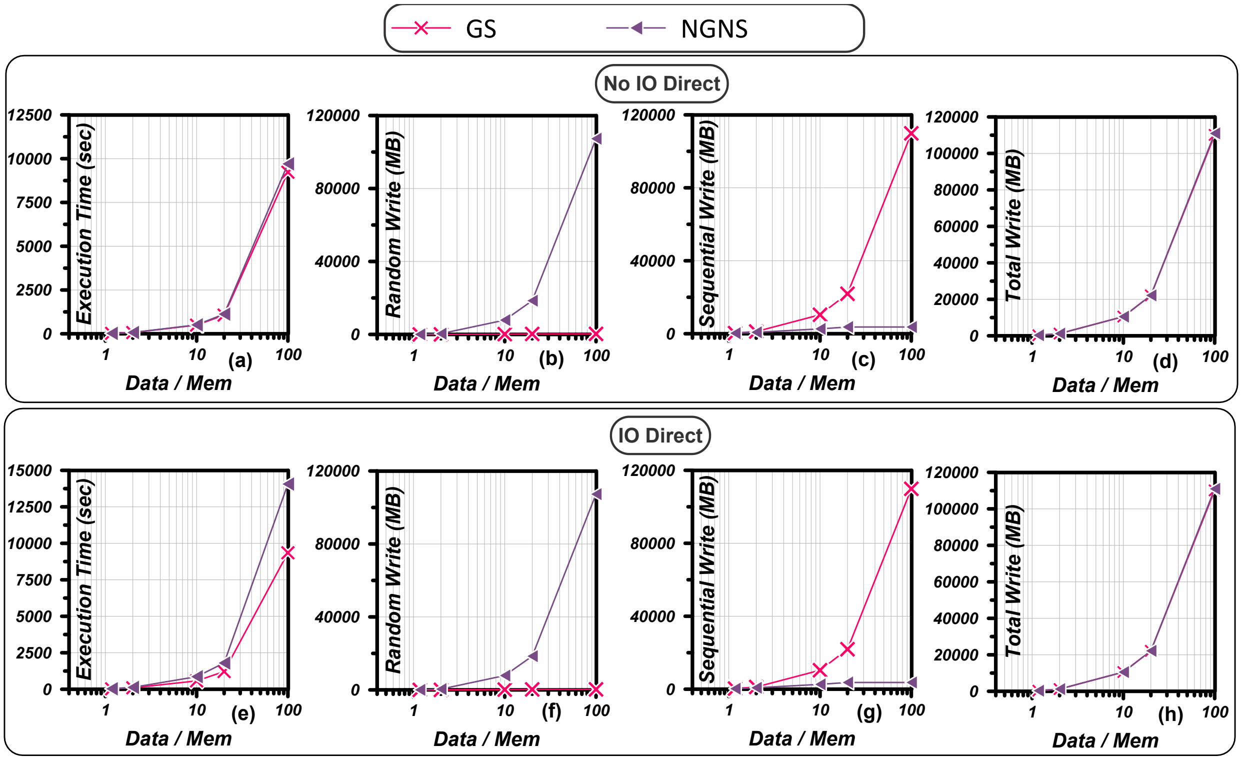


Figure 3.10: Spilled Partition Growth Policies. (a,b,c,d) - Statistics of GS and NG-NS policies with filesystem cache in use. (e,f,g,h) - Statistics of GS and NG-NS policies with filesystem cache disabled.

spilled partitions under two variations of writing to disk: direct or through the filesystem cache. Some database management systems disable the filesystem cache and manage the buffer cache memory themselves. We use the IO_DIRECT library [5] for directly writing data to disk and bypassing the filesystem cache in Linux systems. Figures 3.10-d and 3.10-h show that G-S and NG-NS do the same amount of writing regardless of using or bypassing the filesystem cache. However, as Figures 3.10-c and 3.10-g show, G-S does up to 120x more sequential writes than NG-NS, while NG-NS does up to 120x more random writes than G-S (Figures 3.10-e and 3.10-f). This difference in the I/O patterns of the G-S and NG-NS while writing the same amount of data to disk aligns with our results from the previous subsection. Increasing the input sizes causes more spilling to disk, making the difference between these two policies even more significant.

Next, we study the performance of these growth policies with and without filesystem cache being present. Figure 3.10-e shows the execution time of G-S and NG-NS policies when data is written directly to disk (disabled filesystem cache). In this case, NG-NS takes a longer time than G-S to finish due to performing more random writes. The impact of random writes of NG-NS on its performance becomes more significant as the size of the data relative to memory increases; this is because more data is written randomly and the storage device is an HDD. However, Figure 3.10-a shows that using a filesystem cache minimizes the difference in execution times of these two policies. This is because the filesystem cache collects write requests and orders them based on their target file location on disk (Elevator Algorithm) before sending them to disk; as a result, many of the random writes turn into sequential ones in NG-NS.

Based on our results, choosing the preferred growth policy depends on whether the DBMS performs its own caching or uses the filesystem cache. In AsterixDB, we decided to use NG-NS for two reasons: 1. The filesystem cache is used. 2. NG-NS does not fully utilize its given memory. In future work, we intend to use this leftover memory for other operators of the same query or other queries under a more global memory management policy.

## 3.7 Victim Selection Policies

One or more memory-resident partitions must be written to disk to regain enough space for the incoming records if the available memory is insufficient. In-memory partitions may have different sizes if records have variable sizes or if their distribution between partitions is unbalanced due to skew in join attribute values. In the case of variable-sized partitions, we must decide which partition(s) should spill to disk, considering that we do not know how much data is left to be processed. The partition selected for spilling is called a victim partition, and the policy based on which victim partitions are selected is called the victim selection policy.

In the original HHJ algorithm [68, 27], one partition is selected upfront (before query execution) as the in-memory partition, while the rest of the partitions are disk partitions. To ensure that the chosen partition can indeed remain in memory, we must know the sizes of the inputs and the distribution of join attribute values.

As mentioned earlier, the authors of [59] and [33] instead use dynamic destaging to choose victim partitions at runtime. They always select the largest memory-resident partition as the victim partition and limit the spilled partitions to acquiring a maximum of one frame, following the NG-NS growth policy. Neither of these studies considers other victim selection policies or spilled-partition growth policies. Additionally, they do not provide any experiments to show the superiority of their approach.

In the following, we consider 13 possible policies for selecting the next victim partition among non-spilled partitions. These victim selection policies are designed for the NG-NS growth policy.

The design space for these policies is based on data size and frame fragmentation considerations. Largest Size, Largest Records, Median Size, Median Records, Smallest Size, Smallest Records, Record Size Ratio, Half Empty, and Low High are designed with respect to the data size. The Largest Records and Largest Size are expected to perform well when a large portion of the build dataset is left to be processed. In contrast, the Smallest Records and Smallest Size are expected to perform well when a small portion of the build input remains to be processed. Record Size Ratio considers the number of records in choosing a large partition as victim and is expected to perform well when a large portion of the build input remains to be processed. Half Empty, Low High, Median Size, Random, and Median Records are designed to take a middle ground and are expected to have an average but stable performance for various cases. Least Fragmentation considers the frames' fragmentation to choose a victim and is expected to have an average amount of spilling since it may choose a partition of any size to spill. Smallest Size Self Victim and Largest Size Self Victim are policies which take both the frame fragmentation and data size into consideration and are expected to have an average amount of spilling since the victim partition can be of any size when it is the inserting partition itself.

The overall input dataset sizes are unknown to the DBMS during these experiments. The following list describes the considered victim selection policies:

**Largest Size:** Choose the partition with the largest size in memory as a victim to maximize sequential writes and to defer the next spill(s) as long as possible.

**Largest Records:** Choose the partition with the maximum number of records to spill.

**Largest Size Self Victim:** Choose the partition into which the record is hashed if it has at least one frame. Otherwise, choose the largest partition to spill.

**Median Size:** Choose the partition with the median size among all of the memory-resident partitions as the victim partition.

**Median Records:** Choose the partition with the median number of records to spill.

**Smallest Size:** Choose the smallest partition with at least one memory frame as the victim partition to avoid overspilling.

**Smallest Records:** Choose the memory-resident partition with the minimum number of records ($\geq 1$) for spilling.

**Smallest Size Self Victim:** Choose the partition into which the record is hashed to spill if it has any frames. Otherwise, the smallest-size partition will be selected as the victim.

**Random:** Choose randomly any of the memory-resident partitions as the victim partition.

**Half Empty:** This victim selection policy starts optimistically by guessing that the remainder of the build input is small and spills the smallest partition. However, it acts pessimistically and spills the largest partition if more than half of the partitions have spilled.

**Least Fragmentation:** Choose those partitions that have the least amount of fragmentation in their frames, thus trying to reduce I/O.

**Low High:** Alternate between spilling the smallest and the largest partition.

**Record Size Ratio:** Choose a partition that holds the smallest number of records among partitions whose size is equal to or exceeds 80% of the largest partition size (low ratio of the number of records to the partition size); this expedites record processing by storing more records in the memory.

### 3.7.1 Victim Selection Policy Experiments

This section studies the impacts of join attribute value skew and record size variation on the victim selection policies.

#### Impact of Join skew

In our first experiment, we study the impact of join attribute value skew on the 13 different victim selection algorithms. In Figure 3.11-a, both the build and probe datasets use the All Small Record configuration, and the join attribute values are unique integers (Non Skewed join attribute value case).

In Figure 3.11-b, the join attribute values of the build dataset are integers drawn from a Normal Distribution to make them skewed, while the probe dataset uses unique integers as its join attribute (Skewed join attribute value case). Both relations are 1GB in size and contain 985,000 records. The authors of [65, 15] used a Normal Distribution in which 99% of the join attribute values are coming from 5% of the possible values, justifying this as similar to the skew found in real-world data. To achieve this data skew, we use a Normal Distribution on an integer attribute with the mean of 492500 (equal to half of the cardinality), a standard deviation of 8208, and a range of possible values varying from 1 to the dataset cardinality.

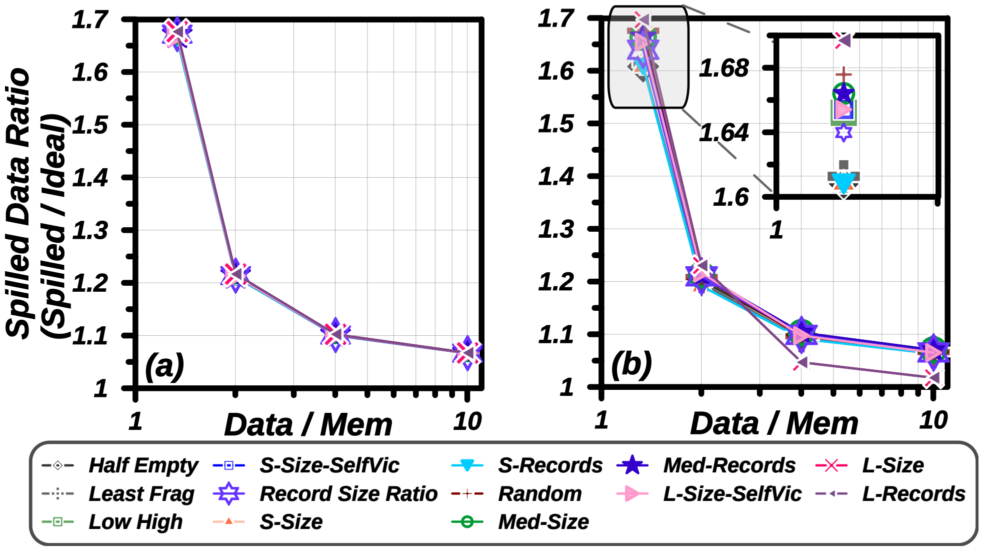


Figure 3.11: Impact of Join Attribute Value Skew in Victim Selection Policies. (a) - No skew. (b) - Skewed.

The metric used in Figure 3.11 is the ratio of the amount of spilled data over the ideal amount of spilling. The ideal amount of spilling is the minimum amount of data that must be spilled to disk during the build phase. We determine this ideal amount by using a simple simulation program. This simulator minimizes the data spilling by maximizing the memory used in each round of HHJ by the in-memory partition. Eq. 3.2 with accurate a priori information and with a fudge factor of 1.4 is used in this simulator to ensure that the amount of spilling is minimal.

As Figure 3.11-a shows, all of the algorithms have a similar performance if records are similar in size and the join attribute values are uniformly distributed. Figure 3.11-b shows that skew in the join attribute values can cause different spilling behavior for some victim selection policies. In Figure 3.11-b, the Largest-Size and Largest-Record policies overspill when data is slightly larger than the available memory. However, as the data size increases, spilling the larger partitions releases more frames, saving other partitions from spilling.

The Smallest-Size and Smallest-Record policies, which spill less data initially, will spill more when the ratio of data to memory is higher. All other policies show a spilling behavior that lies between these two categories of policies. However, the overall difference between most of the policies is almost insignificant.

### Impact of Variable-Sized Records

Next, we study the impact of variable-sized records on the performance of the victim selection policies. We used a set of 1GB relations based on the 1-Large Record Coexist and 3-Large Record Coexist dataset configurations.

As Figures 3.12 and 3.13 show, most of the policies perform similarly as the ratio of data over memory is increased in both experiments. The Largest-Size and Largest-Record policies spill less data and fewer partitions to disk than the other victim selection policies in most of the data points. This is because the number of frames that larger partitions free can save more partitions from spilling.

In both Figures 3.12 and 3.13, increasing the population of large records leads to a larger differences between victim selection policies. The variations in the size of the records and the high impact of large records on the partitions' sizes, compared to fixed sized records in

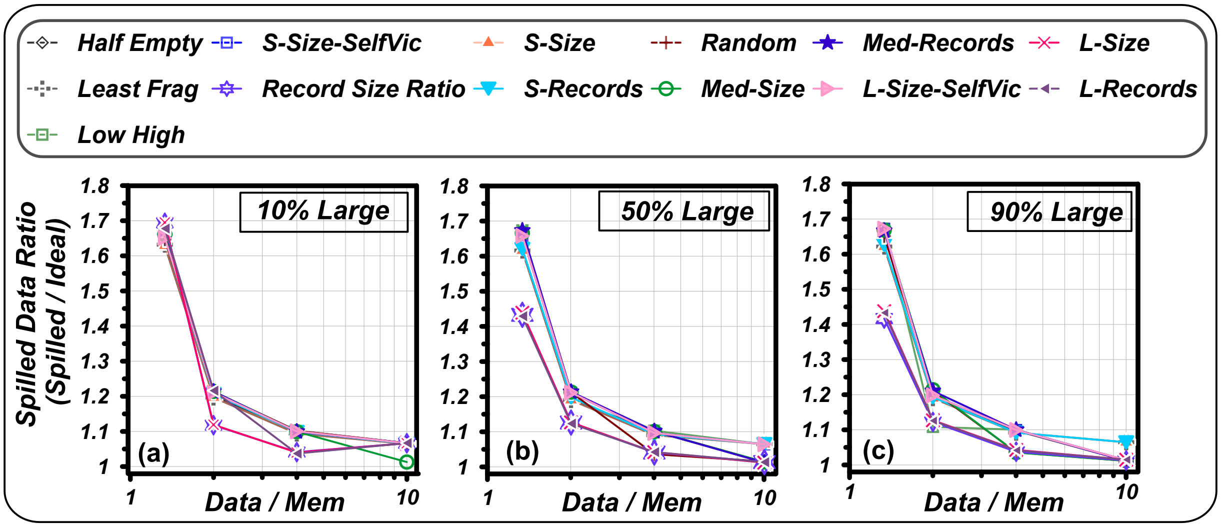


Figure 3.12: Impact of Variable Record Size (1-Large Record Coexist) in Victim Selection Policies. (a,b,c) - Spilled Data Ratio when 10%, 50%, and 90% of the records are large, respectively.

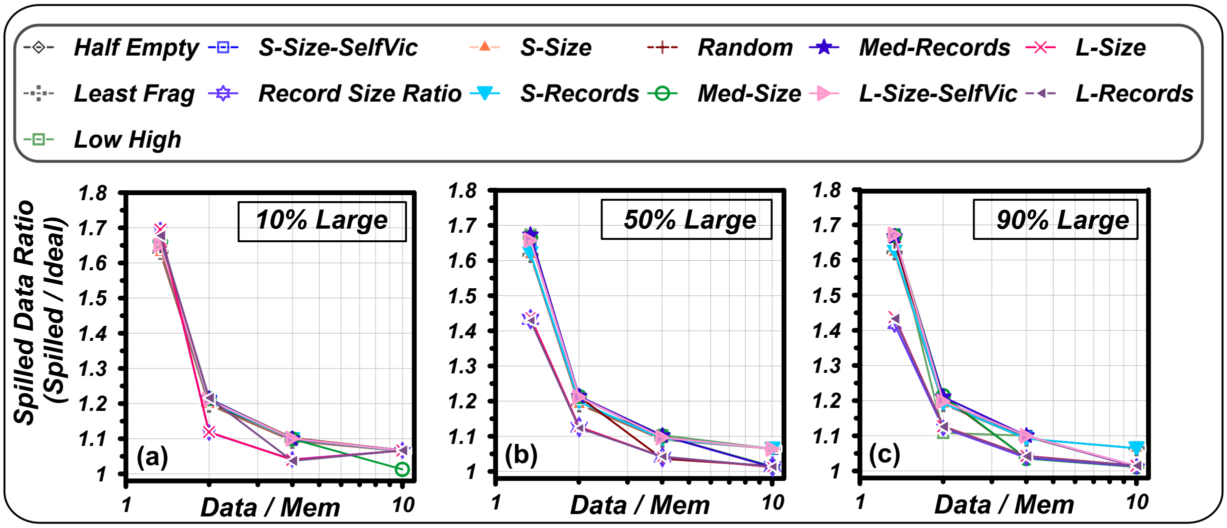


Figure 3.13: Impact of Variable Record Size (3-Large Records Coexist) in Victim Selection Policies. (a,b,c) - Spilled Data Ratio when 10%, 50%, and 90% of the records are large, respectively.

Figure 3.11-a, make it possible to see differences between these victim selection policies. In both 1-Large Record Coexist and 3-Large Record Coexist cases (Figures 3.12 and 3.13), the Largest-Size, Largest-Records, and in some cases Largest-Size-Smallest-Record policies spill the least amount of data and the fewest number of partitions in most of the data points by spilling the largest partitions first. This difference between policies in the 3-Large Record

Coexist experiment is less obvious since the large records are 1/3 of the size of the large records in 1-Large Record Coexist dataset. In Figure 3.12-a most policies perform similarly as there are fewer large records thus, less opportunity for these policies to perform differently.

### Impact of Join Skew & Variable-Sized Records

In this experiment, we study the impact of the combination of join attribute value skew and variable-sized records on the proposed victim selection policies. The Normal distribution discussed in Section 3.7.1 is used for making the build dataset skewed. The record sizes are chosen from the same distribution used for 1-Large Record Coexist (Figure 3.14) and 3-Large Record Coexist (Figure 3.15) cases. The probe inputs have the same cardinality and record size distribution as the build input, while their join attributes are unique integers.

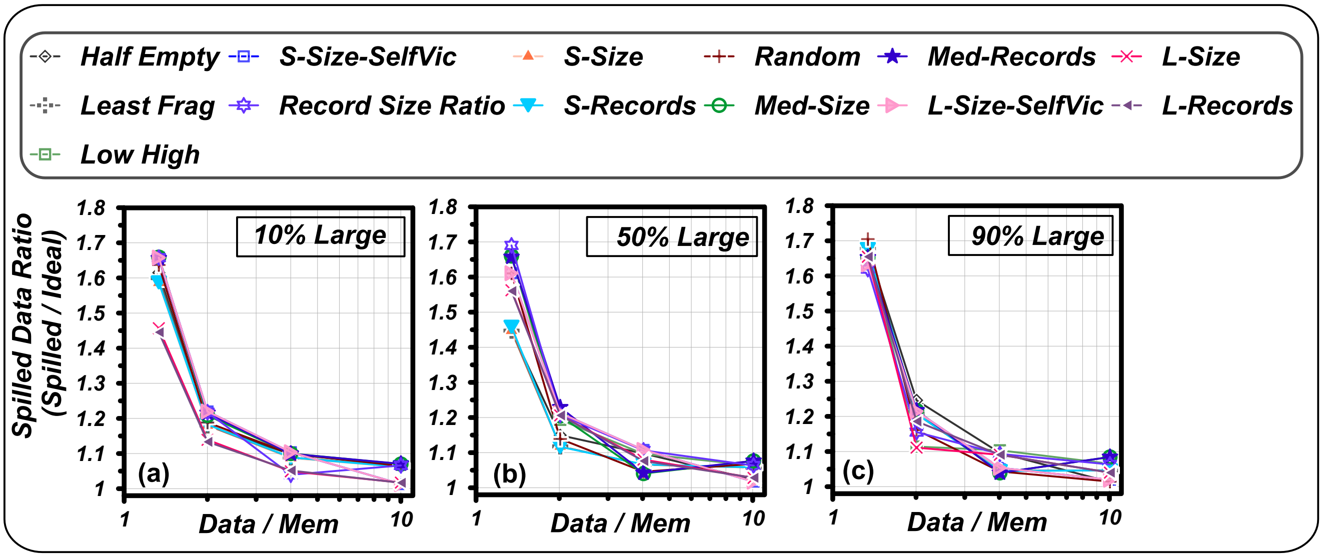


Figure 3.14: Impact of Skew & Variable Record Sizes (1-Large Record Coexist) in Victim Selection Policies.

Similar to the previous experiment, Largest-Size and Largest-Record are two well-performing policies when larger records have a lower population. The Median Size and Median Records policies perform well by taking a middle route if data is skewed and most of the records are large. The skew in data makes some partitions get more records; partitions with more

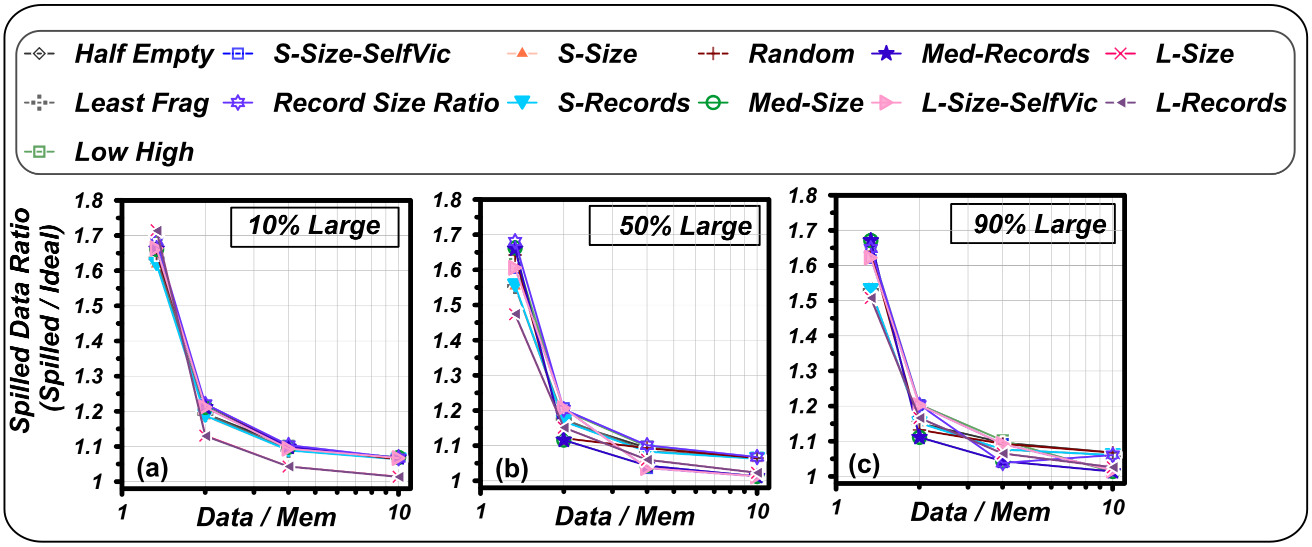


Figure 3.15: Impact of Skew & Variable Record Sizes (3-Large Records Coexist) in Victim Selection Policies.

records will have larger sizes if records are mostly large-sized, and thus the Largest-Size and Largest-Record algorithms can overspill. In the case of very limited memory for the 1-Large Record Coexist case (the first data point in Figure 3.14-a, 3.14-b, and 3.14-c), Smallest-Records and Smallest-Size are two of the best performing policies. Since most of the data is located in a few partitions, there are many small partitions with only a few frames. As such, Smallest-Records and Smallest-Size can avoid overspilling by spilling these small partitions when data is just slightly larger than memory.

In the 3-Large Records Coexist case, the victim selection policies' performance is similar to the 1-Large Record Coexist case with the difference that algorithms such as Median Records also perform well in this case due to the smaller sizes of large records. Largest-Size and Largest-Records tend to write larger numbers of frames sequentially, while others such as Smallest-Size and Smallest-Records write a smaller number of frames in a more random manner. As our experiments for G-S and NG-NS showed, this difference in their I/O patterns may not impact performance as much as otherwise expected if filesystem caching is enabled.

### 3.7.2 Results for Victim Selection Policy

Based on our experiments in the previous subsection, the Largest-Size and Largest-Record policies result in less I/O in most cases than the other alternative policies. Our results confirm the conjecture of [59, 33] that the Largest-Size policy (as well as the Largest-Record policy, based on our results) is a good selection policy for the following two reasons: 1. Larger partitions release many frames; thus, they save other partitions from spilling to disk. 2. Writing larger partitions leads to more sequential and less random writes.

However, our results also show that the difference in the amount of spilled data makes only a slight difference in performance. The gained benefits for having a more sequential pattern by spilling larger partitions are diminished if filesystem caching is enabled.

## 3.8 Single Core vs. Multi-Core

So far all of our experiments have used one thread for executing join queries in one data partition. In this section, we study the impact of the number of partitions and threads (and hence the cores) on the performance of Dynamic HHJ for various storage types.

For these experiments, both the build and probe datasets contain 1GB of data following the All Small Records dataset design. We used one Node Controller with 1,2, and 4 data partitions to utilize 1,2, and 4 CPU cores respectively. All joins use Append(8) as the partition insertion algorithm and NG-NS and Largest Size as their Growth and Victim Selection policies, respectively. Figure 3.16 shows the execution time of a single join executed on HDD, SSD, and AWS EBS. As this figure shows, increasing the number of cores (threads) causes disk contention and degrades the performance for HDD. On the other hand, SSD and AWS EBS benefit from more worker threads due to the efficiency of SSD storage device in handling random I/Os. The overall execution time of AWS EBS is higher than that of SSD

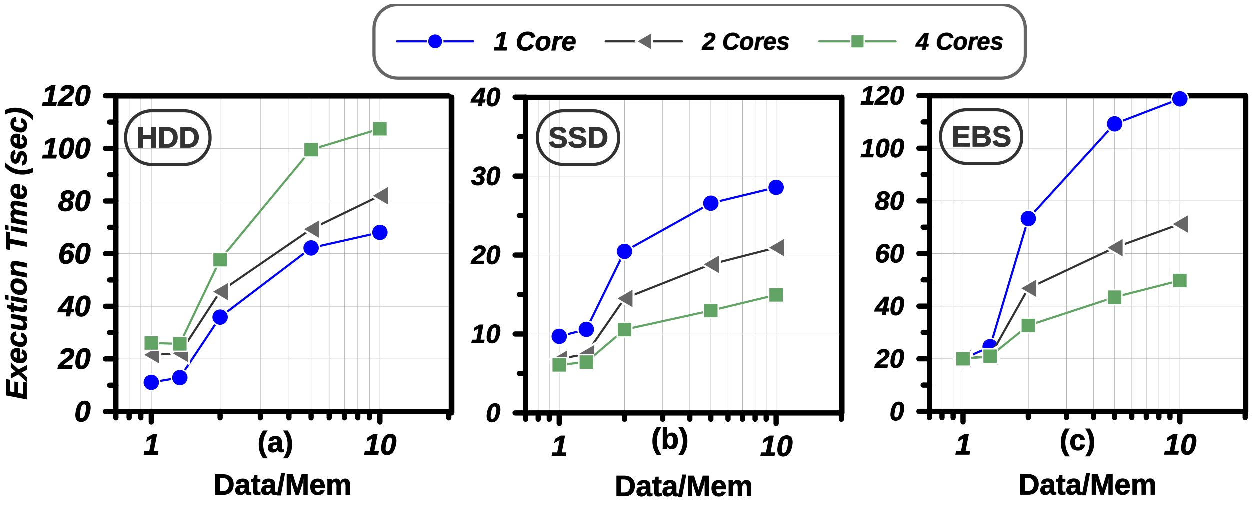


Figure 3.16: Impact of number of cores on the performance of Dynamic Hybrid Hash Join (a)- HDD, (b) - SSD, (c) - EBS

due to the network latency associated with over-the-network storage.

## 3.9 Conclusion and Future Directions

Our experimental study has investigated different policies to design a robust Dynamic HHJ operator when no accurate a priori information about the input datasets is available.

Although previous studies have suggested an upper bound for the number of partitions, no lower bound for this parameter has been proposed to the best of our knowledge. Not having a reasonable lower bound can lead to having too few partitions, causing detrimental overspilling. Based on a simulation study, we recommend using 20 as a minimum number of partitions so that each spilled partition writes only 5% or less of the build input to disk.

Furthermore, we have explored different partition insertion algorithms for incoming records to find a frame with enough space among a partition's in-memory frames. Append(8) showed the best performance among the partition insertion algorithms.

Next, we considered two potential post-spilling growth policies for spilled partitions, Grow-Steal and No Grow-No Steal. Our cost model showed that Grow-Steal should perform better than No Grow-No Steal due to doing more sequential I/O. However, our experiments showed that a modest file system cache can mitigate this difference by turning most random I/Os into sequential ones.

Additionally, we designed and evaluated 13 different victim selection policies. Our results confirmed the conjecture in previous work that the Largest Size policy is one of the best policies in most cases. However, this difference was not large enough to significantly impact overall system performance.

As a future direction, we would like to compare the performance of Dynamic HHJ with the radix join algorithm suggested in [13].

# Chapter 4

# Memory Management for Multi-Join Queries

After studying the performance of a single Dynamic Hybrid Hash Join in Chapter 3, we now investigate various memory management and scheduling techniques for executing multi-join queries under different memory availability for efficient execution. Specifically, we study the impact of different memory distributions for join operators, intra-query parallelism, and execution parallelism on different classes of multi-join query plans under the different assumptions of memory availability and storage devices such as HDD, SSD, EBS, and EBS-Hybrid (an architecture made of EBS and SSD storage types). We consider the Left Deep Tree, Bushy Tree, and variations of the Right Deep Tree, including Parallel Right Deep Tree, Sequential Right Deep Tree, Static Right Deep Tree, and Sequential Static Right Deep Tree in this study. We believe this provides the foundation for understanding basic "join physics".

## 4.1 Introduction

Processing and executing multi-join queries, one of the most important and common queries in a database management system, can be done in many ways, each with different resource requirements. Although the processing and performance evaluation of multi-join queries has been the topic of research for the past decades, the problem's complexity and multidimensional nature make it a poorly understood problem for the database community.

For a scientific approach, it is essential to reproduce the results of prior studies before proceeding with more complex cases. Accordingly, we decided first to re-evaluate the results of a key study done by Schneider & Dewitt [66] in 1990 in which they studied the performance of multi-join queries in shared-nothing clusters. They used the Hybrid Hash Join operator as their join operator and used a simulator made for the Gamma database system on HDD.

In this chapter, we study the performance of various query plan shapes for a multi-join query, as well as the impact of different memory allocation and intra-query parallelism techniques on their performance. In addition to Left Deep Tree (LDT), Right Deep Tree (RDT), and Static Right Deep Tree (Static-RDT), we considered Sequential Right Deep Tree (Sequential-RDT), Sequential Static Right Deep Tree (Sequential-Static-RDT), and an example of Bushy Tree in this study. Several decades of advancement in hardware requires the re-examination and verification of previous work results, which were largely based on simulators. Thus, we re-evaluated the results of [66] using Apache AsterixDB utilizing HDD, SSD, AWS EBS, and EBS-Hybrid. We believe this provides an important foundation for understanding basic "join physics".

## 4.2 Introduction to Activity Clusters and Stages

Each query tree consists of a set of operators and data flow connectors. Thus, a query can be represented by an Activity Dependency Graph where operators are the graph nodes and data flow edges are graph edges [17]. Figure 4.1 shows two activity cluster graphs. Each operator constitutes one or more phases or activities. As mentioned earlier in Chapter 3, HHJ and DHHJ are two-phase join operators with a blocking dependency between their build and probe activities. This blocking dependency provides an ordering between the execution of a join's build and probe activities by preventing the probe activity from starting before the build activity has been finished.

Each group of activities is connected using data flow edges with no blocking dependency constructs a pipeline or an activity cluster. Thus, activities of an activity cluster can be co-scheduled together, and data can flow between them page by page through the data flow connectors. In Figure 4.1 each dashed area represents an activity cluster, arrows represent the blocking dependency between two activities, and solid black lines represent data flow connectors. As this figure shows, each operator with a blocking dependency between its operators introduces a new activity cluster since its activities cannot be co-scheduled.

As Figure 4.1 indicates, the blocking dependencies between operators' activities may lead to a partial or total ordering for the execution of activity clusters of a query. Figure 4.1-a represents an example of a query tree with total ordering in execution, as each activity cluster apart from activity cluster $A$ is blocked by another activity cluster. Activity cluster $X$ is blocked by activity cluster $Y$ if at least one of $X$ 's activities is blocked by an activity from $Y$. In Figure 4.1-a, activity cluster $A$ can start its execution immediately since it is not dependent on any other activity clusters. Activity cluster $B$ may start its execution as soon as the execution of activity cluster $A$ is finished. Activity cluster $C$ starts after activity cluster $B$ is finished. Similarly, activity cluster $D$ can start its execution as soon as activity cluster $C$ is finished.

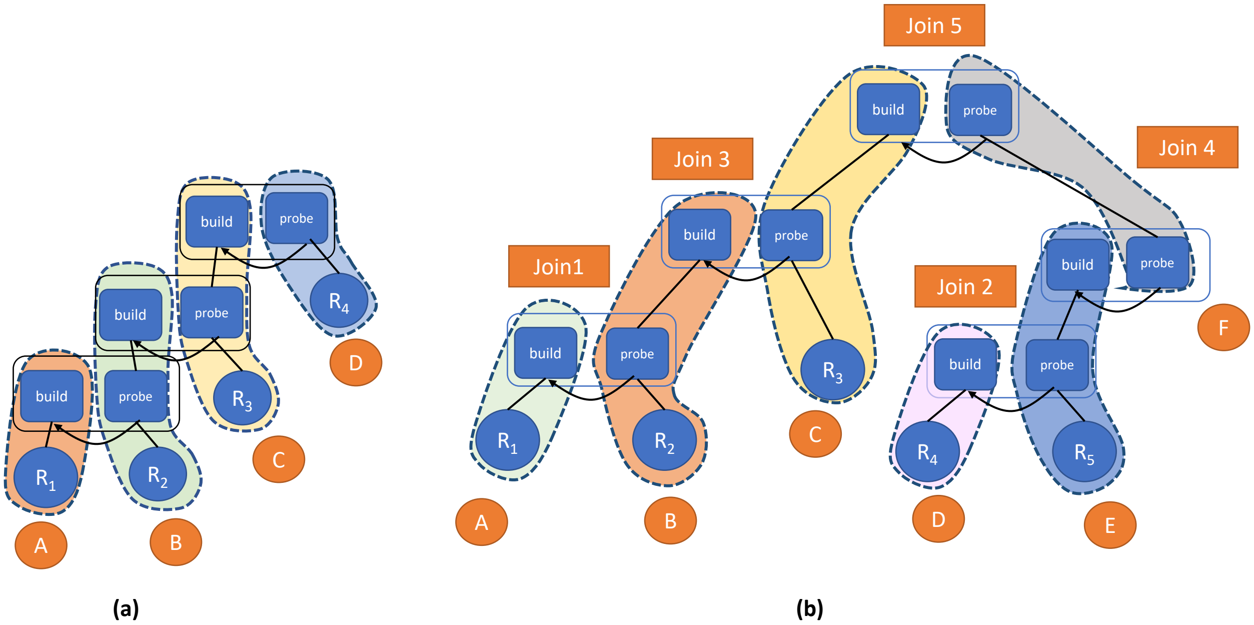


Figure 4.1: Activity Graph Dependency (a) Total Ordering (b) Partial Ordering

Figure 4.1-b shows an example of a partial ordering in executing a Bushy query tree. In this figure, activity clusters $A$ and $D$ can start their execution simultaneously since they are not dependent on any other activity cluster. Once activity cluster $A$ has been finished, activity cluster $B$ can start its execution. Activity cluster $E$ can start its execution as soon as activity cluster $D$ is over. After the execution of activity cluster $B$, activity cluster $C$ can start its execution. Finally, activity cluster $F$ can start its execution after activity clusters of $E$ and $C$ are done. In this example, some execution ordering exists; however, it does not define a total ordering for executing the activity clusters independent of one another. For example, in Figure 4.1-b, the execution of activity clusters $B$ and $E$ may or may not overlap at runtime.

A DBMS may introduce a *control dependency* between activities to enforce a total ordering for a query tree with more than one independent activity cluster. Control dependencies can be introduced to make the execution order of different activity clusters deterministic. Having a deterministic execution order makes the task of resource usage estimation and its management as a function of time more straightforward. A query will be executed stage by stage, where a stage is a set of activity clusters with no direct or indirect blocking dependency between them; thus, they can be co-scheduled. The execution order for a query's stages is defined based on the blocking dependencies between their activity clusters. A stage can only start execution after all the activity clusters in the previous stage are entirely over.

In Figure 4.1-a, each activity cluster is also a stage, as each of them is dependent on another activity. However, in 4.1-b, activity clusters $A$ and $D$ make one single stage. This stage can start its execution since it is not dependent on any other stage. By introducing control dependencies, the next stage, which consists of activity clusters $B$ and $E$, can start its execution as soon as all the activity clusters of the previous stage are finished. The next stage only contains the $C$ activity cluster, and the final stage would contain the activity cluster $F$.

Since only one stage from each query will be active at a time, the memory given to a query should be divided between the operators involved in its active stage only.

## 4.3 Design Space

In this work, we study the performance of multi-join queries in a four-dimensional design space. The first design dimension that we consider is the query plan shape which includes variations of Left Deep Trees, Right Deep Trees, and Bushy Trees. As the next design dimension, we consider various memory management techniques for distributing memory between the join operators of a query including equal and bottom-up memory management techniques. We consider the execution strategy as the third dimension of our design space. We investigate various execution strategies to achieve different degrees of parallelism in execution of the Right Deep Trees. As the last design dimension, we consider four different storage alternatives including HDD, SSD, AWS EBS, and AWS EBS-Hybrid and evaluate the performance of multi-join queries executing with different values for the first three dimensions on these storage choices.

In the next few sections we explain the details of these design dimensions and their possible variations in depth.

## 4.4 Query Shapes

As discussed earlier, a multi-join query can be processed and executed in different shapes and formats. There are three main classes of query shapes, including Left-Deep Trees, Right-Deep Trees, and Bushy Trees. This section reviews these three query shape classes and their general features and characteristics.

### 4.4.1 Left-Deep Trees

A Left-Deep Tree (LDT) has a left-oriented shape shown in Figure 4.2. Each enclosed dashed area represents an activity cluster and stage in this figure. As this figure shows, in LDT, the output of the probe phase from the $i^{th}$ join is the input to the build phase of the $(i+1)^{st}$ join. Since each probe phase is blocked by its corresponding build phase, at most two joins are active at each time during the execution of an LDT. Thus, LDT is a sequential query plan with the available memory being shared between, at most, two consecutive joins simultaneously. The order in which these activity clusters execute is defined based on the intra-operator control dependencies shown in Figure 4.2.

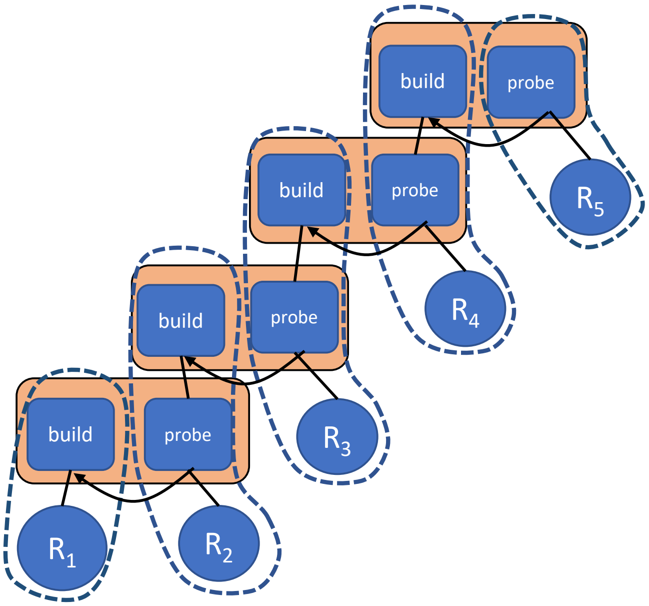


Figure 4.2: An Example of Left Deep Tree

### 4.4.2 Right-Deep Tree

Right Deep Tree (RDT) has the highest parallelism compared to other query plan types. As Figure 4.3 shows, the output of the $i^{th}$ join will be input to the probe phase of the *(i+1)*$^{st}$ join. In this type of query plan shape, the build phases of all joins can execute concurrently;

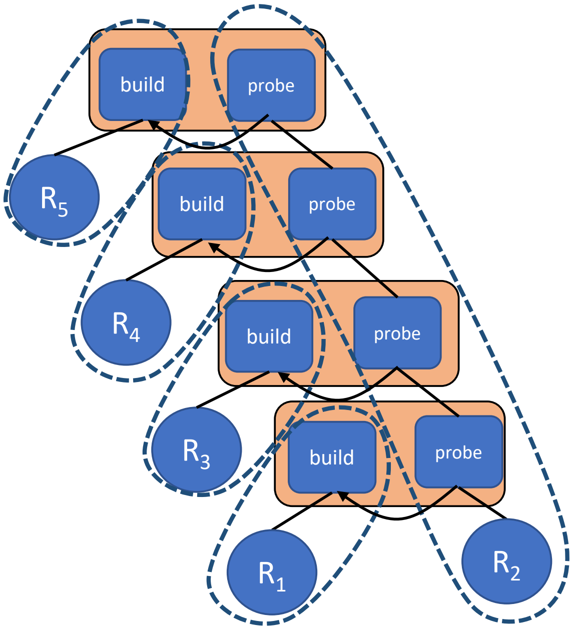


Figure 4.3: An Example of Right Deep Tree

as a result, the given memory will be divided and shared between all of the joins. If the query includes $n$ joins, then $n$ hash tables will be created in the memory, one hash table per join. Once all the build activities have finished processing their input data, their probe phase starts simultaneously, and the matched records will flow frame by frame through a pipeline from one join to another. In Figure 4.3 $R_2$'s contents will drive the probe pipeline.

### 4.4.3 Bushy Tree

Bushy Trees have always attracted researchers' attention due to their vast spectrum of shapes and their unique properties. Bushy Trees are a hybrid version of the two strict query shapes of LDT and RDT; thus, they inherit some of the benefits and drawbacks of their parent tree shapes. For example, some of the joins in a Bushy Tree may run in parallel while others execute sequentially depending on how the query plan is shaped. Bushy Trees, as opposed to strict query shapes such as LDT and RDT, may have one or more joins in which both of the inputs are non-base datasets. Bushy Trees have their specific benefits and drawbacks as well. For example, Bushy Trees can benefit from *Independent Parallelism*, as some subtrees can execute concurrently due to no blocking dependencies. While Independent Parallelism provides more freedom in execution for Bushy Trees compared to the stricter query plan

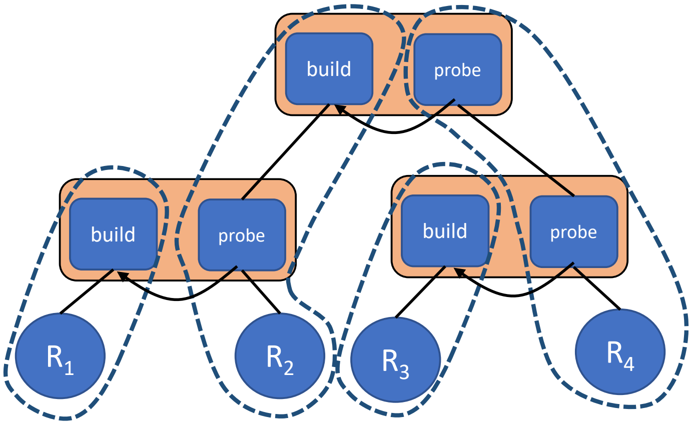


Figure 4.4: An Example of Bushy Tree

shapes, it causes Bushy Tree not to have a total order in the execution of its subtrees. Thus, scheduling and estimating the maximum resource requirement of a Bushy Trees can become a challenge due to its independent subtrees. Figure 4.4 shows an example of a Bushy Tree. Since the variations of Bushy Trees can be numerous, we use the techniques suggested in [51] for generating sample Bushy Trees in our experiments.

## 4.5 Memory Management

Memory is one of the most influential factors in choosing the type of query plan for executing a multi-join query. One of the main questions that a DBMS needs to answer is: considering the available memory, what query plan should be used for a specific multi-join query, and how should the memory should be distributed between its operators to reduce its execution time?

In a Left Deep Tree, memory is always distributed between two adjacent joins in the query plan. Memory distribution between the join operators of Bushy Trees needs to consider the joins whose executions may overlap. For an accurate and proper memory distribution for a Bushy Tree, one solution is to control and order the execution of independent activity clusters. A more conservative but simpler approach is to divide the memory equally between all the join operators. We chose the latter solution for its simplicity. Next, we introduce equal and bottom-up memory distribution strategies for Right Deep Tree query shapes.

### 4.5.1 Equal Memory Distribution

In the Equal Memory Distribution strategy for an RDT, each join in the query will get an equal share of memory as the other joins in the query. If some joins require less memory than their equal share, other joins in the query can use this leftover memory in case of need. A DBMS with proper knowledge of the build input sizes and join selectivities can statically assign the memory to each join operator to enable this memory sharing between join operators. As mentioned earlier, the current version of AsterixDB does not collect and use any information or statistics about the datasets and their attributes. However, a user can add hints to their query to provide AsterixDB with information to be used during the query's execution time. For example, a user can use a skip-secondary-index hint to ask AsterixDB not to use a given secondary index in that query. In this chapter, we use Dynamic HHJ as the join operator, and as we saw in Chapter 3, HHJ's performance is highly dependent on having the accurate statistics for its inputs. Thus, we augmented AsterixDB with a new size hint to use to estimate the build input size. AsterixDB will soon replace this size hint mechanism with a new statistics component and cost-based optimizer currently under implementation.

### 4.5.2 Bottom-Up Memory Distribution

In the bottom-up Memory Distribution strategy, the DBMS distributes the join memory starting from the operators at the bottom of the query plan and assigns each join operator its required memory (the memory needed to avoid spilling). The authors of [23] refer to a RDT with bottom-up memory distribution as "Static-RDT", we will use the same terminology here as well.

By assigning each join its required memory, the bottom-up strategy assures that no data will spill within each join at execution time, assuming that sufficient memory is available to hold the largest join in memory. At any time where the remaining available memory is not enough to hold the next join operator in memory, we "break" the query plan by materializing the output of the last fitting join and using the materialized intermediate results later as the probe input to the next join. Figure 4.5 shows an example of a Static-RDT. In this query plan, the first two builds at the bottom will start their execution simultaneously. The probe

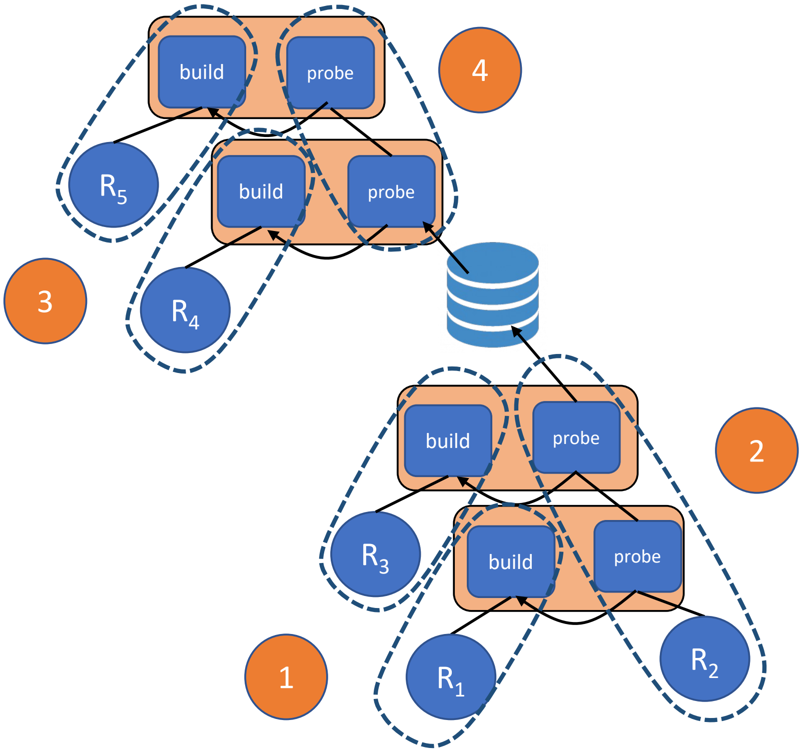


Figure 4.5: An Example of Static Right Deep Tree

phase using $R_2$ dataset will start its execution as soon as these first two build phases are over. The results of this probe phase will be materialized to be used as the probe input to the next section of the tree. The build phases of the joins with inputs $R_4$ and $R_5$ starts after the probe phase from the previous section of the tree is over. After finishing the build phases of the second section of the tree, their probing phase will start by reading the materialized results from the previous section of the tree as input.

## 4.6 Execution Scheduling

In addition to the shape of the query plan and the amount of available memory, the execution pattern and schedule of a query's activity clusters can significantly impact the system's and the query's performance. The execution pattern becomes especially important for cases where the underlying storage system is HDD. In HDDs, having many parallel disk I/Os can challenge the mechanical disk arm and dramatically impact the overall system's performance [45, 53, 64]. This section introduces some possible execution scheduling strategies for Left Deep, Right Deep, and Bushy Trees.

### 4.6.1 Execution Scheduling for LDT

In Left Deep Tree, there is a total order of execution between the phases of various joins in a query. LDT therefore follows the following pattern to execute a multi-join query sequentially:

1)$Build(j_1)$

2) Probe($\mathrm{j_1}$), Build($\mathrm{j_2}$)

3)$Probe(j_2), Build(j_3)$

4) Probe($\mathrm{j_3}$), Build($\mathrm{j_4}$)

.....

$n-1)Probe(j_{\text{n-1}}), Build(j_{\text{n}})$

n) Probe($\mathrm{j_n}$)

Due to the total ordering between various join activities and their sequential pattern of execution, there is only one scheduling strategy for LDT as shown above.

### 4.6.2 Execution Scheduling for RDT

In RDT, all build phases may execute concurrently as they are not blocked by other activities. Once all the build phases are over, the records of the probe dataset probe all the build inputs using a long pipeline. RDT has been attractive to many researchers and developers due to its unique features, including being the highest-parallel type of query plan, not materializing intermediate results, and allowing a more accurate estimate of the sizes of hash tables since all of the build inputs come from base datasets (as opposed to previous join results). As mentioned earlier, the build phases of all joins can execute fully in parallel. However, we can choose to define other execution schedules with different parallelism degrees since there is no partial or total ordering between these build activities. The degree of parallelism in the execution of query plans can significantly impact the execution time depending up on the storage device used and its capability to handle concurrent random vs. sequential I/Os. Next, we introduce various scheduling strategies for RDT query plans, each with a different degree of parallelism, and evaluate them in Section 4.8.2.

**Parallel Execution.** In the Parallel scheduling strategy, the build phases of an RDT query's joins start at the same time. Once all the build phases are finished, their probe phases execute using a long pipeline. This is the most-parallel way to execute a multi-join query.

**Semi-Sequential.** In the Semi-Sequential scheduling strategy, the build activities of an RDT query are grouped in a bottom-up fashion. Groups run sequentially, one after another, while the build activities within a group run in parallel. Note that this impact the query's I/O parallelism but not its memory requirements.

We introduced a new dependency named an "inter-operator dependency" to create blocking dependencies between build phases of different joins. Additionally, we developed a hint to specify the size of each group. Figure 4.6 shows an example of RDT query executing in Semi-Sequential manner. In this example, the build phases of $R_1$ and $R_3$ are part of the same group, which no other group blocks. Hence, they will be the first activities to execute in parallel. Next, the build phases of $R_4$ and $R_5$ can start their execution in parallel as soon as the previous group of build phases is finished. The long pipeline of probe activities can start its execution when both $R_4$ and $R_5$ have finished building their hash table.

**Sequential Execution** The Sequential scheduling strategy uses the inter-operator depen-

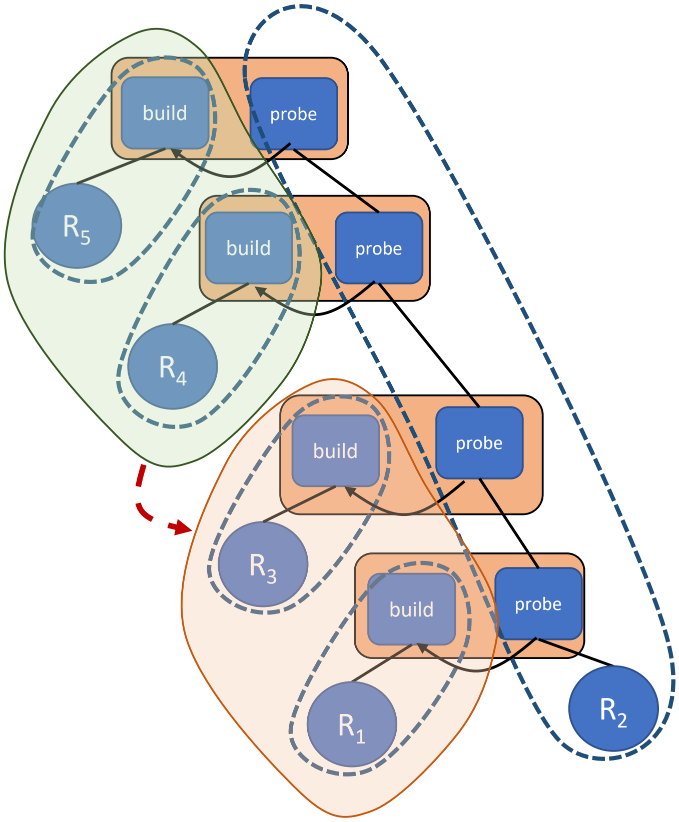


Figure 4.6: An Example of Semi-Sequential Execution in RDT

dency to put each build activity in its own separate group. Since each group has only one join and each group is blocked by the next join in the query, the build activities run sequentially from the bottom of the query tree to the top. Figure 4.7 shows an example of RDT executing its build phases sequentially. In this example, the build phases of the R$_1$, R$_3$, R$_4$, and R$_5$ datasets execute sequentially. The pipeline of the probe phase starts as soon as the last build phase is finished. We call this query plan shape a *Sequential-RDT* query plan. The memory usage of Sequential-RDT is again the same as the ones for the other RDT forms of the same query using the same join order. Sequential-RDT is considered the most disk arm-friendly version of RDT.

Sequential execution scheduling can also be used together with Static-RDT. We call this query plan shape the *Sequential-Static-RDT* query plan. In this query plan, the build phases within each segment of the Static-RDT are themselves executed sequentially. We use the

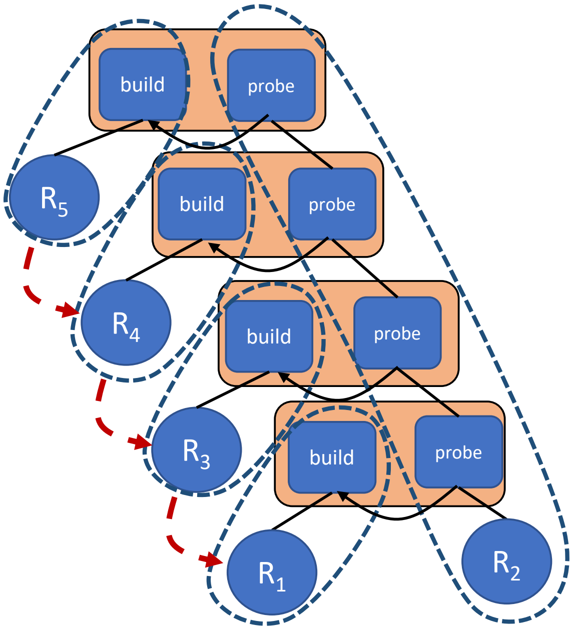


Figure 4.7: An Example of Sequential Execution in RDT

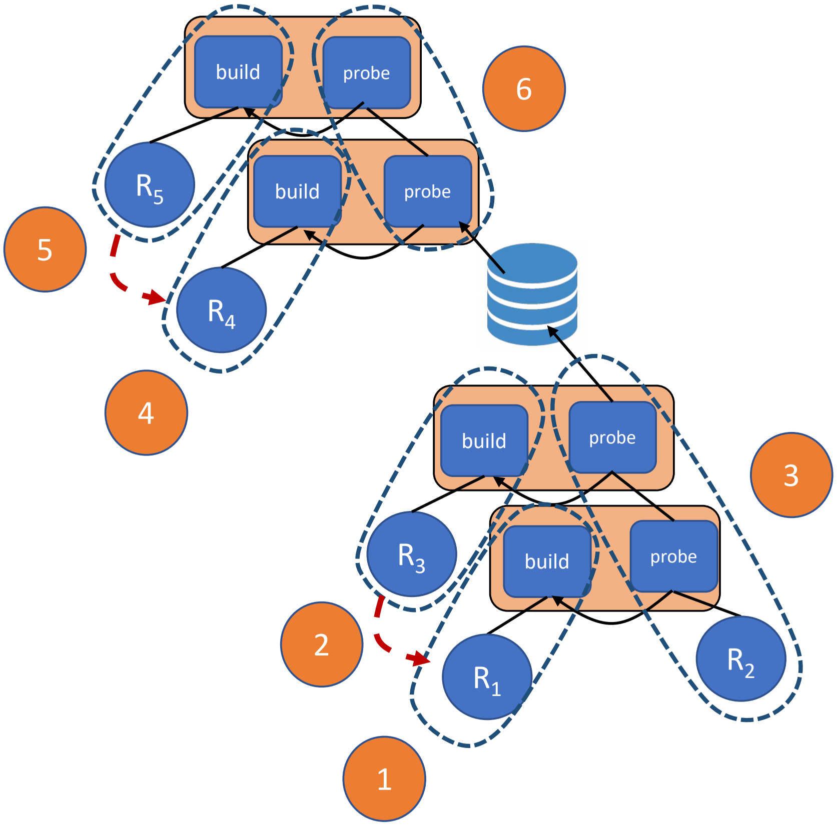


Figure 4.8: An Example of Sequential Execution in Static-RDT

inter-operator dependencies to enforce the sequential execution between the build phases of each segment.

### 4.6.3 Execution Scheduling for Bushy Trees

As mentioned earlier, Bushy Tree is a hybrid version of RDT and LDT; so in general, its scheduling order can be sequential in some parts and parallel in others. In Bushy Tree, some independent activity clusters may be co-scheduled as part of one stage, which increases the possibility for parallel execution for this query plan. Since Bushy Trees have many different shapes, they have no specific order of execution. Instead, blocking dependencies can be used to guide the scheduling order of their different activity clusters.

### 4.6.4 Storage Devices

In this chapter, in addition to exploring the memory and scheduling options, we explore various storage devices to study the performance of various query plans for a multi-join query under different yet most modern storage settings. In this subsection, we introduce these storage devices and node architectures. Figure 4.9 shows a simple design of these storage alternatives.

**HDD.** The Hard Disk Drive (HDD) is one of the oldest storage device types; it uses a mechanical disk arm to read/write data from/on the disk platters. Due to this mechanical arm, HDD is not very efficient in handling random I/Os. In this architecture, HDD storage is on the same system node as the CPU and memory, and all base relations and spilling data are stored on this storage device.

**SSD.** The Solid State Drive (SSD) is a newer storage device compared to HDD; it is made of non-volatile flash memory and is more efficient in managing random I/Os. The base relations

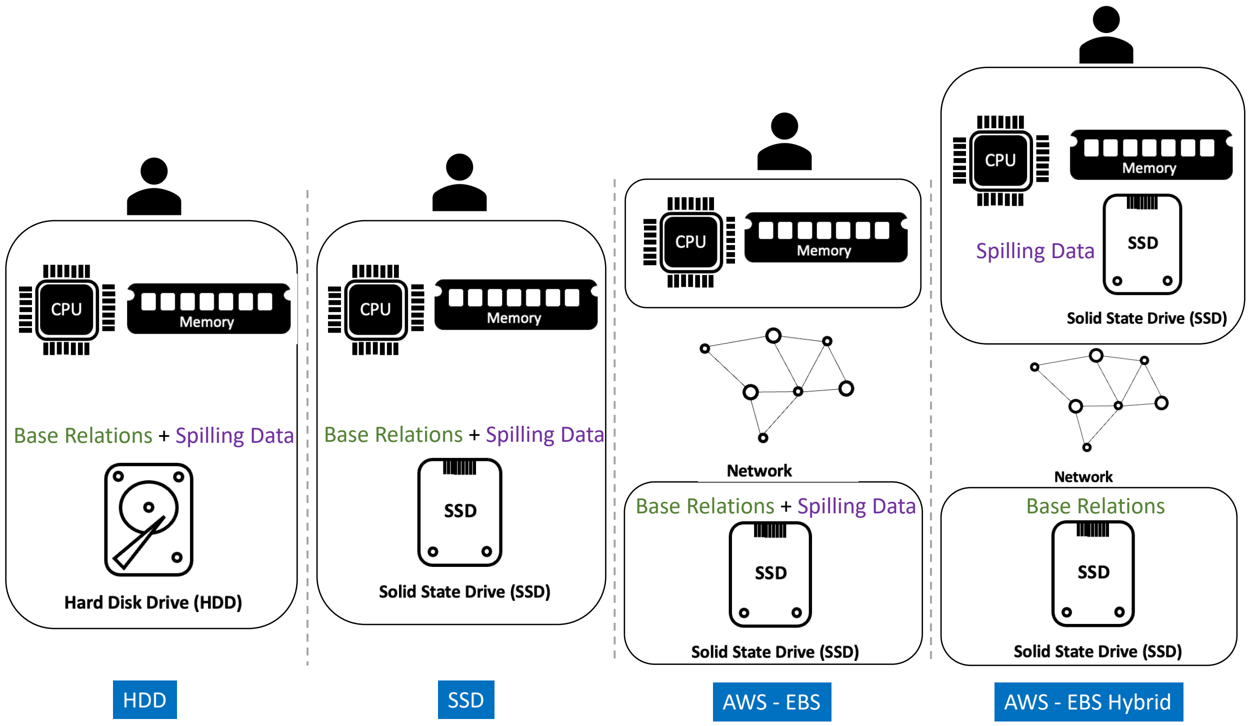


Figure 4.9: Storage Devices

and spilling data reside on the same SSD device in this architecture.

**EBS.** AWS Elastic Block Storage (AWS EBS) is used with the AWS EC2 cloud service to store persistent data over the network. The base datasets and spilling data reside on the same storage device on the EBS node. In our experiments, we used SSD as the storage device on the EBS node.

**EBS-Hybrid.** In this storage architecture, the base datasets are stored on SSD storage on an EBS node. However, the node (EC2 instance) that contains the CPU and memory also utilizes limited-volume local SSD storage for reading and writing the spilling data. (If a query does not spill any data to disk, EBS-Hybrid will act similarly to the basic EBS architecture.)

## 4.7 Related Work

Resource management and multi-join query scheduling have been one of the database community's research topics for over three decades. Schneider and Dewitt authored one of the key studies on tradeoffs between various query shapes for processing multi-join queries in 1990 [66]. They used a simulator of the Gamma parallel database system [24] to study the performance of Left Deep Tree (LDT) and Right Deep Tree (RDT) query plans in processing join queries made of four to ten joins. Additionally, they proposed various memory distribution and scheduling strategies for executing the RDT query plan, which led to new variations of RDT. One of those RDT variations was Static-RDT, a variation in which an RDT is divided up into one or more segments (sub-trees) in case of insufficient memory. They proposed both static and dynamic bottom-up memory assignment strategies for Static-RDT and compared their performance against equal-share memory assignment between all joins. This study showed that RDT is one of the best-performing query plan shapes due to its high parallelism if most of the build inputs remain in memory.

In another study [19], Philip Yu, et al. proposed a new variation of RDT called Segmented-RDT. A Segmented-RDT is a bushy tree made of smaller RDT subtrees. Only one subtree will be active at any time, and it can become the input to the build or probe phase to one of the joins of the next subtree in the plan. They proposed a heuristic to generate the subtrees flexibly and show the gained performance benefits through simulation studies. Through several simulation studies, this paper showed that Segmented-RDT can outperform other query plan shapes, including RDT.

ZigZag Tree, a right-oriented tree, was proposed in [72] as a competitor plan for Static-RDT. The execution of a ZigZag tree possibly leads to less I/O than the execution of a Static-RDT. Instead of dividing the query tree into segments and materializing the intermediate results to be used as the probe input to the next subtree, a ZigZag tree makes a "left-turn". It always uses an intermediate result as the build phase of the next join. Hence, it keeps the intermediate results in memory (if it fits) and avoids the materialization of intermediate result. The authors of [72] compare the performance of Static-RDT and ZigZag Tree in an experimental study on DBS3, showing the superiority of ZigZag Tree due to its needing less disk I/O.

In another study, the authors of [70] used PRISMA/DB, an in-memory database, to study different strategies for processor assignment and execution scheduling of multi-join queries on an 80-processor system. They considered LDT, left-oriented Bushy Trees, wide Bushy Trees, right-oriented Bushy Trees, and RDT as their candidate query plan shapes. They found that query plans with sequential executions are better choices for a system with limited number of processors and query plans with parallel executions are more suitable choices for systems with large number of processors.

## 4.8 Experimental Analysis

In this section, we compare the performance of the different query plan shapes under various memory availability, query complexities, and join and scan selectivities using HDD, SSD storage devices, AWS EBS, and AWS EBS-Hybrid.

### 4.8.1 Datasets and Benchmark

Similar to Chapter 3, we use the Wisconsin Benchmark's schema and the updated JSON data generator for this chapter as well. Since in this chapter our goal is to re-evaluate the results of experiments from [66], we have used the provided descriptions for the chosen experiments from their paper to make the structure of our experiments as close and comparable to theirs as we could. We used the same cardinalities and join and scan selectivities as in [66]. However, we used larger records to increase the dataset sizes since several decades have passed. We introduced this change in the dataset sizes to take into account the possible advancements in storage devices' capacity and efficiency since [66] was published.

For our experiments, each record is 1073B, and each memory frame is 32KB. The dataset sizes and join and scan selectivities are varied between different experiments; thus, we have included their details in each experiment's description.

Each query is running in isolation in this chapter and utilizes one CPU core per NC node. All node instances are chosen from the US-West-2 availability zone of AWS and have four vCPUs and 30.5GB of RAM. We use the d2.xlarge instance type for the HDD experiments and i3.xlarge and r4.xlarge for the SSD and EBS experiments, respectively. For EBS-Hybrid experiments, we use an i3.xlarge instance and attach EBS storage of type SSD to it for holding the base datasets.

### 4.8.2 Evaluating Semi-Sequential RDT Query Plans

We first evaluate and compare the performance of different Semi-Sequential RDT query plans along with fully parallel and sequential versions of RDT. For this experiment, we used an eight-join query and modified the amount of available memory that it was given.

Figure 4.10 shows the results of this experiment when executed on HDD, SSD, EBS, and EBS-Hybrid using Apache AsterixDB. The x-axis shows the ratio of available memory to the total memory needed for all eight joins to remain entirely in memory. The y-axis shows the execution time of the query. Sequential-RDT has the lowest execution time in the HDD case since its sequential execution favors the mechanical disk arm's nature. Conversely, increasing the parallelism in execution increases the execution times of query plans in HDD due to higher contention in disk arm movement, which causes the RDT, which executes fully in parallel, to

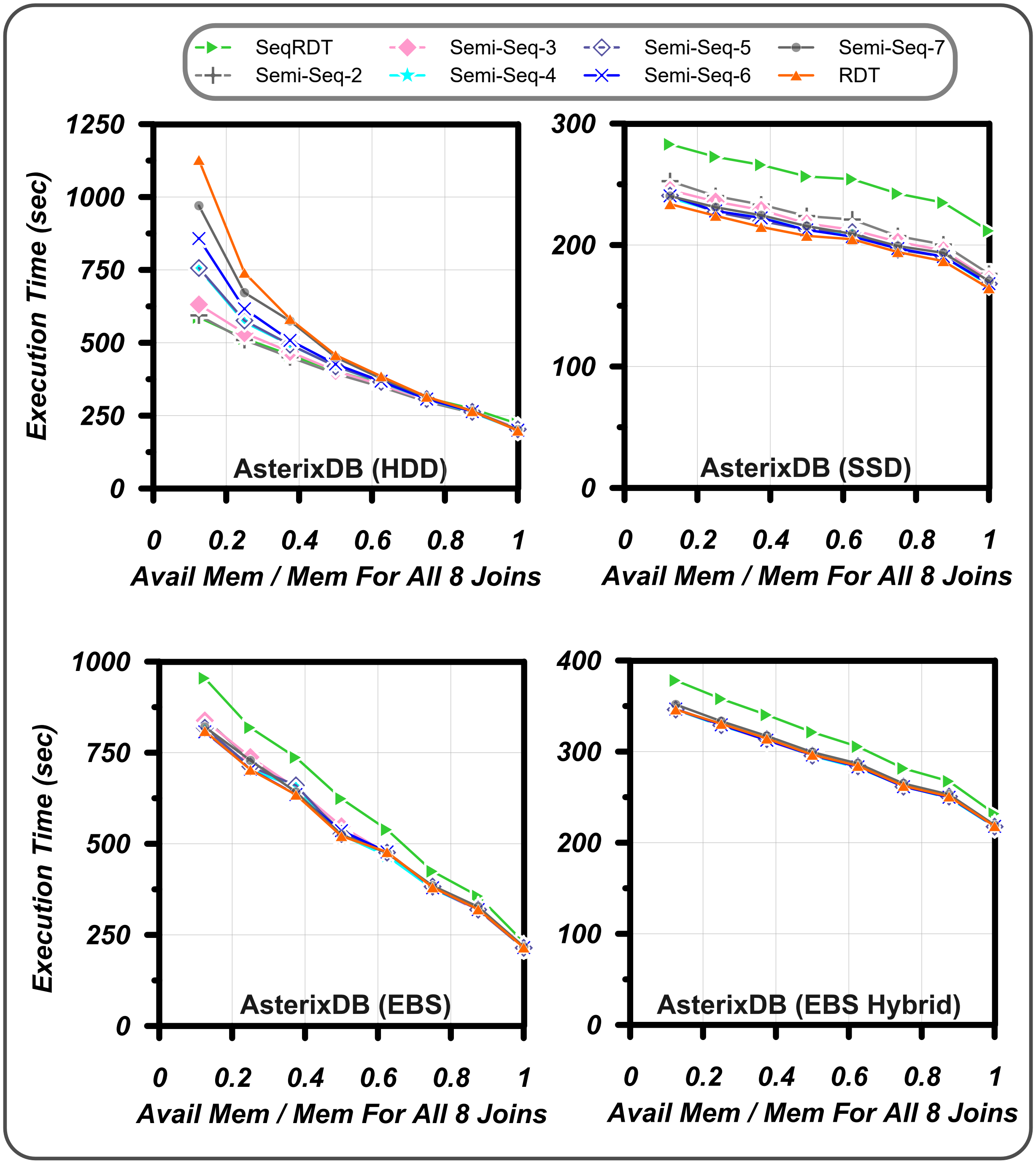


Figure 4.10: Evaluation of Sequential, Semi-Sequential, and Parallel Execution Variations of RDT Query Plan

have the highest execution time. On the other hand, when using SSD, EBS, or EBS-Hybrid architecture as the storage device, the fully parallel execution and fully sequential execution of RDT leads to the lowest and highest execution times, respectively. The efficiency of the

SSD storage used in these alternative architectures in managing random I/O is the reason for the better performance of parallel query plans. As Figure 4.10 shows, the performance of the various semi-sequential query plans is always between the fully-parallel and fully-sequential execution strategies of RDT; thus, we will use only these two execution strategies for RDT throughout the remainder of this chapter.

### 4.8.3 Experiment 1 - Unlimited Memory

In this subsection, we study the impact of query complexity on the execution time of a join query in the case where none of the joins spills to disk. We designed this study similar to the "Unlimited Memory" experiment of [66] and used their provided descriptions to reconstruct the queries.

In this experiment, we used 1 GB of data containing $1,000,000$ records for each build and probe dataset. Similar to [66], we designed the queries in a way that the output size of each join is also 1GB and contains 1,000,000 records. While making the size of intermediate results fixed is unrealistic, it simplifies the comparison between different query shapes and helps understand their main differences. For future work, we plan to consider more realistic and complex queries using more advanced benchmarks, including TPC-H and TPC-DS.

Figure 4.11-a shows the results of the Unlimited Memory experiment from the original Gamma simulator [66], and Figures 4.11-b, 4.11-c, and 4.11-d show the results of AsterixDB for the same experiment utilizing HDD, SSD, and AWS EBS storages, respectively. In addition to LDT and RDT plans which were considered in Gamma simulator experiment, we consider Sequential-RDT and Bushy Trees for AsterixDB.

The Gamma simulator's results were based on simulating HDD storage devices from the 1990s. As Figure 4.11-a shows, in the Gamma simulator, RDT generally had a lower exe-

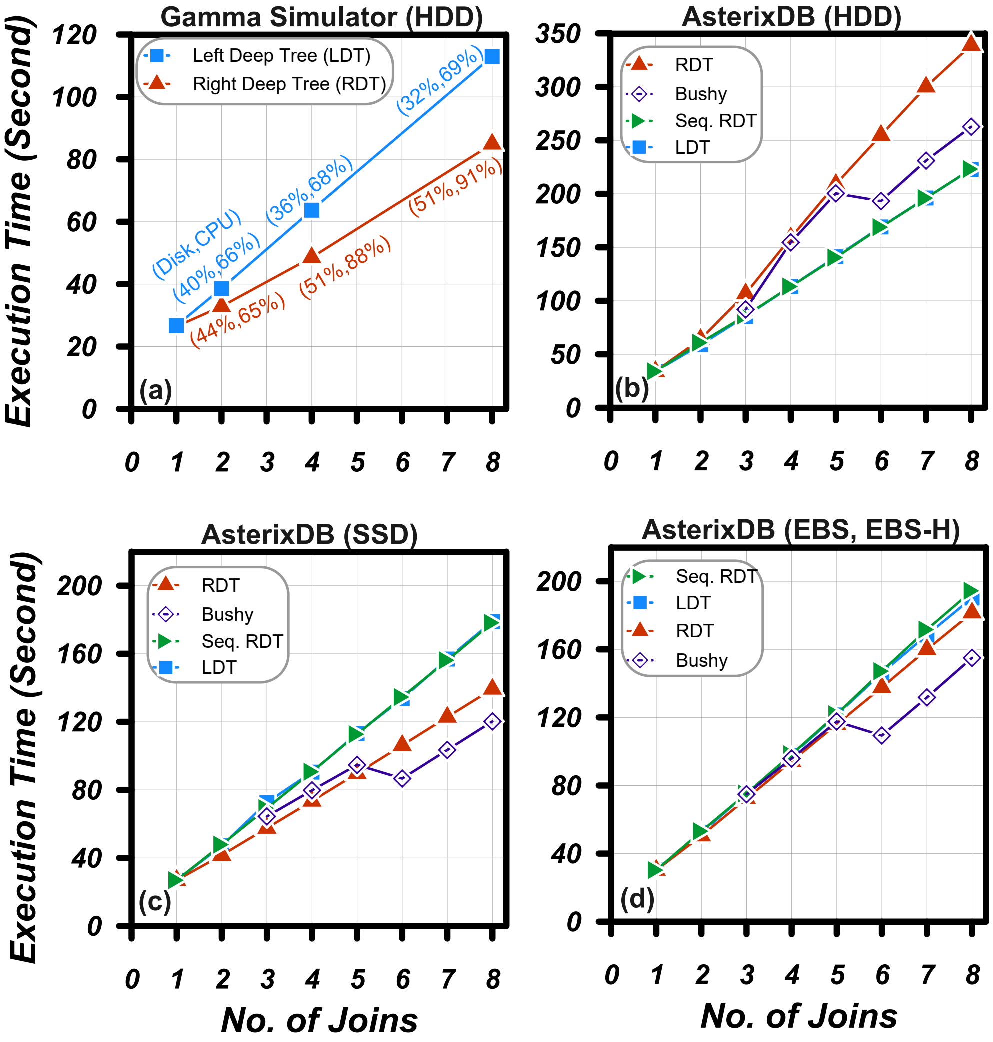


Figure 4.11: Experiment 1 - Unlimited Memory

cution time than LDT. In this figure, disk utilization in RDT was only slightly increasing, while its CPU became almost fully utilized as the number of joins in the query increases. However, we would have expected high disk utilization to be the bottleneck instead of the CPU since concurrently reading all the build datasets can cause high disk arm contention in HDD. From the device utilization and comparing the reported execution times of RDT with LDT and Sequential-RDT, we believe that the Gamma simulator was not properly simulating the disk arm movement and its impact on disk.

In AsterixDB, as Figure 4.11-b shows, increasing the query complexity leads to increasing

RDT's execution time when the storage device is HDD. Reading all the build datasets concurrently from HDD leads to high disk arm movement and contention in RDT. On the other hand, LDT and Sequential-RDT have lower execution times as their sequential execution patterns are less disruptive to HDD's disk arm movement. Bushy Tree is another parallel query plan with shorter pipelines than RDT. These shorter pipelines and the independent parallelism make some build and probe phases of different joins overlap. Bushy Trees take the middle path between RDT and LDT. The jump in the execution time of the Bushy Tree is due to the change of the query shape due to adding more join to the query. We are using the algorithm suggested in [51] for generating Bushy Trees, which keeps the pipelines' length to less than four joins.

This experiment's performance results become very different when SSD is the underlying storage device instead of HDD. As Figure 4.11-c shows, RDT now has a lower execution time than LDT and Sequential-RDT since it takes advantage of SSD's efficiency in handling parallel and random disk I/O due to the lack of a mechanical disk arm.

In the EBS storage device case, as Figure 4.11-d shows, RDT is slightly better than LDT and Sequential-RDT since the storage type is SSD and it can handle parallel and random I/O efficiently. However, the network latency between the disk and the computation node has masked some of the difference in the execution times of different query shapes.

In the case of utilizing SSD, AWS EBS, or AWS EBS-Hybrid as the underlying storage device, Bushy Tree takes advantage of its parallel execution pattern on arm-less storage devices. The independent parallelism and short pipelines improve the performance of Bushy Trees due to better CPU utilization and make it a better plan than RDT under these settings.

Stepping back, as Figure 4.11 shows, the results of the Gamma simulator for HDD are more similar to the results of AsterixDB for SSD. This clearly shows that the Gamma simulator did not properly model the disk arm movement; hence, its impact on the system's performance was not captured. Additionally, we can conclude that RDT's parallel I/O access pattern makes it a better choice for SSD under this setting. At the same time, we see that the more sequential query plans, including Sequential-RDT and LDT, perform better on HDD due to their sequential I/O access patterns.

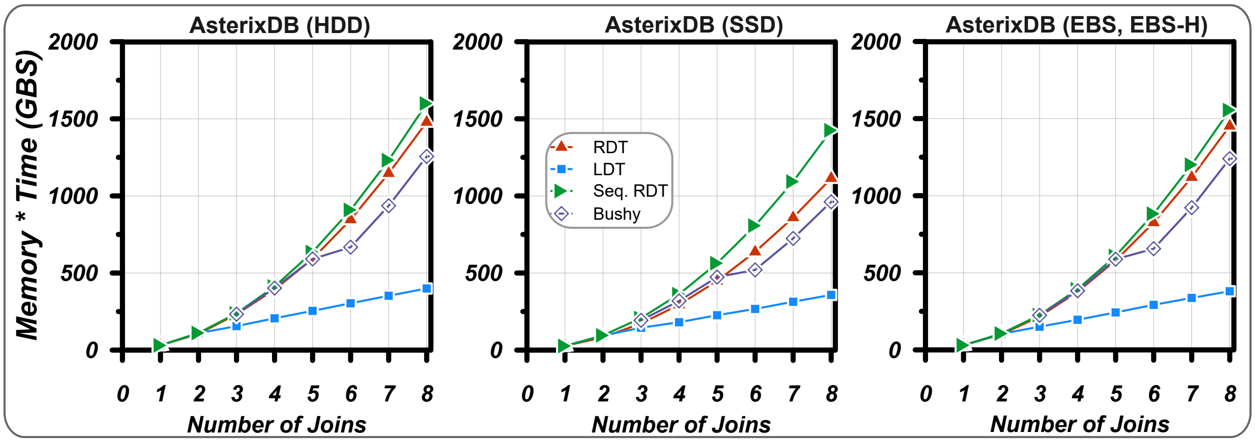


Figure 4.12: Resource Cost - Unlimited Memory Experiment

Since many users use cloud service providers for storing and querying their data, it is very interesting to compare different query plans based on how much they would cost in terms of money based on their resource usage and duration of execution. For this purpose, we introduce another metric to take the amount of resource usage, mainly memory, and the duration of its usage into account. Figure 4.12 shows the resource cost (or footprint) of each query plan in terms of its memory usage times its execution time. For all storage choices, the low memory usage of the LDT plan makes it the cheapest query plan -by far- among others. For the other query plans with similar memory usage, their execution times define which query plan is more expensive in this experiment.

### 4.8.4 Experiment 2 - Limited Memory

For the second experiment, we study the impact of the amount of available memory on the execution times of various query shapes and memory distribution strategies for an eight-join query. This experiment was designed similarly to the "Limited Memory - High Resource Contention" Experiment of [66]. Similar to [66], we evaluate the performance of each query plan as a function of memory availability; thus, the x-axis represents the ratio of available memory over the required amount of memory to keep all eight joins in memory.

All input datasets contain 1 GB of data, and the size of the intermediate results remains constant and equal to 1 GB throughout the query plan's joins. Each record is 1073B.

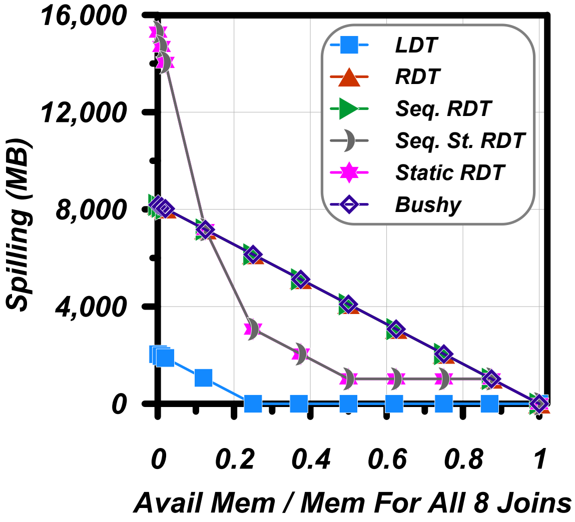


Figure 4.13: Spilling to Disk - Limited Memory Experiment

As Figure 4.13 shows, RDT and Sequential-RDT perform the highest amount of I/O for most data points since memory is divided between all the joins in the query leading to the spilling of the joins. Static-RDT and Sequential-Static-RDT perform a lower amount of I/O than RDT (except for the first three data points) since only intermediate results spill to disk at each breaking point. The high spilling amount in the first three data points of Static-RDT and Sequential-Static-RDT is because all joins spill results to disk (see Figure 4.8) in addition to all the required intermediate results being materialized. On the other hand, LDT performs the lowest amount of I/O since memory is divided between two consecutive joins at each time. Regarding parallelism, RDT and Bushy Trees are two of the most parallel query plans. Static-RDT parallelism is highly dependent on the amount of memory since the joins in each segment run concurrently.

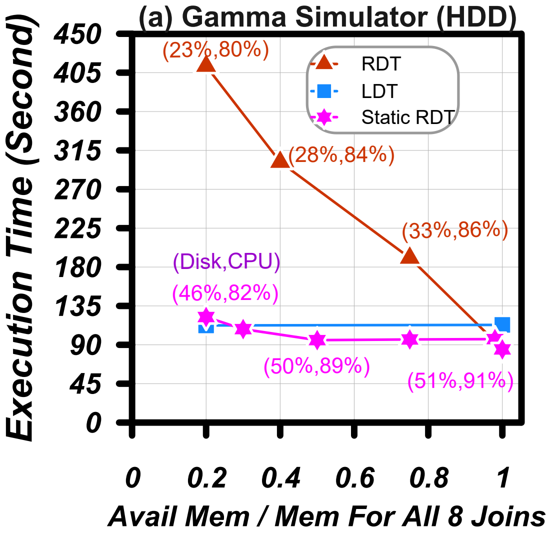


Figure 4.14: Experiment 2 - Limited Memory - Gamma Simulator

Figure 4.14 shows the results of the Gamma simulator as reported in [66], and Figures 4.15-a, 4.15-b, 4.15-c, and 4.15-d show the results of similar queries executed using Apache AsterixDB on HDD, SSD, AWS EBS, and AWS EBS-Hybrid, respectively. As Figure 4.15-a shows, LDT has the fastest execution time since it performs the least amount of I/O and its sequential execution pattern is disk arm-friendly. After LDT, the Static-RDT and Sequential-Static-RDT have the lowest execution times due to their smaller amount of I/O and sequential execution patterns. Sequential-Static-RDT is fully sequential, and Static-RDT becomes more parallel as more memory becomes available. RDT has the worst performance on HDD when limited memory is available due to its large amount of spilling and frequent random access to the disk. Although Sequential-RDT performs the same amount of I/O as RDT, it has a lower execution time due to its sequential I/O execution pattern. Bushy Tree is the second-worst-performing query plan after RDT due to its high amount of I/O and parallel execution pattern, which causes random disk access patterns. However, Bushy Tree benefits from the independent parallelism that causes disk and CPU operations to overlap, leading to better CPU utilization than RDT.

As Figure 4.15-b shows, parallel query plans such as RDT and Bushy Tree perform better

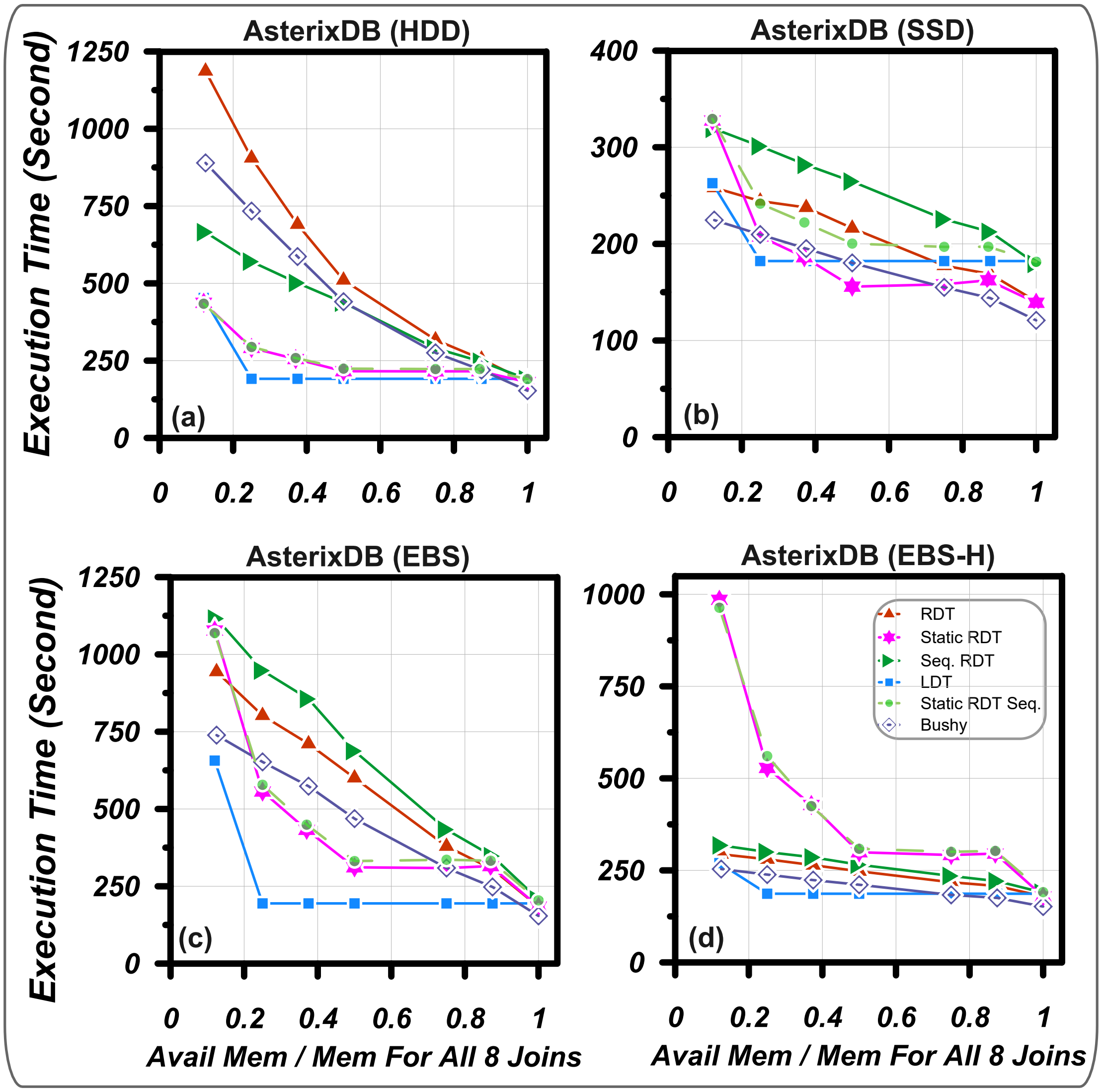


Figure 4.15: Experiment 2 - Limited Memory - AsterixDB

in SSD than HDD due to the lack of the disk arm issue in SSD and its capability to handle random disk I/Os and large volumes of I/Os efficiently. RDT, Bushy Tree, and Static-RDT outperform LDT when the available memory is very large. LDT is still one of the best-performing query plans due to its small spilling to disk, especially when memory is very scarce. Static-RDT performs well, especially with more memory, due to its semi-parallel execution pattern and relatively little spilling to disk. The sequential versions of RDT and Static-RDT have worse performance than their parallel versions since they do not take advantage of the SSD's capability to handle random and concurrent disk I/Os.

In AWS EBS, the execution times of the various query plans are higher than with SSD in general due to the network latency for accessing the over-the-network SSD storage as Figure 4.15-c shows. In this setting, the query plans with lower I/O and less disk access have better performance due to less network overhead. RDT and Static-RDT performed better than their sequential versions since the underlying storage system is SSD and can handle random disk access efficiently. The limited authorization provided by AWS prevented us from measuring the network performance between the processing node and its storage, but we suspect some optimization techniques are used to improve the over-the-network disk accesses and that they are more effective for the more parallel query plans. As for the AWS EBS-Hybrid, the performance trend for various query plans looks similar to the ones from AWS EBS and SSD; however, compared to SSD, these query plans take a slightly longer time due to reading the base relations from the disk. Compared to AWS EBS, the same query plans perform about four times faster in AWS EBS-Hybrid, especially when memory is very limited. This is because the intermediate join results are spilled to the locally attached SSD storage; therefore, the network overhead does not exist for reading and writing the spilling data.

As our results for AsterixDB show, for this simplified setting in which all the base relations and the intermediate results have the same size, LDT is one of the best-performing query plan shapes, especially when memory is very limited. LDT outperforms the other query plans in HDD due to its fewer data spilling and disk-friendly sequential execution pattern. For other node settings where SSD is utilized as the storage type, LDT is still one of the best-performing query plans mainly due to its small amount of I/O and higher CPU utilization. LDT has a consistently high CPU usage (between 42% and 48%) due to the possible overlapping of disk and CPU operations. Variations of parallel query plans including RDT, Static-RDT, and Bushy Tree perform better as the memory increases since the amount of their spilling to disk drops significantly.

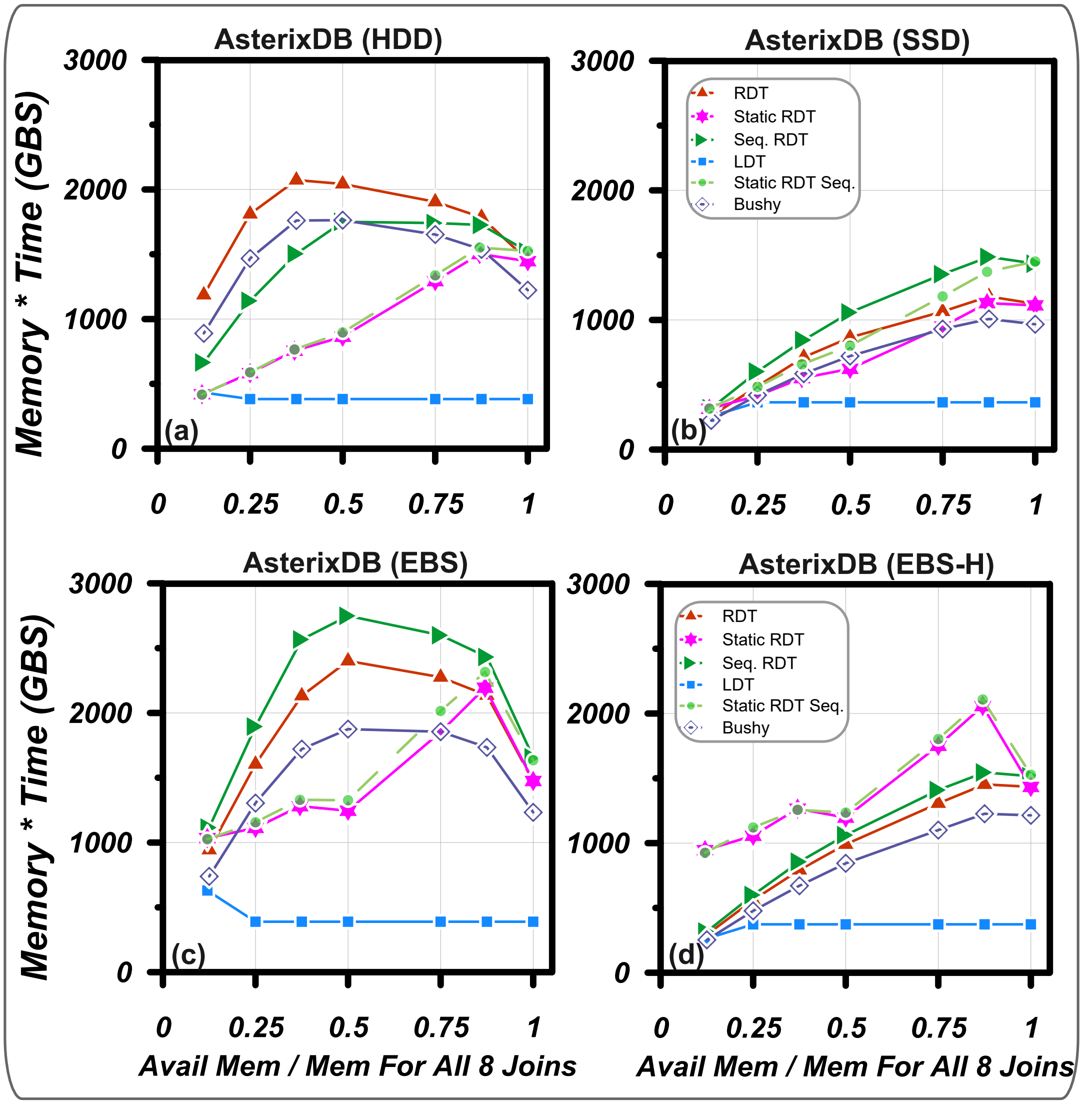


Figure 4.16: Resource Cost - Limited Memory Experiment

By comparing the results from AsterixDB with the results of the Gamma simulator, we can see that even without an accurate simulation model for disk contention, the excessive data spilling of RDT made it the worst-performing query plan when memory is less than 80% of the required memory. The trend of these results matches the observed results of AsterixDB for HDD.

Figure 4.16 shows the resource cost of each query plan in terms of its memory usage times its execution time as a function of the available memory. Similar to Experiment 1, the low memory usage of the LDT plan makes it the cheapest query plan among the others.

### 4.8.5 Experiment 3 - Non-Restrictive Select Conditions

In this experiment, we re-evaluate the results of the “Large Building Relations - Full Declustering” experiment from [66] and study the performance of various multi-join query plan shapes with low-restrictive select conditions which reduces the size of build and probe inputs that use base datasets as their inputs.

Similar to [66], we used a four-join query that uses the following relation cardinalities and selectivity factors: 1,000,000 records with 50% selectivity, 1,000,000 records with 50% selectivity, 1,000,000 records with 20% selectivity, 500,000 records with 10% selectivity, and 200,000 records with 25% selectivity. To have a fair comparison with the results of [66], we also used join selectivities such that the size of the four intermediate join results were 50,000, 50,000, 100,000, and 100,000 tuples.

Additionally, we also explored a smaller amount of memory here, called the “Very Limited Memory” case, to model “Big Data” cases where the ratio of available memory over the input sizes is less than 0.1. Having large input datasets without highly-restrictive selection predicates and small outputs from each join makes the settings favor LDT rather than variations of RDT. Under these settings, variations of the RDT plan require more memory, and thus the lack of memory makes RDT spill more data to disk than LDT. Figure 4.17 shows this experiment’s amount of spilling for various query plan shapes.

The Gamma simulator results in Figure 4.18 show that RDT had the worst performance with very limited memory due to spilling a large amount of data to disk, while LDT was the best-performing query plan shape due to its minimal spilling to disk. Static-RDT performed similarly to LDT when the available memory is significant.

As Figure 4.19-a shows, RDT is the worst-performing query plan on HDD under this experiment’s settings. RDT spills the most data to disk while four build activities run and

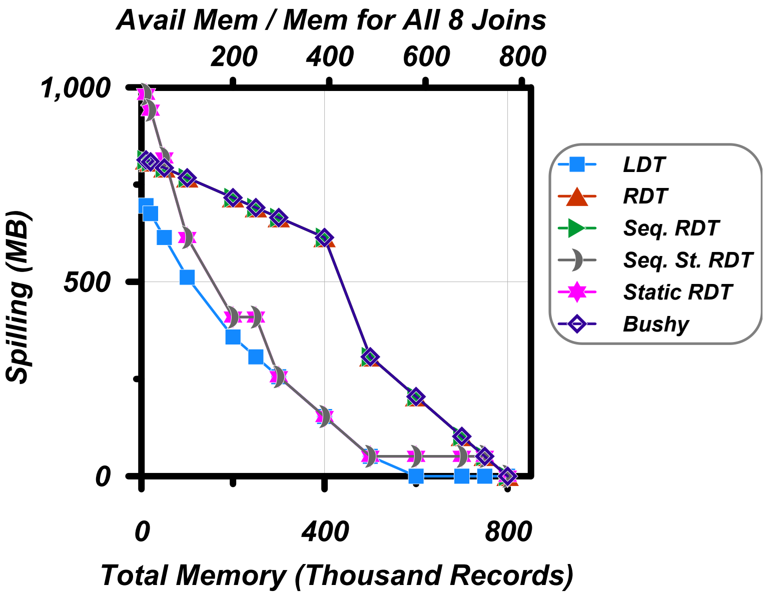


Figure 4.17: Spilling to Disk in AsterixDB - Non-Restrictive Select Conditions Experiment

access the hard disk concurrently. The high spilling and random accesses to the disk, which worsens the disk arm contention, make RDT perform poorly on HDD. Sequential-RDT performs slightly better than RDT due to its sequential pattern of executing build phases of different joins. Bushy Tree performs better than RDT and Sequential-RDT since it uses the intermediate results (smaller than any of the base datasets) as the build input for one of the joins, and its parallelism is less than RDT. LDT, Static-RDT, and Sequential-Static-RDT are the best-performing query plans for this setting where the storage device is HDD. LDT outperforms Sequential-Static-RDT and Static-RDT when less memory is available since its inputs are smaller and memory is shared only between two consecutive joins. Its sequential execution pattern also helps with not causing extra disk contention on HDD. Static-RDT and Sequential-Static-RDT perform better than other variations of RDT since they will spill less. Static-RDT's parallelism is also less when memory is limited, which helps with not causing excessive disk contention.

As Figure 4.19-b shows, RDT performs better on SSD since the random access to disk and the general I/O cost is less in SSD. Additionally, the highly parallel nature of RDT makes it utilize the CPU better than its sequential variant, Sequential-RDT. Note that LDT is still

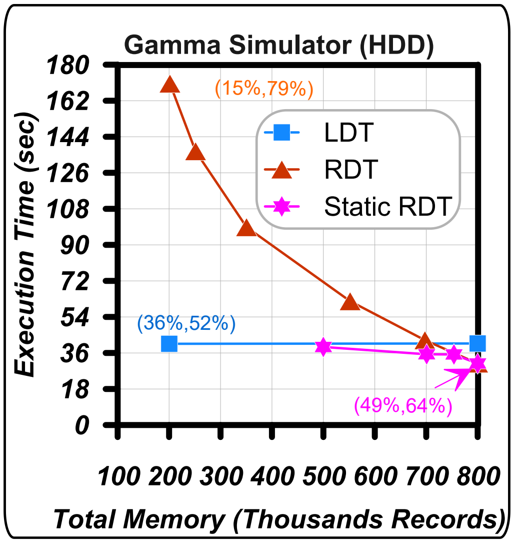


Figure 4.18: Experiment 3 - Non-Restrictive Select Conditions - Gamma Simulator

one of the best-performing query plans , though, since even with a small amount of memory, a large portion of each build input remains in memory. This is because the join selectivities are low, and the output of each join becomes the build input of the next join. Additionally, LDT utilizes the CPU better than other query plans due to its execution schedule, which results in more overlapping of disk and CPU operations. Bushy Tree and Static RDT's good performance are due to their semi-parallel execution. Sequential-Static-RDT performs slightly worse than Static RDT due to executing sequentially and not exploiting the SSD's capability in handling random I/O efficiently.

The performance trend for various query plans in AWS EBS and AWS EBS-Hybrid is very similar to SSD since the underlying storage device in all three cases is SSD. However, the query plans that perform less spilling to disk, such as LDT, perform better than the others on AWS EBS since they avoid network latency for accessing the remote spilling disk.

One interesting observation in Figure 4.19 which is more pronounced for AWS EBS than for the other storage choices is the two jumps on the lines for the performance of Static RDT and Sequential-Static RDT at the memory points for 250K and 750K records. The reason that Static-RDT and Sequential-Static RDT take a longer time to finish at these memory

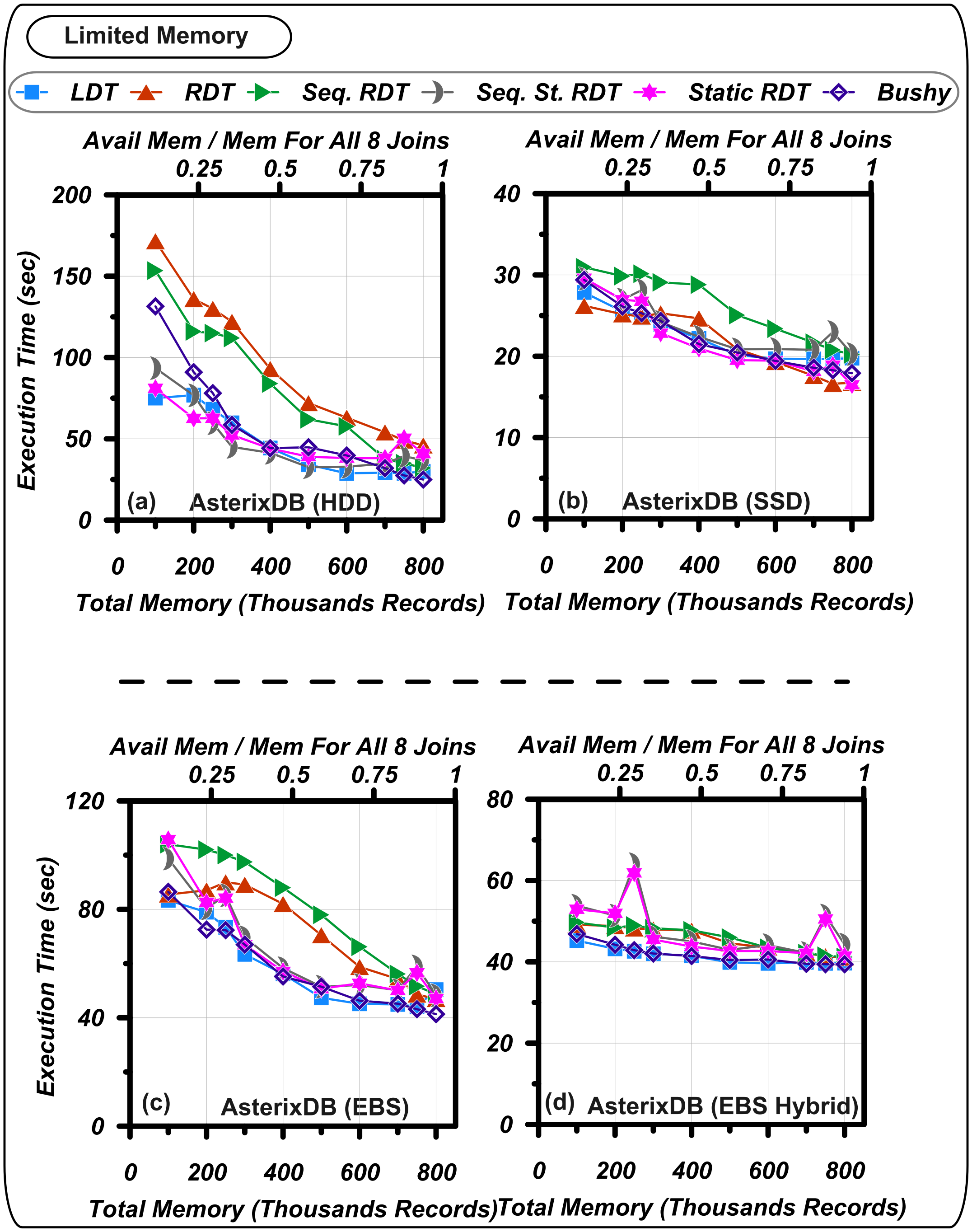


Figure 4.19: Experiment 3 - Non-Restrictive Select Conditions - AsterixDB (Limited Memory)

settings is that with the given memory, the tree will break after the third join, which causes 100,000 tuples to spill to disk. However, if the breaking point was set after the first or second join, only 50,000 records would spill to disk. This shows the importance of considering the

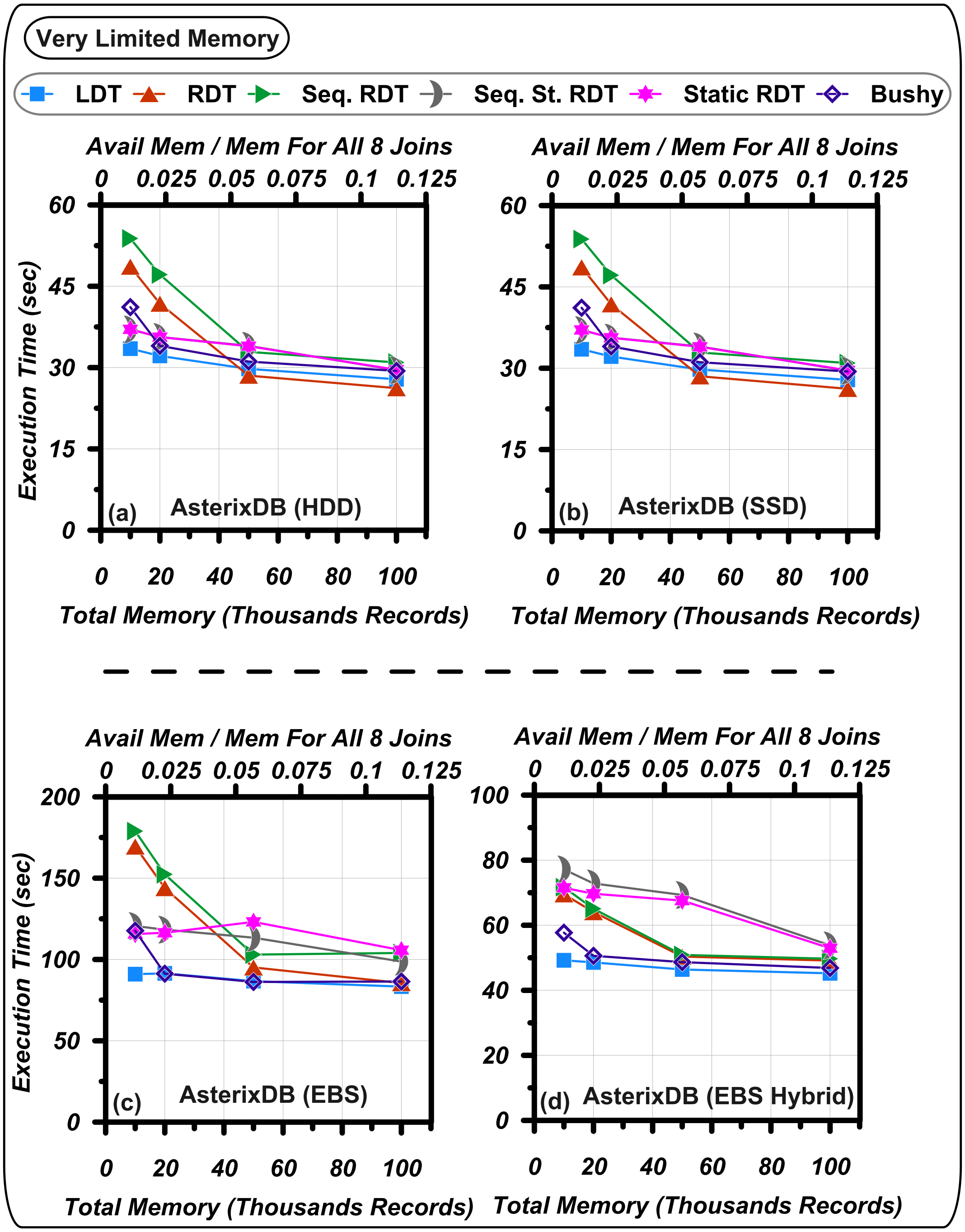


Figure 4.20: Experiment 3 - Non-Restrictive Select Conditions - AsterixDB (Very Limited Memory)

size of intermediate results in choosing the breaking points in Static-RDT and Sequential-Static-RDT. We would have expected similar observations in Gamma simulator results, but those results do not reflect such observations.

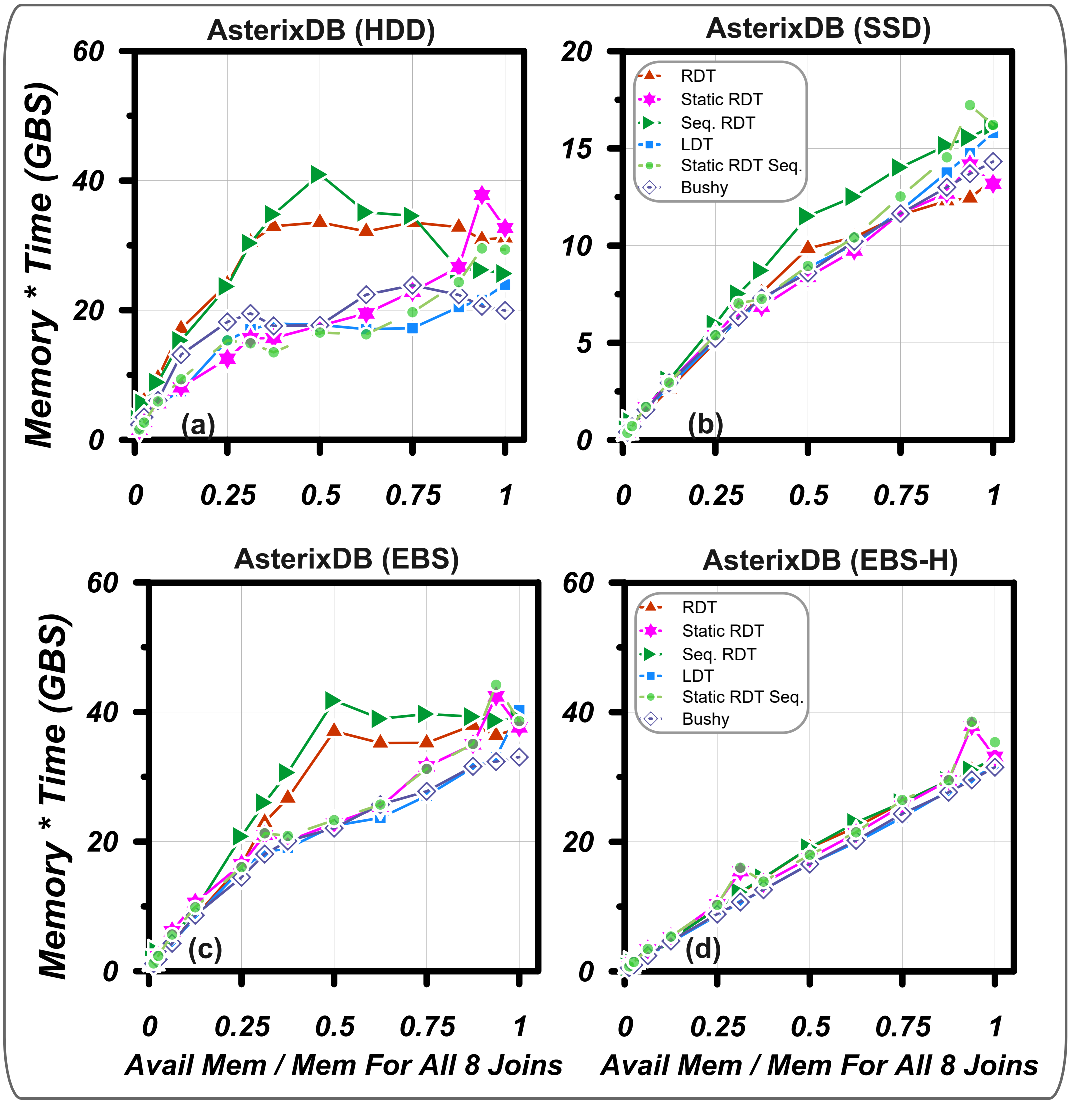


Figure 4.21: Resource Cost - Non-Restrictive Select Conditions Experiment

Figure 4.21 shows the resource cost of each query plan in terms of memory usage times its execution time. Similar to previous experiments, LDT is one of the low-cost query plans in HDD, SSD, and EBS due to its low execution time and memory usage. The cost of the various query plans becomes close to one another in AWS EBS-Hybrid due to the similarity in their execution times.

### 4.8.6 Experiment 4 - Non-Restrictive Join Conditions

Next, we study the performance of different query plan shapes when the query's joins do not have a highly restrictive condition; thus, each join may produce numerous output records per input record. This setup is in favor of plan shapes that use base datasets as the input to their build phase of the joins, including variations of RDT. Similar to the "High Join Selectivity" experiment of [66], the base datasets have sizes of 1000000, 1000000, 1000000, 500000, and 200000 records with scan selectivities of 50%, 50%, 20%, 10%, and 25%, respectively. The join selectivities cause the joins to produce 50000, 200000, 400000, and 500000 records as their outputs.

Figure 4.22 shows the total amount of data spilled to disk here for each query plan under different memory availability. RDT and Sequential-RDT spill the same amount of data to disk, where this amount reduces as the size of the available memory increases. Although RDT and Sequential-RDT have smaller build input sizes than LDT, their overall memory requirements are higher since they divide the memory among all the join operators in the query. In contrast, LDT uses the intermediate results, which are large in this experiment, as the input to the build phase of all joins except for one. LDT thus spills less data and requires less memory for a join to fit entirely in memory since the given memory is divided only between two joins at a time. The amount of spilling in Static-RDT and Sequential-Static-RDT is equal and dependent on the location of the breaking points in the query tree.

Sometimes using a larger amount of memory in Static-RDT and Sequential-Static-RDT causes the breaking points in their query trees to move up higher in the tree and thus be placed after a join with a large amount of output. Thus, a queryy optimizer should carefully decide where to put the breaking points on the Static-RDT and Sequential-Static-RDT based on memory availability.

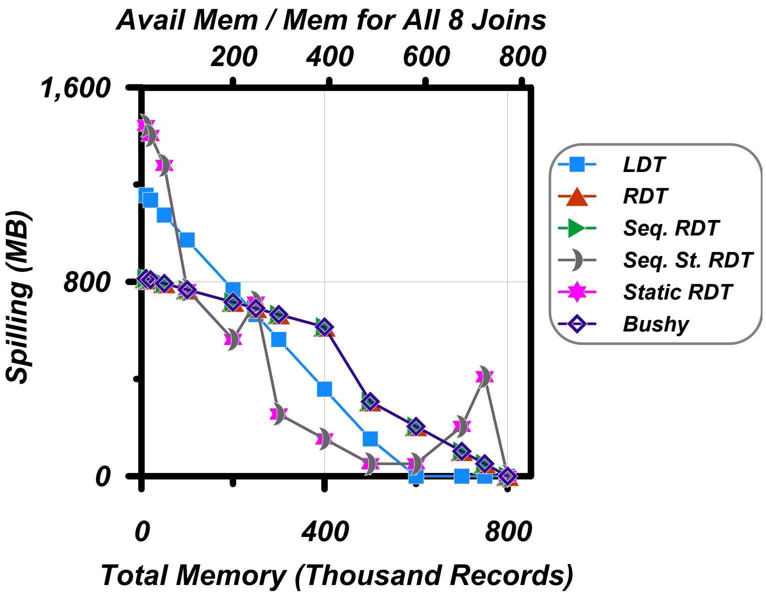


Figure 4.22: Spilling to Disk - Non-restrictive Join Conditions Experiment

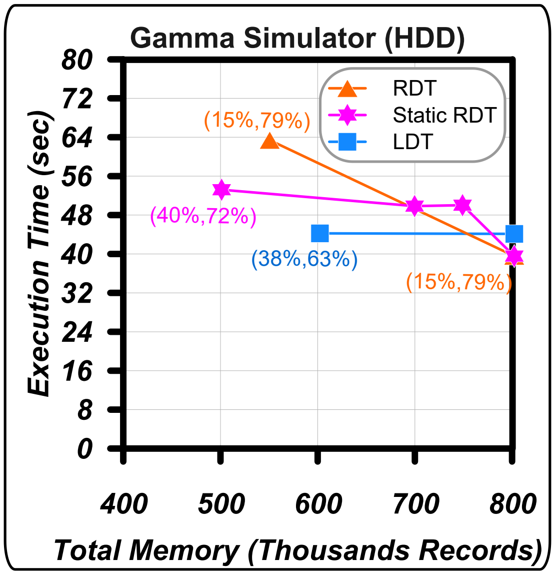


Figure 4.23: Experiment 4 - Non-restrictive Join Conditions - Gamma Simulator

As Figure 4.23 shows, RDT had the highest execution time due to extensive spilling to disk when memory is limited in the Gamma simulator. However, due to its parallel execution nature, RDT outperformed LDT and Static-RDT when a large amount of memory is available. Unfortunately, the figure for the Gamma simulator does not show data points on the far left-hand side to present the performance of these query plan shapes when memory is highly scarce. In the AsterixDB experiments, we also consider both cases where the available

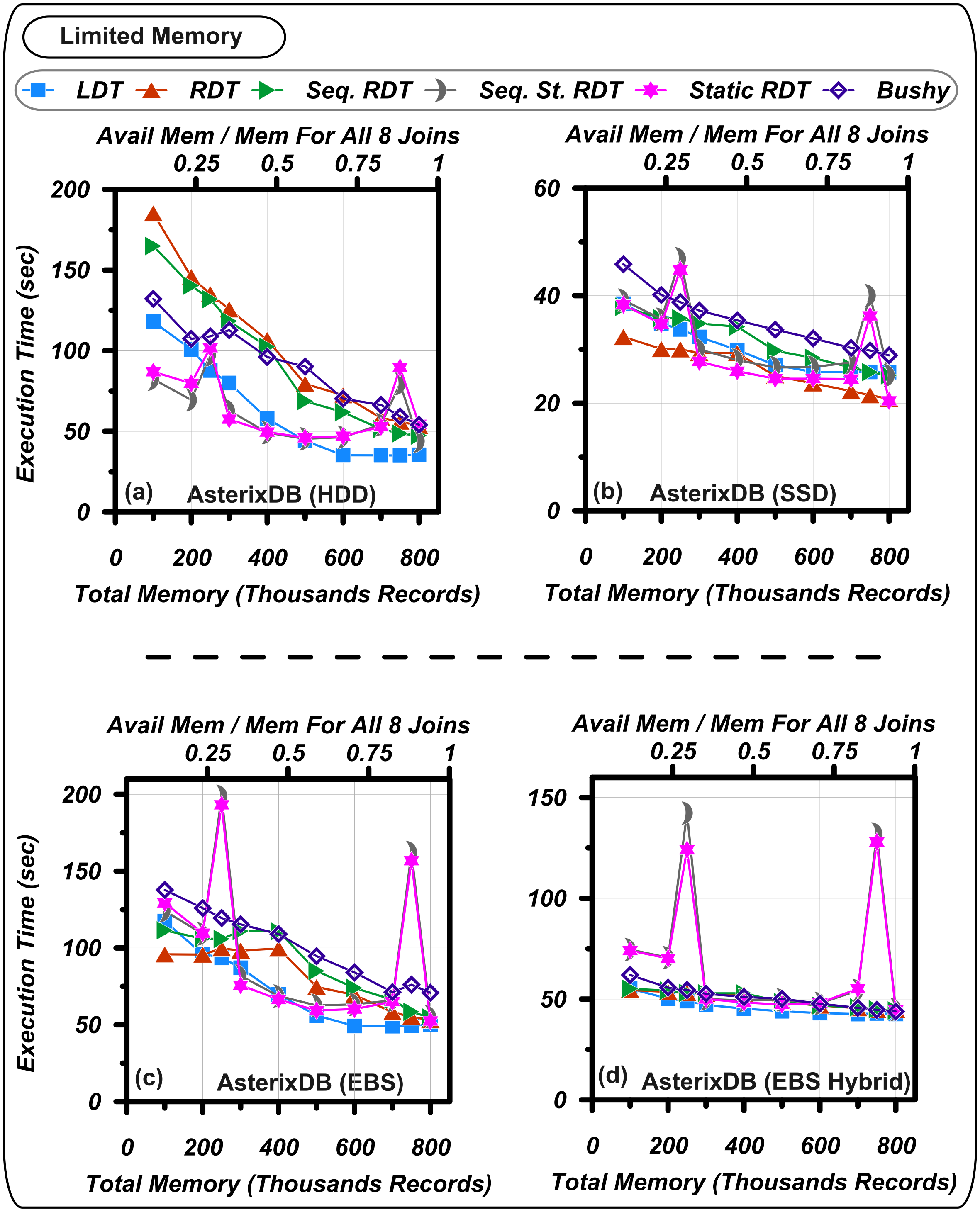


Figure 4.24: Experiment 4 - Non-restrictive Join Conditions - AsterixDB (Limited Memory)

memory is less than 0.125 of the data sizes, called the *Very Limited Memory* case, and where the available memory is somewhat larger, called the *Limited Memory* case.

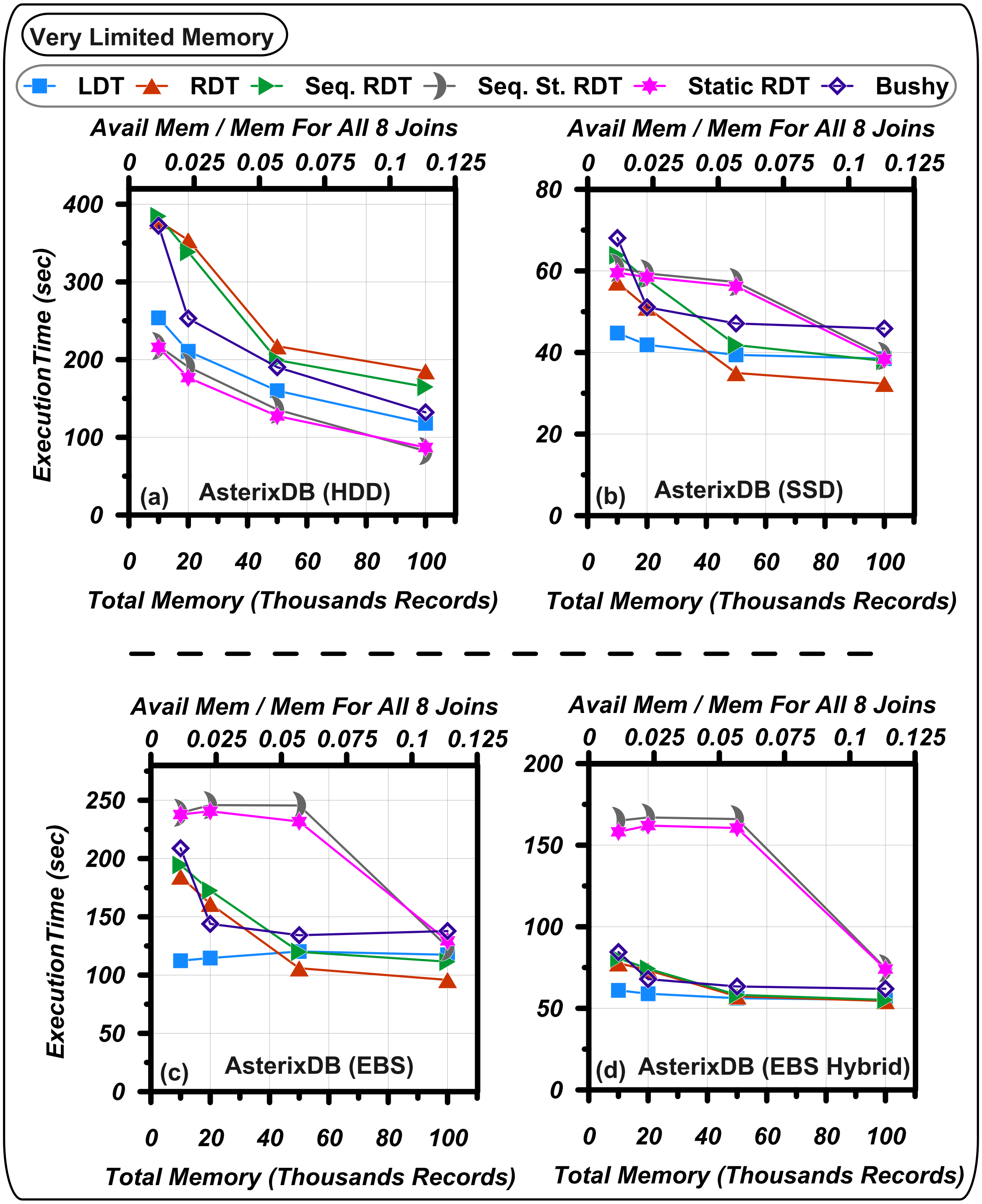


Figure 4.25: Experiment 4 - Non-restrictive Join Conditions - AsterixDB (Very Limited Memory)

As discussed earlier and as Figure 4.24-a shows, the high disk arm contention caused by parallel access to disk makes RDT one of the worst-performing query plan shapes when the underlying storage system is HDD. The performance of RDT improves as memory becomes more available, and Sequential-RDT outperforms RDT for HDD due to its sequential access pattern. Despite using non-base relations with large sizes as the build inputs, LDT is one of the best-performing query plan shapes in this experiment since it divides memory between only two consecutive joins and follows a sequential access pattern to disk. Static-RDT and Sequential-Static-RDT performed similarly in limited memory as the query tree has been divided into multiple subtrees. By increasing the available memory, Sequential-Static-RDT slightly outperforms Static-RDT due to its sequential access pattern and since the query tree has been divided into fewer segments with longer lengths. As Figure 4.24 exhibits, there are two spikes in the performance of Static-RDT and Sequential-Static-RDT since the increment of memory has shifted the breaking point to a higher point in the tree with a larger intermediate result size.

Query plans that run in parallel tend to perform better than sequential ones when SSD is utilized for this experiment. As Figure 4.24-b shows, RDT is the best-performing query plan in this case since its memory requirement is low due to using small-sized base datasets as the build inputs. Additionally, its parallel execution pattern benefits from the efficient random-access capability of SSD. Although Sequential-RDT performs the same amount of I/O as RDT, it performs worse than RDT due to its sequential execution pattern. The difference between Static-RDT and Sequential-Static-RDT is small in this experiment due to the short query pipeline. LDT performs relatively well in this experiment despite its sequential pattern and large sizes of inputs. Each join spills less data in LDT since memory is shared between just two joins at a time, and the pipeline between the probe phase of the previous join and the build phase of the next join allows for a higher CPU utilization.

The trends in performance for the various query plan shapes in AWS EBS (Figure 4.24-c) is very similar to the ones in SSD (Figure 4.24-b) for the majority of the data points. However, the execution times of different query plans are higher in AWS EBS than those from SSD due to the network overhead for accessing the disk. Query plans that execute sequentially and transfer fewer data to and from over-the-network disk at a time,including LDT, tend to have lower execution times due to low disk overhead. On the other hand, parallel query plans, including RDT, benefit from the efficiency of SSD storage in handling random disk access; however, they may face a high latency due to network overhead and bandwidth saturation.

Despite their higher amount of I/O, the more parallel query plans have comparable performance to the sequential query plans when the underlying storage system is AWS EBS-Hybrid. The data spills to a fast SSD storage attached to the node, making writing the data to disk faster. In this experiment, when using AWS EBS-Hybrid, the different query plan execution strategies perform similarly. Although LDT's sequential execution plan does not benefit from the efficiency of SSD in managing random I/O, it spills less to disk due to how it manages its memory; also, it maintains a constantly high CPU utilization and its sequential reading from the base relations does not hit the network bandwidth limitation of storage over the network. With regard to the RDT and Bushy Trees, although they spill more data to disk and their parallel reads from base relations may hit the network's bandwidth limitation and degrade performance, the spilling data under AWS EBS-Hybrid is written to the local fast SSD storage, and all of the I/Os are done in parallel which makes up for the mentioned performance degradation. The network overhead masks the differences between the sequential and non-sequential variations of RDT and Static-RDT. Static-RDT and Sequential-Static-RDT still face degradation in their performance when the breaking points are positioned after a join with numerous output results.

Figure 4.26 shows the resource cost of each query plan in terms of their memory usage times their execution time. Like in the previous experiments, LDT is one of the lowest-cost query plans for HDD due to its low execution time and memory usage. In SSD, RDT has the lowest execution time, which makes it one of the lowest-cost query plans. In AWS EBS, LDT has the lowest cost among other query plans since its execution time and memory usage is less

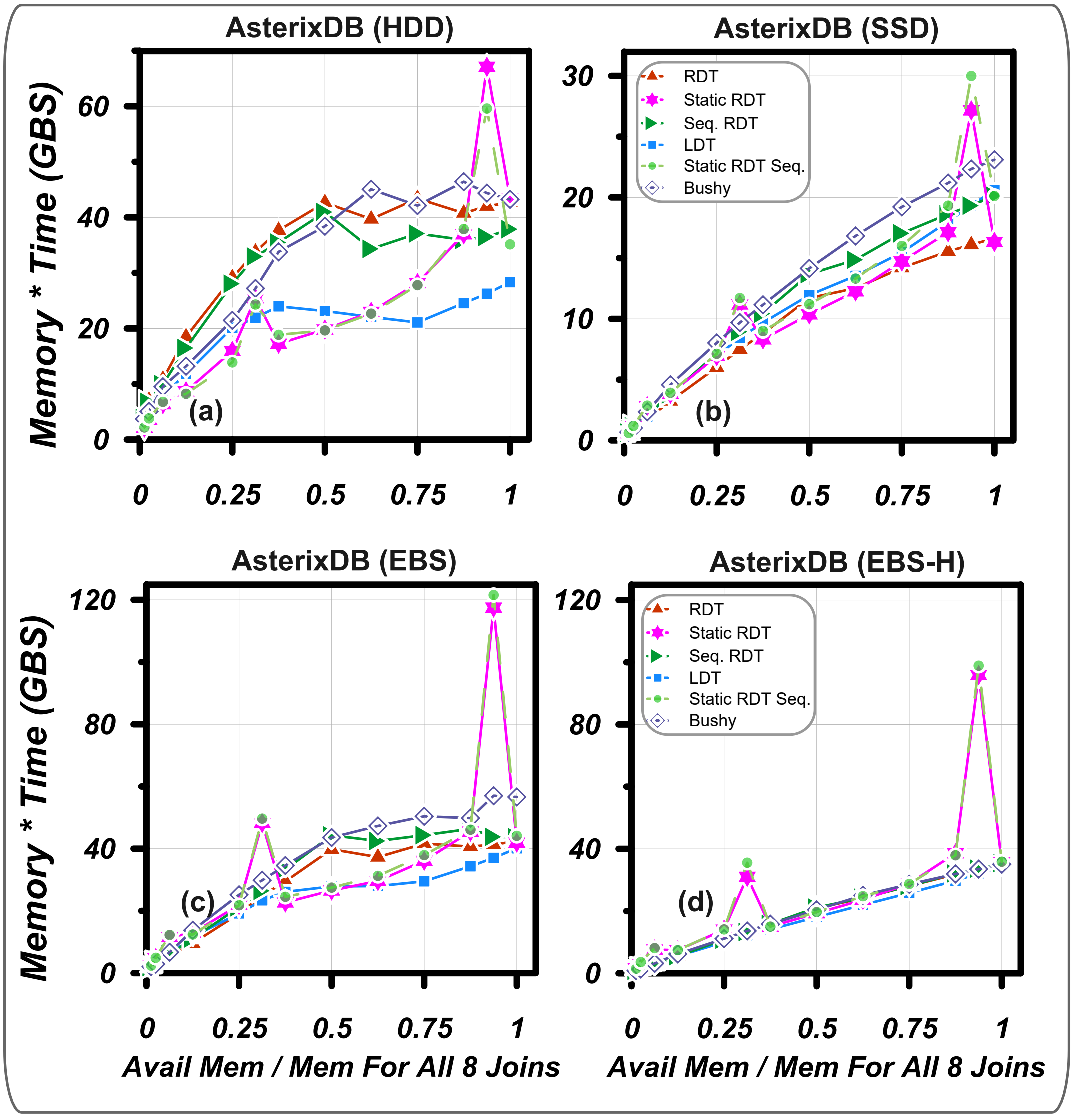


Figure 4.26: Resource Cost - Non-Restrictive Join Conditions Experiment

than others. In AWS EBS-Hybrid the similarity in the execution times of the various query plans at most data points makes their costs very similar.

In conclusion, even when the join conditions are not restrictive, the LDT query plan is one of the best-performing query plans due to its memory distribution. Additionally, its sequential I/O positively impacts its performance when the underlying system is HHD or when the storage device is over the network (AWS EBS and AWS EBS-Hybrid). When a local SSD is used as the storage device, in this case, RDT performs the best due to its parallel execution pattern and small build input sizes. Additionally, although it may spill more data to disk than LDT, the cost of the spilling is not prohitive since SSD is used as the storage device.

## 4.9 Conclusion and Recommendations

In this chapter, we re-evaluated the results of a key study done by Schneider & Dewitt [66] in which they studied the performance of multi-join queries in shared-nothing clusters. They used a simulator made for the Gamma database system on HDD. We re-evaluated their results using Apache AsterixDB utilizing HDD, SSD, AWS EBS, and AWS EBS-Hyrbid as storage alternatives. In addition to their mentioned plan shapes, we studied the performance of an example of Bushy Tree, Sequential-RDT, and Sequential-Static-RDT.

RDT has been thought for a long time, largely due to [66], to be the most efficient plan shape due to its parallel execution pattern. However, our studies show that RDT is only a good query plan for SSD-based storage systems (including AWS EBS and AWS EBS-Hybrid) if the number of joins in the query result in a few build datasets that are not too large, eg. if the available memory is enough to hold more than 80% of the sum of the sizes of all build datasets in memory. Static-RDT may spill less data to disk than RDT if there is enough memory available to at least hold each build dataset in memory. In cases where multiple consecutive build phases can fit in memory, precautions must be applied in Static-RDT and Sequqntial-Static-RDT when selecting the breaking points on the tree not to put it after a join that produces large output.

Our experiments showed that LDT is actually one of the best-performing query plan shapes in most cases. LDT tends to spill less data to disk since the memory is always divided between two consecutive joins at a time. Moreover, its sequential execution pattern makes it less-disruptive for disk arm contention in HDD. Also, the LDT's short pipelines between the build phase of the previous join and the probe phase of the next join lead to a consistently high CPU utilization due to the overlap of these activities.

Bushy Trees benefit from independent parallelism, which leads to its high CPU utilization due to overlapping subtrees of the query. The Bushy Trees we considered in this study are wide and have short pipelines. Bushy Trees may use either a base dataset or the intermediate results of a previous operator as the input to their phases. Thus, caution should be taken in choosing the inputs to the build phases, as that determines the amount of spilling to disk.

The Sequential-RDT and Sequential-Static-RDT plans perform better than their parallel versions when the underlying storage device is HDD and the query contains multiple joins with large build inputs.

Our results from this chapter show the importance of considering the underlying storage architecture in choosing the query plan. They also show that re-evaluation of previous studies is necessary every so often due to changes and improvements in the underlying hardware. We also saw that simulators, while helpful and essential tools for understanding systems' behavior, can produce incorrect results if not verified against real systems carefully.

As bottom line recommendations for AsterixDB and other systems, we suggest using LDT query plans as the default query plan shape for all queries and all storage types. We have seen that LDT maintains high performance under different join and scan selectivities as well as for different storage types and memory availability. RDT can outperform LDT in a few cases where the underlying storage device is a locally attached SSD and the available memory for a query is more than 80% of data. However, the difference in its execution time is not all that high compared to that of the LDT. Thus, using LDT as the default query plan is recommended.

# Chapter 5

# Fair Scheduling for Concurrent Queries

After studying the best practices for implementing a robust HHJ in Chapter 3 and memory management and efficient execution of individual multi-join queries in Chapter 4, we study and evaluate various scheduling techniques for queries executing in a concurrent environment in this chapter.

Since the responsibility of the schedulers may vary from one system to another, we first explain what a scheduler does in AsterixDB and what modules are involved in the execution process of a query. In AsterixDB, each incoming query must pass through several modules before starting its execution as Figure 5.1 shows. After compilation and optimization, a query enters the admission controller module. As mentioned earlier, the current version of AsterixDB does not dynamically allocate memory to the queries and operators; instead, it uses memory budgets for each operator. Since only one stage of the query is active at any point in time, the admission controller calculates the memory and CPU requirements for the most resource-intensive stage and considers this value as the total CPU and memory requirements for the query. The memory budget assigned to each operator is used for the memory requirement calculation. Queries with resource requirements less than the system's total resources will be admitted and passed to the scheduler module to be considered for execution, all other queries are rejected. The scheduler in our design is only responsible for choosing the next query for execution. The next chosen query is sent to the query execution module as soon as the resource manager has enough resources to meet the needs of that query.

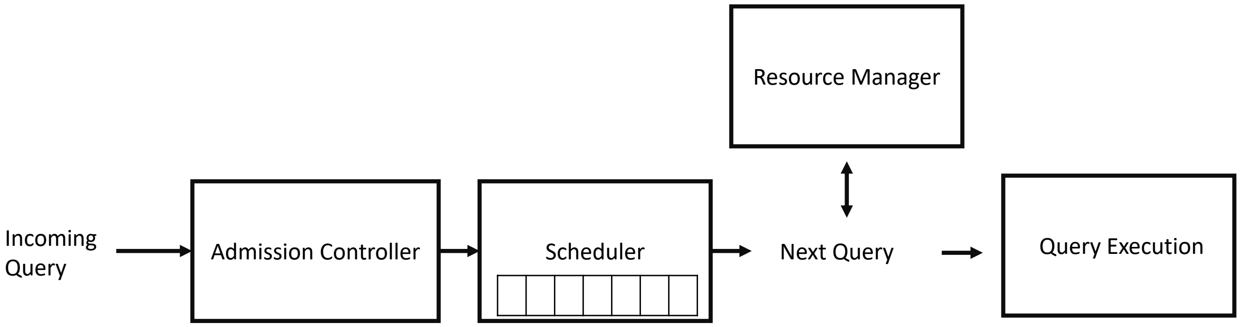


Figure 5.1: Query Scheduling and Resource Management Components in AsterixDB

The goal of this chapter is to design a fair scheduler for a DBMS that is effective at scheduling the execution order of queries with widely different execution times and resource requirements. We call a scheduler fair if, under its management, each query waits in the queue for a time proportional to its execution time. We classify the queries into multiple classes based on their memory requirements and the number of records accessed during their execution.

We define a few measurable metrics to evaluate and to compare the fairness of various scheduling algorithms. In our work, *stretch factor* or *slowdown* measures how much, on average, a query of a specific class has waited in the scheduler's queue compared to its execution time in a concurrent environment. We calculate the stretch factor for a query class as the ratio of its average response time to the execution time for its successfully executed queries. Next, we define *fairness* as a measure of the evenness of the slowdown of all query classes. We will calculate the fairness metric by using the standard deviation of the stretch factors across the query classes.

Furthermore, we present the design of seven schedulers and evaluate them using different metrics, including fairness, when the system handles concurrent queries from different query classes.

## 5.1 Related Work

In this section, we review two main areas of related work:

1. Resource allocation techniques for concurrent online query workloads.
2. Scheduling and order of execution for queries with widely varying resource requirements and execution times.

Query scheduling and resource allocation have been a research topic for many different workload settings and database system types. As one of the early works from the 1980's, [18] used a fairness metric in studying the problem of dynamically assigning queries with different resource requirements to sites in a distributed database system. They defined fairness as the similarity in the response time ratio to the wait time for various queries. They use a simulator of a fully replicated distributed database system to evaluate their suggested heuristics for the dynamic allocation of queries to sites.

Several key studies in resource allocation and query scheduling appeared in the 1990's. Phillip S. Yu, et al. introduced the concept of Return on Consumption (ROC) in [71] and used it as a basis to allocate memory to various operators in a multi-query environment. They defined ROC as the ratio of benefit to cost of giving additional memory to an operator compared to the operator's minimum memory allocation. The authors used single-join queries with Hybrid Hash Join as the join operator. The minimum memory requirement for a Hybrid

Hash Join was calculated as $\sqrt{F \times R}$ where **R** is the size of the build input in pages and **F** is the fudge factor for considering the hash table's overhead in size. Their experiments showed that the highest reduction in average response time is achieved by allocating the maximum required memory to the smallest queries and allocating the minimum required memory to the large queries. The results of this study were subsequently used in several other works on resource allocation [22, 56, 62].

In 1993, Mehta and DeWitt studied the problem of memory allocation and query scheduling for queries with widely varying resource requirements [56]. Their goal was to maximize fairness among various classes of queries. Based on a detailed simulation study, they evaluated the performance of several memory allocations and scheduling schemes. They used single-join queries that were classified based on their build input sizes into three classes – small, medium, and large queries. In one of their schedulers named *adaptive*, each query class has its own independently serviced FIFO queue and the scheduler is responsible for dynamically adjusting each query class's multi-programming level (MPL). The authors of [56] used the results from [71] assigning the maximum required memory to the small queries and minimum required memory to the large queries. The remaining memory was then equally divided between the medium queries such that each query receives at least its minimum required memory.

For metrics, Mehta and DeWitt defined slowdown as the ratio of the observed average response time to the average standalone response time for each class. Furthermore, they defined fairness as the standard deviation of the slowdowns for all three classes. We will refer to this scheduler later as Wisconsin scheduler. We have implemented and evaluated three schedulers inspired by the Wisconsin scheduler in our work.

One year later, Davison and Graefe from the University of Colorado suggested a framework for fair resource allocation and query scheduling based on concepts from microeconomics [22]. They used a resource broker to sell the resources to the operators based on which allocation policy maximizes the profit. This framework defines a currency derived from the system's performance objective that is then used by operators to bid on the resources such as CPU, memory, and disk. Since the bid price has a direct relationship to the value of an allocation, a higher price indicates a more favorable allocation to the performance objective. The ROC concept from [71] was used to estimate the amount of profit for each allocation. In case of a new possible allocation that can increase the profitability, the broker may also "buy back" resources from the operators that are actively running and sell them to the new operators with higher performance benefits. An adaptive memory version of Hybrid Hash Join [21] was used to adjust memory usage possible during the execution of operators. In [21], the authors showed that using only three queues for various queries, as was done in [56], may degrade the performance and fairness since queries with very different execution times can be placed in the same queue. Thus, they used more than ten queues in order to put only queries with similar execution times into the same queue. They assigned more queues to smaller queries since they tend to have shorter execution times.

The performance objective for the prototype broker in [21] was to minimize the slowdown among various queries with the consideration given to fairness. For the calculation of fairness and slowdown, the authors considered each query individually instead of categorizing them into classes. In [21], slowdown measures the expansion of the response time of a query when executed concurrently with other queries as compared to its standalone response time. To calculate the slowdown metric, they first calculated the ratio of the observed response time to the standalone response time for each query separately. The mean of the ratios for all the queries is then the slowdown. Fairness, which measures the evenness in the degradation of the response time of all queries as the system load increases, was defined as the standard deviation of all of the individual queries' slowdown values. Several experiments using a simulator showed that their suggested scheduler is fairer and outperforms other schedulers, including [56]. We will refer to this scheduler as the Colorado scheduler. In our work, we implemented and evaluated two schedulers inspired by the Colorado scheduler.

Other recent research papers studied resource allocation and scheduling problems on various workloads and use case scenarios. The authors of [49] proposed adaptive query scheduling techniques for workloads containing different queries, each with a different performance objective. That paper aimed to demonstrate the complexity of searching for solutions when scheduling dynamic mixed workloads. The authors of [48] studied the workload management for business intelligence (BI) workloads specifically by considering actual workloads and objectives from industrial users' interviews.

## 5.2 Query Types and Classes

In this section, we will consider five different query classes for queries that users submit. This classification is based on the amount of memory that a query would require to execute fully in memory. In the following, we explain each query class.

- **ZeroMemory-Short.** ZeroMemory-Short queries use a secondary index to scan selected records returning their count. They have a selection condition chosen so that very few records in our case satisfy it. The ZeroMemory-Short queries do not contain any memory-intensive operators. Queries in this class have short execution times due to the number of records they access.

- **ZeroMemory-Long.** The queries belonging to the ZeroMemory-Long query class are scan queries that scan most of a dataset's records without using any index and return their count. The ZeroMemory-Long queries do not contain any memory-intensive operator but they have long execution times due to the number of records they access.

- **Small.** The Small queries are single-join queries where the sizes of build and probe inputs are between 5 and 25 percent of total memory. The dataset scan selectivities and ranges vary between different queries within this class. We use Hybrid Hash Join

(HHJ) as the join operator for the queries of this query class. Our initial study in this chapter will focus on join memory budget allocations that allow for non-spilling query execution.

- **Medium.** Similar to the Small query class, the Medium query class contains single-join queries; however, the size of their build and probe inputs is between 26 an 75 percent of the total memory. In these queries, HHJ is used with a join memory budget allocations that allow for the join to execute entirely in memory with no spilling to disk.

- **Large.** The Large query class follows the same structure as the Small and Medium query classes, but the sizes of build and probe inputs is between 76 and 95 percent of the total memory.

## 5.3 Scheduling Algorithms

This section introduces the different scheduling algorithms that we consider in this chapter to schedule concurrent queries.

### 5.3.1 FIFO-Ordered Scheduler

All queries are placed in a single queue under the FIFO-Ordered scheduling algorithm. Queries will then be executed based on their arrival order enforced by the FIFO policy. As a result, under this simple baseline approach, queries with smaller resource requirements may be blocked by queries with much larger resource requirements.

### 5.3.2 FIFO-Semi-Ordered Scheduler

The FIFO-Semi-Ordered scheduler, used in Apache AsterixDB, seeks to overcome the issue of small queries being blocked by the large queries in the FIFO-Ordered scheduler.

This scheduler also uses a single queue following FIFO order. However, in this scheduler, queries may be picked to start their execution earlier than their turn. Under this scheduler, any newly admitted query that can execute immediately using the available resources will directly start its execution without entering the FIFO queue. An incoming query will be added to the end of the queue only if the currently available resources do not meet its resource requirements. Additionally, upon completion of a running query, queries from the head to the end of the queue are checked and allowed to proceed if enough resources for their execution are currently available.

In short, FIFO-Semi-Ordered scheduler tries to avoid the blocking of smaller queries by larger queries by letting them bypass the queue totally or else bypassing larger queries that are ahead of them in the queue and would otherwise cause them to wait.

### 5.3.3 Wisconsin-V1 Scheduler

The Wisconsin-V1 scheduler's design is inspired by the adaptive scheduler introduced in [56]. Figure 5.2 shows the structure of the Wisconsin-V1 scheduler. This scheduler consists of three queues, each assigned to one of the Small, Medium, and Large query classes. These queues manage the multi-programming level (MPL) of each query class; hence, they are called MPL queues. Each MPL queue is serviced independently in FIFO order. Since in [56] there was no distinction between Small and ZeroMemory query classes, the ZeroMemory-Short and ZeroMemory-Long queries in our workload will be grouped with the small queries and enter their MPL queue. The Wisconsin-V1 scheduler dynamically adjusts the MPL of each query class to increase the fairness among different query classes to the maximum extent possible. Each MPL queue also has a minimum MPL value of one to prevent starvation. The incoming queries of each query class wait in their corresponding MPL queue if the current MPL of the class has reached a dynamically determined level. Whenever a query from each query class finishes, the next query waiting in the MPL queue of the same query class will be moved to a different queue named the *resources queue*. Queries may still have to wait in the resources queue if their required resources are not currently available. The resources queue is serviced in FIFO order as well.

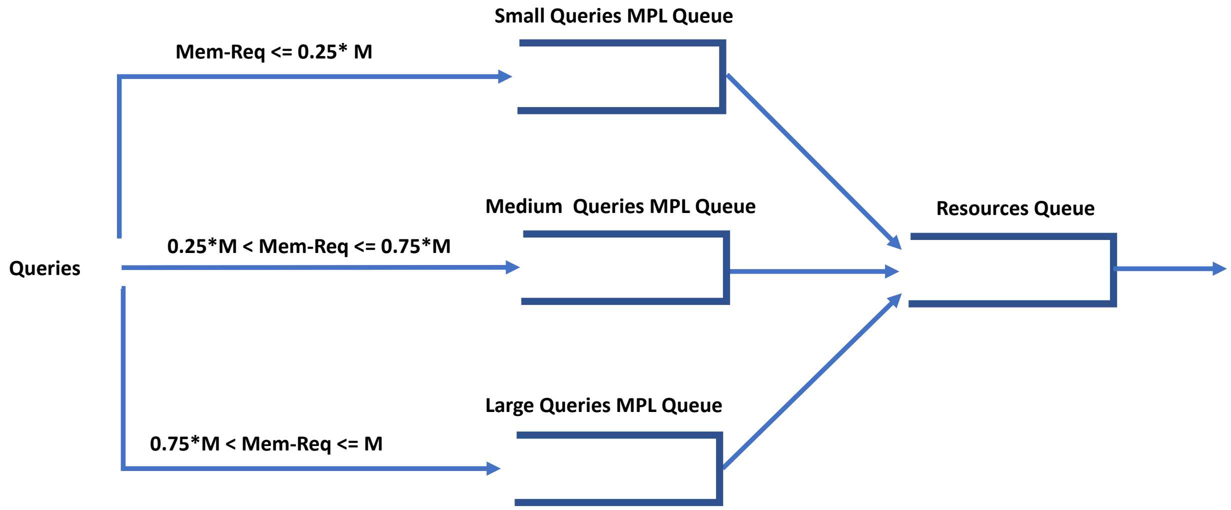
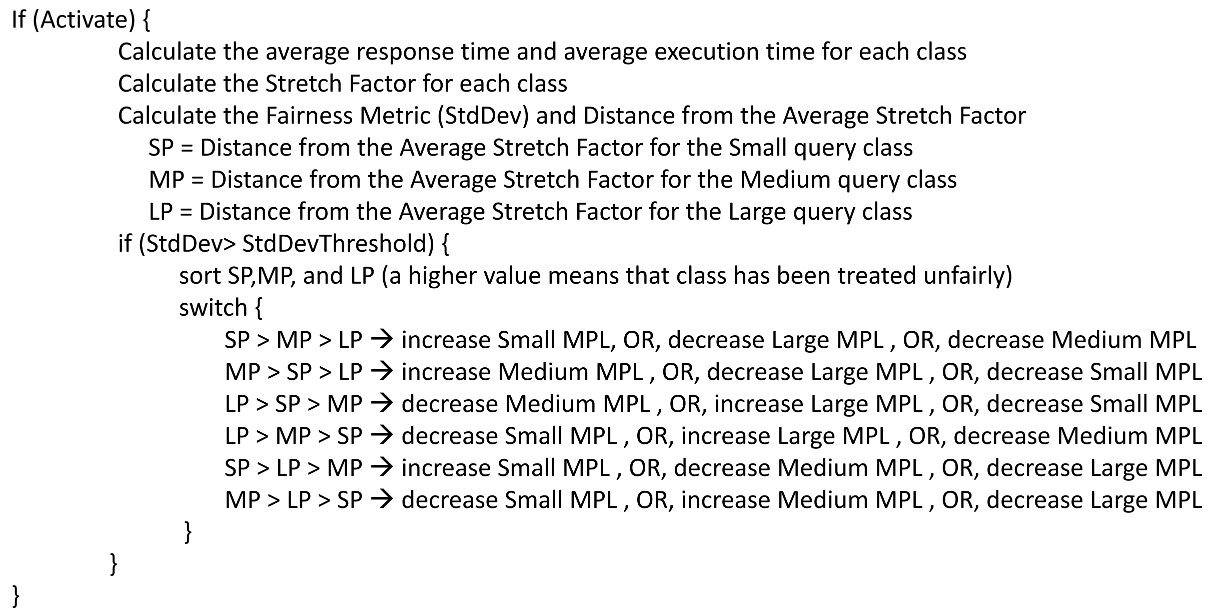


Figure 5.2: Wisconsin-V1 Scheduler Structure

Similar to [56], the calculation of fairness and the adjustments of the MPL values are done after the completion of each medium query. We call this the *activation* condition.

Similar to [56], in Wisconsin-V1 scheduler, slowdown is defined as the ratio of the observed average response time to the average standalone response time for each class and fairness is defined as the standard deviation of the slowdowns for all query classes. The fairness metric for the Wisconsin-V1 scheduler is determined by the standard deviation in the average response time ratio to each query class's execution time. If the standard deviation is beyond a predefined threshold, this means that some classes are doing better than others, and hence



```
If (Activate) {
            Calculate the average response time and average execution time for each class
            Calculate the Stretch Factor for each class
            Calculate the Fairness Metric (StdDev) and Distance from the Average Stretch Factor
               SP = Distance from the Average Stretch Factor for the Small query class
               MP = Distance from the Average Stretch Factor for the Medium query class
               LP = Distance from the Average Stretch Factor for the Large query class
            if (StdDev> StdDevThreshold) {
                  sort SP,MP, and LP (a higher value means that class has been treated unfairly)
                  switch {
                       SP > MP > LP → increase Small MPL, OR, decrease Large MPL , OR, decrease Medium MPL
                       MP > SP > LP → increase Medium MPL , OR, decrease Large MPL , OR, decrease Small MPL
                       LP > SP > MP → decrease Medium MPL , OR, increase Large MPL , OR, decrease Small MPL
                       LP > MP > SP → decrease Small MPL , OR, increase Large MPL , OR, decrease Medium MPL
                       SP > LP > MP → increase Small MPL , OR, decrease Medium MPL , OR, decrease Large MPL
                       MP > LP > SP → decrease Small MPL , OR, increase Medium MPL , OR, decrease Large MPL
                   }
            }
}
```

Figure 5.3: Pseudo-code for Wisconsin-V1 Scheduler

the scheduler has not been fair. Upon detecting such unfairness conditions, the scheduler takes action by either increasing the MPL of classes that scheduler has been unfair to or by decreasing the MPL of classes that the scheduler has favored more than others. Figure 5.3 shows the pseudo-code for the Wisconsin-V1 scheduler algorithm.

### 5.3.4 Wisconsin-V2 Scheduler

The Wisconsin-V2 is another scheduler inspired by [56]. This scheduler separates the ZeroMemory-Short and ZeroMemory-Long queries from the Small queries by directly adding them to the resources Queue instead of first placing them in an MPL queue.

The reason for separating ZeroMemory queries from the Small queries is that the memory requirement for ZeroMemory queries is likely be much less than the memory requirements of Small queries. Additionally, they may have very different execution times than the Small queries, which makes them unsuitable for being treated the same as Small queries. Figure 5.4 shows the structure of the Wisconsin-V2 scheduler. The idea of this scheduler is similar

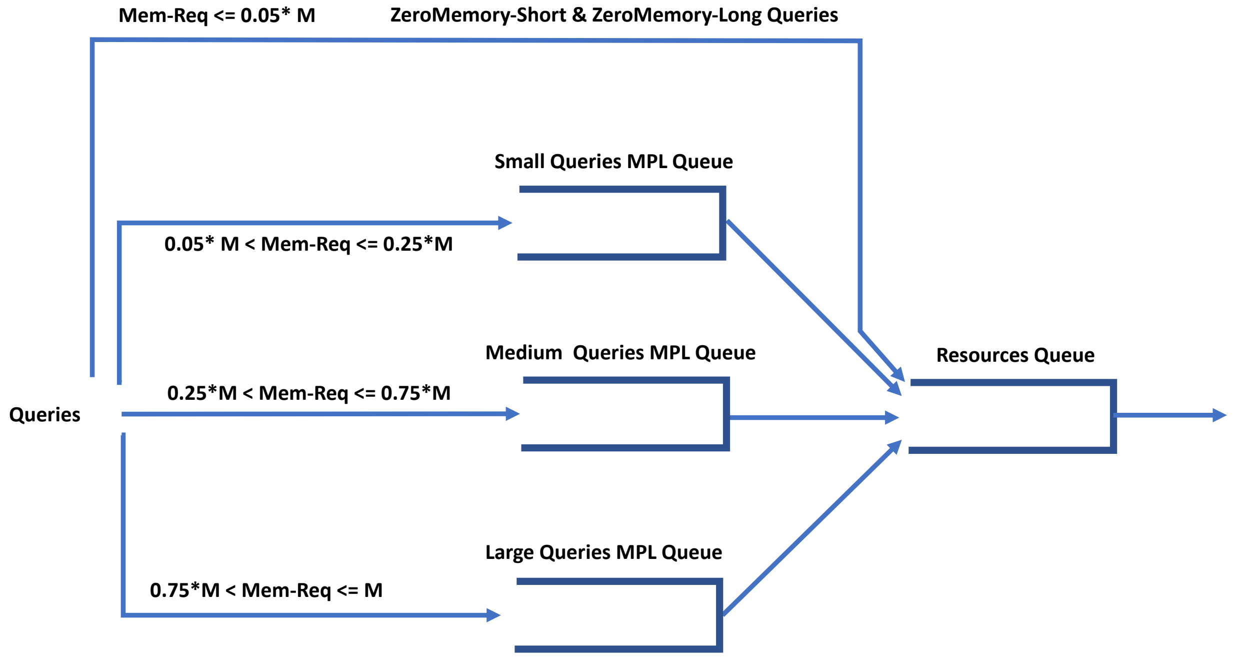


Figure 5.4: Wisconsin-V2 Scheduler Structure

to the *responsible* scheduler in [56] in which the Small queries would skip any MPL queues and would be directly added to the resources queue. We use the same approach used in Wisconsin-V1 scheduler for adjusting the MPL values in the Wisconsin-V2 scheduler.

### 5.3.5 Wisconsin-V3 Scheduler

The Wisconsin-V3 scheduler is another scheduler implemented and evaluated in this chapter that is inspired by [56] as well.

As Figure 5.5 shows, the Wisconsin-V3 scheduler consists of one MPL queue for each of the five query classes. The main motivation for the Wisconsin-V3 scheduler is to separate the ZeroMemory-Short class from other query classes, as its execution times are significantly lower than those of other classes.

We use the same fairness metric as used for the Wisconsin-V1 and Wisconsin-V2 schedulers; However, we adjust the MPL queues more frequently by evaluating fairness upon comple-

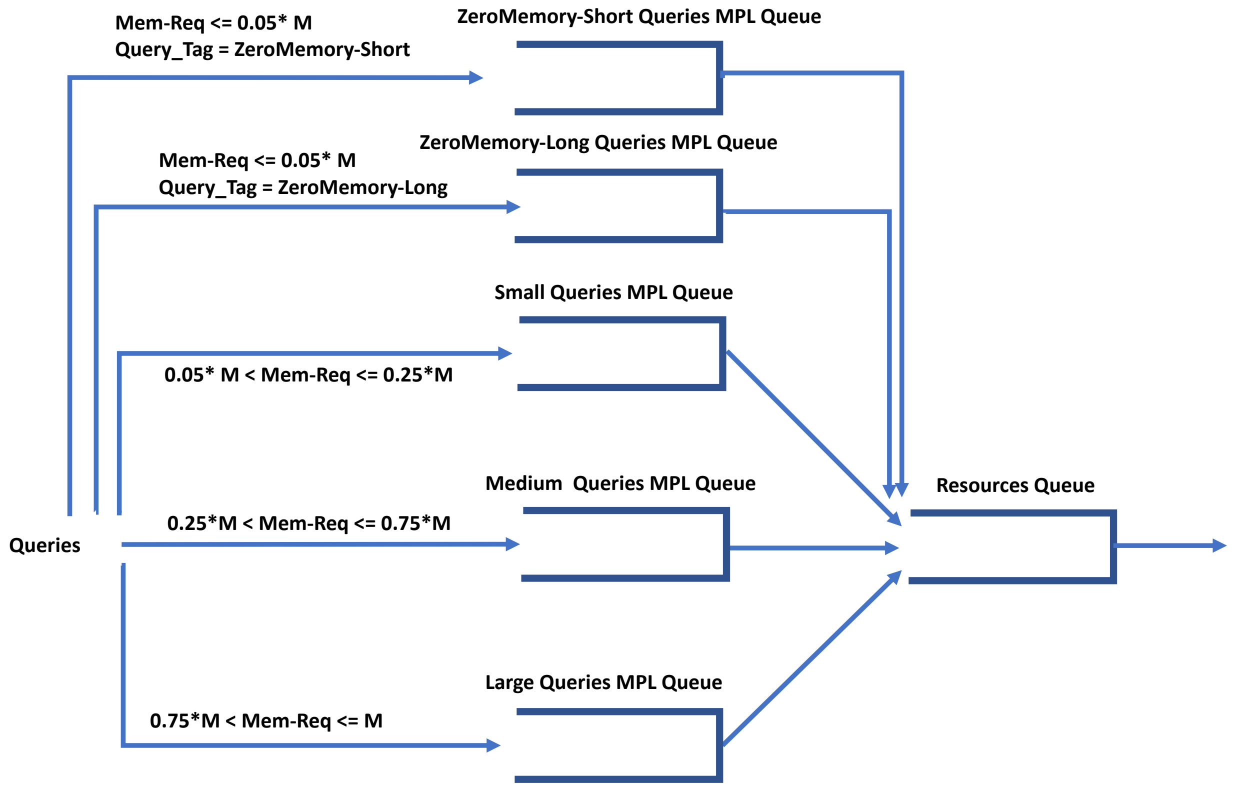
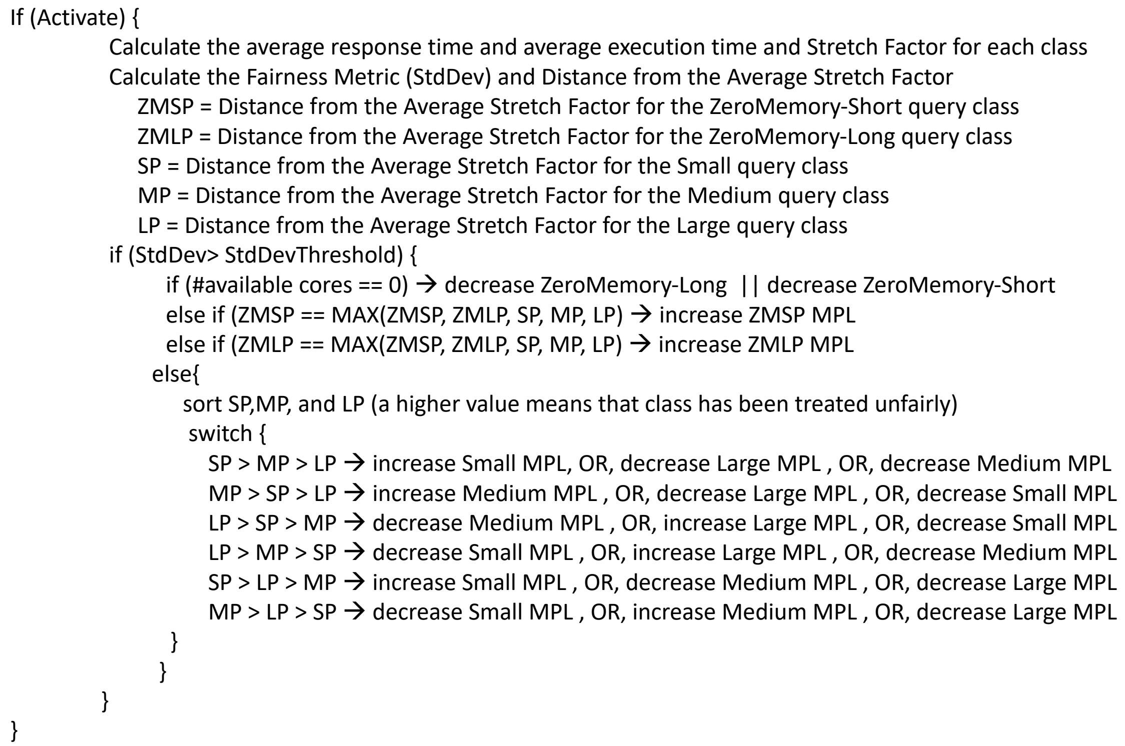

Mem-Req <= 0.05* M
Query_Tag = ZeroMemory-Short
ZeroMemory-Short Queries MPL Queue
Mem-Req <= 0.05* M
Query_Tag = ZeroMemory-Long
ZeroMemory-Long Queries MPL Queue
Small Queries MPL Queue
0.05* M < Mem-Req <= 0.25*M
Medium  Queries MPL Queue
Resources Queue
Queries
0.25*M < Mem-Req <= 0.75*M
Large Queries MPL Queue
0.75*M < Mem-Req <= M

Figure 5.5: Wisconsin-V3 Scheduler Structure

tion of each Small query. During the evaluation of fairness, we increase the MPL value of ZeroMemory-Short and ZeroMemory-Long query class if they have the maximum positive distance from the average stretch factor. However, increasing the MPL of the ZeroMemory-Short or ZeroMemory-Long could allow them to occupy all of the CPU cores and block other queries from running even if enough memory is available. To avoid this blocking and starvation problem, we decrease the MPL of the ZeroMemory-Long class, and if needed the MPL of the ZeroMemory-Short class, if all the CPU cores are occupied. The rest of the fairness enforcement algorithm of the Wisconsin-V3 scheduler is similar to the previous two variations. Figure 5.6 shows the pseudo-code of the algorithm used for the Wisconsin-V3 scheduler.



```
If (Activate) {
            Calculate the average response time and average execution time and Stretch Factor for each class
            Calculate the Fairness Metric (StdDev) and Distance from the Average Stretch Factor
                ZMSP = Distance from the Average Stretch Factor for the ZeroMemory-Short query class
                ZMLP = Distance from the Average Stretch Factor for the ZeroMemory-Long query class
                SP = Distance from the Average Stretch Factor for the Small query class
                MP = Distance from the Average Stretch Factor for the Medium query class
                LP = Distance from the Average Stretch Factor for the Large query class
            if (StdDev> StdDevThreshold) {
                    if (#available cores == 0) → decrease ZeroMemory-Long  || decrease ZeroMemory-Short
                    else if (ZMSP == MAX(ZMSP, ZMLP, SP, MP, LP) → increase ZMSP MPL
                    else if (ZMLP == MAX(ZMSP, ZMLP, SP, MP, LP) → increase ZMLP MPL
                  else{
                      sort SP,MP, and LP (a higher value means that class has been treated unfairly)
                       switch {
                          SP > MP > LP → increase Small MPL, OR, decrease Large MPL , OR, decrease Medium MPL
                          MP > SP > LP → increase Medium MPL , OR, decrease Large MPL , OR, decrease Small MPL
                          LP > SP > MP → decrease Medium MPL , OR, increase Large MPL , OR, decrease Small MPL
                          LP > MP > SP → decrease Small MPL , OR, increase Large MPL , OR, decrease Medium MPL
                          SP > LP > MP → increase Small MPL , OR, decrease Medium MPL , OR, decrease Large MPL
                          MP > LP > SP → decrease Small MPL , OR, increase Medium MPL , OR, decrease Large MPL
                     }
                   }
            }
}
```

Figure 5.6: Pseudo-code for Wisconsin-V3 Scheduler

### 5.3.6 Colorado-V1 Scheduler

This scheduler is designed with inspiration from [22]. In this scheduler, a larger number of queues are used to separate further the queries from each other with different memory requirements. We use ten queues, out of which one queue is allocated to the ZeroMemory-Short and ZeroMemory-Long queries, four queues to the Small queries, three queues to the Medium queries, and two queues to the Large queries. Each of these queues is serviced in FIFO order independently from the other queues. Figure 5.7 shows the general structure of the Colorado-V1 scheduler.

In this scheduler, upon arrival of a new query or completion of an existing query, all the queries at the head of each queue are evaluated to be considered as its next query for execution. In this evaluation, we measure the slowdown of the queries at the head of the queues, the query with the highest slowdown value is selected as the next query to execute.

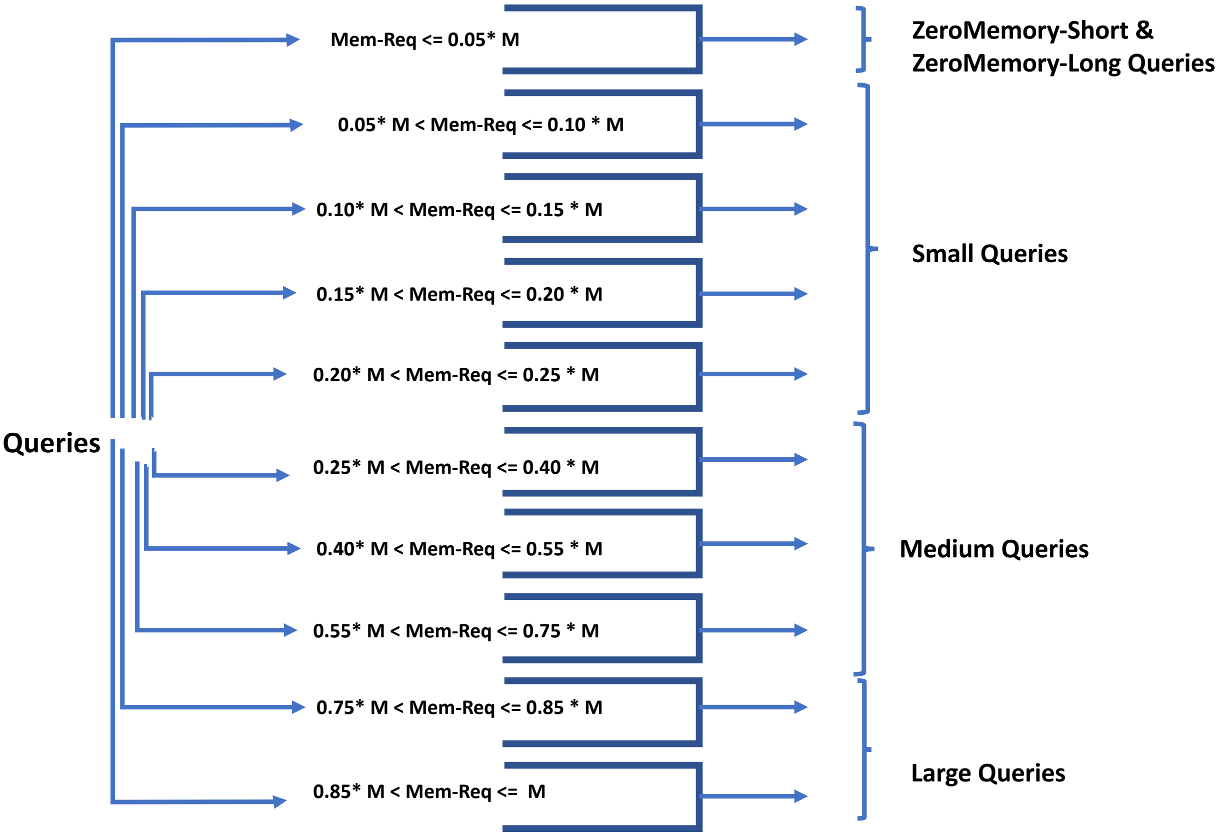

Queries
Mem-Req <= 0.05* M
0.05* M < Mem-Req <= 0.10 * M
0.10* M < Mem-Req <= 0.15 * M
0.15* M < Mem-Req <= 0.20 * M
0.20* M < Mem-Req <= 0.25 * M
0.25* M < Mem-Req <= 0.40 * M
0.40* M < Mem-Req <= 0.55 * M
0.55* M < Mem-Req <= 0.75 * M
0.75* M < Mem-Req <= 0.85 * M
0.85* M < Mem-Req <= M
ZeroMemory-Short &
ZeroMemory-Long Queries
Small Queries
Medium Queries
Large Queries

Figure 5.7: Colorado-V1 Scheduler Structure

The authors of [22] defined slowdown as:

$$slowdown = (waitTime + execTime_{\text{est}})/execTime_{\text{est}} \tag{5.1}$$

The $execTime_{\text{est}}$ is estimated using cost formulas. In our Colorado-V1 scheduler, we define the slowdown as follows:

$$slowdown = (waitTime + averageExecTime_{\text{class}})/averageExecTime_{\text{class}}) \tag{5.2}$$

The reason for choosing the query class's average execution time as opposed to estimated execution time is that the current version of AsterixDB does not have a cost-based optimizer or cost functions to estimate the execution time for us.

### 5.3.7 Colorado-V2 Scheduler

The Colorado-V2 Scheduler is another scheduler inspired by [22]. In the Colorado-V1 scheduler, all queries of types ZeroMemory-Short and ZeroMemory-Large are directed to the same queue since they have similar memory requirements; however, they may have very different execution times. Thus, by putting ZeroMemory-Short and ZeroMemory-Long queries in the same queue, ZeroMemory-Short queries may have to wait for a long duration behind ZeroMemory-Long queries. To avoid this problem, the Colorado-V2 scheduler separates the ZeroMemory-Short and ZeroMemory-Long queries by directing them to two different queues. To enable this, each query carries a query tag that indicates which query class it belongs to. In the future, we will remove the query tag concept and use the currently under-implementation statistics and cost functions in AsterixDB to estimate the number of records a query accesses to determine whether it is a short or long query. Figure 5.8 shows

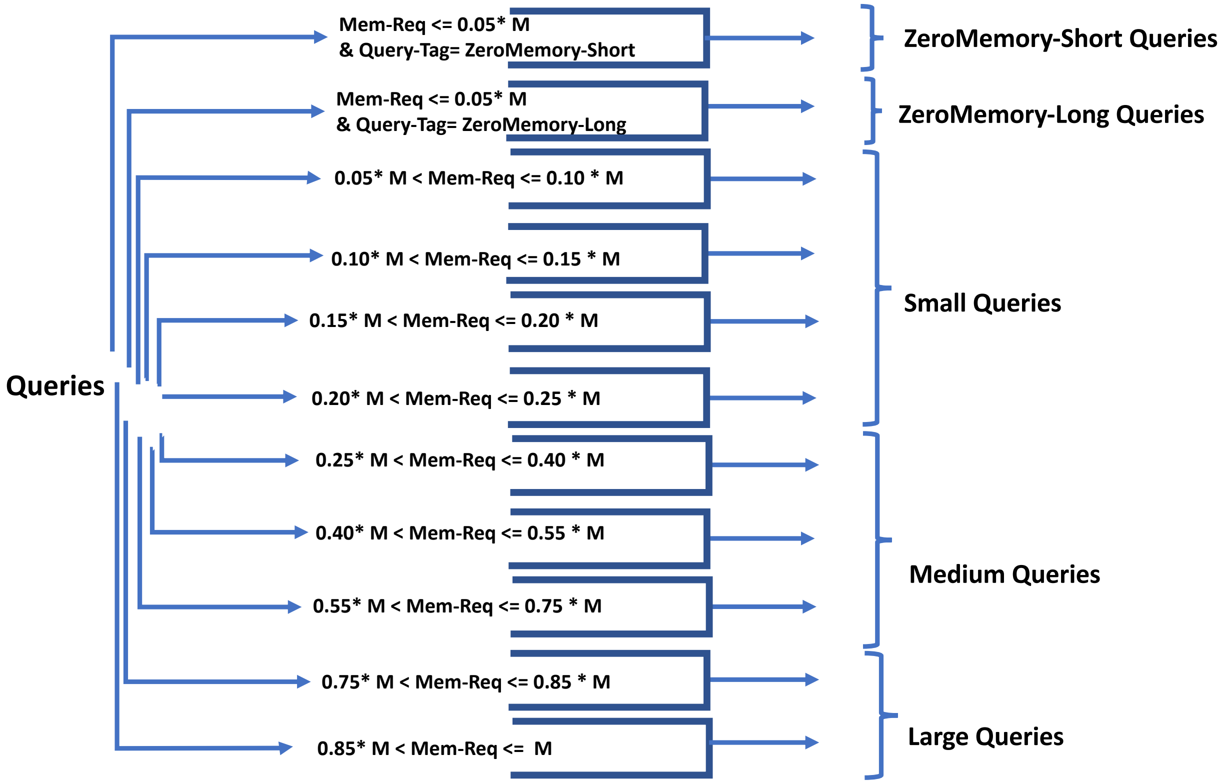

Queries
Mem-Req <= 0.05* M
& Query-Tag= ZeroMemory-Short
Mem-Req <= 0.05* M
& Query-Tag= ZeroMemory-Long
0.05* M < Mem-Req <= 0.10 * M
0.10* M < Mem-Req <= 0.15 * M
0.15* M < Mem-Req <= 0.20 * M
0.20* M < Mem-Req <= 0.25 * M
0.25* M < Mem-Req <= 0.40 * M
0.40* M < Mem-Req <= 0.55 * M
0.55* M < Mem-Req <= 0.75 * M
0.75* M < Mem-Req <= 0.85 * M
0.85* M < Mem-Req <= M
ZeroMemory-Short Queries
ZeroMemory-Long Queries
Small Queries
Medium Queries
Large Queries

Figure 5.8: Colorado-V2 Scheduler Structure

the overall structure of the Colorado-V2 scheduler.

## 5.4 Experiments

In this section, we empirically evaluate the aforementioned scheduling algorithms using workloads containing queries with widely varying memory requirements and execution times.

### 5.4.1 Experiment Settings

For this set of experiments, we use an AsterixDB cluster containing a Cluster Controller (CC) and a Node Controller(NC). Both instances are chosen from the US-West-2 availability zone of AWS and have 4 vCPUs and 30.5GB of RAM. We used the i3.xlarge instance type for the NC which has an attached SSD storage device. The NC has four data partitions, meaning that four sub-instances of each query will execute in parallel, each on one data partition. The core-multiplier is by default set to three, thus at most three such parallel queries may run concurrently if enough memory is available.

**Workload Descriptions**

In this set of experiments, multiple users issue queries to the DBMS concurrently. For simplicity, each user sends queries for one query class only. Each user issues their next query only after receiving the results of their previous query. For simplicity, the thinking time of the users between two consecutive queries is set to zero. We plan to consider variant thinking times relative to the query size in future work.

Section 5.2 discussed the nature of the queries used in this chapter. The workload for these experiments contains twelve, three, ten, eight, and four users sending ZeroMemory-Short, ZeroMemory-Long, Small, Medium, and Large queries, respectively. Each user is modeled by a thread that creates a query based on the query template for its class and the range of predicate values associated with that query class. All of the user threads start at the same time. Each experiment runs for precisely four hours. Only queries that have finished their execution during these four hours will show up in the results. Queries for each class are sent to AsterixDB in a random order every time an experiment starts its execution.

### Datasets and Benchmark

We created five datasets containing identical data for this set of experiments. We used the schema from Wisconsin Benchmark [23] and the Wisconsin Benchmark JSON Data Generator [42] for data generation. Each dataset consists of 40 GB of data with records of length 1073 Bytes. Each ZeroMemory query chooses one dataset to query, and each Small, Medium, and Large query chooses two datasets randomly from these five datasets. As mentioned earlier, each query accesses a specific number of records; however, ranges are chosen randomly for each query.

### Metrics

For this set of experiments, we have used six metrics to evaluate and compare the performance of the different scheduling algorithms from various angles. The description of these metrics is as follows:

- **Throughput.** We report two types of throughput values for each experiment. The first throughput metric is the “Total” throughput, which reports the number of queries that the system executed per minute (across all classes). We calculate the ”Total” throughput by dividing the total number of executed queries by 240 minutes, the duration of the experiment. The Throughputs per query class is the other throughput metric that we consider, which reports the total number of queries of each query class that was executed per minute on average. We calculate the throughput value for each query class by dividing the total number of queries belonging to each class by 240 minutes.

- **Average Execution Time.** This metric reports the average execution time of queries of each query class in a concurrent environment. As mentioned earlier, each query's execution time is the duration when the query was actively executing. We used the average execution time of queries in a concurrent setting rather than their standalone execution times since finding the standalone execution times would require either running queries of each query class individually or having a cost function to predict the standalone execution time accurately. Running candidate queries of each query class individually would not be possible in practice and would add a pre-step for a scheduler even if it were possible. Thus, considering the average execution time of queries from each query class that have completed so far is a more realistic and practical approach since it includes all queries from a query class in the calculation, and they may have different execution times. Additionally, the impact of other concurrent queries on the execution time of each query is captured using this approach.

- **Average Response Time** This metric shows the average response time of queries belonging to each query class. As mentioned earlier, the response time of a query is the duration measured from when a user first sends a query until the results are returned. Thus, the response time includes both the execution time of a query and its waiting time in the queues.

- **Stretch Factor.** As mentioned earlier, this metric indicates how long the queries of each query class have waited in the queue compared to their class average execution times. We use the ratio of the average response time of a query class to its average execution time to calculate the stretch factor for each query class. The stretch factor is a positive number with a minimum value of 1. A scheduler will be considered to be fair if all the query classes have a similar stretch factor. A low stretch factor for each query class is desired.

- **Distance From Average Stretch Factor.** This metric indicates the distance of each query class's stretch factor from the average stretch factor of all query classes. A scheduler has favored query classes with a lower distance from the average stretch factor, as a smaller stretch factor means that they have waited less in the queue in portion to their execution time. On the contrary, a scheduler has discriminated against query classes that have a large distance from the average stretch factor since their wait time in the queue is larger relative to their execution time, meaning they have waited in line for a longer duration than they should have. Ideally, the actions of a fair scheduler will lead to all classes having similar or close values for this metric for all query classes.

- **Fairness.** As mentioned earlier, the fairness metric shows how fair a scheduler has been in terms of scheduling the queries from different classes so they all wait in the queues for a duration proportional to their execution times. We calculate the fairness value for each scheduler by computing the standard deviation of the stretch factor across all query classes. The lower this fairness value is, the more fair a scheduler is being.

Given these metrics, we can now proceed to explore the seven alternative scheduling algorithms of section 5.3.

### 5.4.2 Evaluation of the FIFO-Ordered Scheduler

For the first experiment, we use the basic FIFO-Ordered scheduler as the scheduling policy in AsterixDB. Figures 5.9, 5.10, and 5.11 show timeline plots for the duration of four hours where users were issuing queries to AsterixDB. We show the users sending ZeroMemory-Short

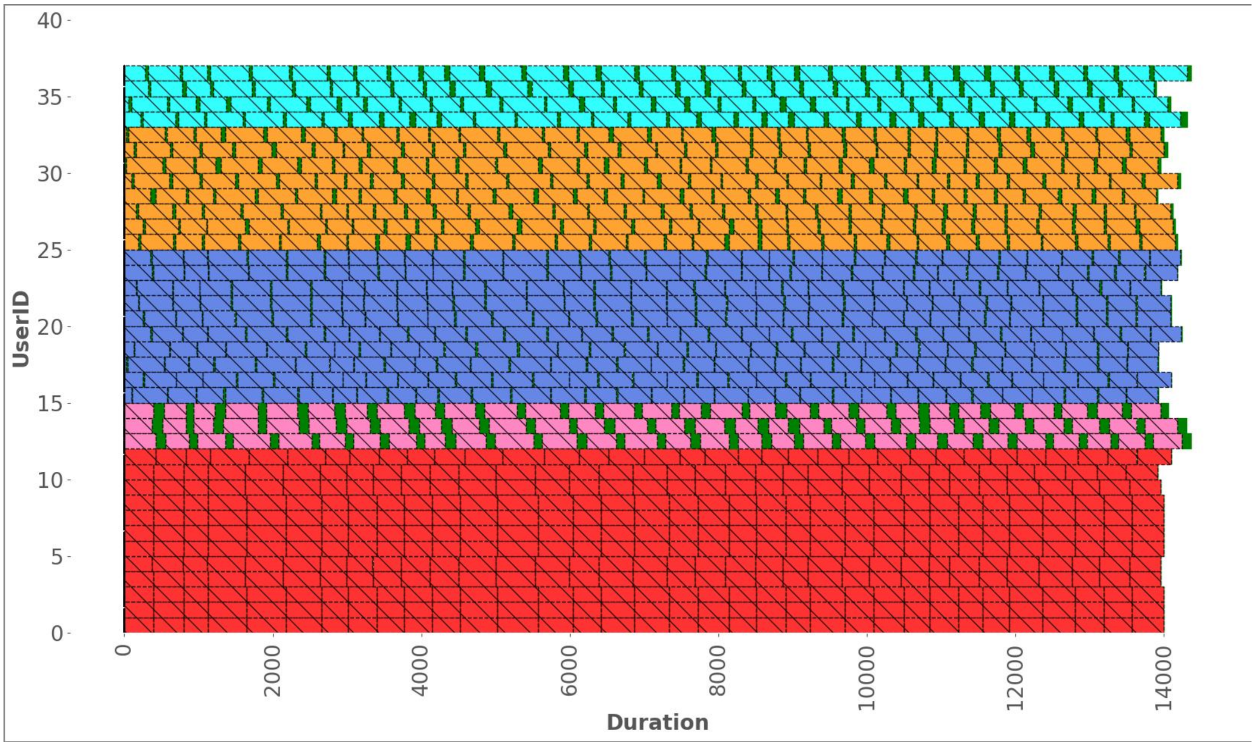


Figure 5.9: FIFO-Ordered Scheduler - Timeline Plot for core-multiplier = 3

queries with the red color, users sending ZeroMemory-Long queries with the blue color, users sending small queries with the pink color, users sending Medium queries with the orange color, and users sending Large queries with the turquoise color. The diagonal lines in each user's query line represent the duration for which its query was waiting in the queue. The green sections for each user show the periods where each query was executing. For the queries with very short execution times their execution duration is shown by a straight black line.

Since in the FIFO-Ordered scheduler all of the queries are added to a single queue and executed in FIFO order, queries with shorter execution times may wait behind queries with longer execution times. This phenomenon is visible in Figures 5.10, and 5.11 where all queries have a similar waiting times regardless of their execution times.

Increasing the core-multiplier from three to twelve decreases the query admission limitations imposed by the CPU on the number of queries that can run concurrently and allows for up

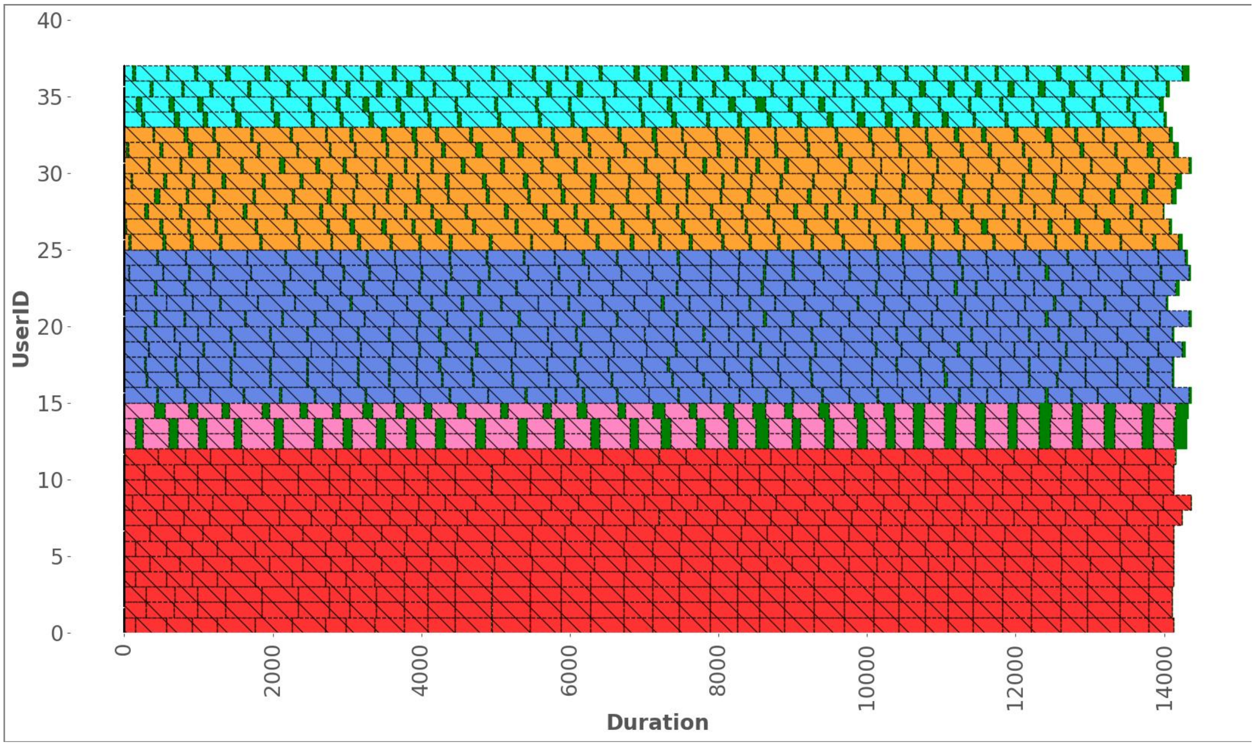


Figure 5.10: FIFO-Ordered Scheduler - Timeline Plot for core-multiplier = 12

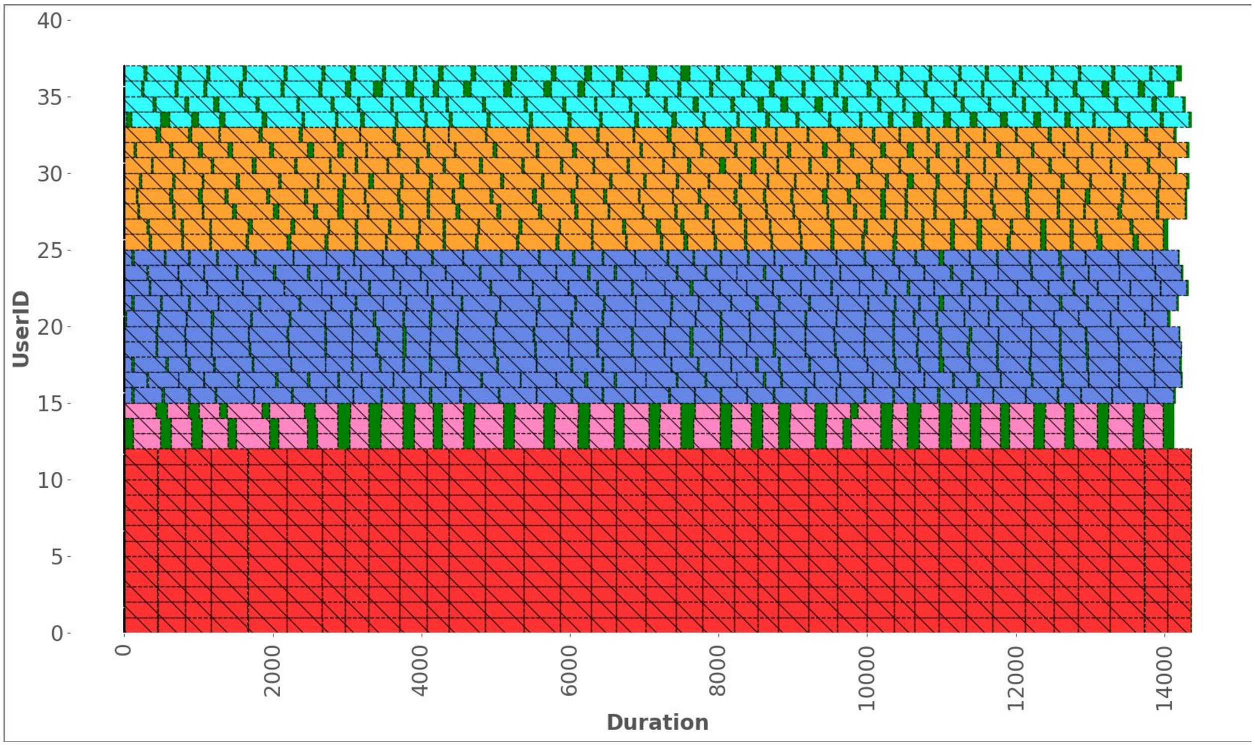


Figure 5.11: FIFO-Ordered Scheduler - Timeline Plot for core-multiplier = 48

to twelve concurrent queries if memory is not a bottleneck. Increasing the core-multiplier to forty-eight lifts any restrictions at all caused by the CPU resource since it allows up to 48 queries to run concurrently, while in this experiment there are only thirty-seven users each with one query present in the system at a time, either executing or waiting in the queue.

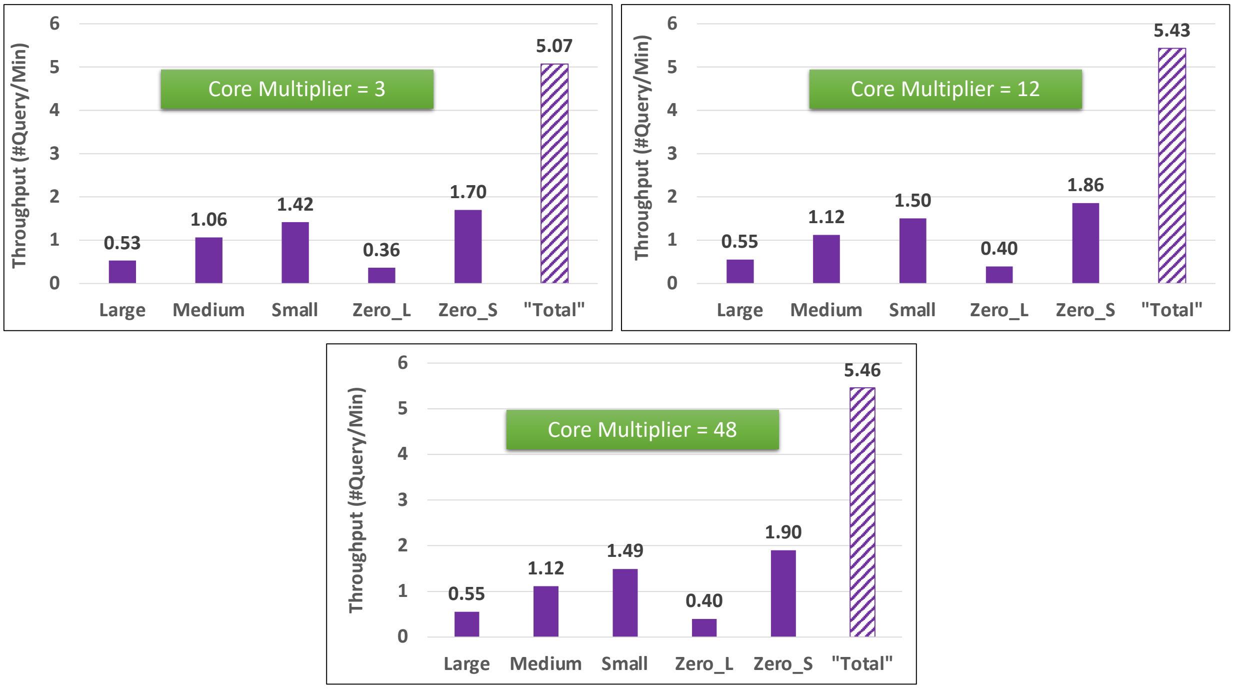


Figure 5.12: FIFO-Ordered Scheduler - Throughput

Figure 5.12 shows the total and per-class throughputs of the system for different core-multipliers. The ZeroMemory-Short and Small query classes have slightly higher throughputs than the other query classes due to having shorter execution times and having more users associated with those query classes issuing queries to the system. Reducing the CPU restrictions improves the total and the per-class throughput values since more queries run concurrently. The throughput improvement is higher for the query classes with shorter execution times and the query classes with lower memory requirements since more of them can be admitted concurrently before the CPU or memory resources reach their limitations with core-multipliers of twelve and forty-eight, respectively. Having more concurrent queries executing in the system leads the average execution times of the query classes with lower memory requirements to increase since they execute concurrently with larger queries. Figure 5.13 shows the impact of increasing the concurrency on the average execution times of the different query classes.

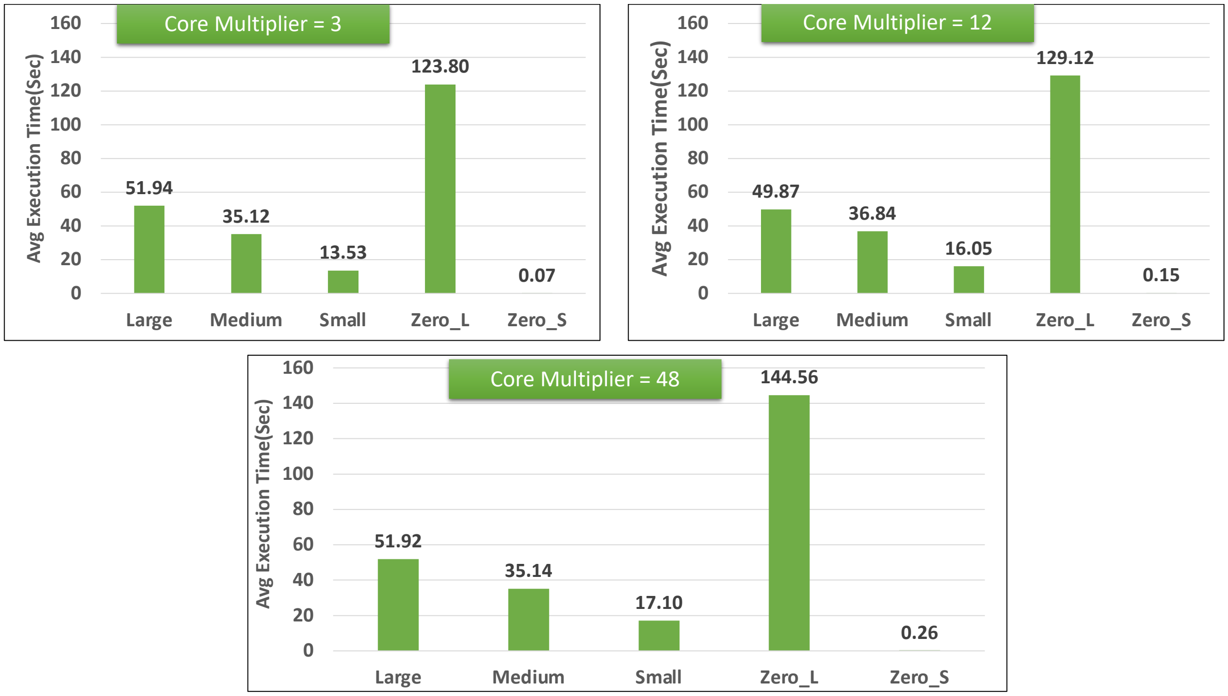


Figure 5.13: FIFO-Ordered Scheduler - Average Execution Time

The average response times for queries of each query class running in a concurrent environment with other queries are shown in Figure 5.14. The similarity in the average response times of the different query classes is because all of the queries, regardless of their sizes, wait in the same queue prior to their execution. The average response times of the ZeroMemory-Short and Small query classes are slightly lower than the average response times of the other query classes due to their shorter execution times once they are allowed to begin running.

We now begin our analysis of the fairness of the FIFO-Ordered scheduler. Figure 5.15 illustrates that the ZeroMemory-Short query class has the highest stretch factor –by far– in all three core-multiplier values. This is because the ZeroMemory-Short queries have a very short execution time, but they must first stay behind the larger queries. Queries from

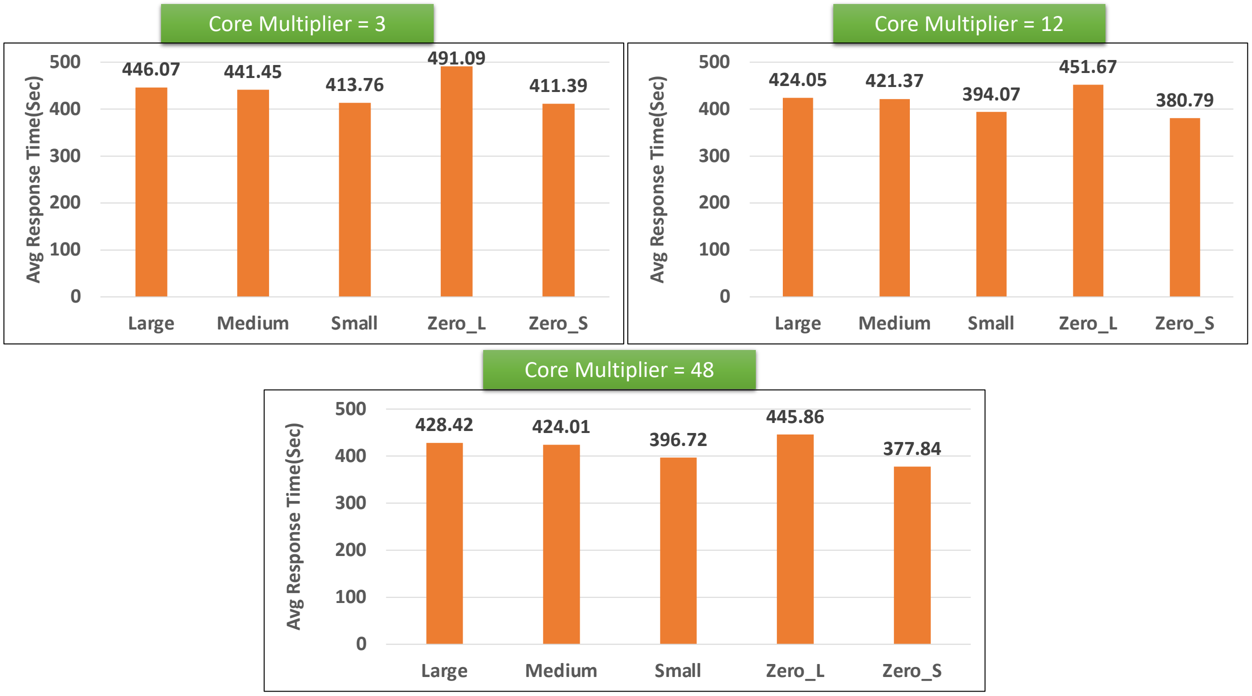


Figure 5.14: FIFO-Ordered Scheduler - Average Response Time

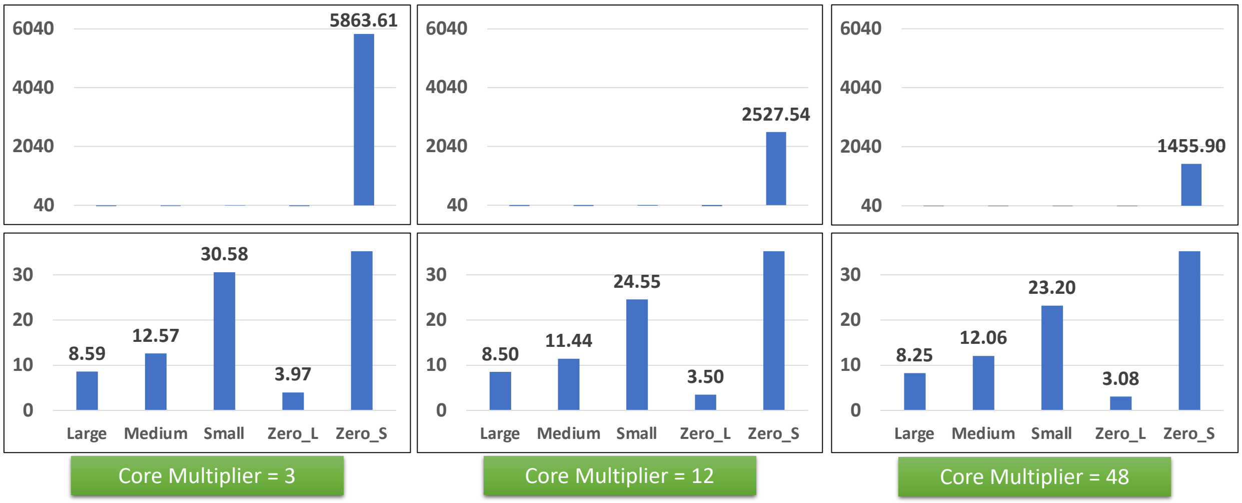


Figure 5.15: FIFO-Ordered Scheduler - Stretch Factor

the Small query class have the second-highest stretch factor under this scheduling policy. Figure 5.16 displays that the FIFO-Ordered schedule is very unfair to the queries from the

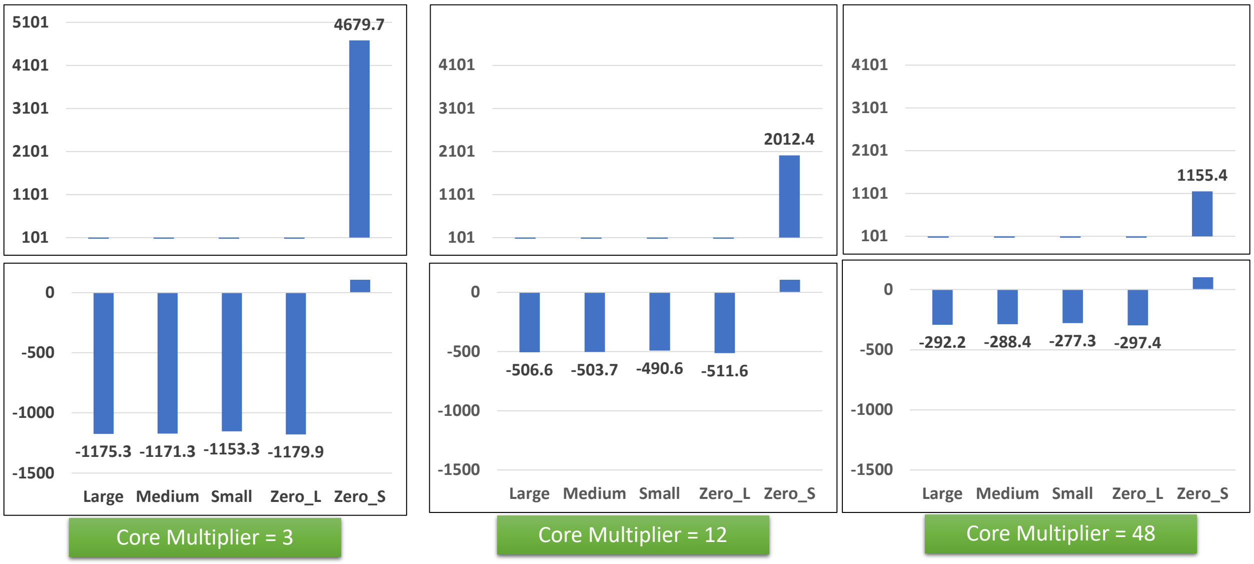


Figure 5.16: FIFO-Ordered Scheduler - Distance From Average Stretch Factor

ZeroMemory-Short query class. In this figure, the query classes with a higher value for the distance from the average stretch factor have been unfairly treated by the scheduler as their wait times in the queue has been disproportional to their execution times. Conversely, the scheduler has favored the query classes that have a lower value for distance from average stretch factor since they have been kept in the queue for shorter durations that they should have based on the average execution time of their query class.

### 5.4.3 Evaluation of the FIFO-Semi-Ordered Scheduler

In this experiment, we use the same query and workload settings as the previous experiment to evaluate the performance of the FIFO-Semi-Ordered scheduler using the different metrics. The execution profile of the users' queries throughout the experiment withe the core-multiplier of three is displayed in Figure 5.17. Comparison of this figure with Figure 5.9 indicates that the queries with shorter execution times wait in the queue for a shorter duration here since they can start their execution by skipping the line totally or partially. (Generating the Timeline plots for core-multipliers of 12 and 48 was not possible in a timely manner due to the large number of executed queries and, thus the numerous data points to be drawn.)

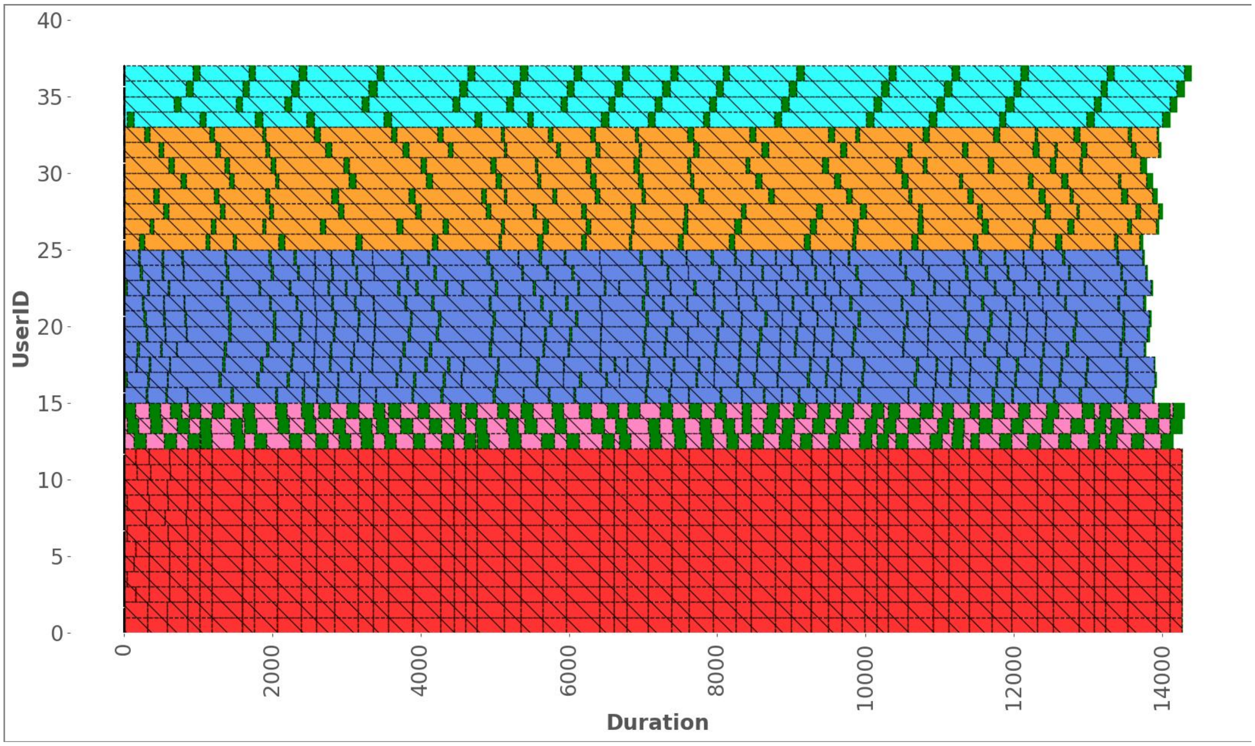


Figure 5.17: FIFO-Semi-Ordered Scheduler - Timeline Plot for core-multiplier = 3

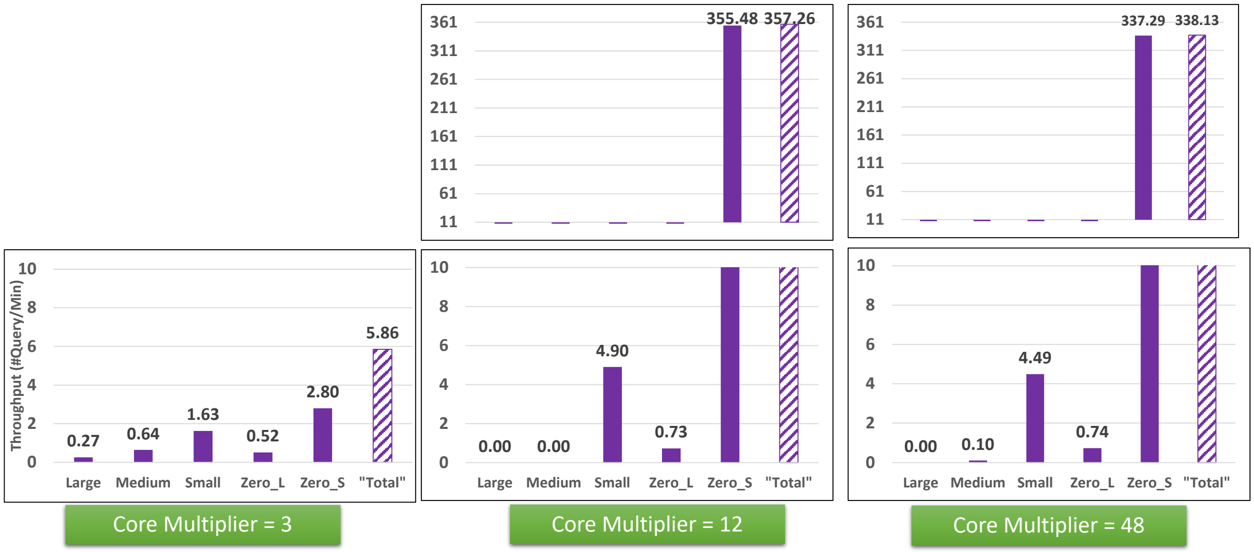


Figure 5.18: FIFO-Semi-Ordered Scheduler - Throughput

Figure 5.18 represents the total and per-class throughputs of the system when the FIFO-Semi-Ordered scheduler is employed. With the core-multiplier set to three, the results for

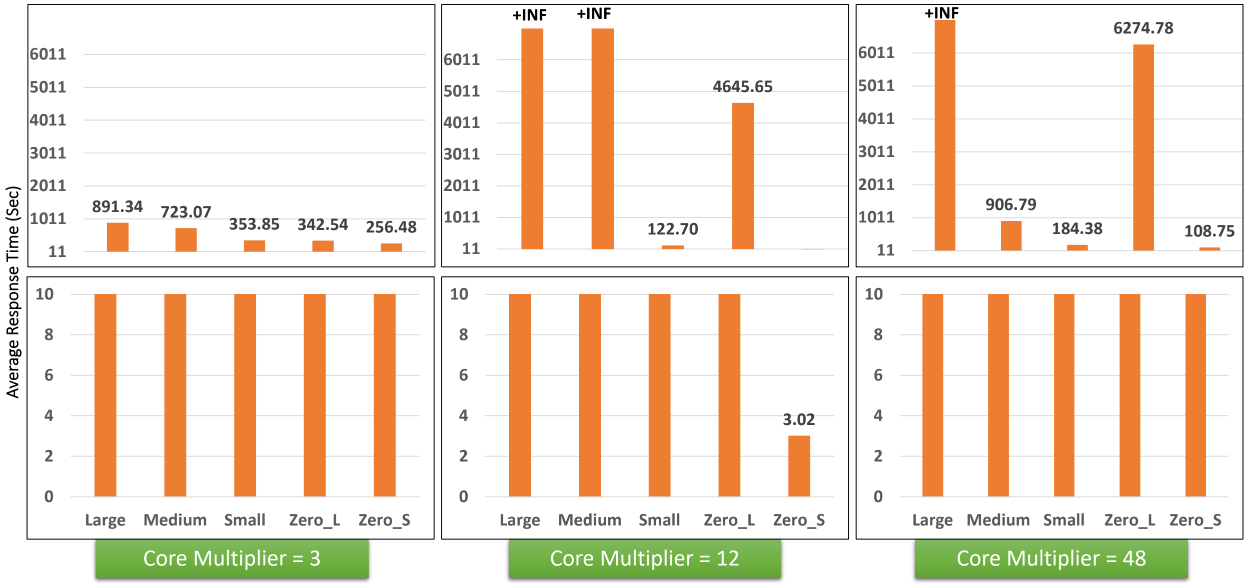


Figure 5.19: FIFO-Semi-Ordered Scheduler - Average Response Time

the total and per-class throughputs of the FIFO-Semi-Ordered scheduler are very similar to the results of the FIFO-Ordered scheduler.

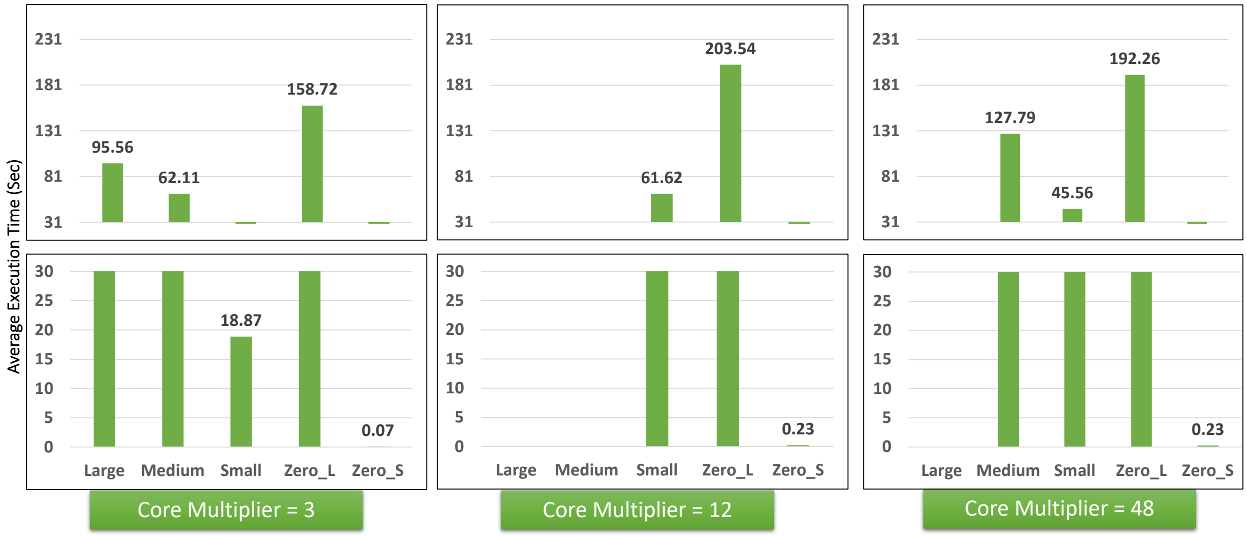


Figure 5.20: FIFO-Semi-Ordered Scheduler - Average Execution Time

Under the FIFO-Semi-Ordered scheduler, the ZeroMemory-Short and Small query classes with their shorter execution times and higher numbers of users associated with them have higher throughput than the other query classes. Since the FIFO-Semi-Ordered scheduler allows the queries with lower resource requirements, including ZeroMemory-Short and ZeroMemory-Long to skip the line and start their executions earlier, their query classes have slightly higher throughputs under the FIFO-Semi-Ordered scheduler than with the FIFO-Ordered scheduler.

Increasing the core-multiplier reduces the restrictions on CPU and leads to executing more queries of the ZeroMemory-Short type. Although the ZeroMemory-Long queries also have a very low resource requirement, there are only three users of this query class in this experiment; thus, the throughput of ZeroMemory-Long query class is not as high as the throughput for ZeroMemory-Short or Small query classes.

In the case of setting the core-multiplier to twelve, the majority of the CPU capacity is used by ZeroMemory-Short queries which leads to the starvation of queries from the Medium and Large query classes. Lifting the CPU limitations entirely by setting the core-multiplier to forty-eight provides a better chance for Medium queries to execute; however, in this case, memory becomes the bottleneck due to executing many queries concurrently. The Large query class thus starves even when the CPU is not a limiting factor. Note also that, the ”Total” throughput is very high when core-multiplier is set to twelve or forty-eight in Figure 5.18, even though some of the query classes have executed zero queries. This indicates that the ”Total” throughput is not a good metric to use for comparing and evaluating schedulers with multi-class workloads.

A comparison of Figures 5.19 and 5.20 outlines that the wait times of the query classes are definitely not proportional to the their execution times.

As Figure 5.21 represents, the ZeroMemory-Short has the highest stretch factor when the core-multiplier is set to three. This is because the ZeroMemory-Short queries have a very small execution time, making them very sensitive to even the slightest wait time. With a higher core-multiplier value, the starved query classes have the highest stretch factors.

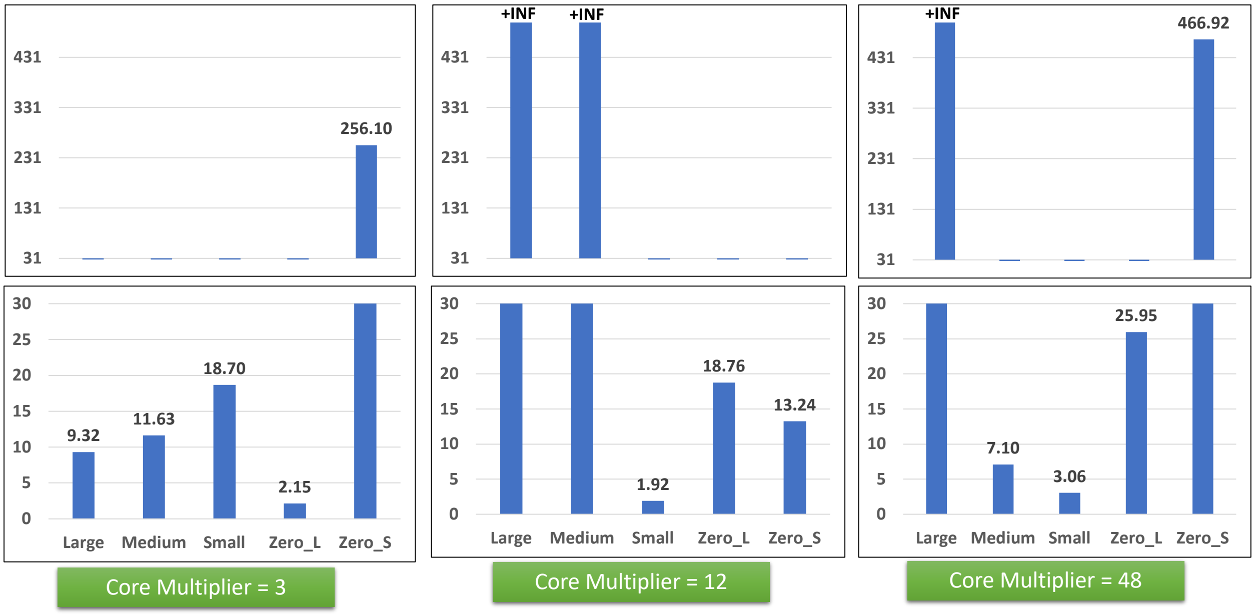


Figure 5.21: FIFO-Semi-Ordered Scheduler - Stretch Factor

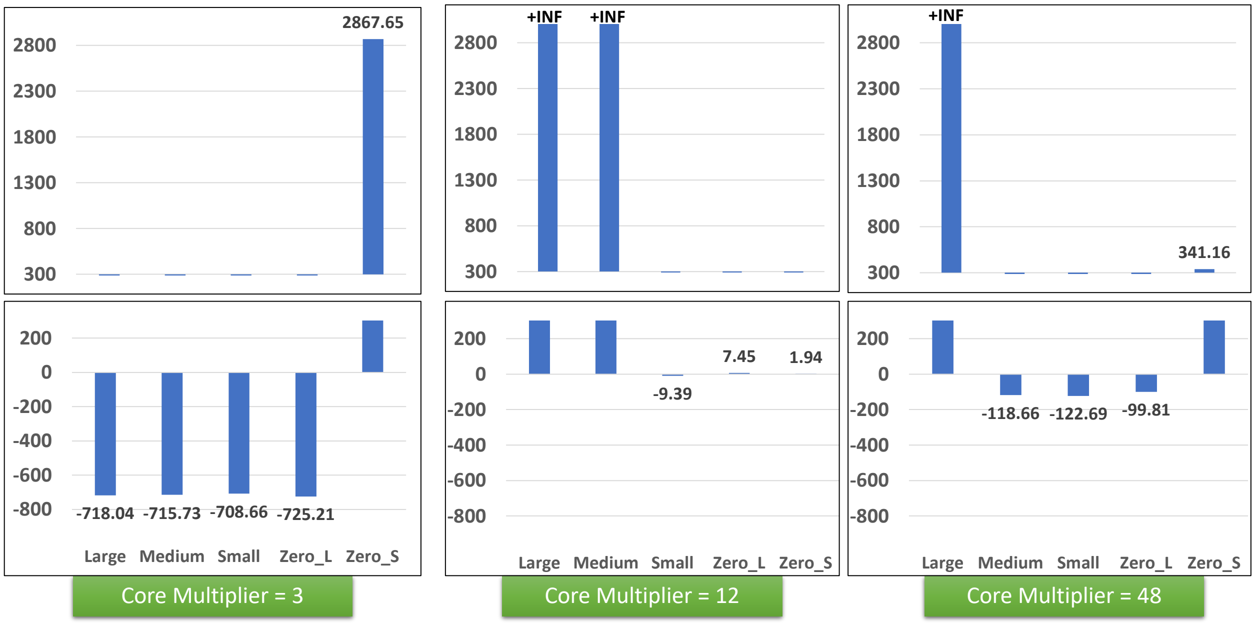


Figure 5.22: FIFO-Semi-Ordered Scheduler - Distance From Average Stretch Factor

The FIFO-Semi-Ordered scheduler cannot be considered a fair scheduler since the improvement in the throughputs of the smaller query classes leads to the starvation of the larger query classes. Figure 5.22 displays the distance from the average stretch factor for the various query classes.

### 5.4.4 Evaluation of the Wisconsin-V1 Scheduler

We use the Wisconsin-V1 scheduler as the scheduling algorithm in this section.

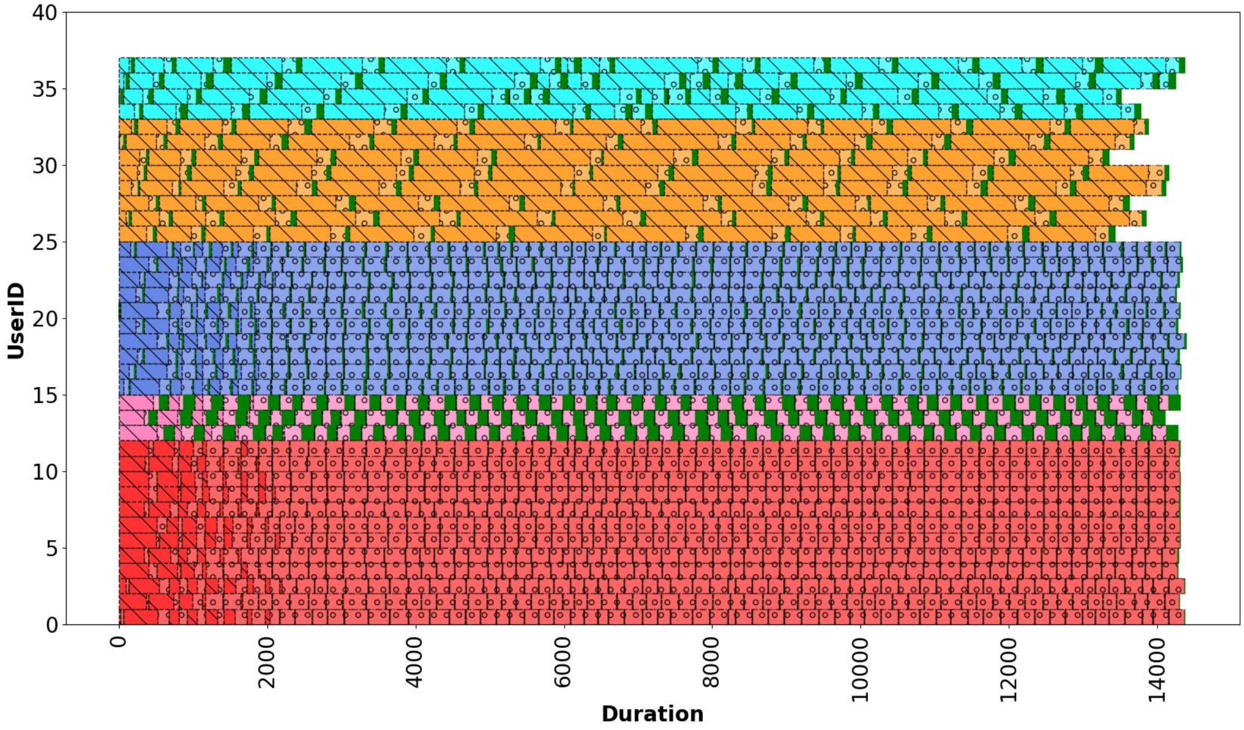


Figure 5.23: Wisconsin-V1 Scheduler - Timeline Plot for core-multiplier = 3

Figuers 5.23, 5.24, and 5.25 illustrate the timeline plots for this experiment using core-multiplier values of three, twelve, and forty-eight respectively. In these plots, the sections with brighter colors and circle-shaped backgrounds represent the time when a query was waiting in the memory queue in Wisconsin-V1 scheduler. In all of these timeline plots, the query classes with shorter execution times wait in the memory queue for some time due to the scheduler's adjustment of MPL values.

By increasing the core-multiplier from three to twelve, we can see that ZeroMemory-Short queries are executed more frequently with less waiting in any MPL or memory queues. However, increasing the MPL further to forty-eight creates more competition for the ZeroMemory-Short queries as more queries of the other types can now execute concurrently until memory

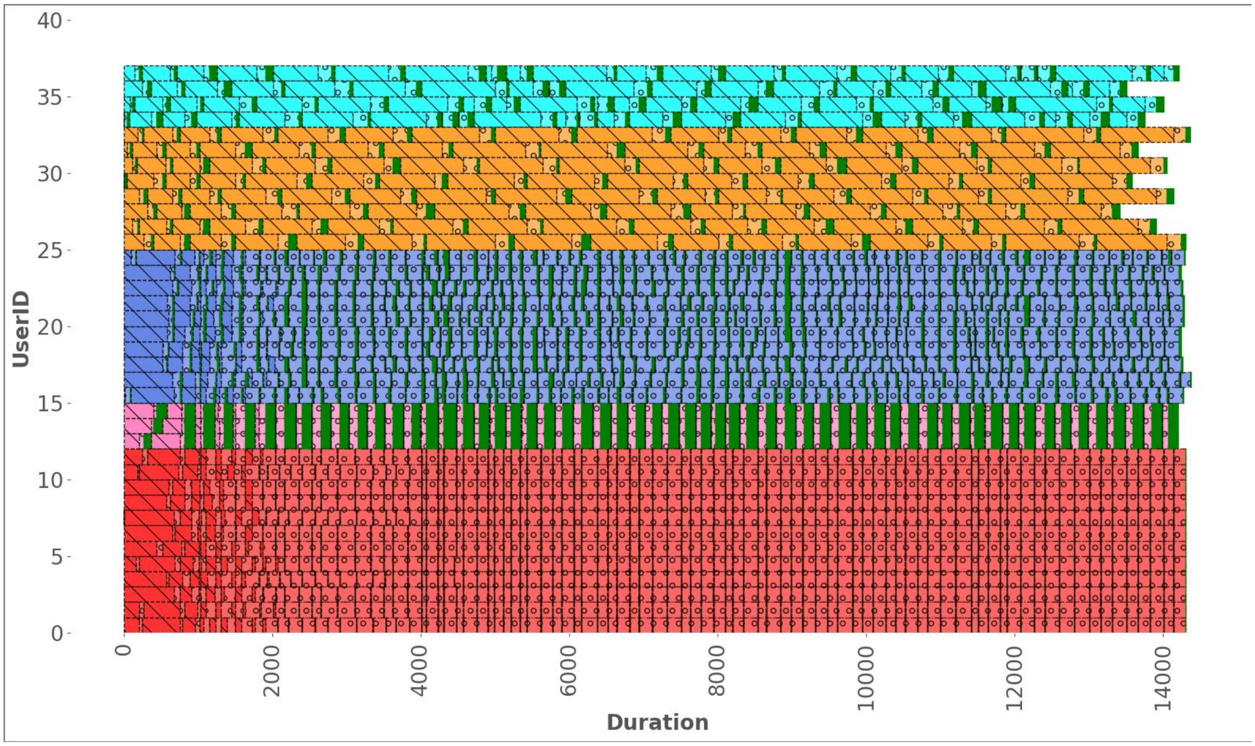


Figure 5.24: Wisconsin-V1 Scheduler - Timeline Plot for core-multiplier = 12

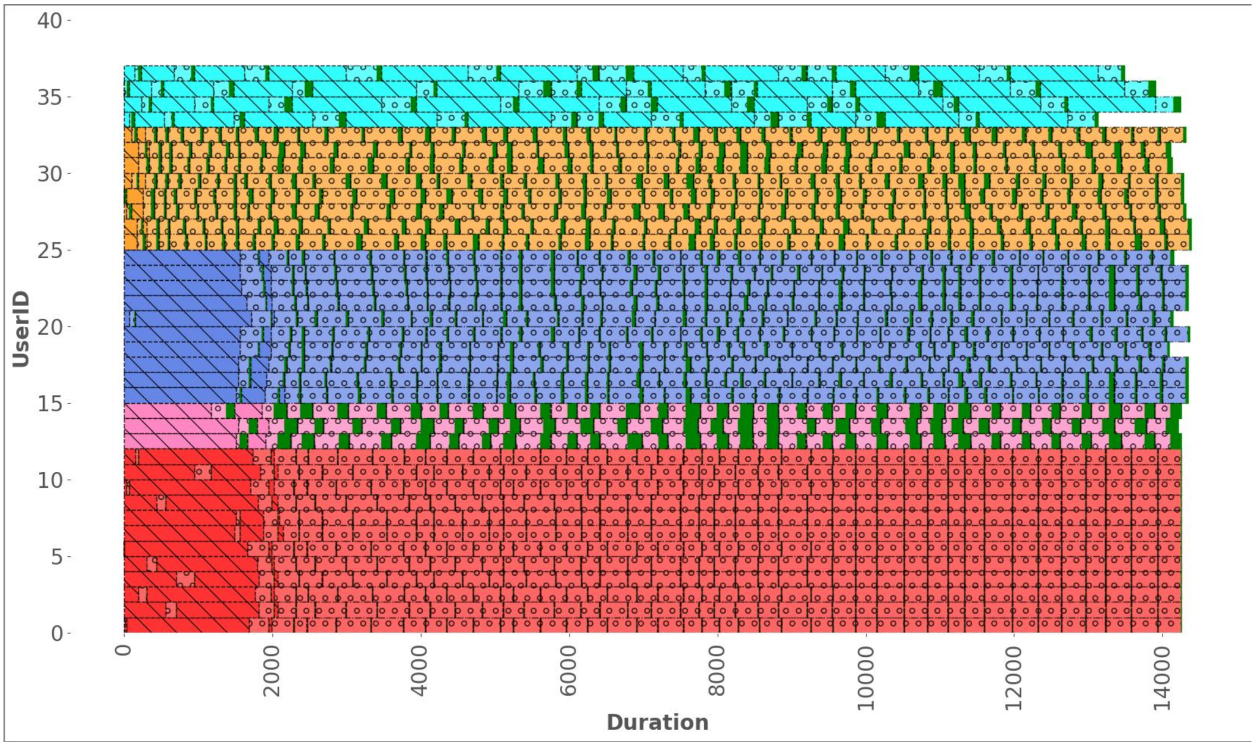


Figure 5.25: Wisconsin-V1 Scheduler - Timeline Plot for core-multiplier = 48

becomes the bottleneck. This competition adds to the waiting time of queries with shorter execution times, including the ZeroMemory-Short and Small queries.

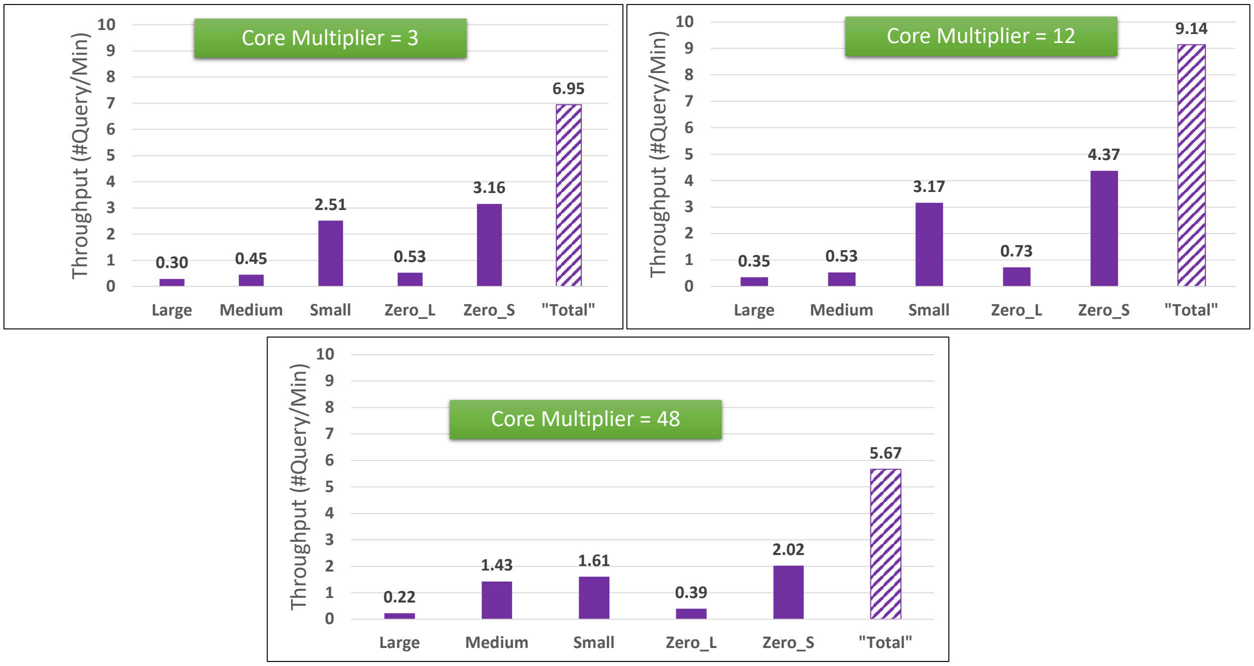


Figure 5.26: Wisconsin-V1 Scheduler - Throughput

As Figure 5.26 outlines, modifying the core-multiplier impacts and changes the trends in the throughput values. In general, the throughputs of the Small and ZeroMemory-Short query classes are slightly higher than those of the other query classes, primarily due to their short execution times and their larger numbers of associated users. Increasing the core-multiplier from three to twelve improves the throughputs of the various query classes and the ”Total” throughput since more queries can be admitted with the increased CPU capacities. However, by increasing the core-multiplier from twelve to forty-eight, more queries of larger classes can start their execution which may block other query classes due to memory limitations. Additionally, with a core-multiplier of forty-eight, more Medium queries have been executed, which means the MPL values have been adjusted more often to improve on the fairness.

By comparing the average execution times (Figure 5.27) and average response times (Figure 5.28) of the various query classes, we can see that the wait times of the various query

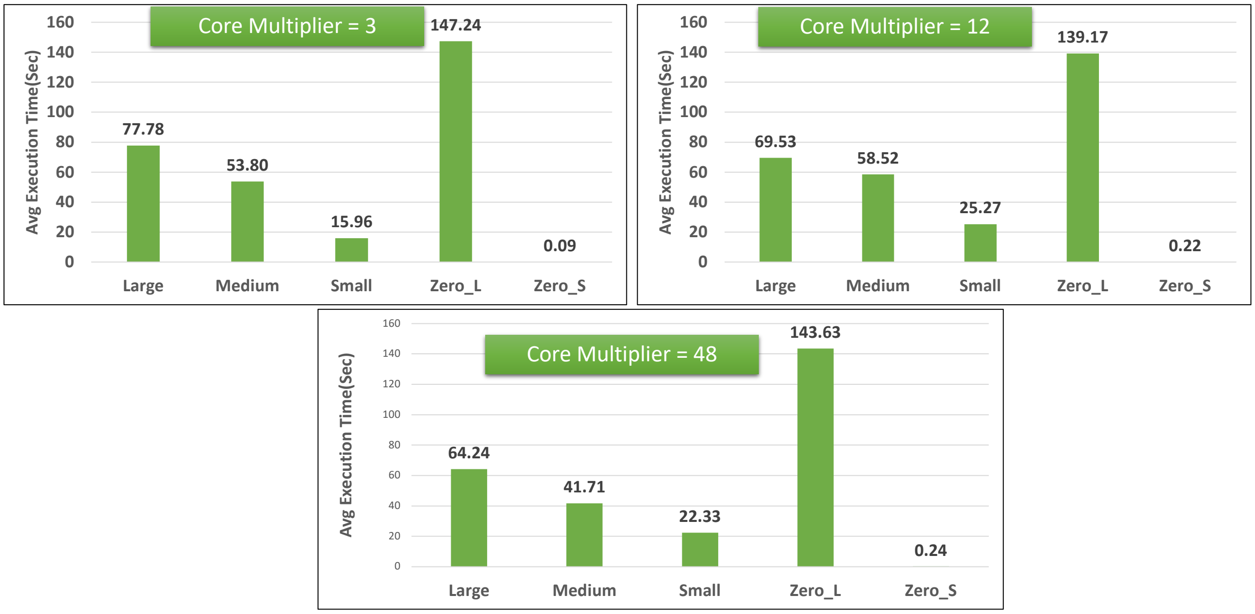


Figure 5.27: Wisconsin-V1 Scheduler - Average Execution Time

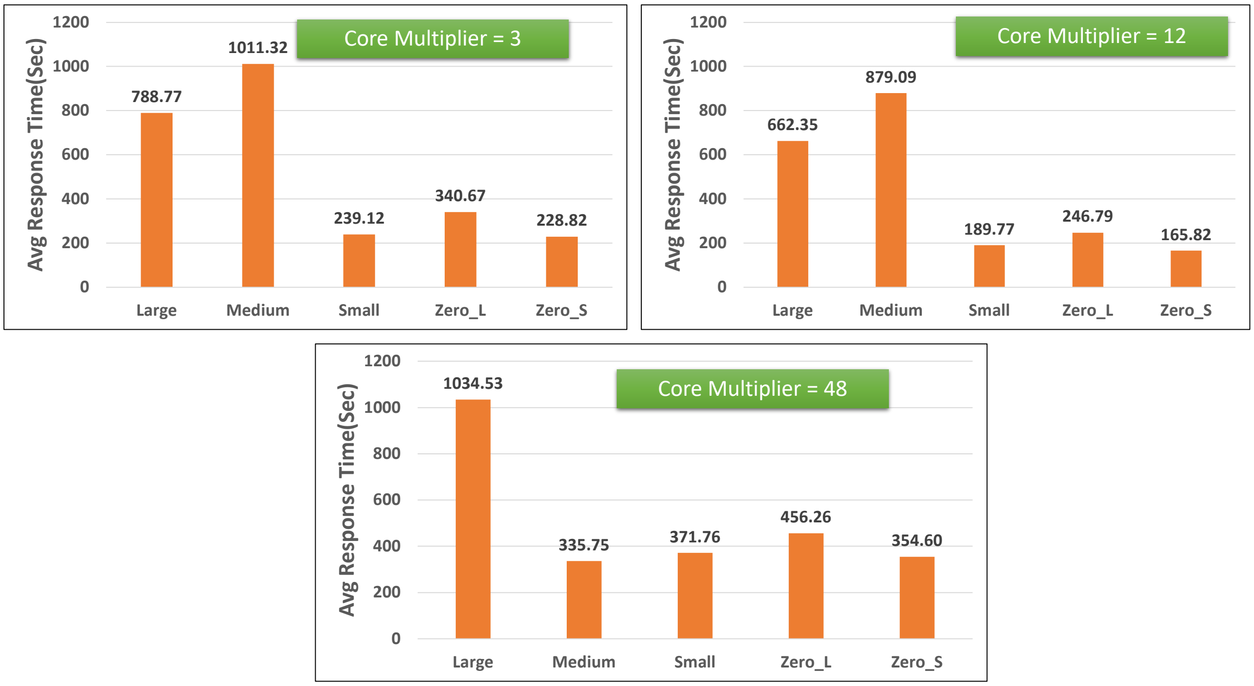


Figure 5.28: Wisconsin-V1 Scheduler - Average Response Time

classes are not proportional to their average execution times. One of the main reasons for this phenomenon is that in the Wisconsin-V1 scheduler, the ZeroMemory-Short, ZeroMemory-Long, and Small queries are all directed to the Small MPL queue. This makes the ZeroMemory-Long queries benefit the most, by having two short query types in its group to reduce their wait time, while, the Small and ZeroMemory-Short queries have to wait longer in the queue due to having the ZeroMemory-Long queries in their group. The second reason for not having a similar slowdown for each query class can be due to how often the MPL adjustment logic is called and what action it takes to adjust the MPLs. The high variance in the slowdown of the query classes seen when modifying the core-multiplier makes the Wisconsin-V1 a potentially non-robust scheduling algorithm with hard-to-predict performance.

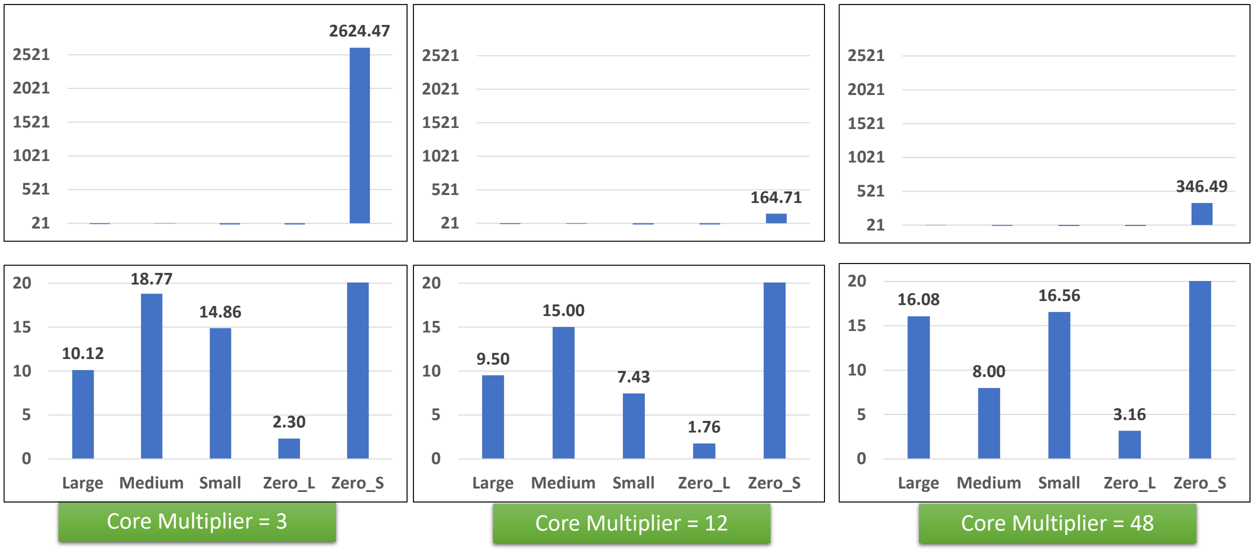


Figure 5.29: Wisconsin-V1 Scheduler - Stretch Factor

Figure 5.29 displays the stretch factors of the query classes for all three core-multiplier values. Based on this figure, the Wisconsin-V1 scheduler is biased towards the ZeroMemory-Long queries since they are inserted into the same MPL queue as Small and ZeroMemory-Short queries whose execution times are much lower than the ZeroMemory-Long queries. The ZeroMemory-Short queries have been treated the most unfairly compared to the other query classes. The stretch factor values of different query classes fluctuate significantly across different core-multipliers, making this scheduler's behavior hard to predict. Similar fluctuations can be seen in the values for the distance from the average stretch factor metric across different core-multiplier values (Figure 5.30).

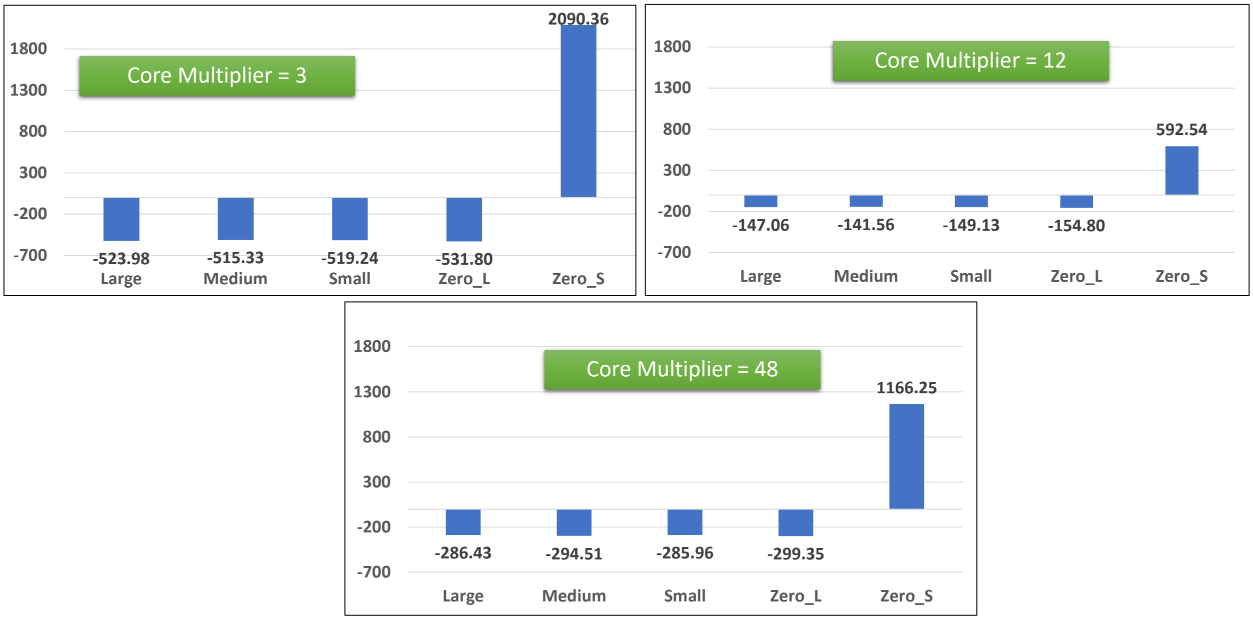


Figure 5.30: Wisconsin-V1 Scheduler - Distance From Average Stretch Factor

### 5.4.5 Evaluation of the Wisconsin-V2 Scheduler

Next, we evaluate the Wisconsin-V2 scheduler using the same experimental settings. The timeline plots for this scheduler have been included in the Appendix. As Figure 5.31 displays, the Small and ZeroMemory-Short query classes have the highest throughputs under all three core-multiplier values, and all throughput values seem to increase as the core-multiplier increases similarly. The better consistency in throughput values in the Wisconsin-V2 scheduler compared to the Wisconsin-V1 scheduler is due to the separation of the ZeroMemory-Short and ZeroMemory-Long queries from the Small queries' MPL queue and the direct addition of them into the resource queue.

A comparison of Figures 5.32 and 5.33 indicates that the waiting times of the query classes is still not proportional to their execution times. This is due to two reasons. First, as mentioned earlier, other queries can get stuck behind ZeroMemory-Long queries in the resource queue as there is no condition to limit its MPL value. Second, the waiting times of queries in the queue depends on how often a Medium query is executed to adjust the MPL values and on

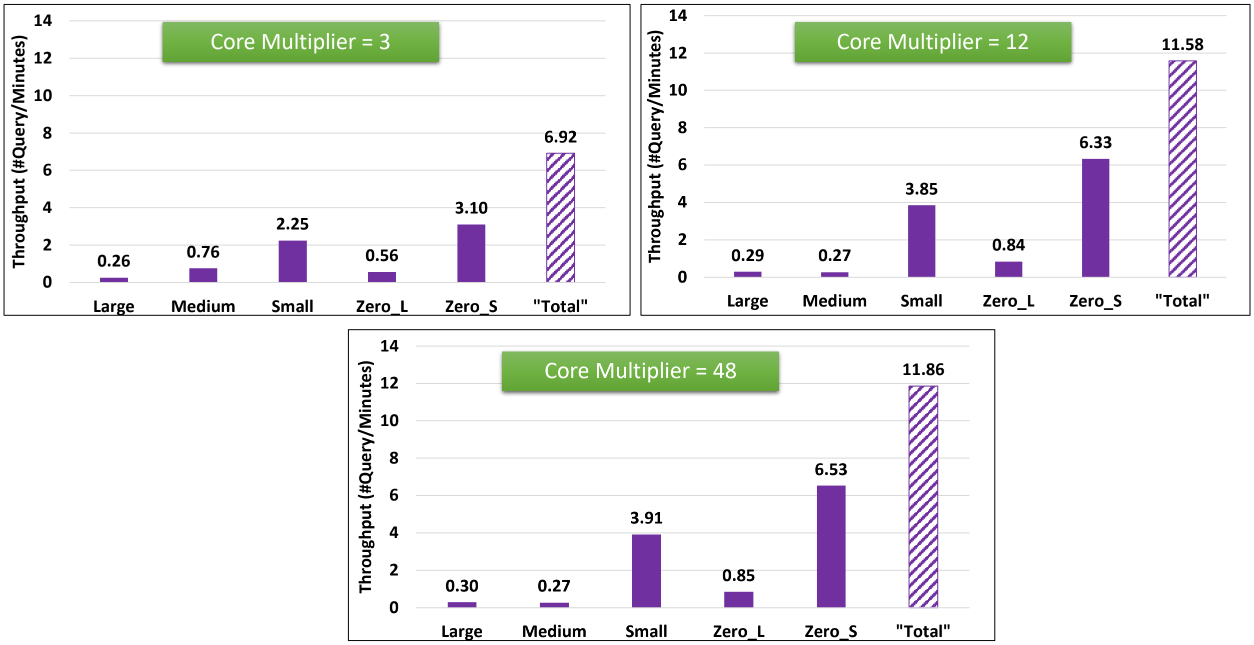


Figure 5.31: Wisconsin-V2 Scheduler - Throughput

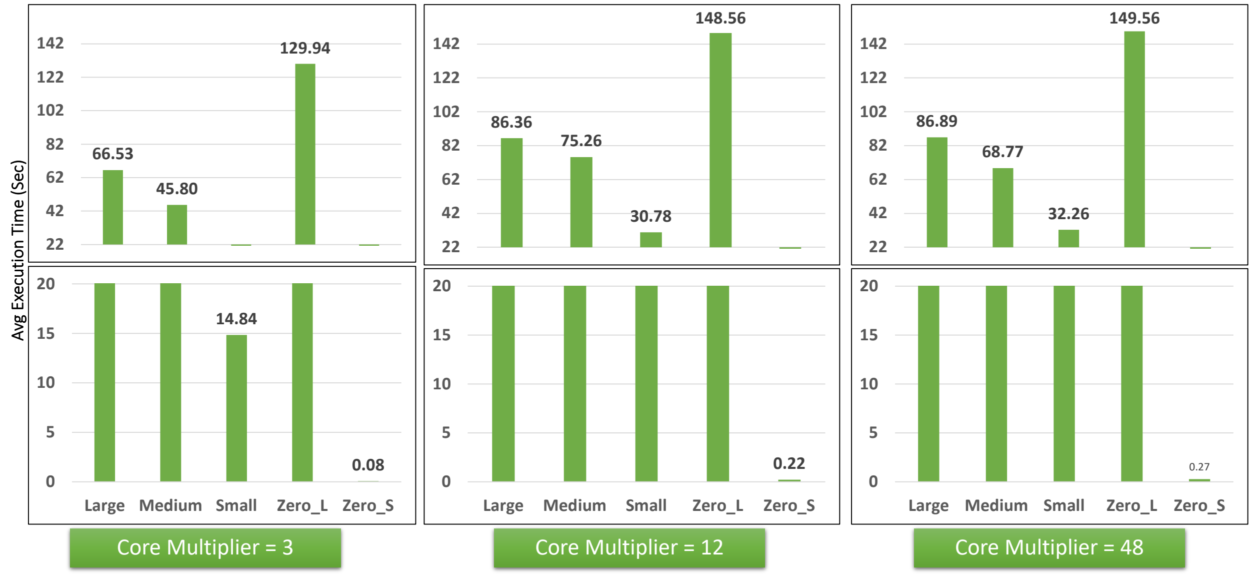


Figure 5.32: Wisconsin-V2 Scheduler - Average Execution Time

the chosen action to adjust the MPL values. Configuring all of the MPL adjustment knobs of the Wisconsin-V2 scheduler to reach a fair scheduler is not a trivial task. Similar to the Wisconsin-V21 scheduler, the Wisconsin-V2 scheduler is also biased towards ZeroMemory-Long queries since they skip the MPL queue and then occupy CPU cores for the (long) duration of their execution, which may make other queries wait in the resource queue for a long time.

Although the average response times of the Small and ZeroMemory-Short classes in Wisconsin-V2 scheduler (Figure 5.33) are lower than those of Wisconsin-V1 (Figure 5.28), not having an MPL limit on the ZeroMemory-Long queries leads to having a high number of them in the resource queue which then makes the shorter queries wait for a long time.

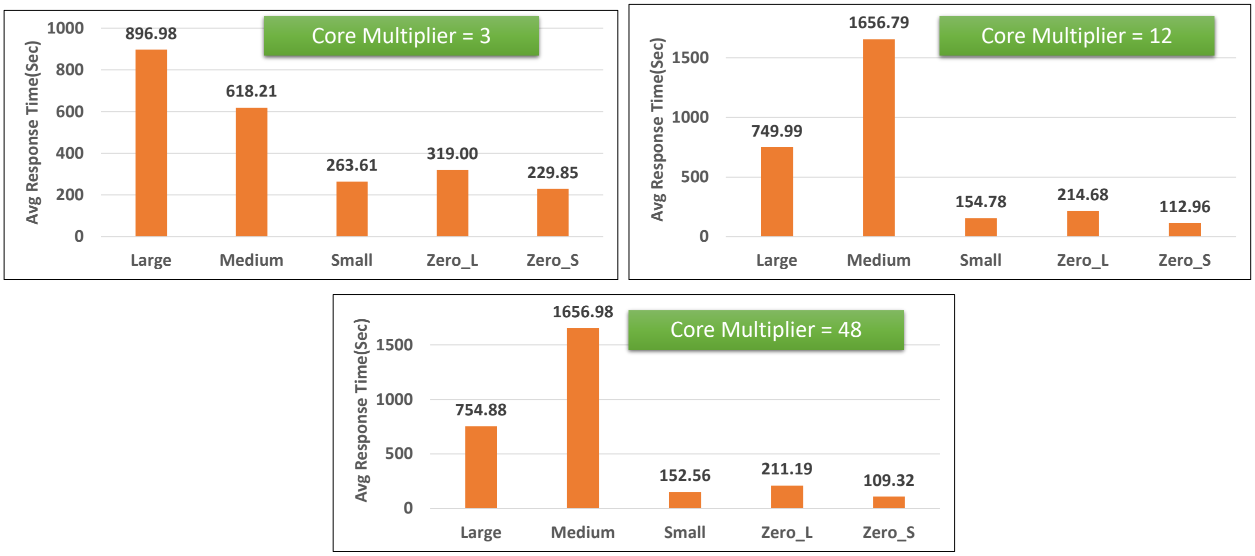


Figure 5.33: Wisconsin-V2 Scheduler - Average Response Time

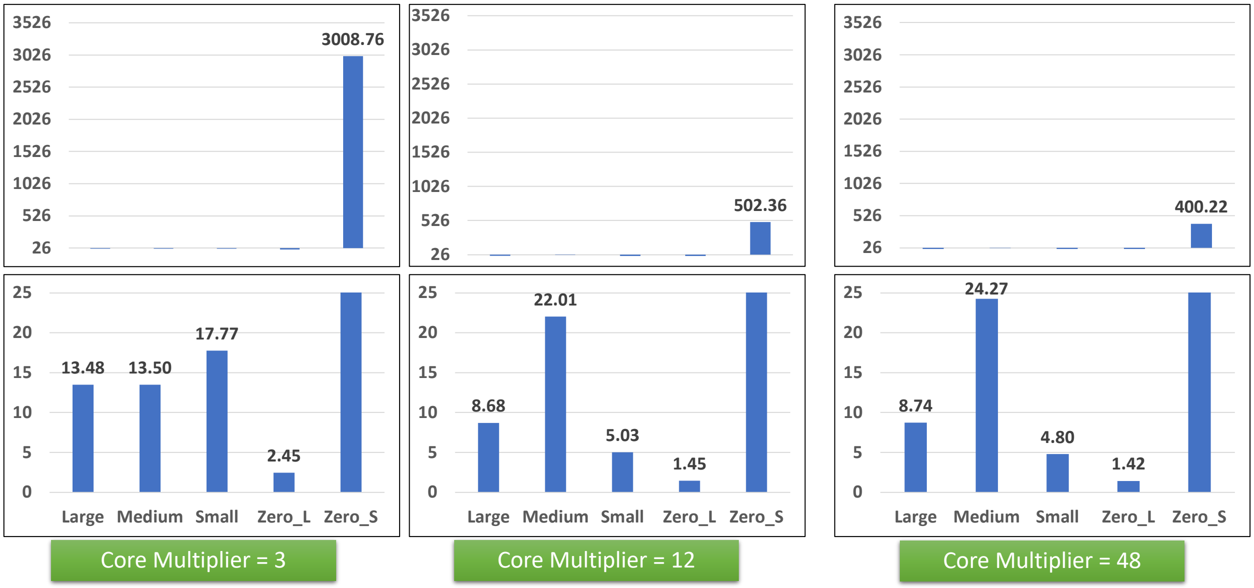


Figure 5.34: Wisconsin-V2 Scheduler - Stretch Factor

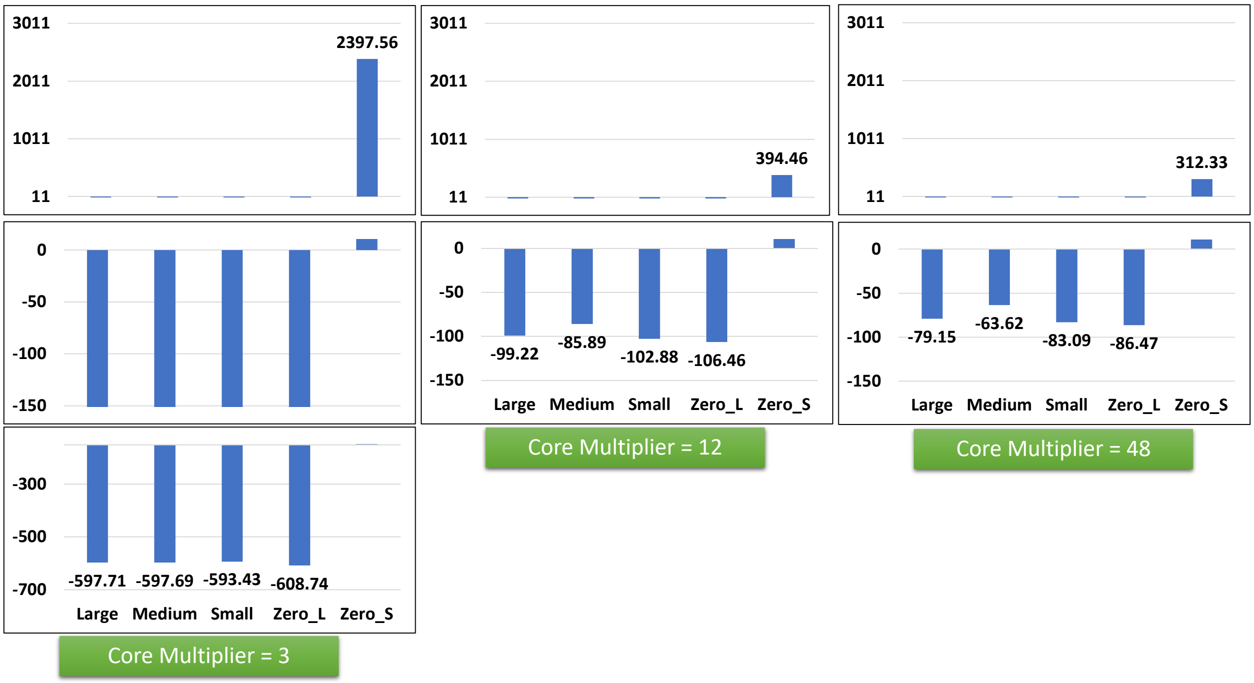


Figure 5.35: Wisconsin-V2 Scheduler - Distance From Average Stretch Factor

Figures 5.34 and 5.35 verify that, despite skipping the MPL queues, the Wisconsin-V2 scheduler is still highly unfair to the ZeroMemory-Short queries and is biased towards the ZeroMemory-Long queries more than any other query class.

### 5.4.6 Evaluation of the Wisconsin-V3 Scheduler

We evaluate the Wisconsin-V3 scheduler as the next scheduling algorithm using the same experimental settings. We have included the timeline plots of this scheduler in the Appendix.

In Figure 5.36, the Wisconsin-V3 scheduler reaches one of the highest throughput values for the ZeroMemory-Short query class and ”Total” throughput. By adding its two new MPL queues for the ZeroMemory-Short and ZeroMemory-Long queries, not only are ZeroMemory-Short queries not blocked behind larger queries in an MPL queue, but also they cannot take over all of the CPU cores because their MPL value will prevent them from doing so. Additionally, by adding the ZeroMemoy-Long queries to their own MPL queue, they cannot block or take advantage of smaller queries in their MPL queue. Additionally, they cannot block other queries in the resources queue for long as since its MPL queue prevents them from over-populating the resource queue.

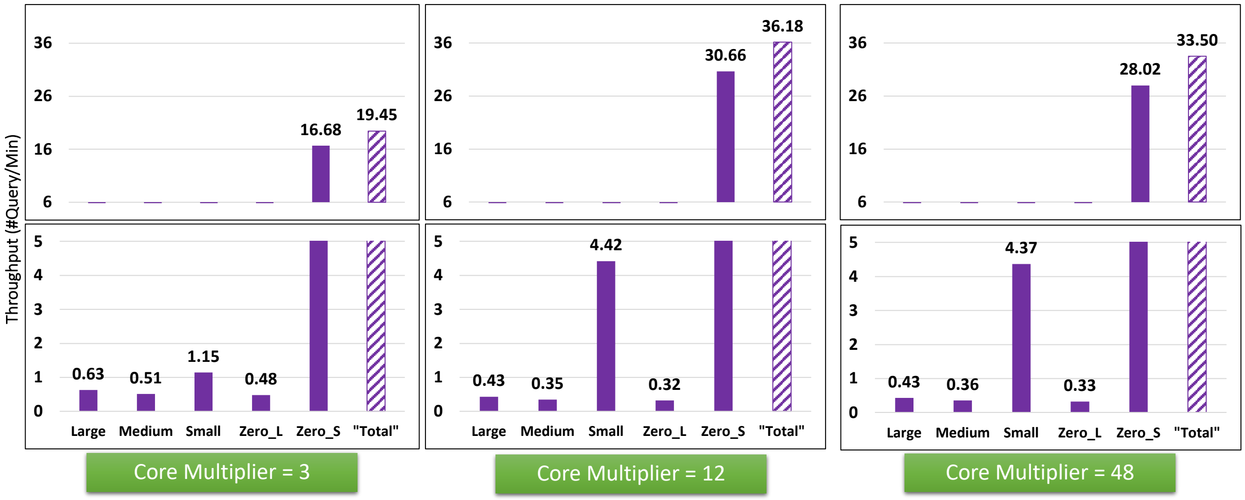


Figure 5.36: Wisconsin-V3 Scheduler - Throughput

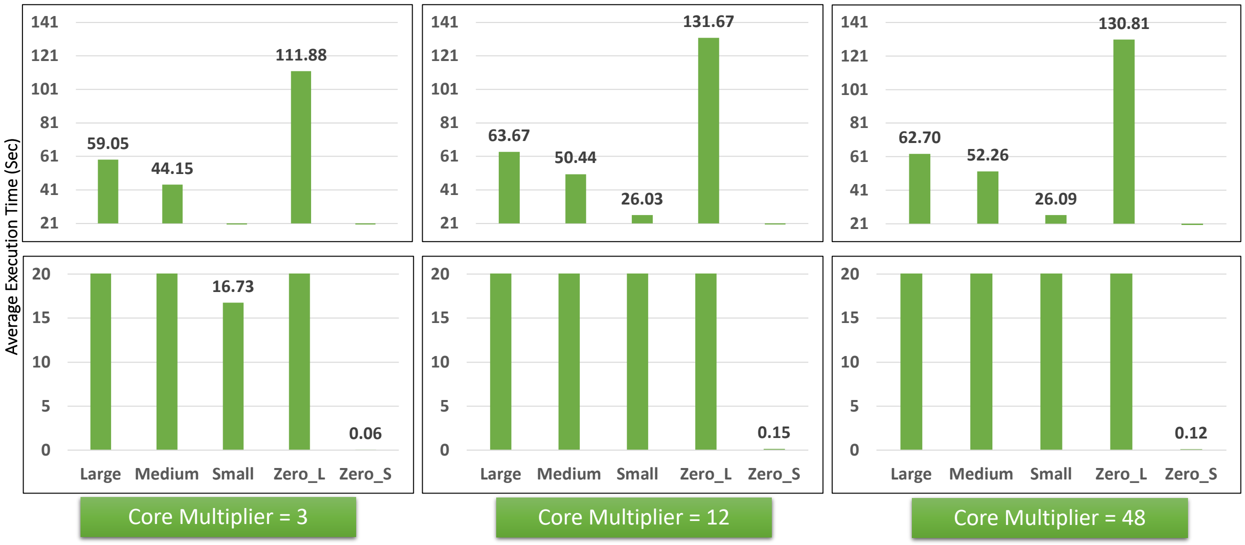


Figure 5.37: Wisconsin-V3 Scheduler - Average Execution Time

By comparing Figures 5.37 and 5.38, we can see that the query classes of ZeroMemory-Short, ZeroMemory-Long, and Small wait in the system for duration proportional to their execution times. However, the Medium and Large query classes wait in the queue for longer than what they should have too. This goes back to how often, and how, the scheduler adjusts the MPLs for the different query classes. Sometimes the action taken to adjust the MPL values are ineffective since the MPL values may grow larger than the available resources can afford. Thus, reducing those MPL values by one may not be effective.

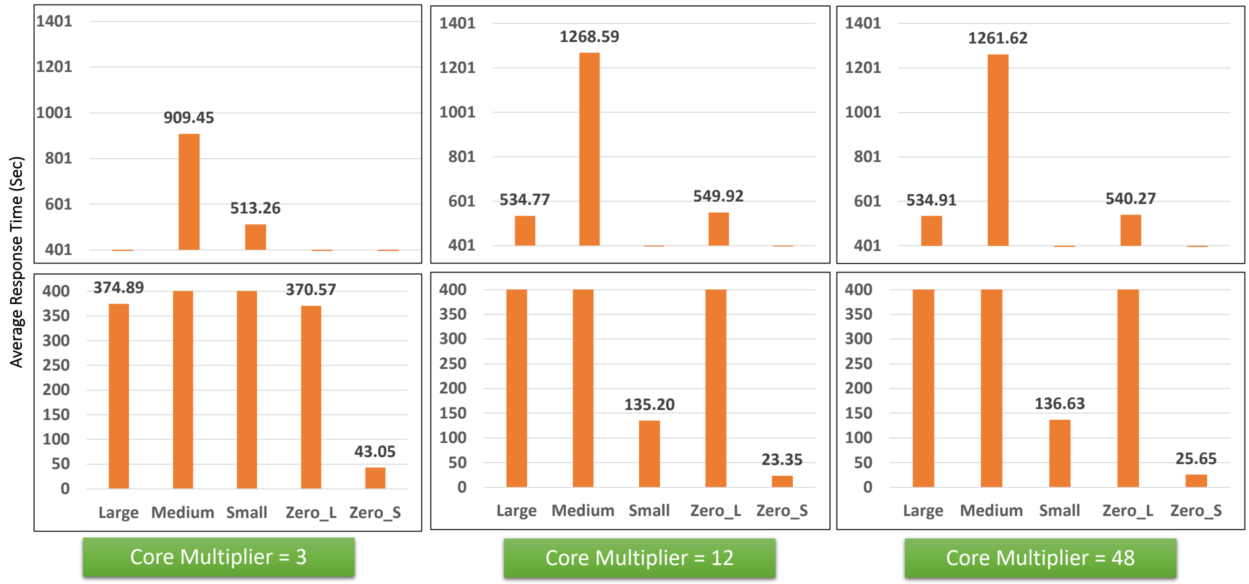


Figure 5.38: Wisconsin-V3 Scheduler - Average Response Time

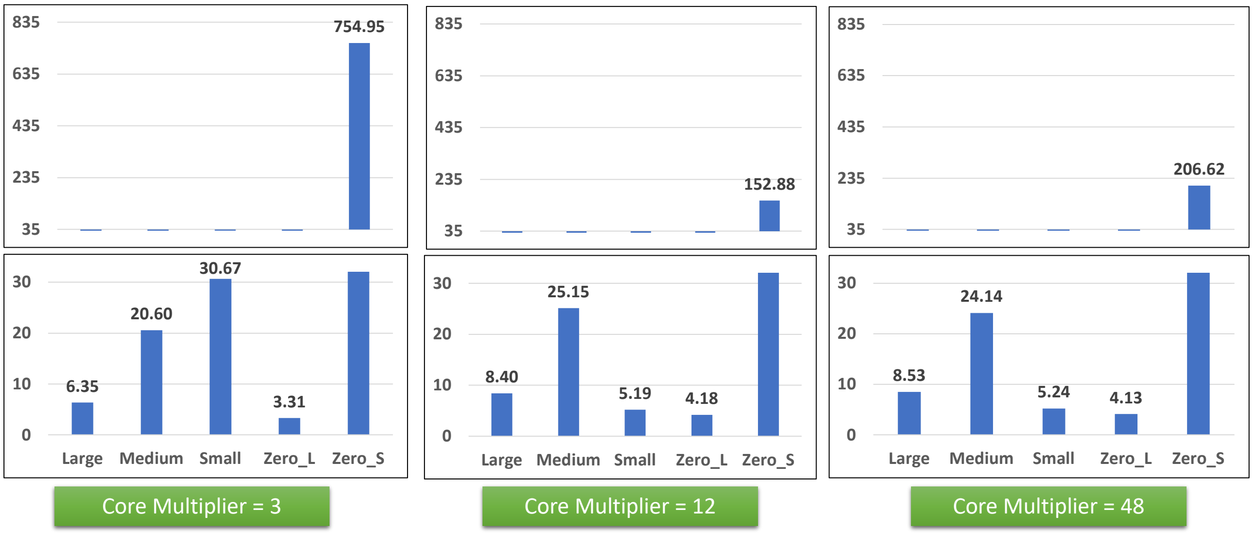


Figure 5.39: Wisconsin-V3 Scheduler - Stretch Factor

As Figures 5.39 and 5.40 display, the Wisconsin-V3 scheduler is one of the fairest schedulers that we have seen so far, as the stretch factors are smaller for all query classes and the values for the distances from the average stretch factor are closer in this scheduler compared

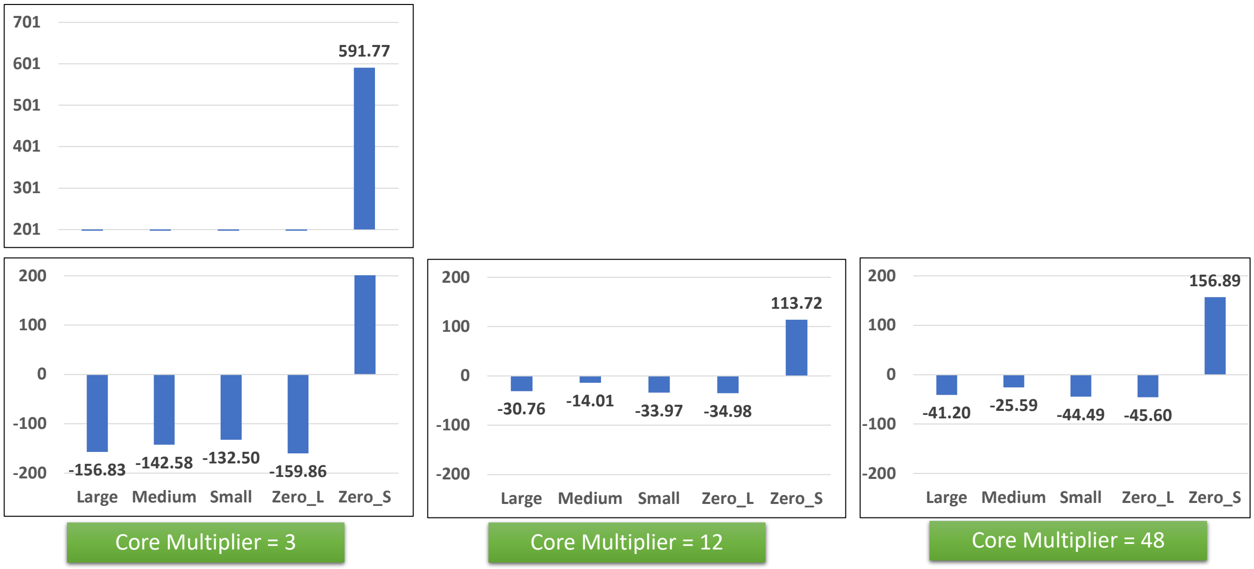


Figure 5.40: Wisconsin-V3 Scheduler - Distance From Average Stretch Factor

to previous schedulers. However, one downside that this scheduler shares with the other variations of the Wisconsin scheduler is that the logic based on which the MPL values get adjusted can cause fluctuations in the different metrics' values and needs to be studied carefully to maintain robustness under different core-multiplier values.

### 5.4.7 Evaluation of the Colorado-V1 Scheduler

As mentioned earlier, in the Colorado-V1 scheduler, queries of different query classes will be inserted in ten queues based on the ratios of their memory usage over the total memory. We have included the timeline plots for this scheduler in the Appendix.

Figure 5.41 outlines the total and per-class throughputs of the system using core-multipliers of three, twelve, and forty-eight when the Colorado-V1 scheduler is used to schedule the queries. The total and per-class throughput values remain fairly consistent when the core-multiplier increases. This throughput consistency is due to the number of queues that are assigned to each query class and how evenly queries of each class have been distributed between its queues. When making its admission decisions, the scheduler checks the ratio of the

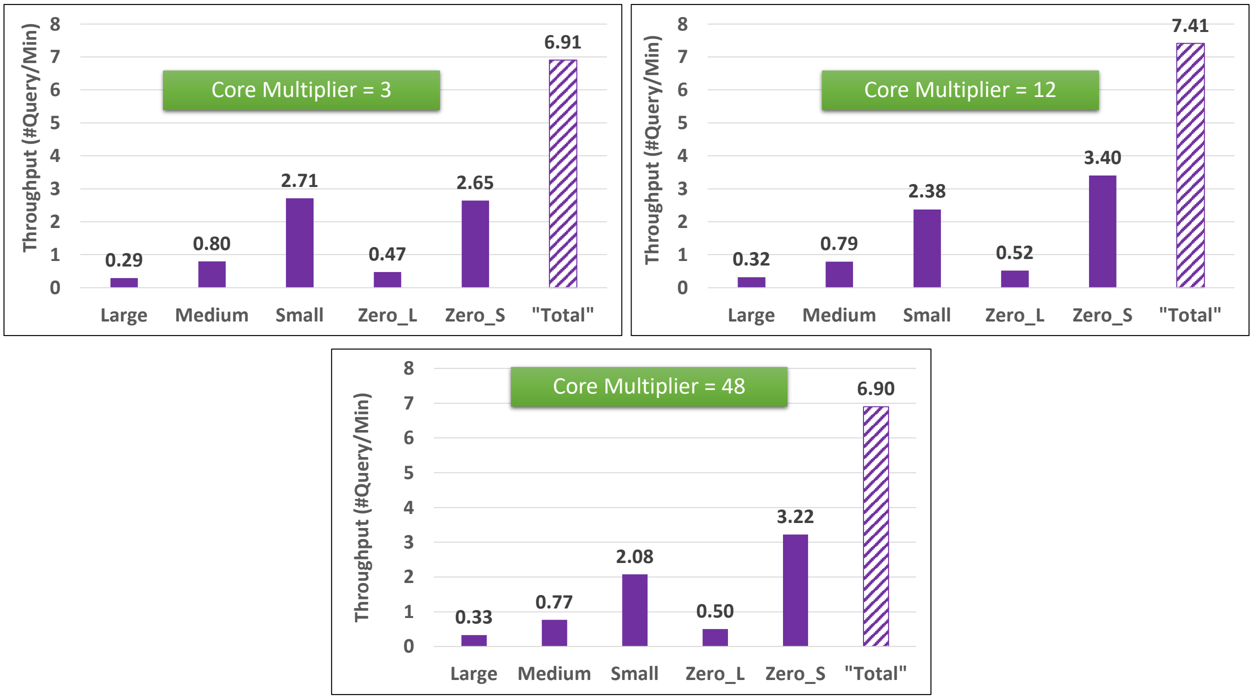


Figure 5.41: Colorado-V1 Scheduler - Throughput

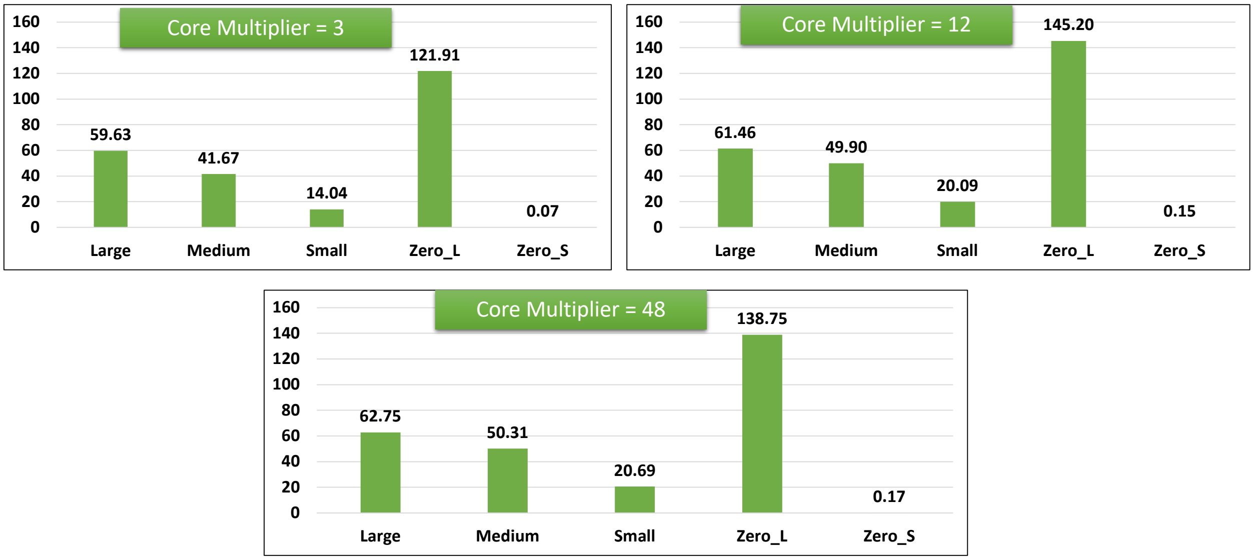


Figure 5.42: Colorado-V1 Scheduler - Average Execution Time

wait time of each query at the head of a queue to the average execution time of it query class to choose the next query to execute. In the Colorado-V1 scheduler, all queries with similar memory requirements will be directed to the same queue meaning that the ZeroMemory-Short and ZeroMemory-Long queries are directed to one queue despite their large difference

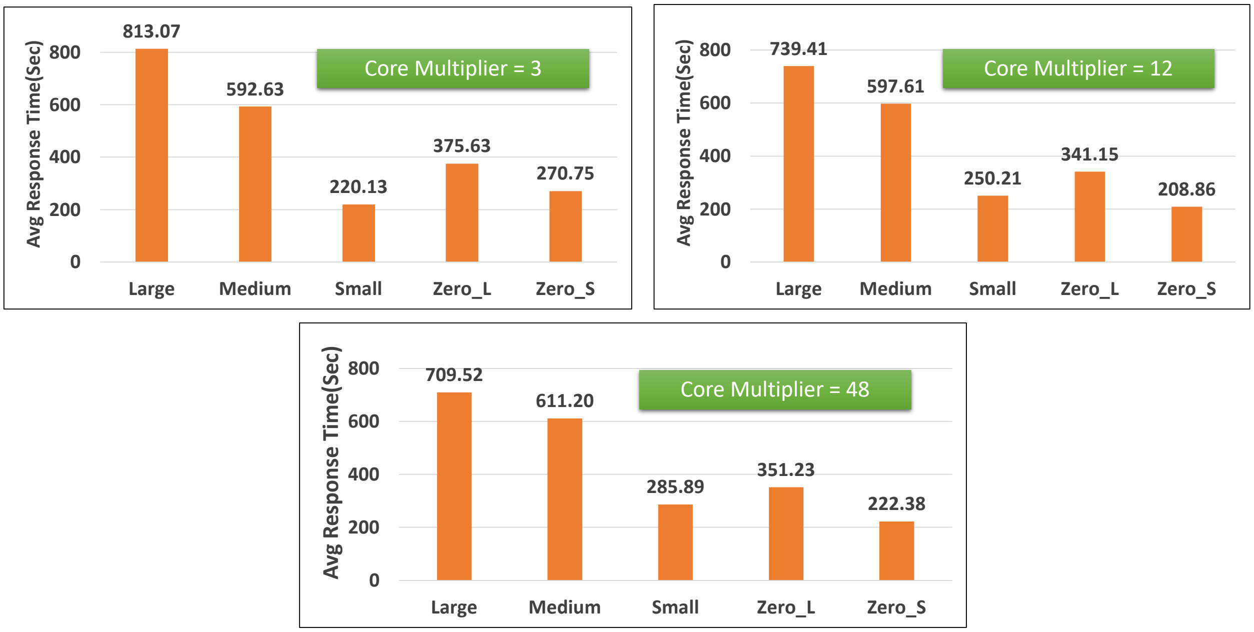


Figure 5.43: Colorado-V1 Scheduler - Average Response Time

in execution times. Thus, ZeroMemory-Short queries can wait behind ZeroMemory-Long queries, which reduces their throughput.

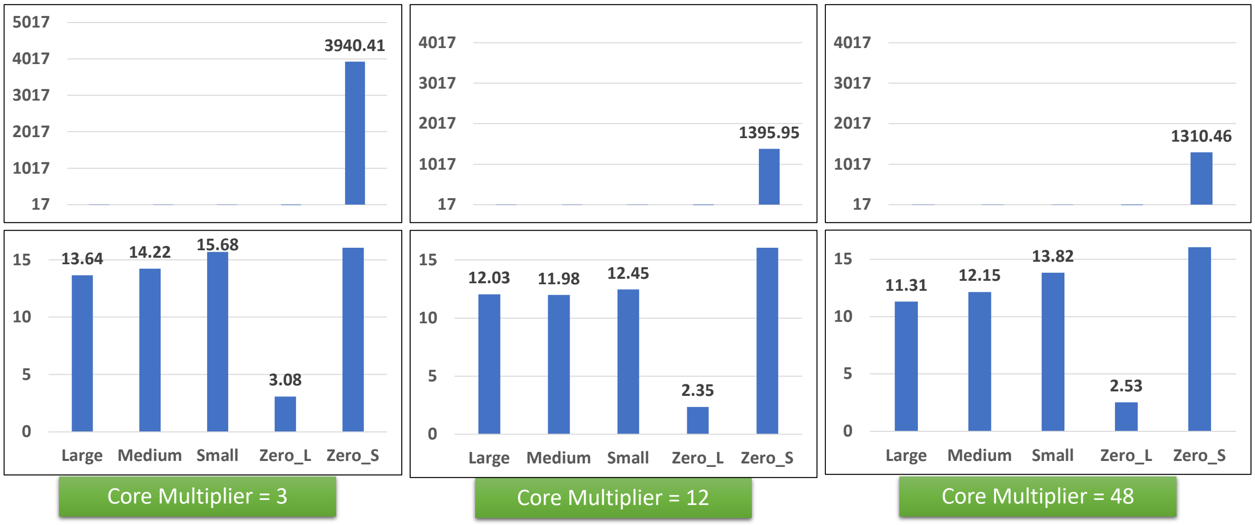


Figure 5.44: Colorado-V1 Scheduler - Stretch Factor

By comparing the average execution times of each query class in Figure5.42 to their average responses time in Figure 5.43, we can see that the ZeroMemory-Short queries have the highest wait times compared to their short execution times. The fact that ZeroMemory-Short queries

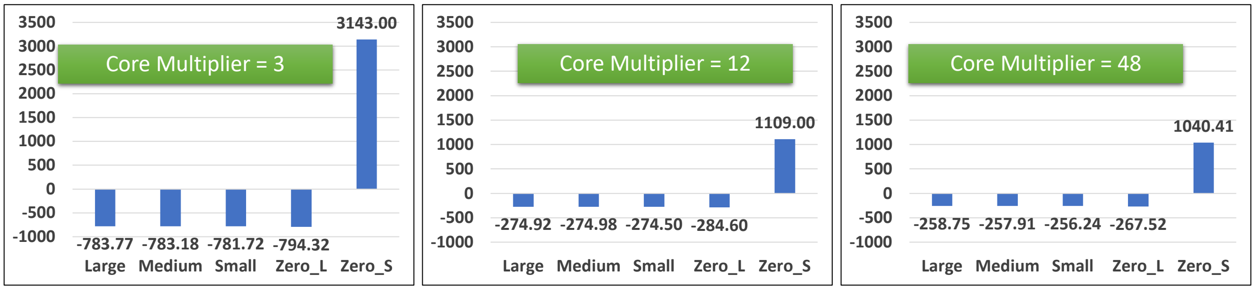


Figure 5.45: Colorado-V1 Scheduler - Distance From Average Stretch Factor

can stay in the queue behind the ZeroMemory-Long queries makes the Colorado-V1 scheduler act unfairly towards the ZeroMemory-Short queries. This phenomenon is seen in Figures 5.44 and 5.45 with large values for the stretch factor and distance from the average stretch factor for the ZeroMemory-Short query class.

### 5.4.8 Evaluation of the Colorado-V2 Scheduler

In the next experiment, we use the same workload settings and datasets used in previous sections to evaluate the Colorado-V2 scheduler under different core-multiplier values and using our different metrics. We have included the timeline plots for this scheduler in the Appendix.

As Figure 5.46 demonstrates, the Colorado-V2 scheduler delivers a higher throughput for the ZeroMemory-Short query class as compared to the Colorado-V1 scheduler since the Colorado-V2 scheduler puts ZeroMemory-Short and ZeroMemory-Long queries into different queues, while as mentioned earlier, in the Colorado-V1 scheduler these two query classes are directed to the same queue. This separation of queues for the ZeroMemory-Short and ZeroMemory-Long query classes helps with the earlier release of ZeroMemory-Short queries from their queue, as they are not waiting behind the ZeroMemory-Long queries. Note that this high throughput for the ZeroMemory-Short query class causes the total throughput of the system to be higher as well. Figure 5.46 outlines that the total throughput and the per-class throughput values remain fairly consistent under different core-multiplier settings.

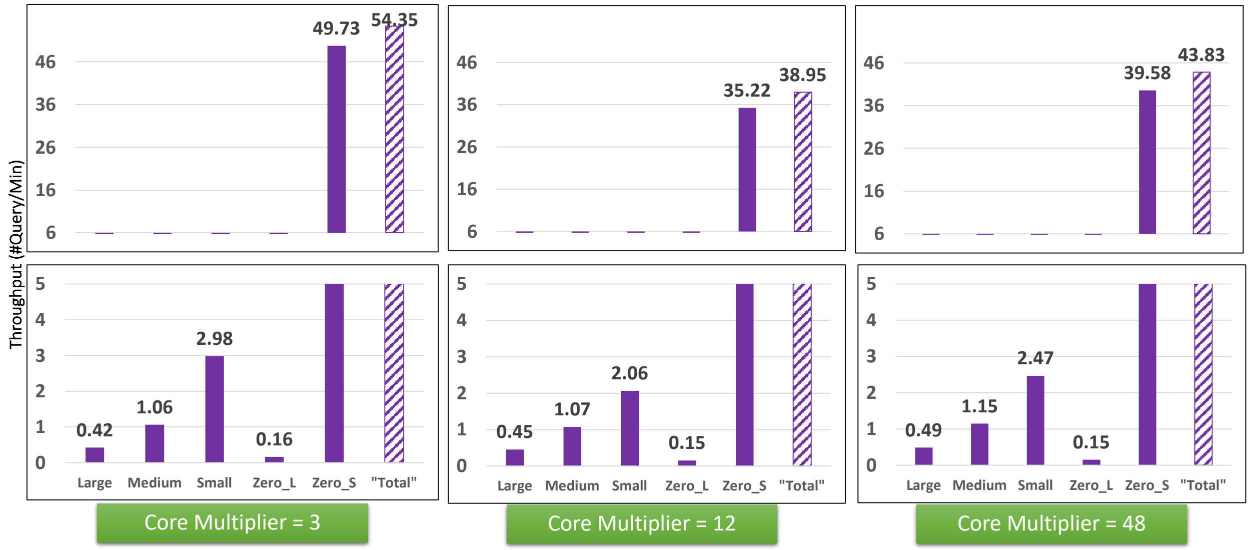


Figure 5.46: Colorado-V2 Scheduler - Throughput

Figure 5.47 represents the average execution times for queries of each query class executing concurrently. By comparing the Figures 5.46 and 5.47 for each core-multiplier value, we can see that the query classes with lower average execution times have the highest throughput and vice versa. This indicates that the Colorado-V2 scheduler has fairly scheduled the queries from different query classes.

By comparing the average response times and the average execution times of each query class across the different core-multiplier settings (Figures 5.47 and 5.48), we can see that the query classes whose queries have a large execution time also have a high response time on average and vice versa. This again verifies that the Colorado-V2 scheduler has successfully kept the wait times of the queries in the queues proportional to their average execution times.

Figure 5.49 demonstrates the stretch factors of the various query classes for the Colorado-V2 scheduler with different core-multiplier settings. Based on this figure, the stretch factor values of all query classes apart from ZeroMemory-Short are very close to each other and are

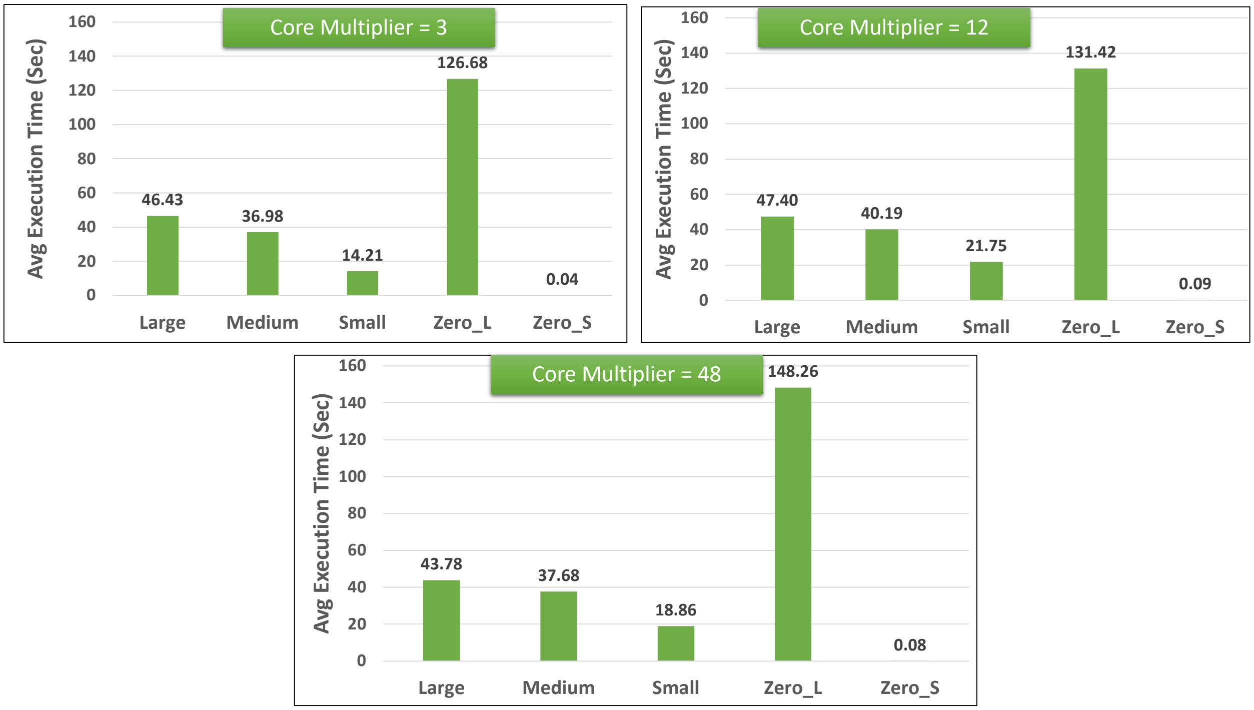


Figure 5.47: Colorado-V2 Scheduler - Average Execution Time

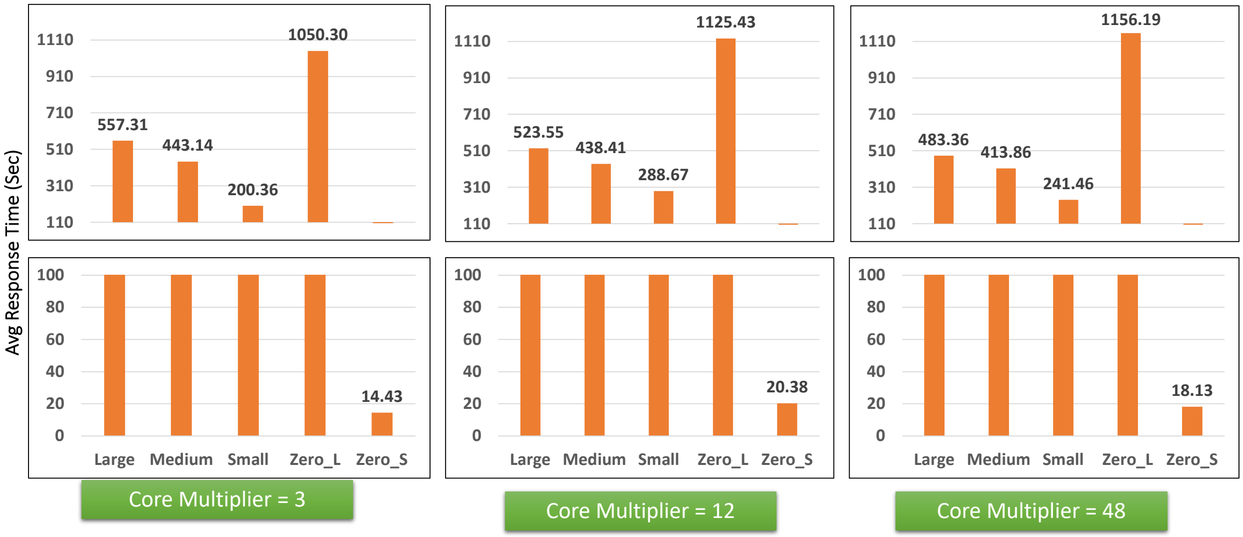


Figure 5.48: Colorado-V2 Scheduler - Average Response Time

consistent for the different core-multiplier settings. This closeness in stretch factor values is what we would expect to see if a scheduler is fairly scheduling the queries. The only exception in this figure is the ZeroMemory-Short class, which has a consistently high stretch factor in all three core-multiplier settings. This high stretch factor value for ZeroMemory-

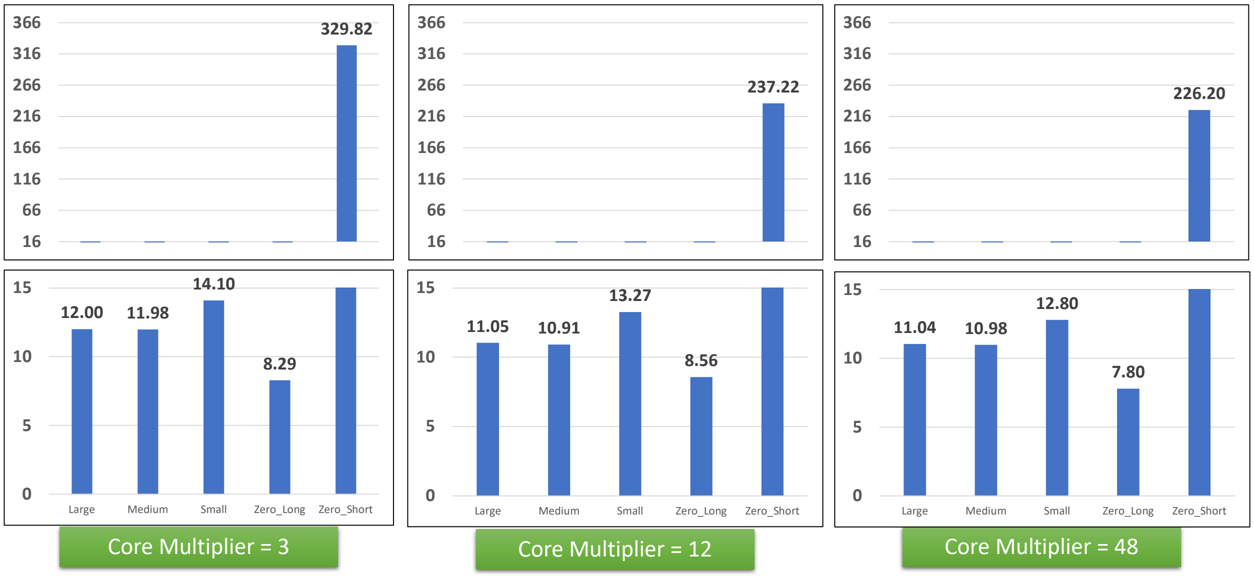


Figure 5.49: Colorado-V2 Scheduler - Stretch Factor

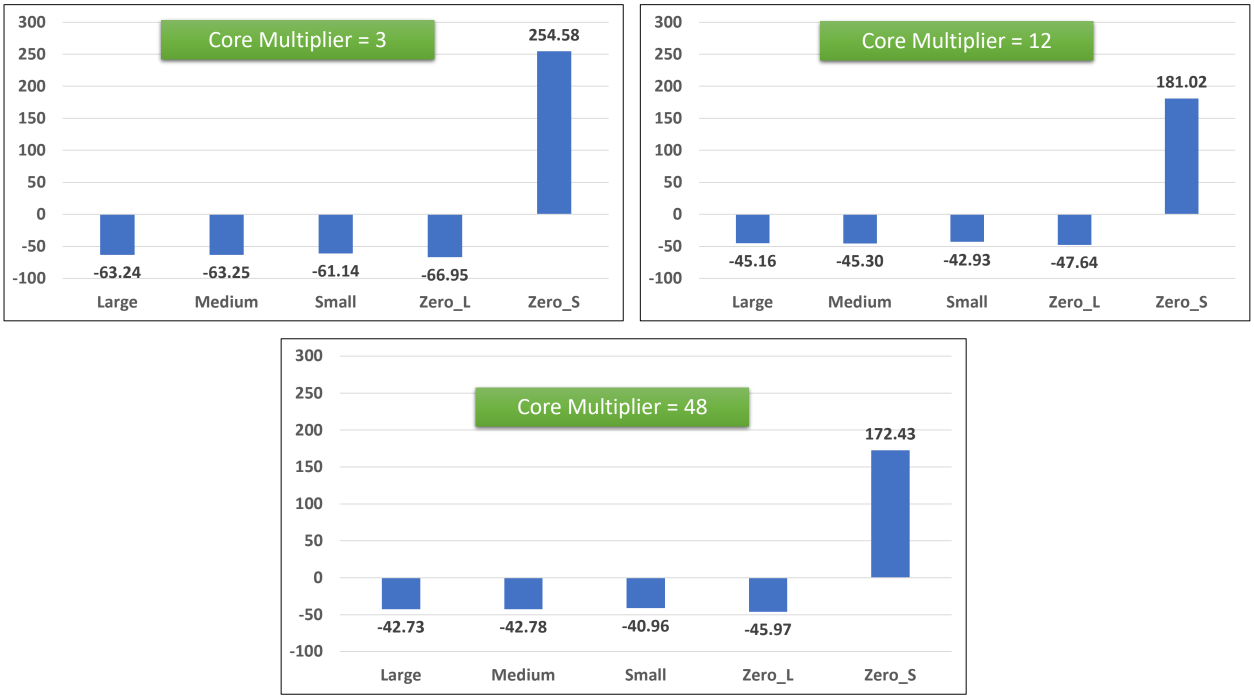


Figure 5.50: Colorado-V2 Scheduler - Distance From Average Stretch Factor

Short is due to the very low average execution time of this query class, which leads to a high penalty in its stretch factor even with the slightest wait time. Regardless, the Colorado-V2 scheduler has the lowest stretch factor and average response time for the ZeroMemory-Short query classes among all other schedulers.

Figure 5.50 displays the distance of the stretch factor of each query class from the average stretch factor value. By comparing the results of the Colorado-V2 scheduler in the distances from the average stretch factor of other schedulers, we can see that the Colorado-V2 scheduler has the smallest difference in the distance from the average stretch factor between the query classes. Thus, the Colorado-V2 scheduler is the fairest scheduler among all the evaluated schedulers in this chapter and provides robust performance across different core-multiplier settings.

### 5.4.9 Fairness

In this section, we compare the schedulers based on our fairness metric. As mentioned earlier, our fairness metric is the standard deviation between the stretch factor values of the different query classes. A lower value for fairness is preferred.

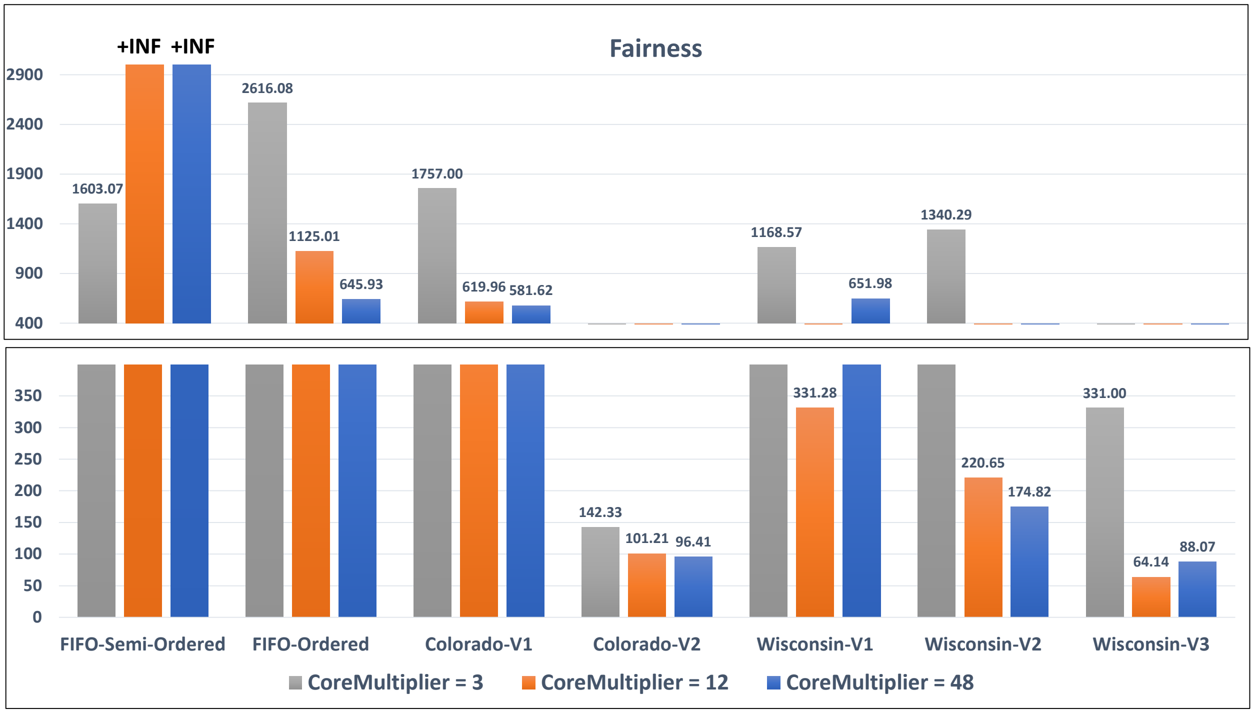


Figure 5.51: Fairness of Various Schedulers

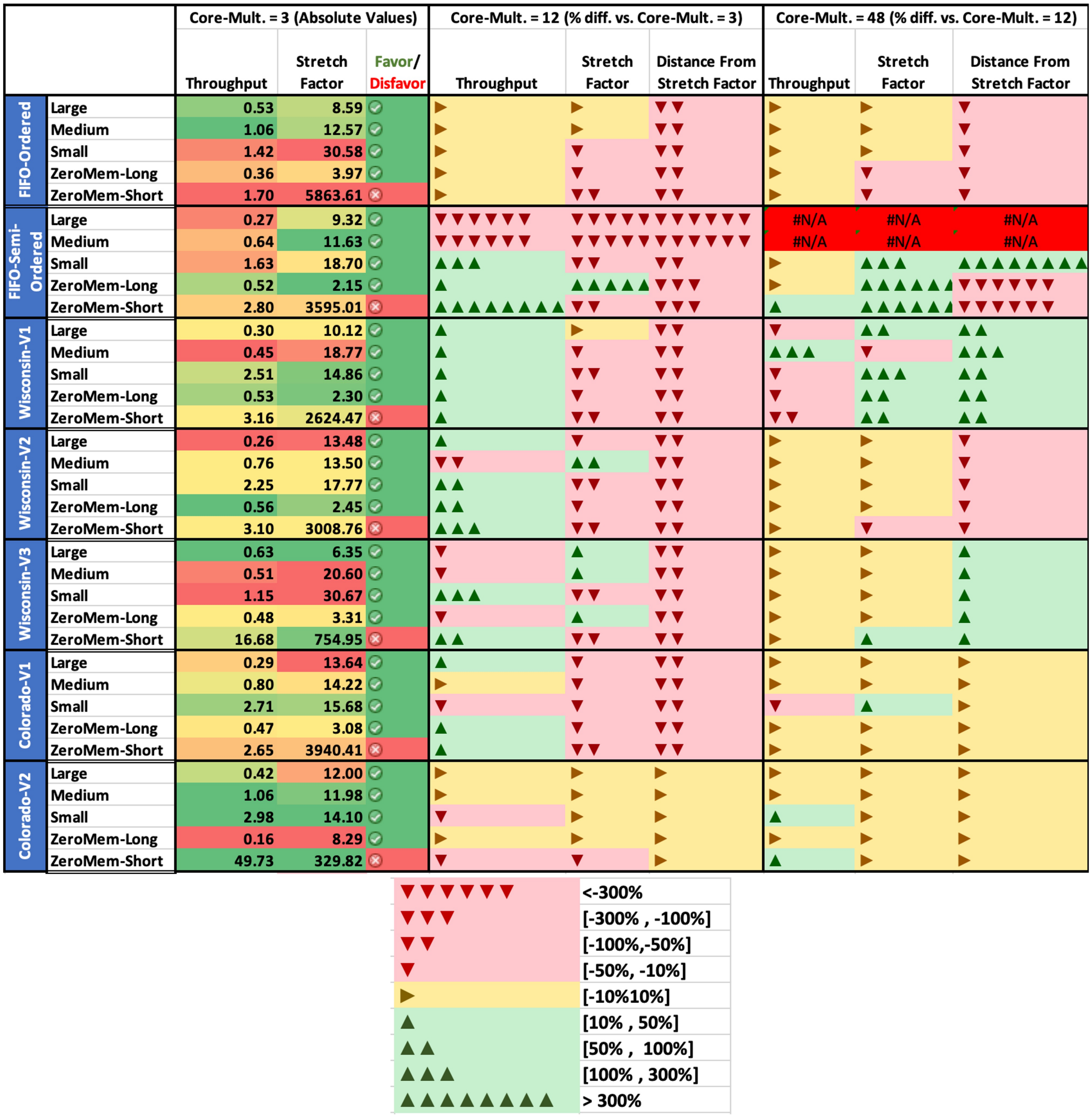
Figure 5.51 represents the fairness values for the different schedulers across the three core-multiplier values. FIFO-Semi-Ordered is the most unfair scheduler, as the larger query classes can be starved under this scheduler. FIFO-Ordered is unfair towards shorter queries, as all queries have to wait in one queue which can cause the short queries to wait for a long time behind longer queries. Colorado-V1 is also unfair towards the ZeroMemory-Short query class as it directs the ZeroMemory-Short queries to the same queue as the ZeroMemory-Long queries which leads to their long waits in the queue.

Among the variations of the Wisconsin schedulers, the Wisconsin-V1 and Wisconsin-V2 schedulers were not fair towards the ZeroMemory-Short queries as they get blocked behind the ZeroMemory-Long queries under both schedulers. The Wisconsin-V3 scheduler performed better than the other two variations of the Wisconsin scheduler, especially with higher core-multiplier values, but its fairness and thus its performance may not remain stable under different scenarios including variant core-multiplier values. The actions taken to adjust the MPL values can have different impacts on the overall fairness of the scheduler.

The Colorado-V2 scheduler is the fairest and most robust scheduler among all the evaluated schedulers. It has relatively simple logic and can be easily managed since it does not need to follow complex logic to adjust the multi-programming levels of different query classes. Instead, a query with the highest stretch factor value will be selected as the next query to be executed. Additionally, allocating separate queues to the ZeroMemory-Short and ZeroMemory-Long improved its fairness metric by preventing long waits of ZeroMemory-Short queries behind ZeroMemory-Long queries.

### 5.4.10 Overall Comparison of Schedulers

In this section, we evaluate each scheduler by comparing its throughput, stretch factor, and the average distance from the stretch factor under different core-multipliers of three, twelve,



|  |  | Core-Mult. = 3 (Absolute Values) |  |  | Core-Mult. = 12 (% diff. vs. Core-Mult. = 3) |  |  | Core-Mult. = 48 (% diff. vs. Core-Mult. = 12) |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
|  |  | Throughput | Stretch Factor | Favor/ Disfavor | Throughput | Stretch Factor | Distance From Stretch Factor | Throughput | Stretch Factor | Distance From Stretch Factor |
| FIFO-Ordered | Large | 0.53 | 8.59 | ✓ | ▶ | ▶ | ▼▼ | ▶ | ▶ | ▼ |
|  | Medium | 1.06 | 12.57 | ✓ | ▶ | ▶ | ▼▼ | ▶ | ▶ | ▼ |
|  | Small | 1.42 | 30.58 | ✓ | ▶ | ▼ | ▼▼ | ▶ | ▶ | ▼ |
|  | ZeroMem-Long | 0.36 | 3.97 | ✓ | ▶ | ▼ | ▼▼ | ▶ | ▼ | ▼ |
|  | ZeroMem-Short | 1.70 | 5863.61 | ✗ | ▶ | ▼▼ | ▼▼ | ▶ | ▼ | ▼ |
| FIFO-Semi-Ordered | Large | 0.27 | 9.32 | ✓ | ▼▼▼▼▼▼ | ▼▼▼▼▼ | ▼▼▼▼▼▼ | #N/A | #N/A | #N/A |
|  | Medium | 0.64 | 11.63 | ✓ | ▼▼▼▼▼▼ | ▼▼▼▼▼ | ▼▼▼▼▼▼ | #N/A | #N/A | #N/A |
|  | Small | 1.63 | 18.70 | ✓ | ▲▲▲ | ▼▼ | ▼▼ | ▶ | ▲▲▲ | ▲▲▲▲▲▲▲▲ |
|  | ZeroMem-Long | 0.52 | 2.15 | ✓ | ▲ | ▲▲▲▲▲ | ▼▼▼ | ▶ | ▲▲▲▲▲▲ | ▼▼▼▼▼▼ |
|  | ZeroMem-Short | 2.80 | 3595.01 | ✗ | ▲▲▲▲▲▲▲▲ | ▼▼ | ▼▼▼ | ▲ | ▲▲▲▲▲▲ | ▼▼▼▼▼▼ |
| Wisconsin-V1 | Large | 0.30 | 10.12 | ✓ | ▲ | ▶ | ▼▼ | ▼ | ▲▲ | ▲▲ |
|  | Medium | 0.45 | 18.77 | ✓ | ▲ | ▼ | ▼▼ | ▲▲▲ | ▼ | ▲▲▲ |
|  | Small | 2.51 | 14.86 | ✓ | ▲ | ▼▼ | ▼▼ | ▼ | ▲▲▲ | ▲▲ |
|  | ZeroMem-Long | 0.53 | 2.30 | ✓ | ▲ | ▼ | ▼▼ | ▼ | ▲▲ | ▲▲ |
|  | ZeroMem-Short | 3.16 | 2624.47 | ✗ | ▲ | ▼▼ | ▼▼ | ▼▼ | ▲▲ | ▲▲ |
| Wisconsin-V2 | Large | 0.26 | 13.48 | ✓ | ▲ | ▼ | ▼▼ | ▶ | ▶ | ▼ |
|  | Medium | 0.76 | 13.50 | ✓ | ▼▼ | ▲▲ | ▼▼ | ▶ | ▶ | ▼ |
|  | Small | 2.25 | 17.77 | ✓ | ▲▲ | ▼▼ | ▼▼ | ▶ | ▶ | ▼ |
|  | ZeroMem-Long | 0.56 | 2.45 | ✓ | ▲▲ | ▼ | ▼▼ | ▶ | ▶ | ▼ |
|  | ZeroMem-Short | 3.10 | 3008.76 | ✗ | ▲▲▲ | ▼▼ | ▼▼ | ▶ | ▼ | ▼ |
| Wisconsin-V3 | Large | 0.63 | 6.35 | ✓ | ▼ | ▲ | ▼▼ | ▶ | ▶ | ▲ |
|  | Medium | 0.51 | 20.60 | ✓ | ▼ | ▲ | ▼▼ | ▶ | ▶ | ▲ |
|  | Small | 1.15 | 30.67 | ✓ | ▲▲▲ | ▼▼ | ▼▼ | ▶ | ▶ | ▲ |
|  | ZeroMem-Long | 0.48 | 3.31 | ✓ | ▼ | ▲ | ▼▼ | ▶ | ▶ | ▲ |
|  | ZeroMem-Short | 16.68 | 754.95 | ✗ | ▲▲ | ▼▼ | ▼▼ | ▶ | ▲ | ▲ |
| Colorado-V1 | Large | 0.29 | 13.64 | ✓ | ▲ | ▼ | ▼▼ | ▶ | ▶ | ▶ |
|  | Medium | 0.80 | 14.22 | ✓ | ▶ | ▼ | ▼▼ | ▶ | ▶ | ▶ |
|  | Small | 2.71 | 15.68 | ✓ | ▼ | ▼ | ▼▼ | ▼ | ▲ | ▶ |
|  | ZeroMem-Long | 0.47 | 3.08 | ✓ | ▲ | ▼ | ▼▼ | ▶ | ▶ | ▶ |
|  | ZeroMem-Short | 2.65 | 3940.41 | ✗ | ▲ | ▼▼ | ▼▼ | ▶ | ▶ | ▶ |
| Colorado-V2 | Large | 0.42 | 12.00 | ✓ | ▶ | ▶ | ▶ | ▶ | ▶ | ▶ |
|  | Medium | 1.06 | 11.98 | ✓ | ▶ | ▶ | ▶ | ▶ | ▶ | ▶ |
|  | Small | 2.98 | 14.10 | ✓ | ▼ | ▶ | ▶ | ▲ | ▶ | ▶ |
|  | ZeroMem-Long | 0.16 | 8.29 | ✓ | ▶ | ▶ | ▶ | ▶ | ▶ | ▶ |
|  | ZeroMem-Short | 49.73 | 329.82 | ✗ | ▼ | ▼ | ▶ | ▲ | ▶ | ▶ |

|  |  |
|---|---|
| ▼▼▼▼▼▼ | <-300% |
| ▼▼▼ | [-300% , -100%] |
| ▼▼ | [-100%,-50%] |
| ▼ | [-50%, -10%] |
| ▶ | [-10%10%] |
| ▲ | [10% , 50%] |
| ▲▲ | [50% ,  100%] |
| ▲▲▲ | [100% , 300%] |
| ▲▲▲▲▲▲▲▲ | > 300% |

Figure 5.52: Overall Comparison of Various Schedulers

and forty-eight. Figure 5.52 demonstrates the changes in the values for the mentioned metrics for all the schedulers when the core-multiplier increases. We have used the core-multiplier of three as the base case. As Figure 5.52 shows, for the core-multiplier of three, the Colorado-V2 scheduler has the highest throughput values for each query class among other schedulers except for the ZeroMemory-Long class. This is since in the Colorado-V2 scheduler, query classes with a longer execution time wait in the queue for a longer duration as well, which reduces their overall throughput. In our experiments, ZeroMemory-Long has the longest average execution time compared to other query classes. Other schedulers that allow the ZeroMemory-Long queries to start their execution early have a higher throughput for this query class with the cost of lower throughput for query classes with shorter average execution times. The next column in the core-multiplier of three shows the stretch factor values for each query class ranked between different query schedulers. The Colorado-V2 scheduler has the lowest stretch factor for most query classes among other schedulers, which shows its fairness in scheduling queries belonging to different classes. The next column demonstrates the query classes that each scheduler has favored with green and the discriminated query classes with red. All schedulers have discriminated against the ZeroMemory-Short query class because of the short average execution time of this query class and its sensitivity to even the shortest wait time in the queue. The Colorado-V2 and Wisconsin-V3 have two of the lowest absolute distance from the average stretch factor for all query classes when core-multiplier is set to three.

So far, the Colorado-V2 scheduler is the best-performing scheduler under different metrics when the core-multiplier is set to three. By further evaluating the performance of each query class under different core-multiplier values, we can see that the Colorado-V2 is the most stable and robust scheduler among the evaluated schedulers when the core-multiplier varies.

## 5.5 Conclusion and Recommendations for Schedulers

In this chapter, we evaluated seven alternative scheduling algorithms for scheduling queries drawn from five query classes – ZeroMemory-Short, ZeroMemory-Long, Small, Medium, and Large – to design a fair scheduler. We define fairness as the provision of similar ratios of wait times to execution times for all query classes. The schedulers do not decide on the per-query memory allocation; they only decide on which query should be executed next given their resource needs.

We evaluated the seven schedulers using different metrics, including throughput (total and per-class), average execution time, average response time, stretch factor, distance from average stretch factor, and fairness. The Colorado-V2 scheduler was seen to be the fairest and most robust scheduling algorithm among all the evaluated schedulers. It uses a number of queues to separate the workload's queries based on their memory requirements. Additionally, this scheduler assigns queries that do not have a large memory requirement, but do have widely variant execution times, including the ZeroMemory-Short and ZeroMemory-Long query classes, to separate queues to prevent the shorter queries from getting stuck behind the longer queries.

Based on our empirical results, we recommend using the Colorado-V2 scheduler as the scheduling algorithm for AsterixDB. (AsterixDB is currently using FIFO-Semi-Ordered as its scheduling algorithm for multi-user workloads.)

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

In this thesis, we have investigated different ways to manage complex join queries in a big data management system.

In Chapter 3, we considered single join queries and studied best practices in having a robust Dynamic Hybrid Hash Join when accurate a priori information is unavailable. Specifically, we empirically investigated the design space of DHHJ to answer the following four questions:

1. Number of partitions: How many partitions should the records be hashed into if the sizes of inputs are unknown or inaccurate?

2. Partition Insertion: How can we find a “good” page (memory frame) within a partition for inserting a new record?

3. Victim Selection Policy: How can we select a “good” partition to spill in the case of insufficient memory?

4. Growth Policy: How many memory frames should a spilled partition be allowed to occupy after having spilled?

Based on our simulation study, we suggest using at least 20 partitions to partition the data to avoid the large spilling penalty for too few partitions.

Our study on partition insertion algorithm showed that Append(8) delivers a similar average frame fullness as other partition insertion algorithms but more efficiently due to its early search termination policy.

For victim selection policy, our experiments showed that policies that spill the largest size partitions or partitions that have the largest number of records are better policies for the majority of the cases due to two reasons: 1. Larger partitions release many frames; thus, they save other partitions from spilling to disk. 2. Writing larger partitions leads to more sequential and less random writes.

Next, we evaluated the performance of DHHJ under two growth policies of Grow-Steal and No Grow-No Steal for spilled partitions. Although our analytical and empirical evaluations showed that No Grow-No Steal performs more random I/Os than Grow-Steal, our experiments showed that having the filesystem cache enabled can diminish the differences between these growth policies by turning most of the random writes into sequential writes (Elevator Algorithm). Thus, choosing the right growth policy highly depends on whether the filesystem cache is enabled.

In Chapter 4, we explore various memory management and execution strategies for efficiently executing multi-join queries. We study the variations of Left Deep Trees, Right Deep Trees, and Bushy Trees. We consider two memory distribution methods for Right Deep Trees. RDT uses the equal memory distribution method while Static-RDT uses the bottom-up memory distribution method.

Additionally, we considered fully parallel, semi-parallel, and fully sequential execution strategies for executing RDT plans. Sequential-RDT and Sequential-Static-RDT are two new variations of RDT where the build phases of the joins execute sequentially.

Our experiments showed that LDT is one of the best-performing query plan shapes in most cases. LDT tends to spill less data to disk since the memory is always divided between two consecutive joins simultaneously. Moreover, its sequential execution pattern makes it less disruptive for disk arm contention in HDD. Also, the LDT's short pipelines between the build phase of the previous join and the probe phase of the next join lead to a consistently high CPU utilization due to the overlap of these activities. Parallel query plans such as RDT and Bushy Trees have a comparable performance on SSD when large amount of memory is available.

As bottom-line recommendations for AsterixDB and other systems, we suggest using LDT query plans as the default query plan shape for all queries and all storage types. We have seen that LDT maintains high performance under different join and scan selectivities and for different storage types and memory availability. RDT can outperform LDT in a few cases where the underlying storage device is a locally attached SSD and the available memory for a query is more than 80% of data. However, the difference in its execution time is not all that high compared to that of the LDT. Thus, using LDT as the default query plan is recommended.

In Chapter 5, we evaluated seven alternative scheduling algorithms for scheduling queries drawn from five query classes – ZeroMemory-Short, ZeroMemory-Long, Small, Medium, and Large – to design a fair scheduler. We define fairness as the provision of similar ratios of wait times to the execution times for all query classes. We evaluated the seven schedulers using different metrics, including throughput (total and per-class), average execution time, average response time, stretch factor, distance from aver- age stretch factor, and fairness. The Colorado-V2 was seen to be the fairest and most robust scheduling algorithm among all the evaluated schedulers. Based on our empirical results, we recommend using the Colorado-V2 scheduler as the scheduling algorithm for AsterixDB. (AsterixDB is currently using FIFO-Semi-Ordered as its scheduling algorithm.)

## 6.2 Future Work

Resource management and query scheduling are essential parts of a big data management system and can significantly impact its performance.

In addition to the join operator, careful considerations should be taken in the design and resource management of other memory-intensive operators. One future work direction would be to carefully study the design of other memory-intensive operators, including sort and group by, and ensure their robustness under different scenarios, including memory and statistics availability. Next, studying resource management for a complex query containing multiple memory-intensive operators is essential.

As another direction for future work, it would be interesting to design and implement memory-adaptive operators that can borrow from and lend memory to the other memory-intensive operators of a query. It could be worth studying the performance changes for various query plans for multi-join queries when memory can dynamically be shared and move between operator.

For concurrent schedulers, it would be worth evaluating the schedulers under much wider variety of different workloads, including cases where there are infrequent queries from a specific query class. Additionally, it will be essential to evaluate the schedulers on other storage alternatives, particularly EBS and EBS-Hybrid.

Furthermore, in most previous studies and this thesis, only single-join queries are considered.

It is essential for a DBMS to be able correctly classify more complex query plans based on their execution times to be able to handle all sorts of queries efficiently and fairly.

Last but not least, in Chapter 5 the scheduler was only responsible for deciding the order of execution for queries. The amount of memory for each operator was decided by the user and by using hints. It is essential for AsterixDB and or other data management system to manage and assign the memory to the operators and queries themselves. Having one module or collaborating modules for memory assignment and query scheduling can be the next step for AsterixDB.

# Bibliography

[1] Amazon Elastic Block Store. `https://aws.amazon.com/ebs/`. 2021.

[2] Apache AsterixDB. `https://asterixdb.apache.org/`. 2021.

[3] Apache Spark. `https://spark.apache.org/`. 2021.

[4] Apache VXQuery. `https://vxquery.apache.org/`. 2021.

[5] IO-Direct Java Library. `https://github.com/smacke/jaydio`. 2021.

[6] Mersenne Twister Fast. `https://cs.gmu.edu/~sean/research/mersenne/MersenneTwisterFast.java`. 2021.

[7] Oracle. `http://www.oracle.com/`. 2021.

[8] TPC benchmark$^{\text{tm}}$ A: standard specification. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[9] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. Asterixdb: A scalable, open source BDMS. *Proc. VLDB Endow.*, 7(14):1905–1916, 2014.

[10] S. Babu and H. Herodotou. Massively parallel databases and mapreduce systems. *Found. Trends Databases*, 5(1):1–104, 2013.

[11] M. M. Baldi and M. Bruglieri. On the generalized bin packing problem. *Int. Trans. Oper. Res.*, 24(3):425–438, 2017.

[12] C. Ballinger. TPC-D: benchmarking for decision support. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[13] M. Bandle, J. Giceva, and T. Neumann. To partition, or not to partition, that is the join question in a real system. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 168–180. ACM, 2021.

[14] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking database systems A systematic approach. In M. Schkolnick and C. Thanos, editors, *9th International Conference on Very Large Data Bases, October 31 - November 2, 1983, Florence, Italy, Proceedings*, pages 8–19. Morgan Kaufmann, 1983.

[15] P. A. Boncz, A. G. Anadiotis, and S. Kläbe. JCC-H: adding join crossing correlations with skew to TPC-H. In R. Nambiar and M. Poess, editors, *Performance Evaluation and Benchmarking for the Analytics Era - 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers*, volume 10661 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2017.

[16] V. R. Borkar, Y. Bu, E. P. C. Jr., N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras. Algebricks: a data model-agnostic compiler backend for big data languages. In S. Ghandeharizadeh, S. Barahmand, M. Balazinska, and M. J. Freedman, editors, *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, pages 422–433. ACM, 2015.

[17] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In S. Abiteboul, K. Böhm, C. Koch, and K. Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1151–1162. IEEE Computer Society, 2011.

[18] M. J. Carey, M. Livny, and H. Lu. Dynamic task allocation in a distributed database system. In *Proceedings of the 5th International Conference on Distributed Computing Systems, Denver, Colorado, USA, May 13-17, 1985*, pages 282–291. IEEE Computer Society, 1985.

[19] M. Chen, M. Lo, P. S. Yu, and H. C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In L. Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 15–26. Morgan Kaufmann, 1992.

[20] A. Crolotte and A. Ghazal. Introducing skew into the TPC-H benchmark. In R. O. Nambiar and M. Poess, editors, *Topics in Performance Evaluation, Measurement and Characterization - Third TPC Technology Conference, TPCTC 2011, Seattle, WA, USA, August 29-September 3, 2011, Revised Selected Papers*, volume 7144 of *Lecture Notes in Computer Science*, pages 137–145. Springer, 2011.

[21] D. L. Davison and G. Graefe. Memory-contention responsive hash joins. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 379–390. Morgan Kaufmann, 1994.

[22] D. L. Davison and G. Graefe. Dynamic resource brokering for multi-user query execution. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 281–292. ACM Press, 1995.

[23] D. J. DeWitt. The wisconsin benchmark: Past, present, and future. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (1st Edition)*, pages 119–165. Morgan Kaufmann, 1991.

[24] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A high performance dataflow database machine. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 228–237. Morgan Kaufmann, 1986.

[25] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[26] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[27] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, page 1–8, New York, NY, USA, 1984. Association for Computing Machinery.

[28] G. Dósa and J. Sgall. Optimal analysis of best fit bin packing. In J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 429–441. Springer, 2014.

[29] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 365–376. ACM Press, 1996.

[30] M. N. Garofalakis and Y. E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 296–305. Morgan Kaufmann, 1997.

[31] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H. Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1197–1208. ACM, 2013.

[32] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[33] G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in microsoft SQL server. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 86–97. Morgan Kaufmann, 1998.

[34] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition).* 1993.

[35] J. Gray. Database and transaction processing performance handbook. In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition).* 1993.

[36] J. Gray. A "measure of transaction processing" 20 years later. *IEEE Data Eng. Bull.*, 28(2):3–4, 2005.

[37] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about ad hoc join costs. *VLDB J.*, 6(3):241–256, 1997.

[38] R. Han, L. K. John, and J. Zhan. Benchmarking big data systems: A review. *IEEE Trans. Serv. Comput.*, 11(3):580–597, 2018.

[39] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. Deep learning models for selectivity estimation of multi-attribute queries. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1035–1050. ACM, 2020.

[40] S. Jahangiri. Re-evaluating the performance trade-offs for hash-based multi-join queries. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference, June 14-19, 2020*, pages 2845–2847, online conference [Portland, OR, USA], 2020. ACM.

[41] S. Jahangiri. "wisconsin benchmark data generator", 2020.

[42] S. Jahangiri. Wisconsin benchmark data generator: To JSON and beyond. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2887–2889. ACM, 2021.

[43] S. Jahangiri, M. J. Carey, and J. Freytag. Design trade-offs for a robust dynamic hybrid hash join (extended version). *CoRR*, abs/2112.02480, 2021.

[44] T. Kim, A. Behm, M. Blow, V. R. Borkar, Y. Bu, M. J. Carey, M. A. Hubail, S. Jahangiri, J. Jia, C. Li, C. Luo, I. Maxon, and P. Pirzadeh. Robust and efficient memory management in apache asterixdb. *Softw. Pract. Exp.*, 50(7):1114–1151, 2020.

[45] R. P. King. Disk arm movement in anticipation of future requests. *ACM Trans. Comput. Syst.*, 8(3):214–229, 1990.

[46] A. Kipf, D. Vorona, J. Müller, T. Kipf, B. Radke, V. Leis, P. A. Boncz, T. Neumann, and A. Kemper. Estimating cardinalities with deep sketches. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1937–1940. ACM, 2019.

[47] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In P. M. G. Apers and G. Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, pages 257–266. Morgan Kaufmann, 1989.

[48] S. Krompass, U. Dayal, H. A. Kuno, and A. Kemper. Dynamic workload management for very large data warehouses: Juggling feathers and bowling balls. In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas, and E. J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1105–1115. ACM, 2007.

[49] S. Krompass, H. A. Kuno, K. Wilkinson, U. Dayal, and A. Kemper. Adaptive query scheduling for mixed database workloads with multiple objectives. In S. Babu and G. N. Paulley, editors, *Proceedings of the Third International Workshop on Testing Database Systems, DBTest 2010, Indianapolis, Indiana, USA, June 7, 2010.* ACM, 2010.

[50] F. M. Liang. A lower bound for on-line bin packing. *Inf. Process. Lett.*, 10(2):76–79, 1980.

[51] B. Liu and E. A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P. Larson, and B. C. Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 829–840. ACM, 2005.

[52] F. Liu and S. Blanas. Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In S. Ghandeharizadeh, S. Barahmand, M. Balazinska, and M. J. Freedman, editors, *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, pages 153–166. ACM, 2015.

[53] W. C. Lynch. Do disk arms move? *SIGMETRICS Perform. Evaluation Rev.*, 1(4):3–16, 1972.

[54] P. Martin, P. Larson, and V. Deshpande. Parallel hash-based join algorithms for a shared-everything. *IEEE Trans. Knowl. Data Eng.*, 6(5):750–763, 1994.

[55] M. L. McAuliffe, M. J. Carey, and M. H. Solomon. Towards effective and efficient free space management. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 389–400. ACM Press, 1996.

[56] M. Mehta and D. J. DeWitt. Dynamic memory allocation for multiple-query workloads. In R. Agrawal, S. Baker, and D. A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 354–367. Morgan Kaufmann, 1993.

[57] M. Mehta and D. J. DeWitt. Managing intra-operator parallelism in parallel database systems. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 382–394. Morgan Kaufmann, 1995.

[58] F. Morvan and A. Hameurlain. Dynamic memory allocation strategies for parallel query execution. In G. B. Lamont, H. Haddad, G. A. Papadopoulos, and B. Panda, editors, *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC), March 10-14, 2002, Madrid, Spain*, pages 897–901. ACM, 2002.

[59] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In F. Bancilhon and D. J. DeWitt, editors, *Fourteenth International Conference on Very Large Data Bases, August 29 - September 1, 1988, Los Angeles, California, USA, Proceedings*, pages 468–478. Morgan Kaufmann, 1988.

[60] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. *CoRR*, abs/1405.3631, 2014.

[61] H. Pang, M. J. Carey, and M. Livny. Partially preemptive hash joins. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 59–68. ACM Press, 1993.

[62] H. Pang, M. J. Carey, and M. Livny. Managing memory for real-time queries. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, pages 221–232. ACM Press, 1994.

[63] F. Raab. TPC-C - the standard benchmark for online transaction processing (OLTP). In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[64] D. Rotem. Analysis of disk arm movement for large sequential reads. In M. Y. Vardi and P. C. Kanellakis, editors, *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*, pages 47–54. ACM Press, 1992.

[65] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, pages 110–121. ACM Press, 1989.

[66] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 469–480. Morgan Kaufmann, 1990.

[67] O. Serlin. The history of debitcredit and the TPC. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. 1993.

[68] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[69] A. M. van Wezenbeek and W. J. Withagen. A survey of memory management. *Microprocess. Microprogramming*, 36(3):141–162, 1993.

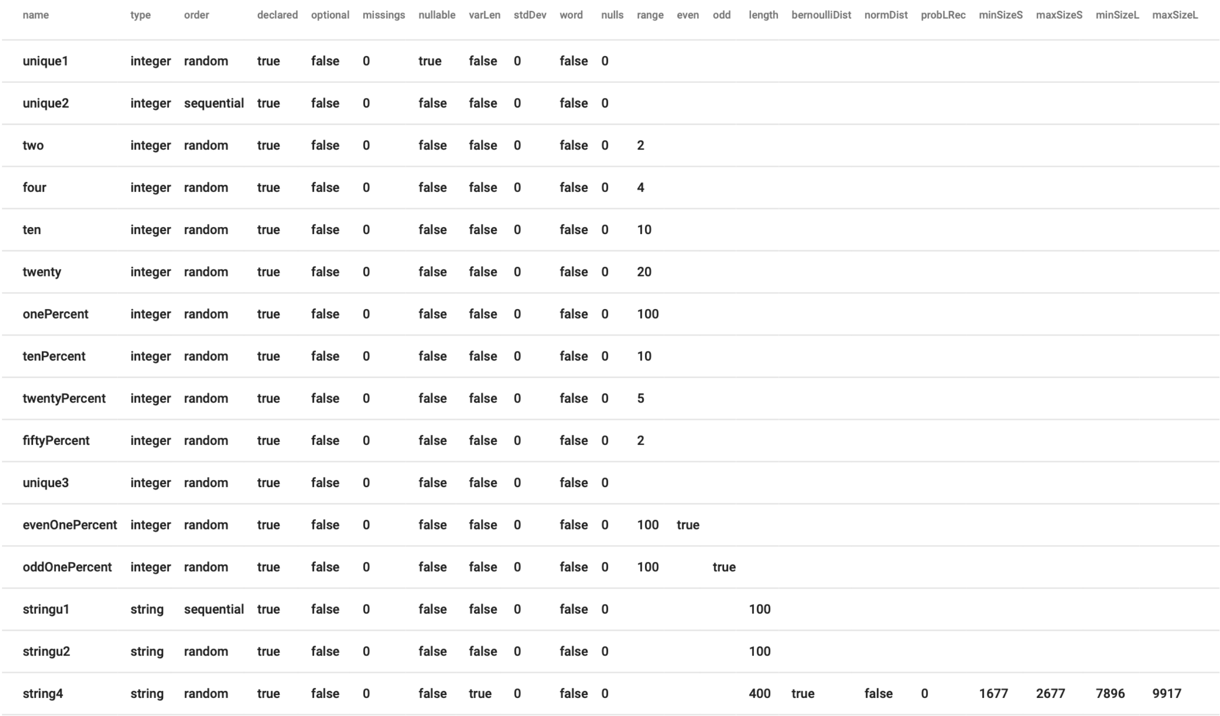
[70] A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallel evaluation of multi-join queries. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 115–126. ACM Press, 1995.

[71] P. S. Yu and D. W. Cornell. Buffer management based on return on consumption in a multi-query environment. *VLDB J.*, 2(1):1–37, 1993.

[72] M. Ziane, M. Zaït, and P. Borla-Salamet. Parallel query processing with zigzag trees. *VLDB J.*, 2(3):277–301, 1993.

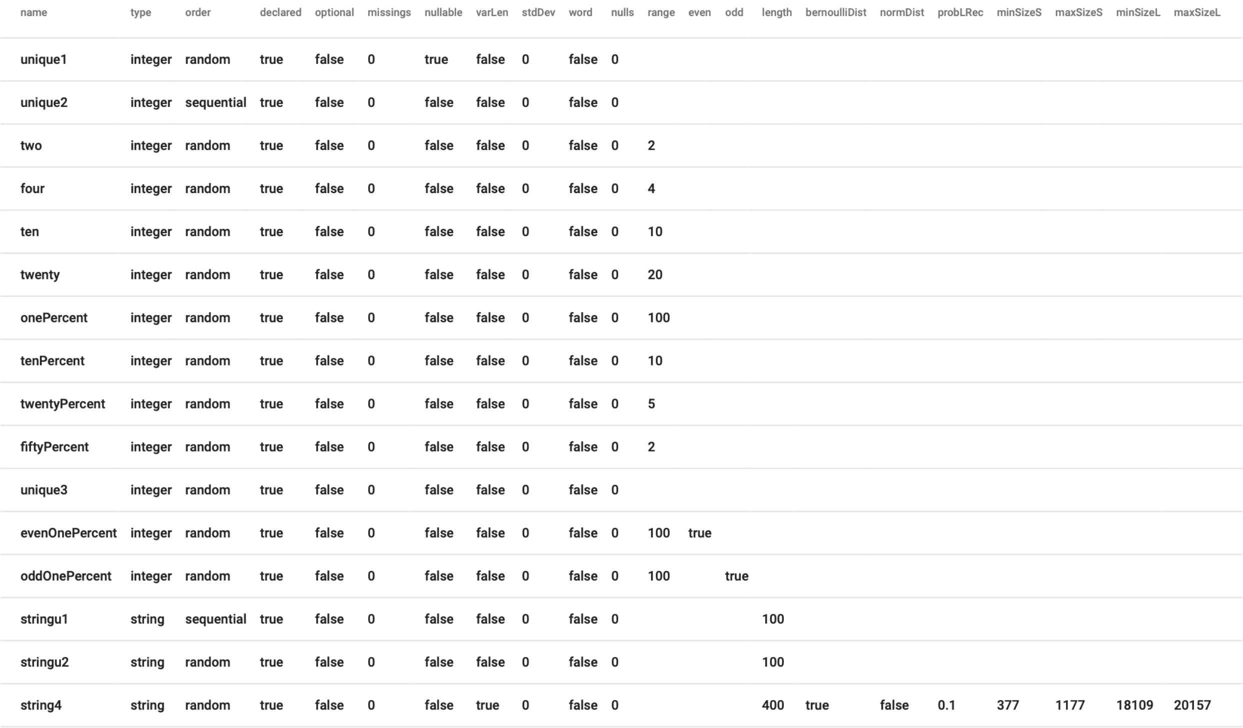
# Appendix A

# Datasets' Schema



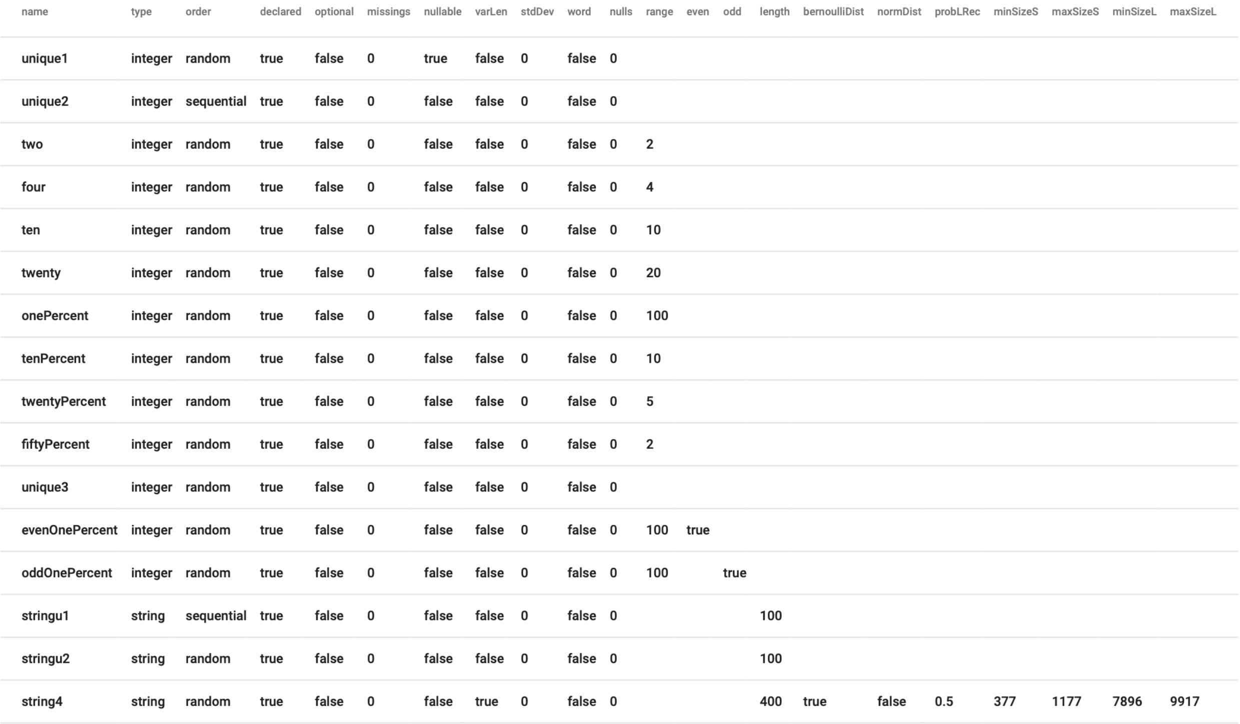
| name | type | order | declared | optional | missings | nullable | varLen | stdDev | word | nulls | range | even | odd | length | bernoulliDist | normDist | probLRec | minSizeS | maxSizeS | minSizeL | maxSizeL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| unique1 | integer | random | true | false | 0 | true | false | 0 | false | 0 | | | | | | | | | | | |
| unique2 | integer | sequential | true | false | 0 | false | false | 0 | false | 0 | | | | | | | | | | | |
| two | integer | random | true | false | 0 | false | false | 0 | false | 0 | 2 | | | | | | | | | | |
| four | integer | random | true | false | 0 | false | false | 0 | false | 0 | 4 | | | | | | | | | | |
| ten | integer | random | true | false | 0 | false | false | 0 | false | 0 | 10 | | | | | | | | | | |
| twenty | integer | random | true | false | 0 | false | false | 0 | false | 0 | 20 | | | | | | | | | | |
| onePercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 100 | | | | | | | | | | |
| tenPercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 10 | | | | | | | | | | |
| twentyPercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 5 | | | | | | | | | | |
| fiftyPercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 2 | | | | | | | | | | |
| unique3 | integer | random | true | false | 0 | false | false | 0 | false | 0 | | | | | | | | | | | |
| evenOnePercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 100 | true | | | | | | | | | |
| oddOnePercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 100 | | true | | | | | | | | |
| stringu1 | string | sequential | true | false | 0 | false | false | 0 | false | 0 | | | | 100 | | | | | | | |
| stringu2 | string | random | true | false | 0 | false | false | 0 | false | 0 | | | | 100 | | | | | | | |
| string4 | string | random | true | false | 0 | false | true | 0 | false | 0 | | | | 400 | true | false | 0 | 1677 | 2677 | 7896 | 9917 |

Figure A.1: Schema for All Small Records Dataset



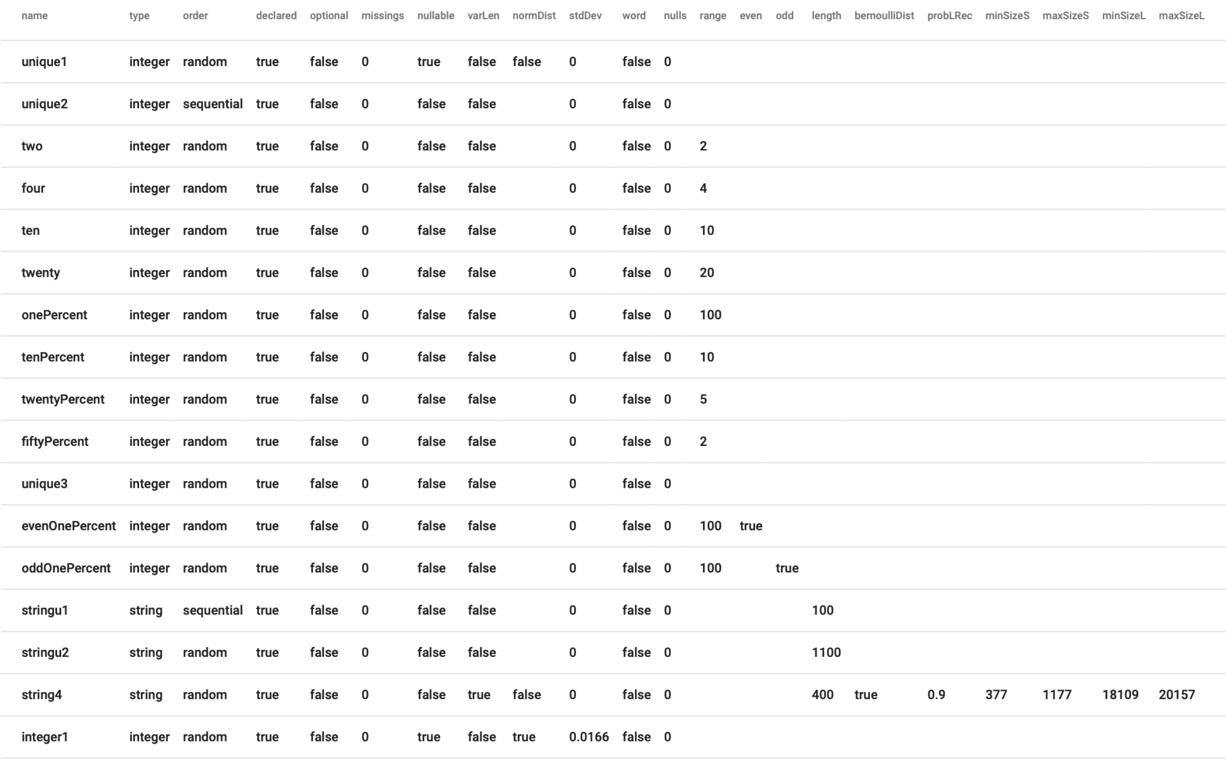
| name | type | order | declared | optional | missings | nullable | varLen | stdDev | word | nulls | range | even | odd | length | bernoulliDist | normDist | probLRec | minSizeS | maxSizeS | minSizeL | maxSizeL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| unique1 | integer | random | true | false | 0 | true | false | 0 | false | 0 | | | | | | | | | | | |
| unique2 | integer | sequential | true | false | 0 | false | false | 0 | false | 0 | | | | | | | | | | | |
| two | integer | random | true | false | 0 | false | false | 0 | false | 0 | 2 | | | | | | | | | | |
| four | integer | random | true | false | 0 | false | false | 0 | false | 0 | 4 | | | | | | | | | | |
| ten | integer | random | true | false | 0 | false | false | 0 | false | 0 | 10 | | | | | | | | | | |
| twenty | integer | random | true | false | 0 | false | false | 0 | false | 0 | 20 | | | | | | | | | | |
| onePercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 100 | | | | | | | | | | |
| tenPercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 10 | | | | | | | | | | |
| twentyPercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 5 | | | | | | | | | | |
| fiftyPercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 2 | | | | | | | | | | |
| unique3 | integer | random | true | false | 0 | false | false | 0 | false | 0 | | | | | | | | | | | |
| evenOnePercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 100 | true | | | | | | | | | |
| oddOnePercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 100 | | true | | | | | | | | |
| stringu1 | string | sequential | true | false | 0 | false | false | 0 | false | 0 | | | | 100 | | | | | | | |
| stringu2 | string | random | true | false | 0 | false | false | 0 | false | 0 | | | | 100 | | | | | | | |
| string4 | string | random | true | false | 0 | false | true | 0 | false | 0 | | | | 400 | true | false | 0.1 | 377 | 1177 | 18109 | 20157 |

Figure A.2: Schema for 1-Large Record Coexist - 10% Large Dataset



| name | type | order | declared | optional | missings | nullable | varLen | stdDev | word | nulls | range | even | odd | length | bernoulliDist | normDist | probLRec | minSizeS | maxSizeS | minSizeL | maxSizeL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| unique1 | integer | random | true | false | 0 | true | false | 0 | false | 0 | | | | | | | | | | | |
| unique2 | integer | sequential | true | false | 0 | false | false | 0 | false | 0 | | | | | | | | | | | |
| two | integer | random | true | false | 0 | false | false | 0 | false | 0 | 2 | | | | | | | | | | |
| four | integer | random | true | false | 0 | false | false | 0 | false | 0 | 4 | | | | | | | | | | |
| ten | integer | random | true | false | 0 | false | false | 0 | false | 0 | 10 | | | | | | | | | | |
| twenty | integer | random | true | false | 0 | false | false | 0 | false | 0 | 20 | | | | | | | | | | |
| onePercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 100 | | | | | | | | | | |
| tenPercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 10 | | | | | | | | | | |
| twentyPercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 5 | | | | | | | | | | |
| fiftyPercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 2 | | | | | | | | | | |
| unique3 | integer | random | true | false | 0 | false | false | 0 | false | 0 | | | | | | | | | | | |
| evenOnePercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 100 | true | | | | | | | | | |
| oddOnePercent | integer | random | true | false | 0 | false | false | 0 | false | 0 | 100 | | true | | | | | | | | |
| stringu1 | string | sequential | true | false | 0 | false | false | 0 | false | 0 | | | | 100 | | | | | | | |
| stringu2 | string | random | true | false | 0 | false | false | 0 | false | 0 | | | | 100 | | | | | | | |
| string4 | string | random | true | false | 0 | false | true | 0 | false | 0 | | | | 400 | true | false | 0.5 | 377 | 1177 | 7896 | 9917 |

Figure A.3: Schema for 3-Large Record Coexist - 50% Large Dataset

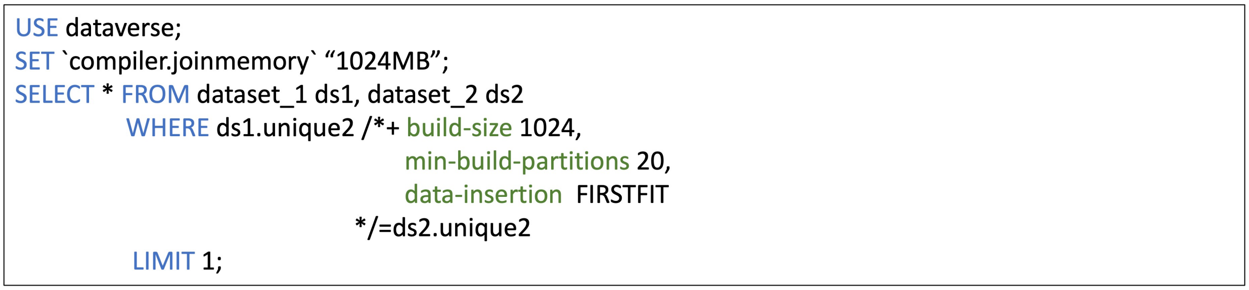


| name | type | order | declared | optional | missings | nullable | varLen | normDist | stdDev | word | nulls | range | even | odd | length | bernoulliDist | probLRec | minSizeS | maxSizeS | minSizeL | maxSizeL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| unique1 | integer | random | true | false | 0 | true | false | false | 0 | false | 0 | | | | | | | | | | |
| unique2 | integer | sequential | true | false | 0 | false | false | | 0 | false | 0 | | | | | | | | | | |
| two | integer | random | true | false | 0 | false | false | | 0 | false | 0 | 2 | | | | | | | | | |
| four | integer | random | true | false | 0 | false | false | | 0 | false | 0 | 4 | | | | | | | | | |
| ten | integer | random | true | false | 0 | false | false | | 0 | false | 0 | 10 | | | | | | | | | |
| twenty | integer | random | true | false | 0 | false | false | | 0 | false | 0 | 20 | | | | | | | | | |
| onePercent | integer | random | true | false | 0 | false | false | | 0 | false | 0 | 100 | | | | | | | | | |
| tenPercent | integer | random | true | false | 0 | false | false | | 0 | false | 0 | 10 | | | | | | | | | |
| twentyPercent | integer | random | true | false | 0 | false | false | | 0 | false | 0 | 5 | | | | | | | | | |
| fiftyPercent | integer | random | true | false | 0 | false | false | | 0 | false | 0 | 2 | | | | | | | | | |
| unique3 | integer | random | true | false | 0 | false | false | | 0 | false | 0 | | | | | | | | | | |
| evenOnePercent | integer | random | true | false | 0 | false | false | | 0 | false | 0 | 100 | true | | | | | | | | |
| oddOnePercent | integer | random | true | false | 0 | false | false | | 0 | false | 0 | 100 | | true | | | | | | | |
| stringu1 | string | sequential | true | false | 0 | false | false | | 0 | false | 0 | | | | 100 | | | | | | |
| stringu2 | string | random | true | false | 0 | false | false | | 0 | false | 0 | | | | 1100 | | | | | | |
| string4 | string | random | true | false | 0 | false | true | false | 0 | false | 0 | | | | 400 | true | 0.9 | 377 | 1177 | 18109 | 20157 |
| integer1 | integer | random | true | false | 0 | true | false | true | 0.0166 | false | 0 | | | | | | | | | | |

Figure A.4: Schema for 1-Large Record Coexist - 90% Large - Skewed Dataset

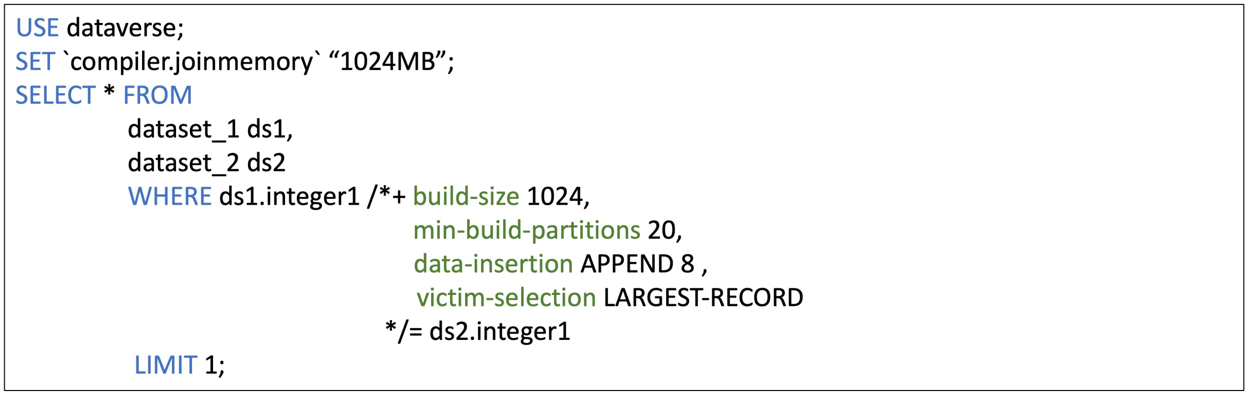
# Appendix B

# Query Examples and Templates



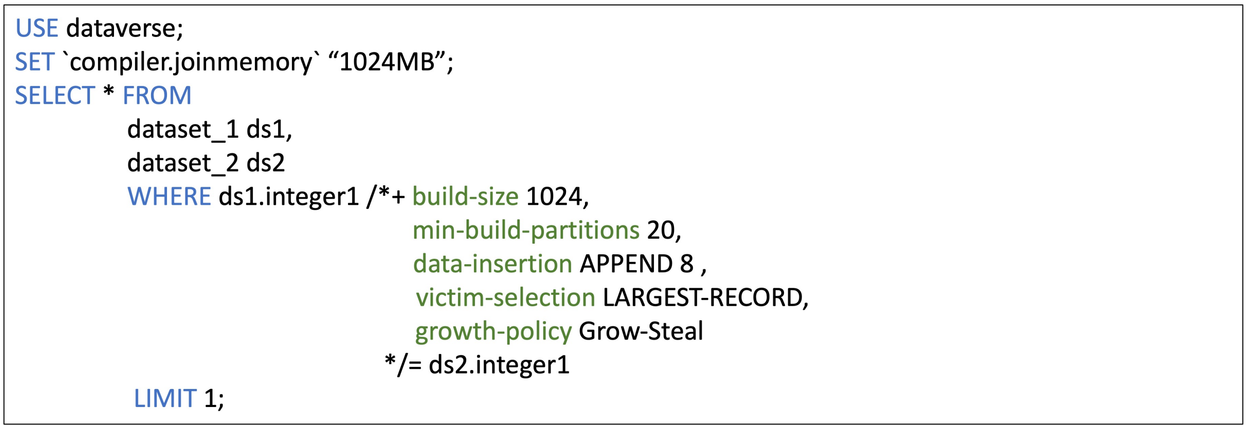
```
USE dataverse;
SET `compiler.joinmemory` “1024MB”;
SELECT * FROM dataset_1 ds1, dataset_2 ds2
            WHERE ds1.unique2 /*+ build-size 1024,
                                     min-build-partitions 20,
                                     data-insertion  FIRSTFIT
                                *//=ds2.unique2
             LIMIT 1;
```

Figure B.1: Partition Insertion - Query Example



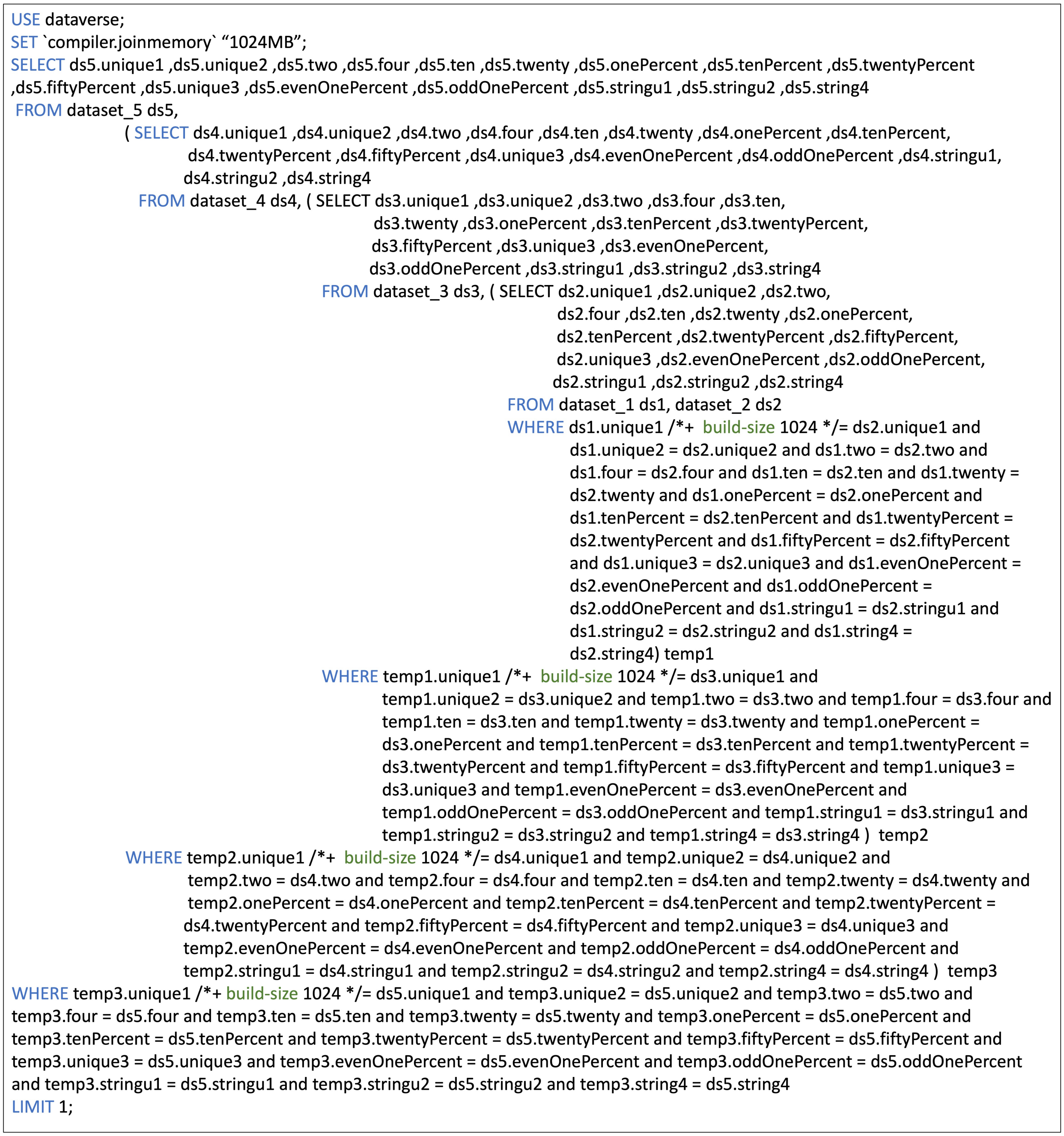
```
USE dataverse;
SET `compiler.joinmemory` “1024MB”;
SELECT * FROM
            dataset_1 ds1,
            dataset_2 ds2
            WHERE ds1.integer1 /*+ build-size 1024,
                                      min-build-partitions 20,
                                      data-insertion APPEND 8 ,
                                       victim-selection LARGEST-RECORD
                                   */= ds2.integer1
             LIMIT 1;
```

Figure B.2: Victim Selection Policy - Query Example



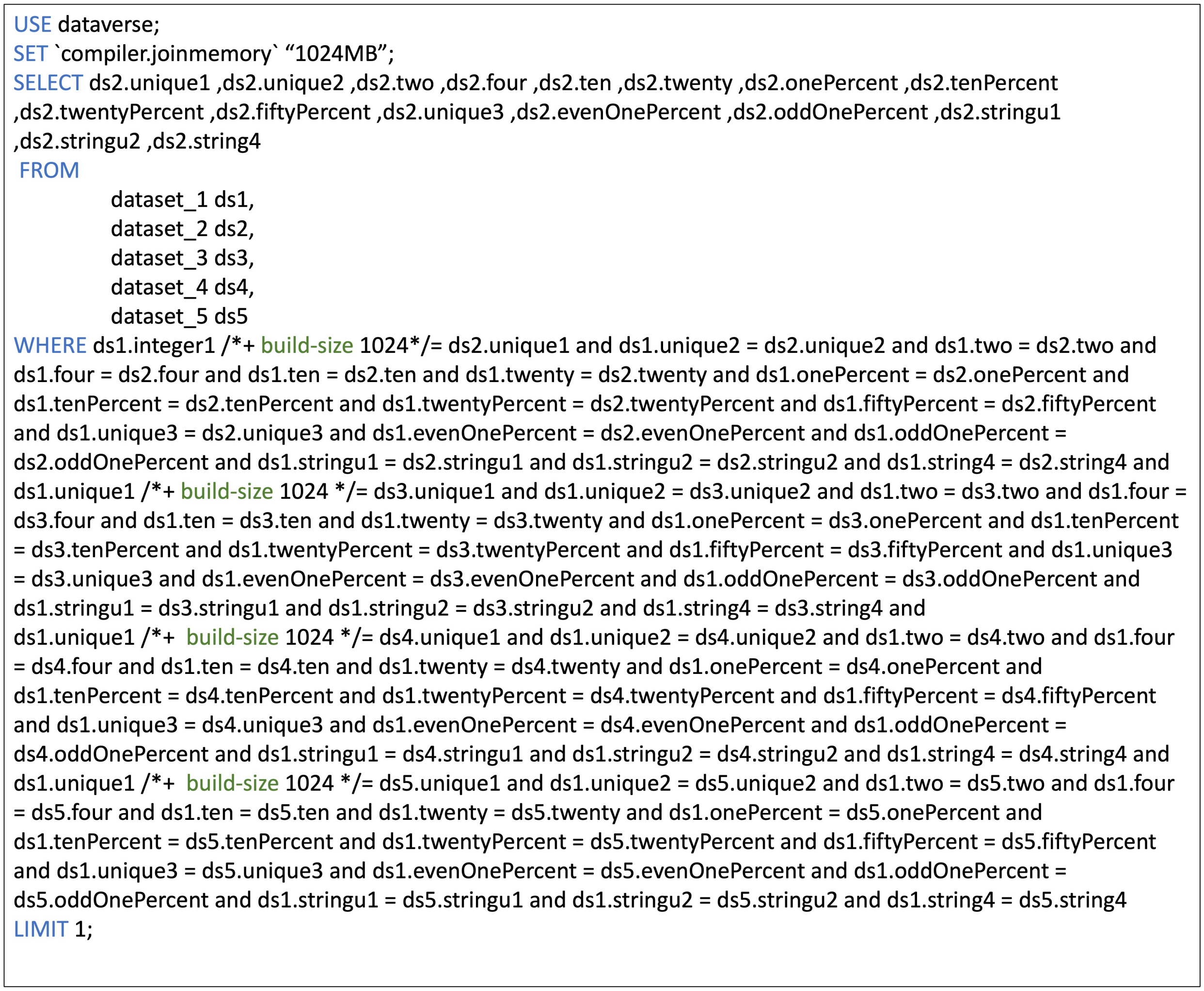
```
USE dataverse;
SET `compiler.joinmemory` “1024MB”;
SELECT * FROM
            dataset_1 ds1,
            dataset_2 ds2
            WHERE ds1.integer1 /*+ build-size 1024,
                                   min-build-partitions 20,
                                   data-insertion APPEND 8 ,
                                   victim-selection LARGEST-RECORD,
                                   growth-policy Grow-Steal
                                 */= ds2.integer1
             LIMIT 1;
```

Figure B.3: Growth Policy - Query Example



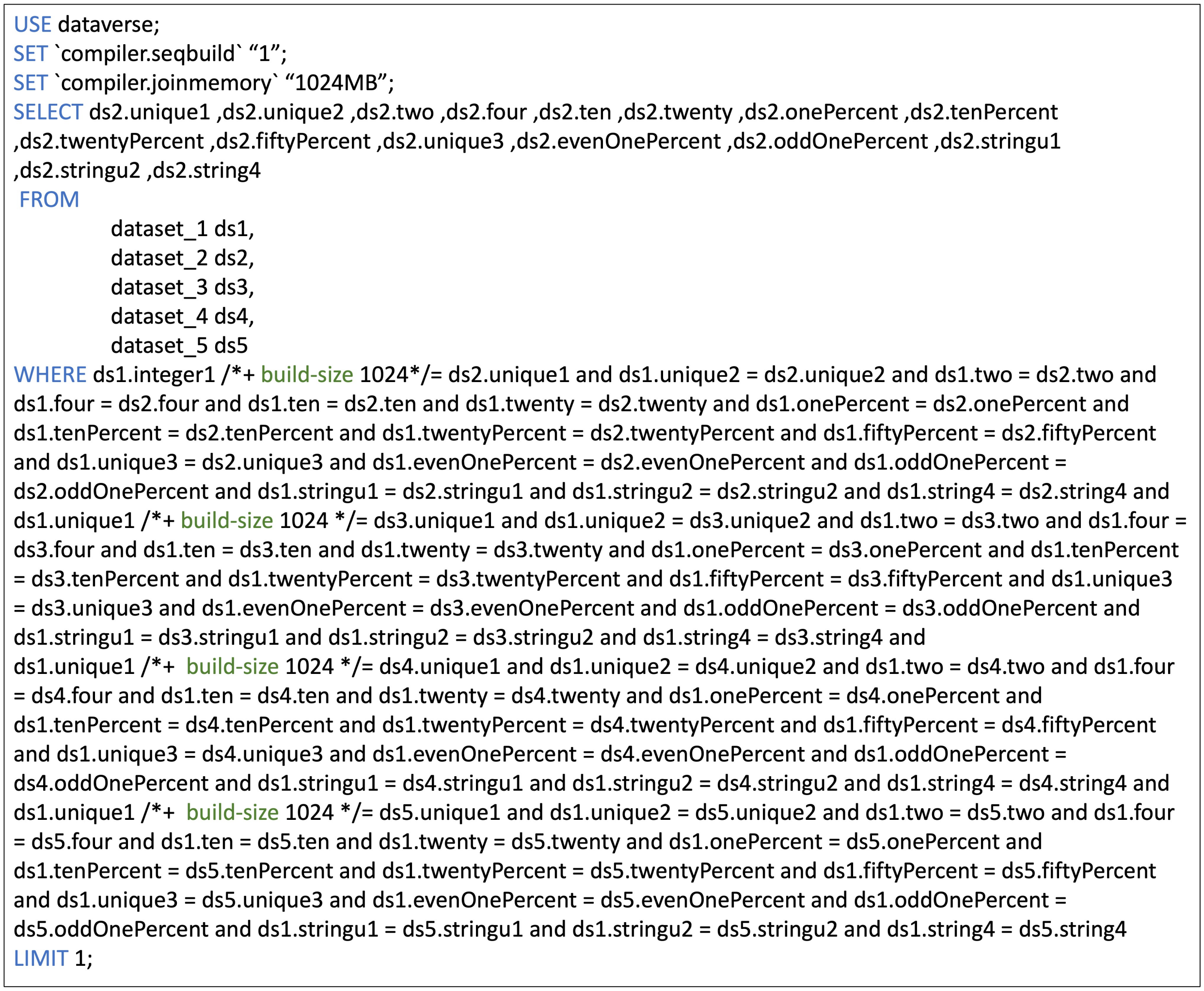
```
USE dataverse;
SET `compiler.joinmemory` “1024MB”;
SELECT ds5.unique1 ,ds5.unique2 ,ds5.two ,ds5.four ,ds5.ten ,ds5.twenty ,ds5.onePercent ,ds5.tenPercent ,ds5.twentyPercent
,ds5.fiftyPercent ,ds5.unique3 ,ds5.evenOnePercent ,ds5.oddOnePercent ,ds5.stringu1 ,ds5.stringu2 ,ds5.string4
 FROM dataset_5 ds5,
              ( SELECT ds4.unique1 ,ds4.unique2 ,ds4.two ,ds4.four ,ds4.ten ,ds4.twenty ,ds4.onePercent ,ds4.tenPercent,
                      ds4.twentyPercent ,ds4.fiftyPercent ,ds4.unique3 ,ds4.evenOnePercent ,ds4.oddOnePercent ,ds4.stringu1,
                      ds4.stringu2 ,ds4.string4
                FROM dataset_4 ds4, ( SELECT ds3.unique1 ,ds3.unique2 ,ds3.two ,ds3.four ,ds3.ten,
                                           ds3.twenty ,ds3.onePercent ,ds3.tenPercent ,ds3.twentyPercent,
                                           ds3.fiftyPercent ,ds3.unique3 ,ds3.evenOnePercent,
                                           ds3.oddOnePercent ,ds3.stringu1 ,ds3.stringu2 ,ds3.string4
                                      FROM dataset_3 ds3, ( SELECT ds2.unique1 ,ds2.unique2 ,ds2.two,
                                                              ds2.four ,ds2.ten ,ds2.twenty ,ds2.onePercent,
                                                              ds2.tenPercent ,ds2.twentyPercent ,ds2.fiftyPercent,
                                                              ds2.unique3 ,ds2.evenOnePercent ,ds2.oddOnePercent,
                                                              ds2.stringu1 ,ds2.stringu2 ,ds2.string4
                                                         FROM dataset_1 ds1, dataset_2 ds2
                                                         WHERE ds1.unique1 /*+  build-size 1024 */= ds2.unique1 and
                                                               ds1.unique2 = ds2.unique2 and ds1.two = ds2.two and
                                                               ds1.four = ds2.four and ds1.ten = ds2.ten and ds1.twenty =
                                                               ds2.twenty and ds1.onePercent = ds2.onePercent and
                                                               ds1.tenPercent = ds2.tenPercent and ds1.twentyPercent =
                                                               ds2.twentyPercent and ds1.fiftyPercent = ds2.fiftyPercent
                                                               and ds1.unique3 = ds2.unique3 and ds1.evenOnePercent =
                                                               ds2.evenOnePercent and ds1.oddOnePercent =
                                                               ds2.oddOnePercent and ds1.stringu1 = ds2.stringu1 and
                                                               ds1.stringu2 = ds2.stringu2 and ds1.string4 =
                                                               ds2.string4) temp1
                                      WHERE temp1.unique1 /*+  build-size 1024 */= ds3.unique1 and
                                            temp1.unique2 = ds3.unique2 and temp1.two = ds3.two and temp1.four = ds3.four and
                                            temp1.ten = ds3.ten and temp1.twenty = ds3.twenty and temp1.onePercent =
                                            ds3.onePercent and temp1.tenPercent = ds3.tenPercent and temp1.twentyPercent =
                                            ds3.twentyPercent and temp1.fiftyPercent = ds3.fiftyPercent and temp1.unique3 =
                                            ds3.unique3 and temp1.evenOnePercent = ds3.evenOnePercent and
                                            temp1.oddOnePercent = ds3.oddOnePercent and temp1.stringu1 = ds3.stringu1 and
                                            temp1.stringu2 = ds3.stringu2 and temp1.string4 = ds3.string4 )  temp2
              WHERE temp2.unique1 /*+  build-size 1024 */= ds4.unique1 and temp2.unique2 = ds4.unique2 and
                      temp2.two = ds4.two and temp2.four = ds4.four and temp2.ten = ds4.ten and temp2.twenty = ds4.twenty and
                      temp2.onePercent = ds4.onePercent and temp2.tenPercent = ds4.tenPercent and temp2.twentyPercent =
                      ds4.twentyPercent and temp2.fiftyPercent = ds4.fiftyPercent and temp2.unique3 = ds4.unique3 and
                      temp2.evenOnePercent = ds4.evenOnePercent and temp2.oddOnePercent = ds4.oddOnePercent and
                      temp2.stringu1 = ds4.stringu1 and temp2.stringu2 = ds4.stringu2 and temp2.string4 = ds4.string4 )  temp3
WHERE temp3.unique1 /*+ build-size 1024 */= ds5.unique1 and temp3.unique2 = ds5.unique2 and temp3.two = ds5.two and
temp3.four = ds5.four and temp3.ten = ds5.ten and temp3.twenty = ds5.twenty and temp3.onePercent = ds5.onePercent and
temp3.tenPercent = ds5.tenPercent and temp3.twentyPercent = ds5.twentyPercent and temp3.fiftyPercent = ds5.fiftyPercent and
temp3.unique3 = ds5.unique3 and temp3.evenOnePercent = ds5.evenOnePercent and temp3.oddOnePercent = ds5.oddOnePercent
and temp3.stringu1 = ds5.stringu1 and temp3.stringu2 = ds5.stringu2 and temp3.string4 = ds5.string4
LIMIT 1;
```

Figure B.4: Left Deep Tree - Query Example



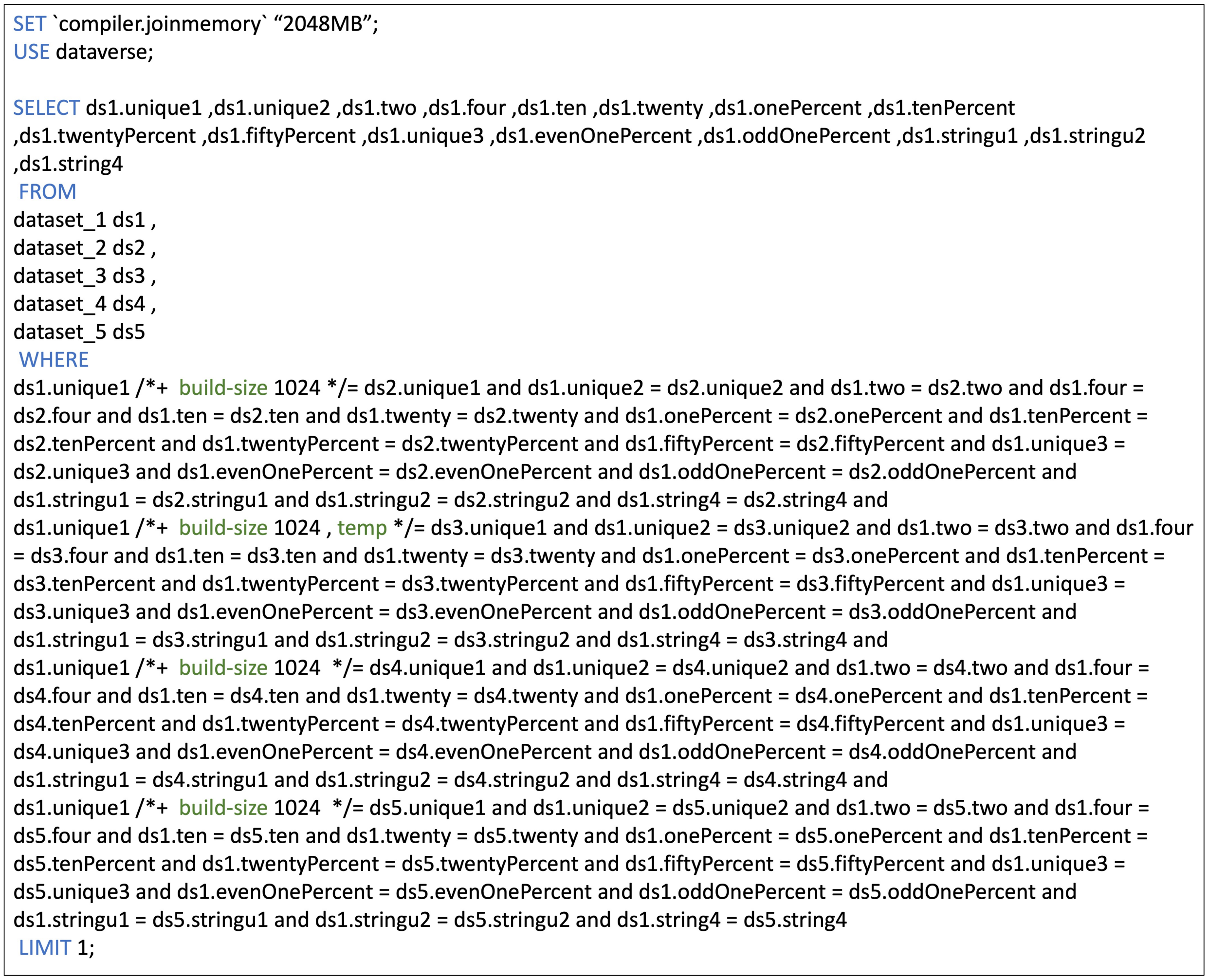
```
USE dataverse;
SET `compiler.joinmemory` “1024MB”;
SELECT ds2.unique1 ,ds2.unique2 ,ds2.two ,ds2.four ,ds2.ten ,ds2.twenty ,ds2.onePercent ,ds2.tenPercent
,ds2.twentyPercent ,ds2.fiftyPercent ,ds2.unique3 ,ds2.evenOnePercent ,ds2.oddOnePercent ,ds2.stringu1
,ds2.stringu2 ,ds2.string4
 FROM
            dataset_1 ds1,
            dataset_2 ds2,
            dataset_3 ds3,
            dataset_4 ds4,
            dataset_5 ds5
WHERE ds1.integer1 /*+ build-size 1024*/= ds2.unique1 and ds1.unique2 = ds2.unique2 and ds1.two = ds2.two and
ds1.four = ds2.four and ds1.ten = ds2.ten and ds1.twenty = ds2.twenty and ds1.onePercent = ds2.onePercent and
ds1.tenPercent = ds2.tenPercent and ds1.twentyPercent = ds2.twentyPercent and ds1.fiftyPercent = ds2.fiftyPercent
and ds1.unique3 = ds2.unique3 and ds1.evenOnePercent = ds2.evenOnePercent and ds1.oddOnePercent =
ds2.oddOnePercent and ds1.stringu1 = ds2.stringu1 and ds1.stringu2 = ds2.stringu2 and ds1.string4 = ds2.string4 and
ds1.unique1 /*+ build-size 1024 */= ds3.unique1 and ds1.unique2 = ds3.unique2 and ds1.two = ds3.two and ds1.four =
ds3.four and ds1.ten = ds3.ten and ds1.twenty = ds3.twenty and ds1.onePercent = ds3.onePercent and ds1.tenPercent
= ds3.tenPercent and ds1.twentyPercent = ds3.twentyPercent and ds1.fiftyPercent = ds3.fiftyPercent and ds1.unique3
= ds3.unique3 and ds1.evenOnePercent = ds3.evenOnePercent and ds1.oddOnePercent = ds3.oddOnePercent and
ds1.stringu1 = ds3.stringu1 and ds1.stringu2 = ds3.stringu2 and ds1.string4 = ds3.string4 and
ds1.unique1 /*+  build-size 1024 */= ds4.unique1 and ds1.unique2 = ds4.unique2 and ds1.two = ds4.two and ds1.four
= ds4.four and ds1.ten = ds4.ten and ds1.twenty = ds4.twenty and ds1.onePercent = ds4.onePercent and
ds1.tenPercent = ds4.tenPercent and ds1.twentyPercent = ds4.twentyPercent and ds1.fiftyPercent = ds4.fiftyPercent
and ds1.unique3 = ds4.unique3 and ds1.evenOnePercent = ds4.evenOnePercent and ds1.oddOnePercent =
ds4.oddOnePercent and ds1.stringu1 = ds4.stringu1 and ds1.stringu2 = ds4.stringu2 and ds1.string4 = ds4.string4 and
ds1.unique1 /*+  build-size 1024 */= ds5.unique1 and ds1.unique2 = ds5.unique2 and ds1.two = ds5.two and ds1.four
= ds5.four and ds1.ten = ds5.ten and ds1.twenty = ds5.twenty and ds1.onePercent = ds5.onePercent and
ds1.tenPercent = ds5.tenPercent and ds1.twentyPercent = ds5.twentyPercent and ds1.fiftyPercent = ds5.fiftyPercent
and ds1.unique3 = ds5.unique3 and ds1.evenOnePercent = ds5.evenOnePercent and ds1.oddOnePercent =
ds5.oddOnePercent and ds1.stringu1 = ds5.stringu1 and ds1.stringu2 = ds5.stringu2 and ds1.string4 = ds5.string4
LIMIT 1;
```

Figure B.5: Right Deep Tree - Query Example



```
USE dataverse;
SET `compiler.seqbuild` “1”;
SET `compiler.joinmemory` “1024MB”;
SELECT ds2.unique1 ,ds2.unique2 ,ds2.two ,ds2.four ,ds2.ten ,ds2.twenty ,ds2.onePercent ,ds2.tenPercent
,ds2.twentyPercent ,ds2.fiftyPercent ,ds2.unique3 ,ds2.evenOnePercent ,ds2.oddOnePercent ,ds2.stringu1
,ds2.stringu2 ,ds2.string4
 FROM
            dataset_1 ds1,
            dataset_2 ds2,
            dataset_3 ds3,
            dataset_4 ds4,
            dataset_5 ds5
WHERE ds1.integer1 /*+ build-size 1024*/= ds2.unique1 and ds1.unique2 = ds2.unique2 and ds1.two = ds2.two and
ds1.four = ds2.four and ds1.ten = ds2.ten and ds1.twenty = ds2.twenty and ds1.onePercent = ds2.onePercent and
ds1.tenPercent = ds2.tenPercent and ds1.twentyPercent = ds2.twentyPercent and ds1.fiftyPercent = ds2.fiftyPercent
and ds1.unique3 = ds2.unique3 and ds1.evenOnePercent = ds2.evenOnePercent and ds1.oddOnePercent =
ds2.oddOnePercent and ds1.stringu1 = ds2.stringu1 and ds1.stringu2 = ds2.stringu2 and ds1.string4 = ds2.string4 and
ds1.unique1 /*+ build-size 1024 */= ds3.unique1 and ds1.unique2 = ds3.unique2 and ds1.two = ds3.two and ds1.four =
ds3.four and ds1.ten = ds3.ten and ds1.twenty = ds3.twenty and ds1.onePercent = ds3.onePercent and ds1.tenPercent
= ds3.tenPercent and ds1.twentyPercent = ds3.twentyPercent and ds1.fiftyPercent = ds3.fiftyPercent and ds1.unique3
= ds3.unique3 and ds1.evenOnePercent = ds3.evenOnePercent and ds1.oddOnePercent = ds3.oddOnePercent and
ds1.stringu1 = ds3.stringu1 and ds1.stringu2 = ds3.stringu2 and ds1.string4 = ds3.string4 and
ds1.unique1 /*+  build-size 1024 */= ds4.unique1 and ds1.unique2 = ds4.unique2 and ds1.two = ds4.two and ds1.four
= ds4.four and ds1.ten = ds4.ten and ds1.twenty = ds4.twenty and ds1.onePercent = ds4.onePercent and
ds1.tenPercent = ds4.tenPercent and ds1.twentyPercent = ds4.twentyPercent and ds1.fiftyPercent = ds4.fiftyPercent
and ds1.unique3 = ds4.unique3 and ds1.evenOnePercent = ds4.evenOnePercent and ds1.oddOnePercent =
ds4.oddOnePercent and ds1.stringu1 = ds4.stringu1 and ds1.stringu2 = ds4.stringu2 and ds1.string4 = ds4.string4 and
ds1.unique1 /*+  build-size 1024 */= ds5.unique1 and ds1.unique2 = ds5.unique2 and ds1.two = ds5.two and ds1.four
= ds5.four and ds1.ten = ds5.ten and ds1.twenty = ds5.twenty and ds1.onePercent = ds5.onePercent and
ds1.tenPercent = ds5.tenPercent and ds1.twentyPercent = ds5.twentyPercent and ds1.fiftyPercent = ds5.fiftyPercent
and ds1.unique3 = ds5.unique3 and ds1.evenOnePercent = ds5.evenOnePercent and ds1.oddOnePercent =
ds5.oddOnePercent and ds1.stringu1 = ds5.stringu1 and ds1.stringu2 = ds5.stringu2 and ds1.string4 = ds5.string4
LIMIT 1;
```

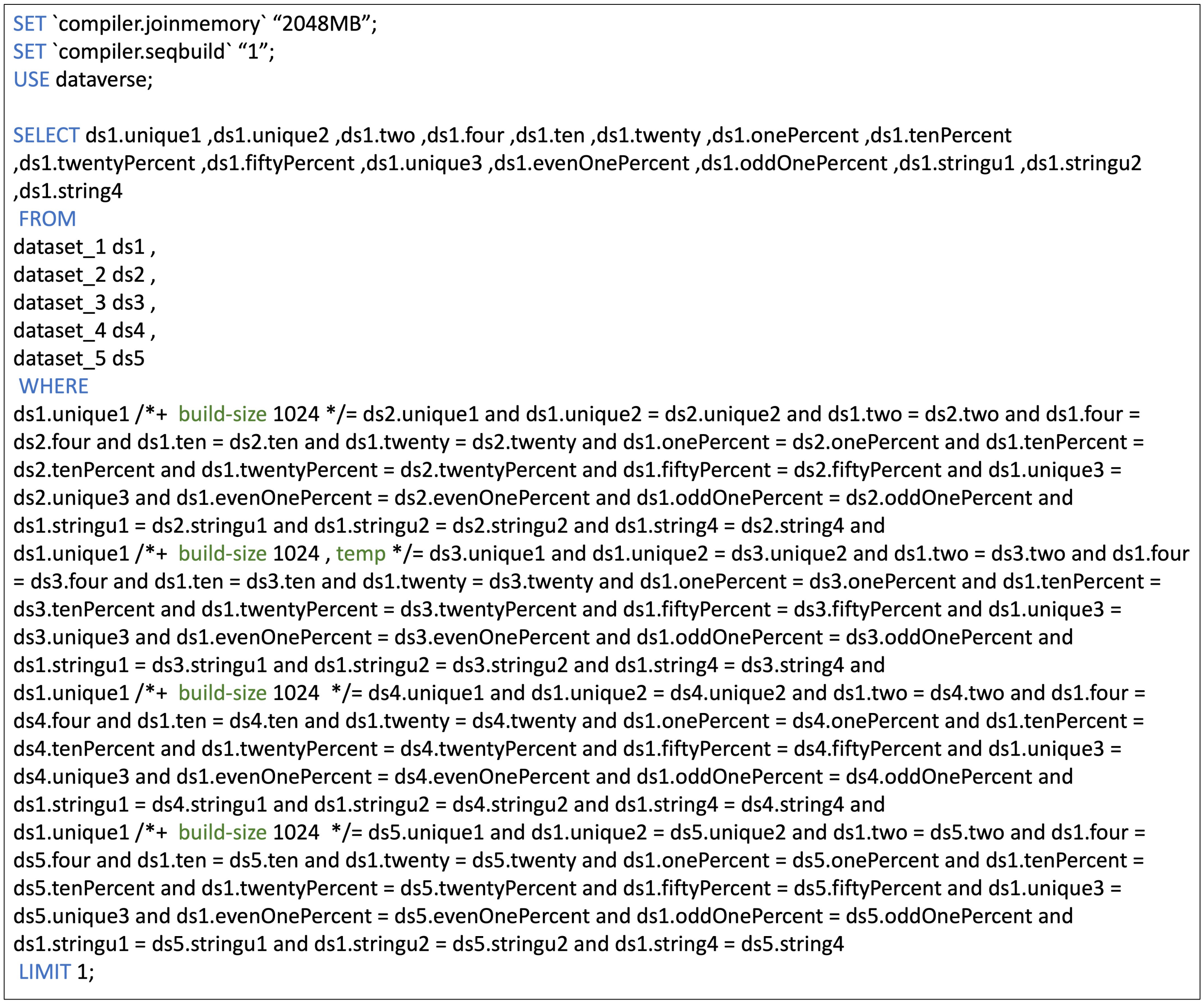
Figure B.6: Sequential Right Deep Tree - Query Example



```
SET `compiler.joinmemory` “2048MB”;
USE dataverse;

SELECT ds1.unique1 ,ds1.unique2 ,ds1.two ,ds1.four ,ds1.ten ,ds1.twenty ,ds1.onePercent ,ds1.tenPercent
,ds1.twentyPercent ,ds1.fiftyPercent ,ds1.unique3 ,ds1.evenOnePercent ,ds1.oddOnePercent ,ds1.stringu1 ,ds1.stringu2
,ds1.string4
 FROM
dataset_1 ds1 ,
dataset_2 ds2 ,
dataset_3 ds3 ,
dataset_4 ds4 ,
dataset_5 ds5
 WHERE
ds1.unique1 /*+  build-size 1024 */= ds2.unique1 and ds1.unique2 = ds2.unique2 and ds1.two = ds2.two and ds1.four =
ds2.four and ds1.ten = ds2.ten and ds1.twenty = ds2.twenty and ds1.onePercent = ds2.onePercent and ds1.tenPercent =
ds2.tenPercent and ds1.twentyPercent = ds2.twentyPercent and ds1.fiftyPercent = ds2.fiftyPercent and ds1.unique3 =
ds2.unique3 and ds1.evenOnePercent = ds2.evenOnePercent and ds1.oddOnePercent = ds2.oddOnePercent and
ds1.stringu1 = ds2.stringu1 and ds1.stringu2 = ds2.stringu2 and ds1.string4 = ds2.string4 and
ds1.unique1 /*+  build-size 1024 , temp */= ds3.unique1 and ds1.unique2 = ds3.unique2 and ds1.two = ds3.two and ds1.four
= ds3.four and ds1.ten = ds3.ten and ds1.twenty = ds3.twenty and ds1.onePercent = ds3.onePercent and ds1.tenPercent =
ds3.tenPercent and ds1.twentyPercent = ds3.twentyPercent and ds1.fiftyPercent = ds3.fiftyPercent and ds1.unique3 =
ds3.unique3 and ds1.evenOnePercent = ds3.evenOnePercent and ds1.oddOnePercent = ds3.oddOnePercent and
ds1.stringu1 = ds3.stringu1 and ds1.stringu2 = ds3.stringu2 and ds1.string4 = ds3.string4 and
ds1.unique1 /*+  build-size 1024  */= ds4.unique1 and ds1.unique2 = ds4.unique2 and ds1.two = ds4.two and ds1.four =
ds4.four and ds1.ten = ds4.ten and ds1.twenty = ds4.twenty and ds1.onePercent = ds4.onePercent and ds1.tenPercent =
ds4.tenPercent and ds1.twentyPercent = ds4.twentyPercent and ds1.fiftyPercent = ds4.fiftyPercent and ds1.unique3 =
ds4.unique3 and ds1.evenOnePercent = ds4.evenOnePercent and ds1.oddOnePercent = ds4.oddOnePercent and
ds1.stringu1 = ds4.stringu1 and ds1.stringu2 = ds4.stringu2 and ds1.string4 = ds4.string4 and
ds1.unique1 /*+  build-size 1024  */= ds5.unique1 and ds1.unique2 = ds5.unique2 and ds1.two = ds5.two and ds1.four =
ds5.four and ds1.ten = ds5.ten and ds1.twenty = ds5.twenty and ds1.onePercent = ds5.onePercent and ds1.tenPercent =
ds5.tenPercent and ds1.twentyPercent = ds5.twentyPercent and ds1.fiftyPercent = ds5.fiftyPercent and ds1.unique3 =
ds5.unique3 and ds1.evenOnePercent = ds5.evenOnePercent and ds1.oddOnePercent = ds5.oddOnePercent and
ds1.stringu1 = ds5.stringu1 and ds1.stringu2 = ds5.stringu2 and ds1.string4 = ds5.string4
 LIMIT 1;
```

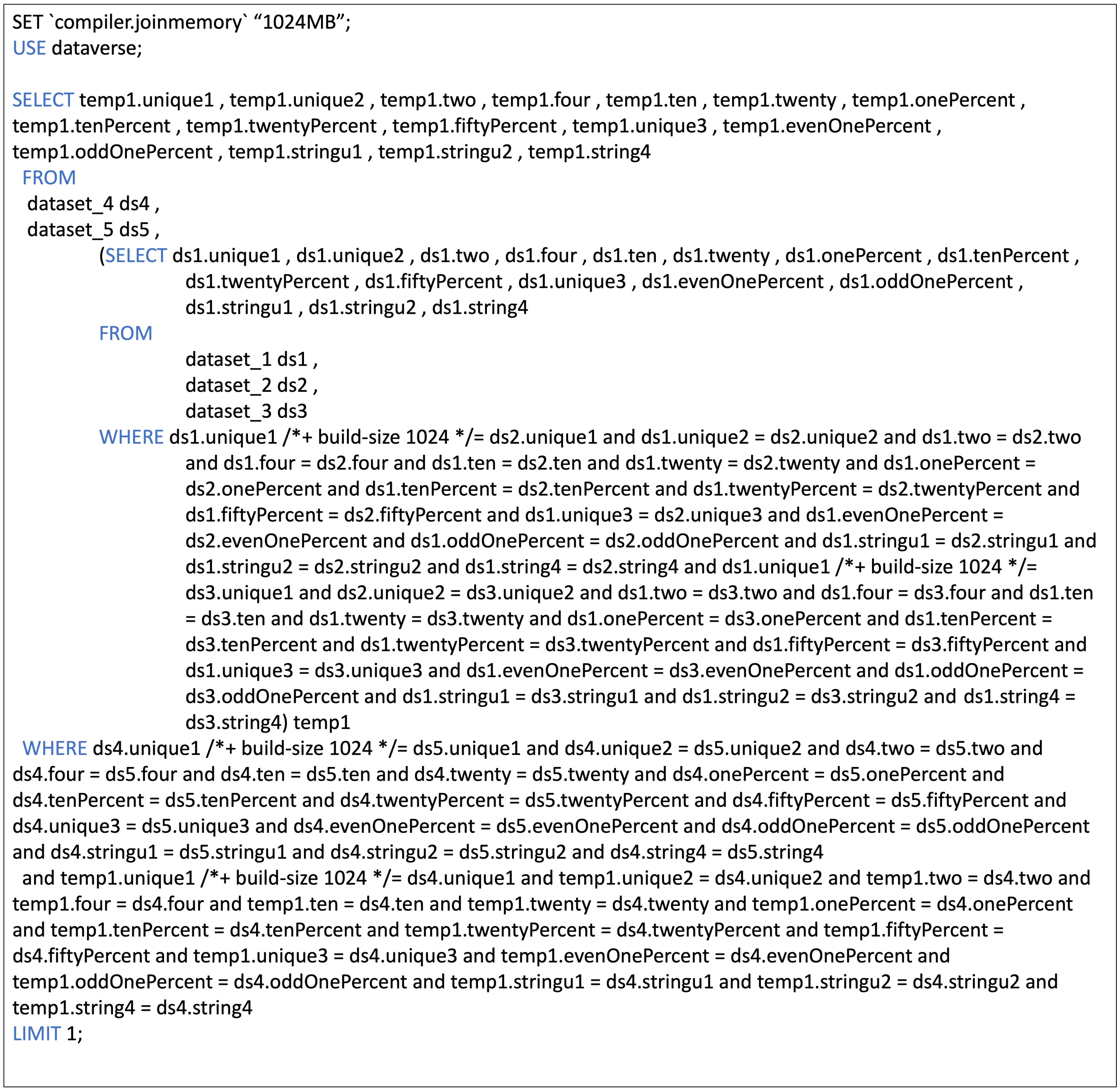
Figure B.7: Static Right Deep Tree - Query Example



```
SET `compiler.joinmemory` “2048MB”;
SET `compiler.seqbuild` “1”;
USE dataverse;

SELECT ds1.unique1 ,ds1.unique2 ,ds1.two ,ds1.four ,ds1.ten ,ds1.twenty ,ds1.onePercent ,ds1.tenPercent
,ds1.twentyPercent ,ds1.fiftyPercent ,ds1.unique3 ,ds1.evenOnePercent ,ds1.oddOnePercent ,ds1.stringu1 ,ds1.stringu2
,ds1.string4
 FROM
dataset_1 ds1 ,
dataset_2 ds2 ,
dataset_3 ds3 ,
dataset_4 ds4 ,
dataset_5 ds5
 WHERE
ds1.unique1 /*+  build-size 1024 */= ds2.unique1 and ds1.unique2 = ds2.unique2 and ds1.two = ds2.two and ds1.four =
ds2.four and ds1.ten = ds2.ten and ds1.twenty = ds2.twenty and ds1.onePercent = ds2.onePercent and ds1.tenPercent =
ds2.tenPercent and ds1.twentyPercent = ds2.twentyPercent and ds1.fiftyPercent = ds2.fiftyPercent and ds1.unique3 =
ds2.unique3 and ds1.evenOnePercent = ds2.evenOnePercent and ds1.oddOnePercent = ds2.oddOnePercent and
ds1.stringu1 = ds2.stringu1 and ds1.stringu2 = ds2.stringu2 and ds1.string4 = ds2.string4 and
ds1.unique1 /*+  build-size 1024 , temp */= ds3.unique1 and ds1.unique2 = ds3.unique2 and ds1.two = ds3.two and ds1.four
= ds3.four and ds1.ten = ds3.ten and ds1.twenty = ds3.twenty and ds1.onePercent = ds3.onePercent and ds1.tenPercent =
ds3.tenPercent and ds1.twentyPercent = ds3.twentyPercent and ds1.fiftyPercent = ds3.fiftyPercent and ds1.unique3 =
ds3.unique3 and ds1.evenOnePercent = ds3.evenOnePercent and ds1.oddOnePercent = ds3.oddOnePercent and
ds1.stringu1 = ds3.stringu1 and ds1.stringu2 = ds3.stringu2 and ds1.string4 = ds3.string4 and
ds1.unique1 /*+  build-size 1024  */= ds4.unique1 and ds1.unique2 = ds4.unique2 and ds1.two = ds4.two and ds1.four =
ds4.four and ds1.ten = ds4.ten and ds1.twenty = ds4.twenty and ds1.onePercent = ds4.onePercent and ds1.tenPercent =
ds4.tenPercent and ds1.twentyPercent = ds4.twentyPercent and ds1.fiftyPercent = ds4.fiftyPercent and ds1.unique3 =
ds4.unique3 and ds1.evenOnePercent = ds4.evenOnePercent and ds1.oddOnePercent = ds4.oddOnePercent and
ds1.stringu1 = ds4.stringu1 and ds1.stringu2 = ds4.stringu2 and ds1.string4 = ds4.string4 and
ds1.unique1 /*+  build-size 1024  */= ds5.unique1 and ds1.unique2 = ds5.unique2 and ds1.two = ds5.two and ds1.four =
ds5.four and ds1.ten = ds5.ten and ds1.twenty = ds5.twenty and ds1.onePercent = ds5.onePercent and ds1.tenPercent =
ds5.tenPercent and ds1.twentyPercent = ds5.twentyPercent and ds1.fiftyPercent = ds5.fiftyPercent and ds1.unique3 =
ds5.unique3 and ds1.evenOnePercent = ds5.evenOnePercent and ds1.oddOnePercent = ds5.oddOnePercent and
ds1.stringu1 = ds5.stringu1 and ds1.stringu2 = ds5.stringu2 and ds1.string4 = ds5.string4
 LIMIT 1;
```

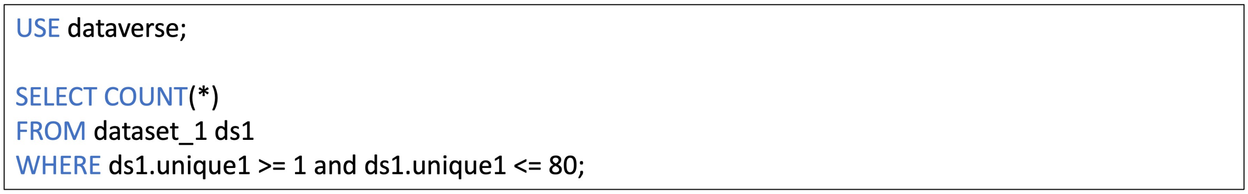
Figure B.8: Sequential Static Right Deep Tree - Query Example



```
SET `compiler.joinmemory` “1024MB”;
USE dataverse;

SELECT temp1.unique1 , temp1.unique2 , temp1.two , temp1.four , temp1.ten , temp1.twenty , temp1.onePercent ,
temp1.tenPercent , temp1.twentyPercent , temp1.fiftyPercent , temp1.unique3 , temp1.evenOnePercent ,
temp1.oddOnePercent , temp1.stringu1 , temp1.stringu2 , temp1.string4
 FROM
  dataset_4 ds4 ,
  dataset_5 ds5 ,
            (SELECT ds1.unique1 , ds1.unique2 , ds1.two , ds1.four , ds1.ten , ds1.twenty , ds1.onePercent , ds1.tenPercent ,
                        ds1.twentyPercent , ds1.fiftyPercent , ds1.unique3 , ds1.evenOnePercent , ds1.oddOnePercent ,
                        ds1.stringu1 , ds1.stringu2 , ds1.string4
            FROM
                        dataset_1 ds1 ,
                        dataset_2 ds2 ,
                        dataset_3 ds3
            WHERE ds1.unique1 /*+ build-size 1024 */= ds2.unique1 and ds1.unique2 = ds2.unique2 and ds1.two = ds2.two
                        and ds1.four = ds2.four and ds1.ten = ds2.ten and ds1.twenty = ds2.twenty and ds1.onePercent =
                        ds2.onePercent and ds1.tenPercent = ds2.tenPercent and ds1.twentyPercent = ds2.twentyPercent and
                        ds1.fiftyPercent = ds2.fiftyPercent and ds1.unique3 = ds2.unique3 and ds1.evenOnePercent =
                        ds2.evenOnePercent and ds1.oddOnePercent = ds2.oddOnePercent and ds1.stringu1 = ds2.stringu1 and
                        ds1.stringu2 = ds2.stringu2 and ds1.string4 = ds2.string4 and ds1.unique1 /*+ build-size 1024 */=
                        ds3.unique1 and ds2.unique2 = ds3.unique2 and ds1.two = ds3.two and ds1.four = ds3.four and ds1.ten
                        = ds3.ten and ds1.twenty = ds3.twenty and ds1.onePercent = ds3.onePercent and ds1.tenPercent =
                        ds3.tenPercent and ds1.twentyPercent = ds3.twentyPercent and ds1.fiftyPercent = ds3.fiftyPercent and
                        ds1.unique3 = ds3.unique3 and ds1.evenOnePercent = ds3.evenOnePercent and ds1.oddOnePercent =
                        ds3.oddOnePercent and ds1.stringu1 = ds3.stringu1 and ds1.stringu2 = ds3.stringu2 and ds1.string4 =
                        ds3.string4) temp1
 WHERE ds4.unique1 /*+ build-size 1024 */= ds5.unique1 and ds4.unique2 = ds5.unique2 and ds4.two = ds5.two and
ds4.four = ds5.four and ds4.ten = ds5.ten and ds4.twenty = ds5.twenty and ds4.onePercent = ds5.onePercent and
ds4.tenPercent = ds5.tenPercent and ds4.twentyPercent = ds5.twentyPercent and ds4.fiftyPercent = ds5.fiftyPercent and
ds4.unique3 = ds5.unique3 and ds4.evenOnePercent = ds5.evenOnePercent and ds4.oddOnePercent = ds5.oddOnePercent
and ds4.stringu1 = ds5.stringu1 and ds4.stringu2 = ds5.stringu2 and ds4.string4 = ds5.string4
 and temp1.unique1 /*+ build-size 1024 */= ds4.unique1 and temp1.unique2 = ds4.unique2 and temp1.two = ds4.two and
temp1.four = ds4.four and temp1.ten = ds4.ten and temp1.twenty = ds4.twenty and temp1.onePercent = ds4.onePercent
and temp1.tenPercent = ds4.tenPercent and temp1.twentyPercent = ds4.twentyPercent and temp1.fiftyPercent =
ds4.fiftyPercent and temp1.unique3 = ds4.unique3 and temp1.evenOnePercent = ds4.evenOnePercent and
temp1.oddOnePercent = ds4.oddOnePercent and temp1.stringu1 = ds4.stringu1 and temp1.stringu2 = ds4.stringu2 and
temp1.string4 = ds4.string4
LIMIT 1;
```

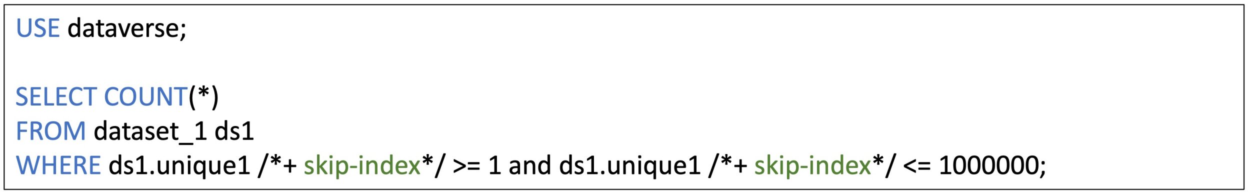
Figure B.9: Bushy - Query Example



```
USE dataverse;

SELECT COUNT(*)
FROM dataset_1 ds1
WHERE ds1.unique1 >= 1 and ds1.unique1 <= 80;
```

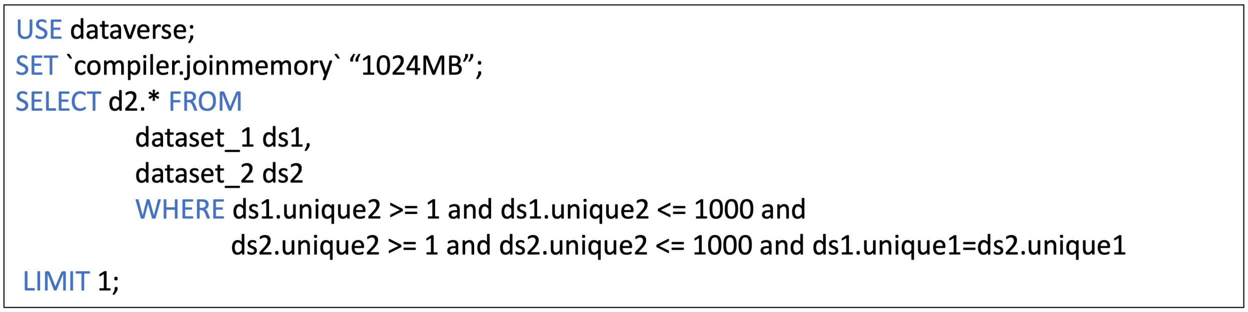
Figure B.10: ZeroMemory-Short - Query Example



```
USE dataverse;

SELECT COUNT(*)
FROM dataset_1 ds1
WHERE ds1.unique1 /*+ skip-index*/ >= 1 and ds1.unique1 /*+ skip-index*/ <= 1000000;
```

Figure B.11: ZeroMemory-Long - Query Example



```
USE dataverse;
SET `compiler.joinmemory` “1024MB”;
SELECT d2.* FROM
            dataset_1 ds1,
            dataset_2 ds2
            WHERE ds1.unique2 >= 1 and ds1.unique2 <= 1000 and
                      ds2.unique2 >= 1 and ds2.unique2 <= 1000 and ds1.unique1=ds2.unique1
 LIMIT 1;
```

Figure B.12: Small, Medium, and Large Query Class - Query Example

# Appendix C

# Fair Scheduling for Concurrent Queries - Timeline Plots

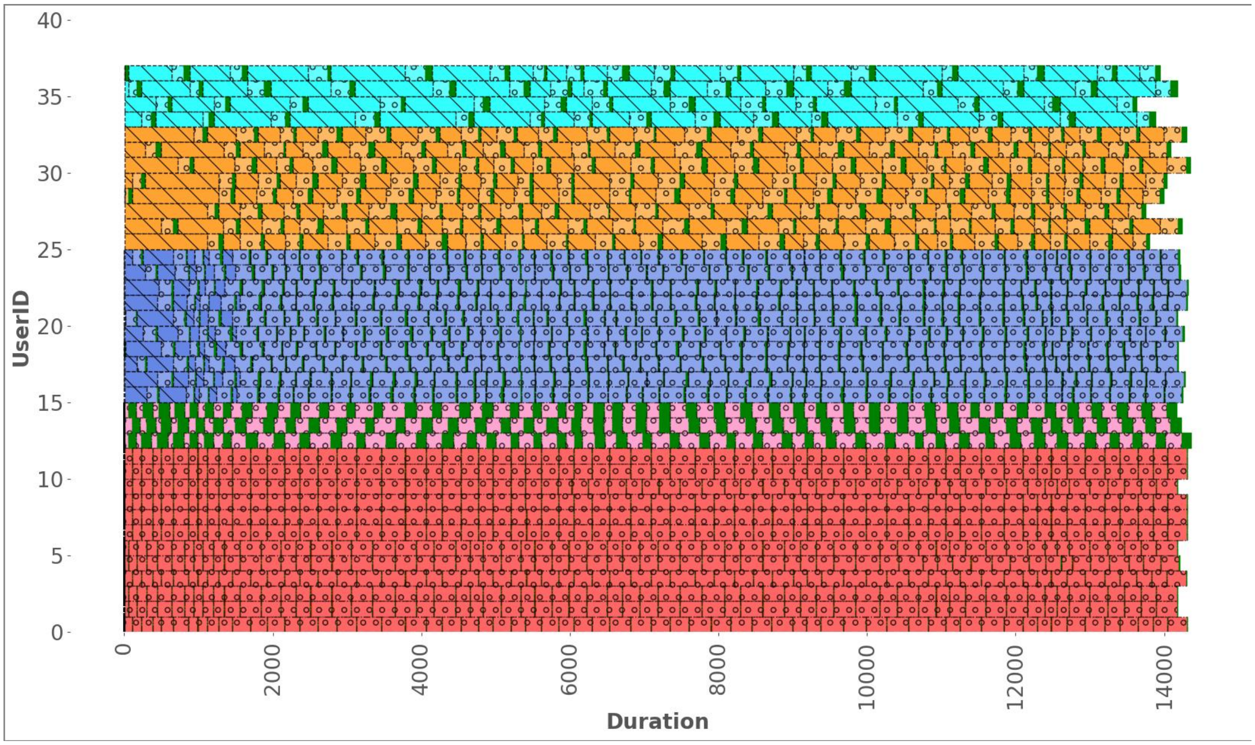


Figure C.1: Wisconsin-V2 Scheduler - Timeline Plot for core-multiplier = 3

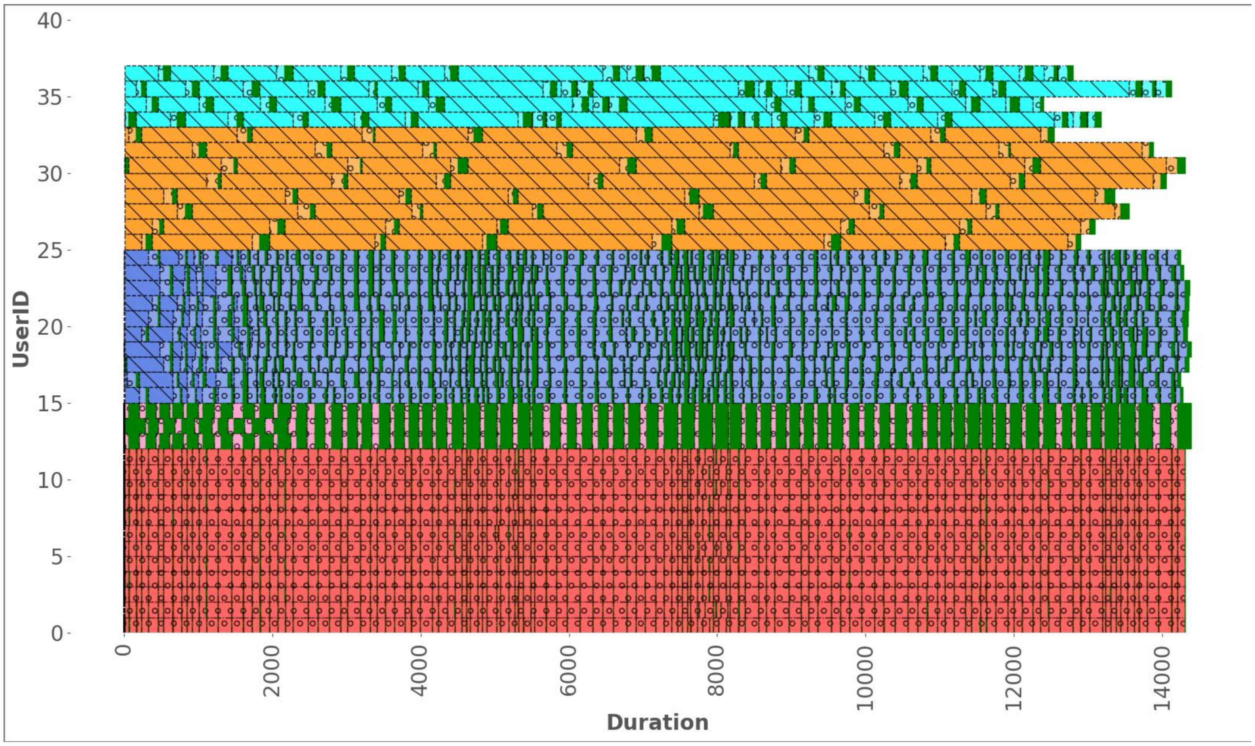


Figure C.2: Wisconsin-V2 Scheduler - Timeline Plot for core-multiplier = 12

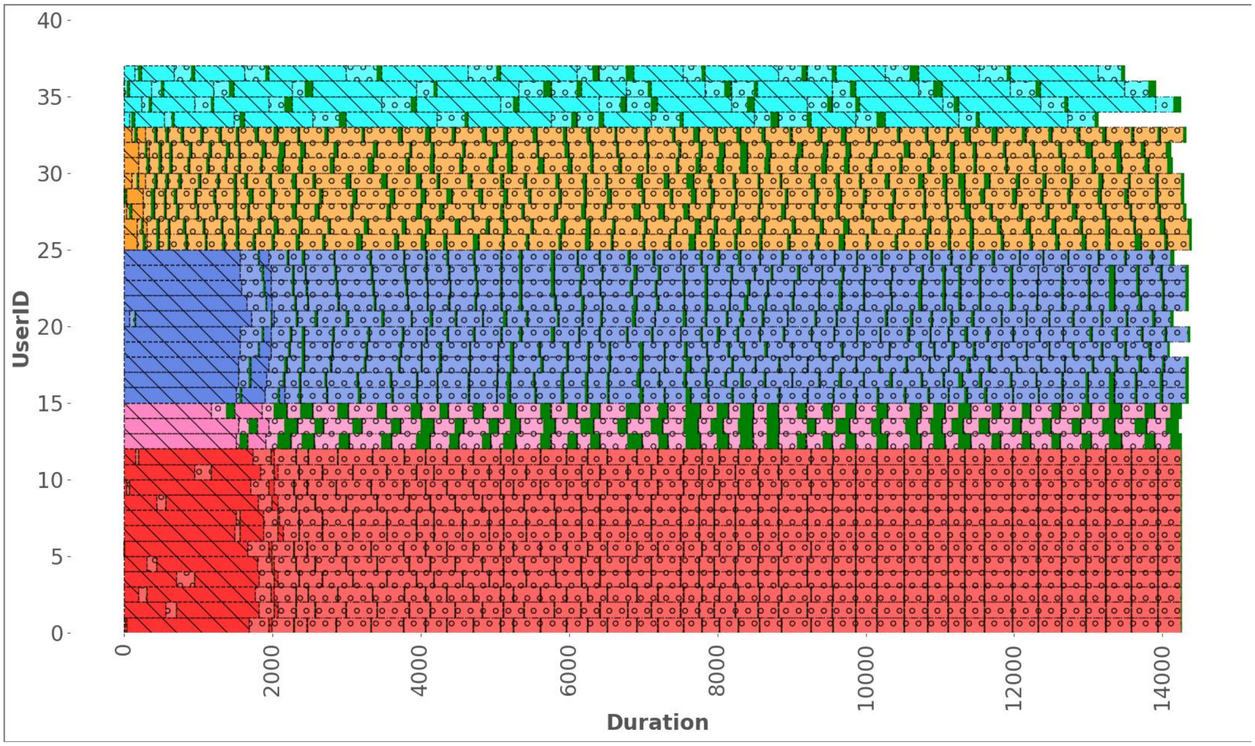


Figure C.3: Wisconsin-V2 Scheduler - Timeline Plot for core-multiplier = 48

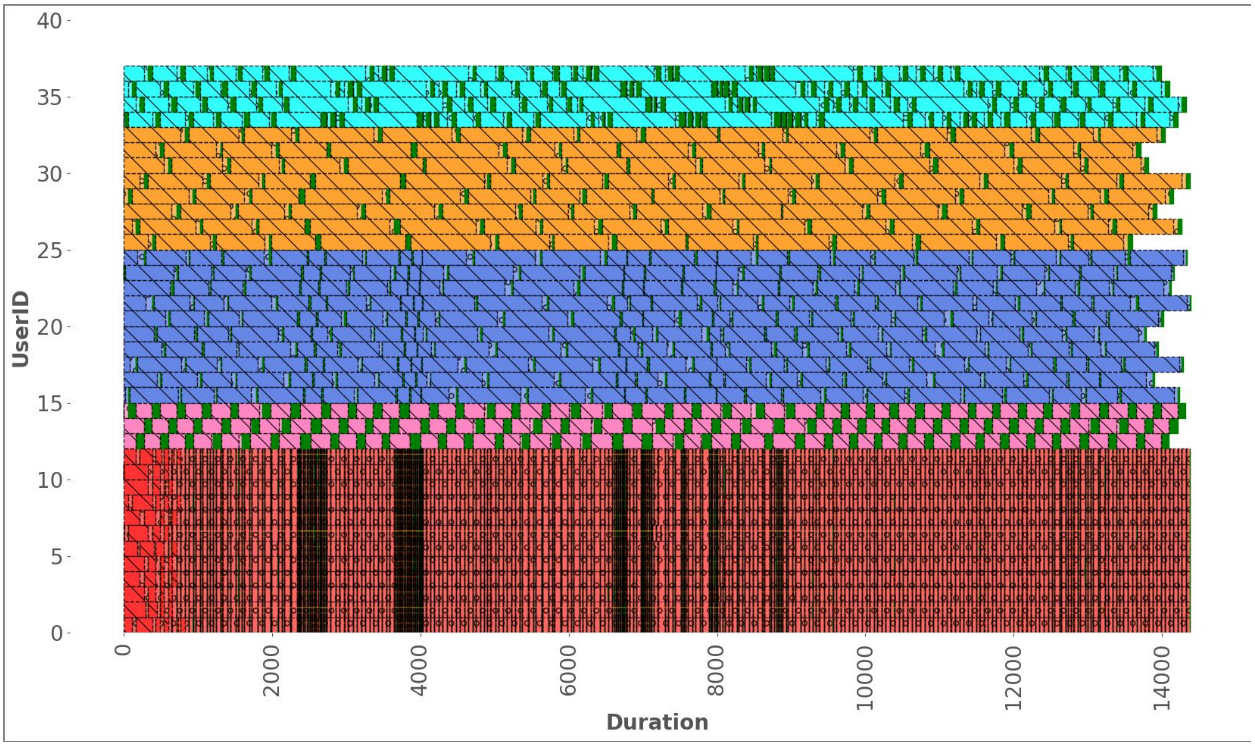


Figure C.4: Wisconsin-V3 Scheduler - Timeline Plot for core-multiplier = 3

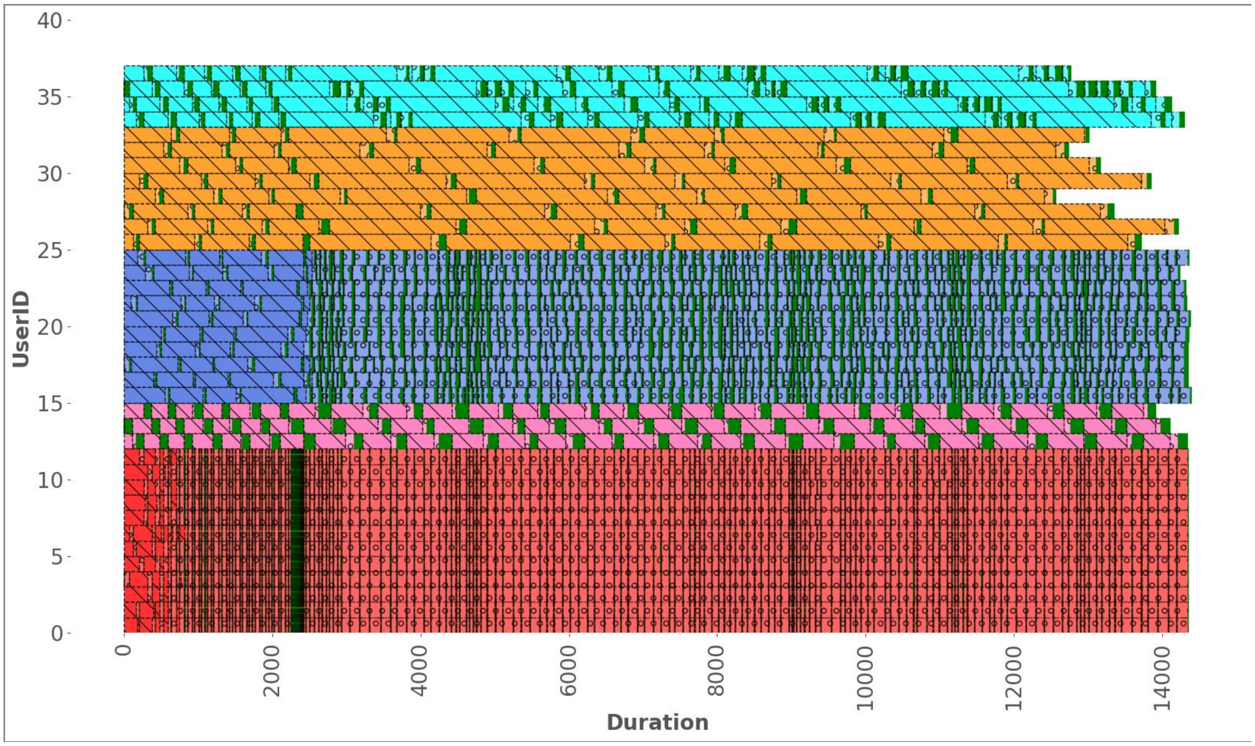


Figure C.5: Wisconsin-V3 Scheduler - Timeline Plot for core-multiplier = 12

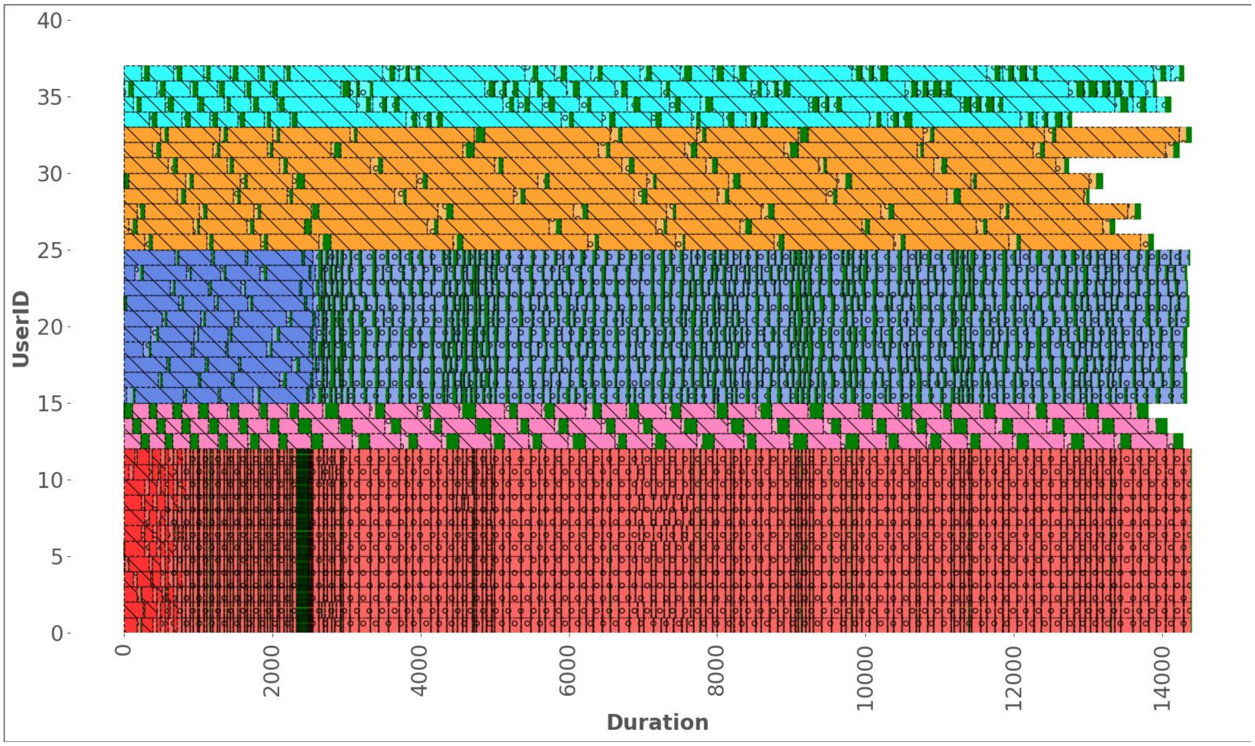


Figure C.6: Wisconsin-V3 Scheduler - Timeline Plot for core-multiplier = 48

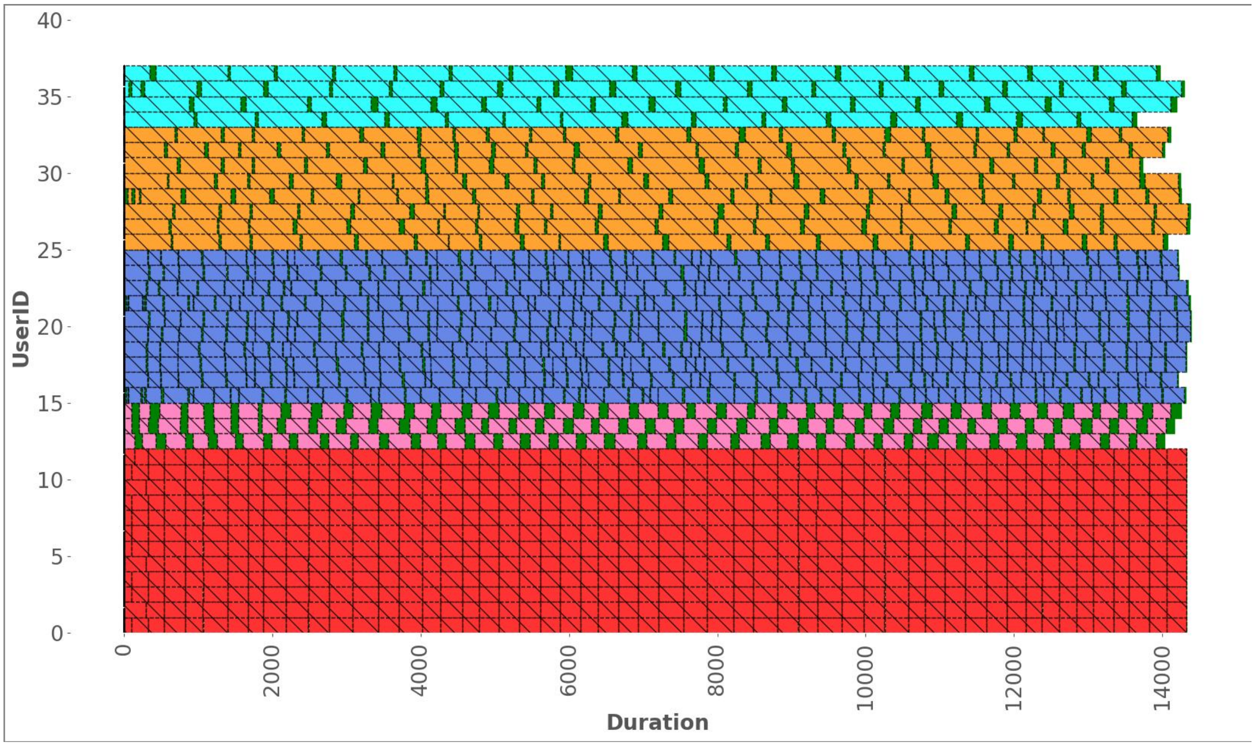


Figure C.7: Colorado-V1 Scheduler - Timeline Plot for core-multiplier = 3

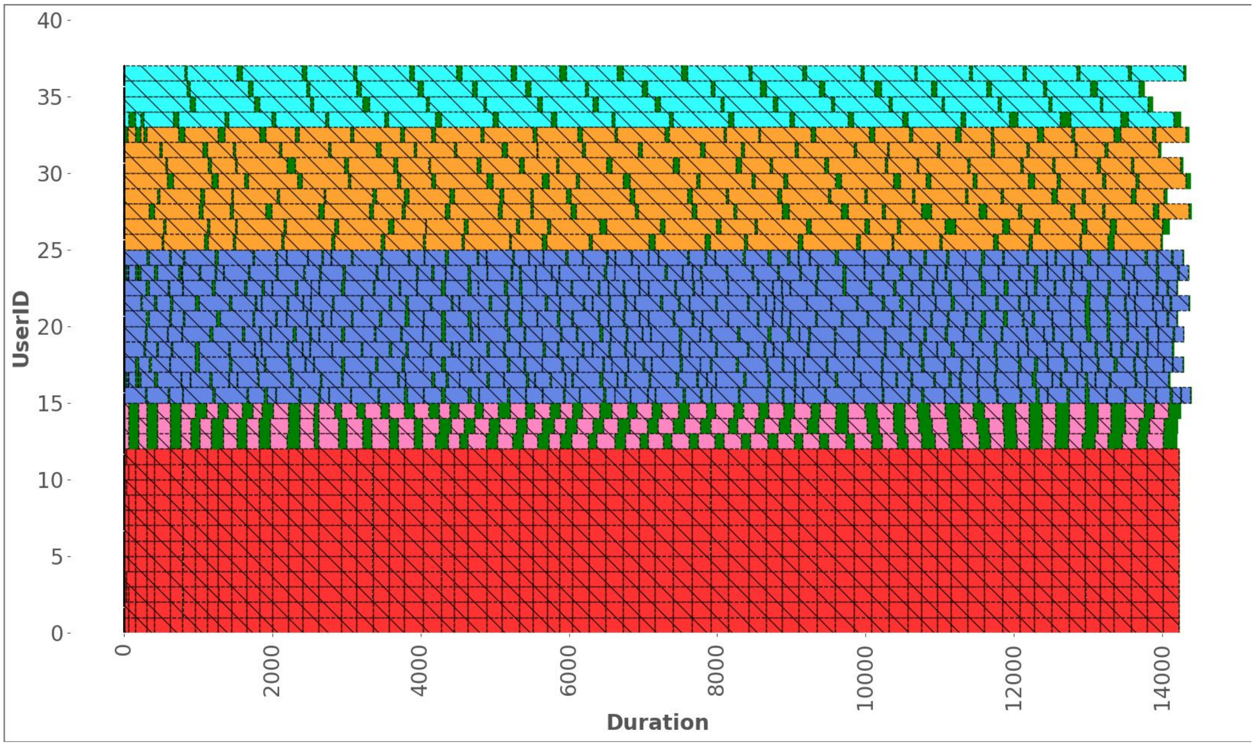


Figure C.8: Colorado-V1 Scheduler - Timeline Plot for core-multiplier = 12

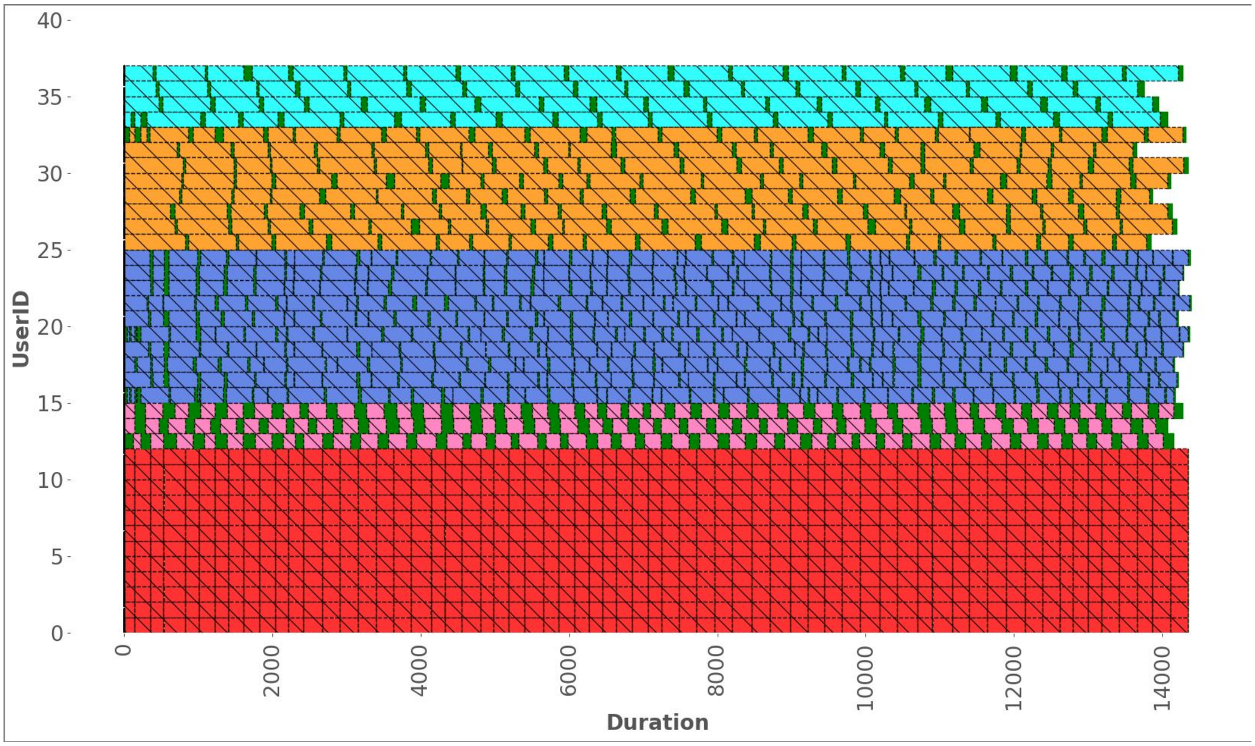


Figure C.9: Colorado-V1 Scheduler - Timeline Plot for core-multiplier = 48

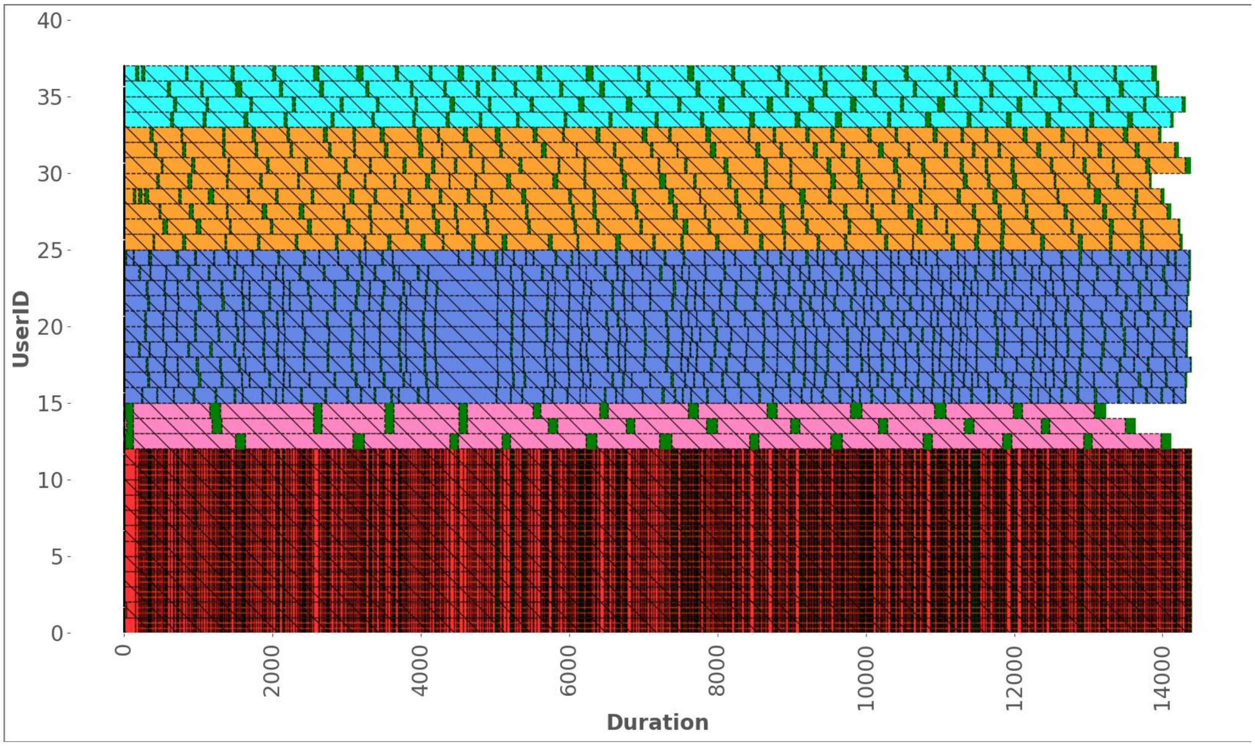


Figure C.10: Colorado-V2 Scheduler - Timeline Plot for core-multiplier = 3

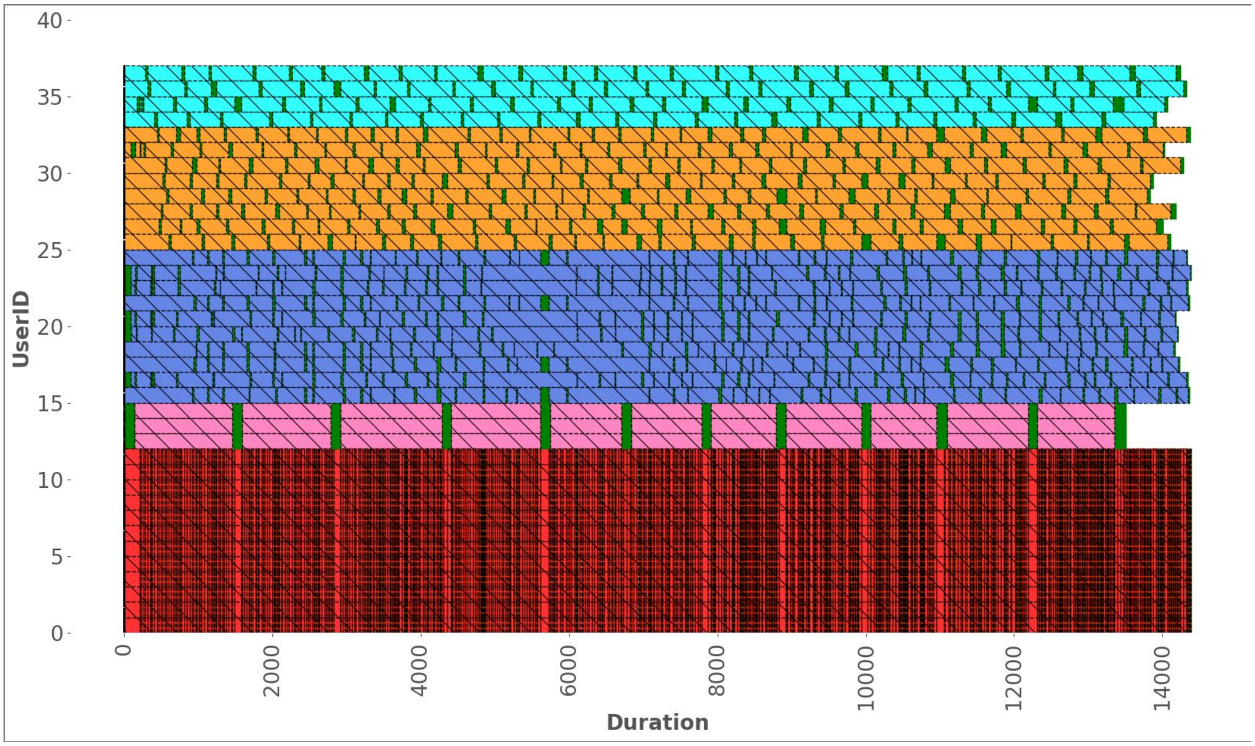


Figure C.11: Colorado-V2 Scheduler - Timeline Plot for core-multiplier = 12

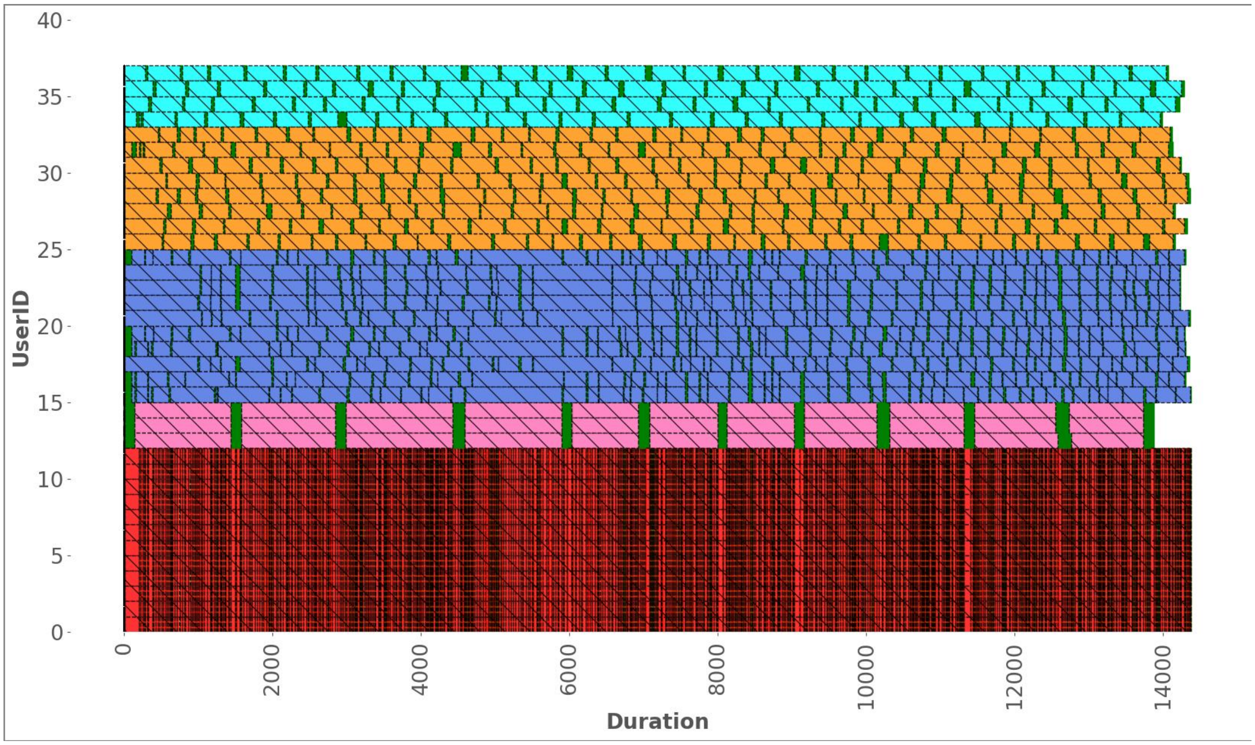


Figure C.12: Colorado-V2 Scheduler - Timeline Plot for core-multiplier = 48