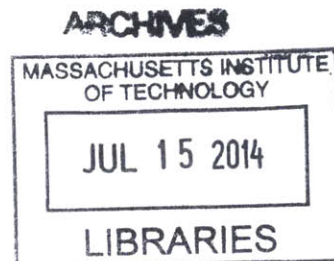


Optimizations to a Massively Parallel Database and Support of a Shared Scan Architecture

by

Saher B. Ahwal

S.B., EECS M.I.T., 2013





Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 2014
[JUNE 2014]
Copyright 2014 Massachusetts Institute of Technology. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author  **Signature redacted** _____
Department of Electrical Engineering and Computer Science

Signature redacted _____ May 23, 2014
Certified by _____
Samuel Madden

 Professor of Electrical Engineering and Computer Science
Signature redacted _____ Thesis Supervisor
Accepted by _____
 Prof. Albert R. Meyer, Chairman
Masters of Engineering Thesis Committee

Optimizations to a Massively Parallel Database and Support of a Shared Scan Architecture

by

Saher B. Ahwal

Submitted to the Department of Electrical Engineering and Computer Science on
May 22, 2014, in partial fulfillment of the requirements for the degree of Masters of
Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents a new architecture and optimizations to MapD, a database server which uses a hybrid of multi-CPU/multi-GPU architecture for query execution and analysis. We tackle the challenge of partitioning the data across multiple nodes with many CPUs and GPUs by means of an indexing framework. We implement a QuadTree spatial partitioning scheme and demonstrate how it improves the latencies of many queries when using the index as opposed to not using any.

Moreover, we tackle the challenge of processing many queries (perhaps issued concurrently) where queries have very fast latency constraints, e.g, for visualization. We implement a software architecture which allows for scheduling concurrent client query requests to share processing of many queries in a single pass through the data (“shared scans”). Our experiments exhibit orders of magnitude improvement in query throughput for both, skewed and non-skewed workloads, for shared scans as opposed to serial execution.

Thesis Supervisor: Samuel R. Madden

Title: Professor

Acknowledgements

I would like to thank my family and friends for their continuous support during my undergraduate years and MEng year at MIT. I thank my family for their encouragement during the busy and stressful times. I would like to thank all my friends for offering support, help, and motivation in the past 5 years as well.

I would like to give special thanks to my thesis supervisor, Professor Sam Madden for providing me with the opportunity to work on a very interesting project during the MEng year. His continuous support and guidance throughout the MEng year have been invaluable. I truly appreciated his readiness to take the time from his busy schedule and meet with me to answer my questions. He was always quick in responding to my questions through email and I really appreciated that. He really made sure I am able to make progress so I can finish my project in time. The technical discussions I had with Professor Madden were always helpful and informative.

I would also like to thank the creator of MapD, Todd Mostak for his help and guidance throughout the MapD2 project. Without the many discussions I had with Todd, this project would've been impossible. I appreciated the hours he spent in helping me and discussing issues on the phone, even on weekends. I would like to thank Todd, for the opportunity to work on MapD and be able to extend the framework with an interesting architecture.

Finally, I would like to thank MIT for everything it offered me through my academic journey. Thanks to my academic advisor, Professor George Verghese for the great guidance and support he offered. Thanks to Anne Hunter, all my professors and TAs for being of great assistance at challenging and stressful times. While writing this, I realize that these are the final moments for me being a student here; this place will be dearly missed.

Contents

CHAPTER 1 - INTRODUCTION	12
1.1 OVERVIEW	13
1.2 MASSIVELY PARALLEL DATABASE (MAPD)	15
1.2.1 <i>Database Table Structure</i>	17
1.2.2 <i>MapD Architecture Overview</i>	18
1.3 MAPD INDEX AND PARTITIONING OPPORTUNITY	19
1.4 MEMORY POOL MANAGEMENT AND REPLACEMENT POLICY	21
1.5 THE DEADLINE SCHEDULING PROBLEM	23
1.6 THESIS ROAD MAP	24
CHAPTER 2 – DATABASE STORAGE AND INDEXING	25
2.1 DATABASE TABLE STRUCTURE	25
2.2 PHYSICAL PARTITIONING SCHEME DESIGN	28
2.2.1 <i>Partition Scheme API</i>	28
2.2.2 <i>Linear Scheme</i>	29
2.2.3 <i>QuadTree Scheme</i>	30
2.3 CATALOG MANAGER	35
2.4 SUMMARY	39
CHAPTER 3 – DATABASE MEMORY MANAGEMENT	40
3.1 BUFFER POOL MANAGER OVERVIEW	40
3.1.1 <i>Three-level Memory Hierarchy</i>	40
3.1.1.1 <i>Architecture</i>	41
3.1.1.2 <i>Data Structure Design and Maintenance</i>	42
3.1.1.3 <i>API summary</i>	46
3.1.1.4 <i>Eviction Policy Design</i>	47
3.1.1.4.1 <i>Overview</i>	47
3.1.1.4.2 <i>Eviction Cases</i>	48
3.2 SUMMARY	49
CHAPTER 4 - SHARED SCAN DESIGN	50
4.1 OVERVIEW	50

4.2 ARCHITECTURE OF SHARED SCAN PIPELINE -----	52
4.2.1 <i>Task Wrapper</i> -----	54
4.2.2 <i>Query Information Estimator</i> -----	55
4.2.3 <i>Query Scheduler</i> -----	55
4.2.3.1 EDF SHARED SCHEDULER -----	56
4.3 SUMMARY -----	57
CHAPTER 5 – EXPERIMENTS AND RESULTS -----	58
5.1 DATASETS -----	58
5.2 BENCHMARK QUERY -----	59
5.3 PARTITIONING SCHEME EXPERIMENTS-----	60
5.3.1 <i>QuadTree Index vs Linear Scheme</i> -----	60
5.4 SHARED SCAN EVALUATION AND ANALYSIS-----	65
5.4.1 <i>Overlap</i> -----	66
5.4.2 <i>Deadlines</i> -----	68
5.4.3 <i>Number of Queries</i> -----	70
5.4.4 <i>Output Buffer Size</i> -----	73
5.4.5 <i>Scheduling (EDF vs. FIFO)</i> -----	75
CHAPTER 6 - RELATED WORK-----	85
CHAPTER 7 - FUTURE WORK-----	88
7.1 PARTITION SCHEME ARCHITECTURE -----	88
7.2 BUFFER POOL MANAGER -----	89
7.3 SHARED SCANS AND EDF -----	89
CHAPTER 8 - CONCLUSION -----	91
BIBLIOGRAPHY-----	92

Table of Figures

FIGURE 1.1-1 - GPU VS CPU PERFORMANCE OVER TIME	14
FIGURE 1.2-1 - MAPD WEB USER INTERFACE.....	16
FIGURE 1.2-2 - DATABASE TABLE STRUCTURE EXAMPLE - MAPD - TWEETS TABLE.....	17
FIGURE 1.2-3 - BASIC SHARED-NOTHING DESIGN	18
FIGURE 1.3-1 – PARTITION SCHEME: ABSTRACT CLASS. THREE EXAMPLE IMPLEMENTATIONS OF PARTITION SCHEME.....	20
FIGURE 1.3-2 - SAMPLE QUADTREE PARTITION SCHEME.....	21
FIGURE 2.1-1 - MAPD2 DATABASE TABLE STRUCTURE.....	26
FIGURE 2.1-2 - DATABASE TABLE PSEUDO CODE.....	27
FIGURE 2.2-1 - POINTS FILE AND FRAGMENT-SIZES FILE - FOR RECOVERY	33
FIGURE 2.2-2 - QUADTREE SCHEME EXAMPLE - FRAGMENT ID ASSIGNMENTS - (NW, NE, SW, SE)	34
FIGURE 2.3-1 - CATALOG METADATA TABLES FOR MULTI-PARTITIONING SCHEMES SUPPORT	36
FIGURE 2.3-2 - CATALOG METADATA – SUPPORT OF MULTIPLE PARTITION SCHEMES.....	38
FIGURE 3.1-1 MAPD2 BUFFER POOL MEMORY HIERARCHY	41
FIGURE 3.1-2 MAPD2 ARCHITECTURE – 4 NODES EXAMPLE.....	42
FIGURE 3.1-3 - BUFFER POOL PARTITIONING EXAMPLE. B = 3, D = 5.....	43
FIGURE 3.1-4 - CHUNKINFO (MEMORY SEGMENT) AND CHUNKKEY DEFINITIONS	44
FIGURE 3.1-5 - EVICTION CASE 3 - BUFFER POOL COALESCING	49
FIGURE 4.1-1 EXAMPLE TASK WITH PREEMPTIVE POINTS	52
FIGURE 4.2-1 - HIGH LEVEL PIPELINE FOR SHARED SCAN	54
FIGURE 5.2-1 AN EXAMPLE HISTOGRAM RESULT OF 9 * 10 PIXELS.....	60
FIGURE 5.3-1 QUADTREE VS. LINEARSCHEME ON GPU – SMALL DATA	62

FIGURE 5.3-2 QUADTREE VS. LINEARSCHEME ON CPU – SMALL DATA.....	62
FIGURE 5.3-3 QUADTREE VS. LINEARSCHEME ON GPU – LARGE DATASET.....	64
FIGURE 5.3-4 QUADTREE VS. LINEARSCHEME ON CPU - LARGE DATASET.....	64
FIGURE 5.4-1 QMD VS. OVERLAP.....	67
FIGURE 5.4-2 AVG. LATENESS VS. OVERLAP.....	68
FIGURE 5.4-3 QMD VS. DEADLINE.....	69
FIGURE 5.4-4 AVG. LATENESS VS. DEADLINE.....	69
FIGURE 5.4-5 THROUGHPUT VS. NUMBER OF QUERIES.....	71
FIGURE 5.4-6 TIME VS. NUMBER OF QUERIES.....	71
FIGURE 5.4-7 QMD VS. NUMBER OF QUERIES.....	72
FIGURE 5.4-8 AVG. LATENESS VS. NUMBER OF QUERIES.....	72
FIGURE 5.4-9 THROUGHPUT VS. OUTPUT BUFFER SIZE.....	74
FIGURE 5.4-10 QMD VS. OUTPUT BUFFER SIZE.....	74
FIGURE 5.4-11 AVERAGE LATENESS VS. OUTPUT BUFFER SIZE.....	75
FIGURE 5.4-12 EDF-FIFO. THROUGHPUT VS. N –SMALL BUFFER.....	76
FIGURE 5.4-13 EDF-FIFO. AVG. LATENESS VS. N - SMALL BUFFER.....	77
FIGURE 5.4-14 EDF-FIFO. QMD VS. N - SMALL BUFFER.....	77
FIGURE 5.4-15 EDF-FIFO. THROUGHPUT VS. N - LARGE BUFFER.....	79
FIGURE 5.4-16 EDF-FIFO. QMD VS. N - LARGE BUFFER.....	79
FIGURE 5.4-17 EDF-FIFO. AVG. LATENESS VS. N - LARGE BUFFER.....	80
FIGURE 5.4-18 SKEWED DEADLINES. AVG. LATENESS VS. DEADLINE.....	82
FIGURE 5.4-19 SKEWED DEADLINES QMD VS. DEADLINE.....	82
FIGURE 5.4-20 SKEWED DEADLINES. AVG. LATENESS VS. DEADLINE.....	83
FIGURE 5.4-21 SKEWED DEADLINES. QMD VS. DEADLINE.....	84

Table of Tables

TABLE 2.2-1 - FRAGMENT GROUPING PHASE - MAP	31
TABLE 5.3-1 QUADTREE EXPERIMENT PARAMETERS – SMALL DATASET.....	61
TABLE 5.3-2 QUADTREE EXPERIMENT PARAMETERS – LARGE DATASET.....	63
TABLE 5.4-1 OVERLAP EXPERIMENT PARAMETERS	66
TABLE 5.4-2 SHARED SCAN VS. SERIAL EXECUTION PERFORMANCE.....	67
TABLE 5.4-3 DEADLINE EXPERIMENTS PARAMETERS	68
TABLE 5.4-4 PARAMETERS FOR NUMBER OF QUERIES EXPERIMENTS	70
TABLE 5.4-5 PARAMETERS FOR OUTPUT BUFFER SIZE EXPERIMENT	73
TABLE 5.4-6 PARAMETERS FOR EDF VS. FIFO EXPERIMENT - SMALL BUFFER	76
TABLE 5.4-7 PARAMETERS FOR EDF VS. FIFO EXPERIMENT - LARGE BUFFER	78
TABLE 5.4-8 PARAMETERS FOR SKEWED DEADLINES EXPERIMENT.....	81

Chapter 1 - Introduction

The growth in social media usage and the increase in web content have contributed to a recent explosive growth in the volume of data that we need to process. As a result, there is greater demand today for dynamic and interactive visualizations of such datasets to look for trends and understand social behavior. This increases the need for low latency analytics.

GPUs (Graphical Processing Units) provide memory bandwidth that is orders of magnitude greater than that of CPUs. Moreover, they deliver hundreds to thousands of cores that allow for higher degrees of parallelization. The use of general purpose GPUs for query and analysis of large data sets is, therefore, promising.

In the implementation of low latency analytics hybrid CPU/GPU system, two key challenges arise. The first is the managing and the partitioning of data sets across multiple nodes with multiple CPUs and GPUs. The second is the processing of many queries (perhaps issued concurrently) where queries have very fast latency constraints, e.g, for visualization. This thesis project presents optimizations to tackle these key challenges. First, it describes the implementation of data partitioning strategy for a hybrid GPU/CPU system called MapD. Second, it demonstrates the implementation of a memory buffer pool to manage the data when it exceeds the size of main memory on GPU and CPU. Finally, it describes the design and implementation of a software architecture which allows for scheduling concurrent client query requests to share processing of many queries in a single pass through the data (“shared scans”).

In this chapter, I discuss the motivation for hybrid multi-CPU/multi-GPU database frameworks. Later, I introduce an existent multi-CPU/GPU database framework, MapD. Then, I present optimization opportunities for MapD and the need of an architecture accustomed for concurrent query requests to the database server.

1.1 Overview

The design and architecture of optimized database management systems remains a challenging task with the growing user demand for real-time big-data analytics and the increasing size of datasets. Cheaper hardware provides computer systems engineers with opportunities to parallelize query processing across many devices to improve performance of query execution. Yet, current technology platforms like MapReduce framework and conventional distributed DBMSs are not optimized for low latency analytics to support real-time visualizations.

New hardware like the GPUs and Intel many-core devices (e.g Xeon Phi) are delivering hundreds to thousands of cores. In particular, the GPUs, Graphical Processing Units, optimized for fast processing for video games and HD videos, have tremendous computational potential in non-graphical applications. In fact, many have been using the GPUs for general purpose computation to leverage their capability of fast parallel computation. Furthermore, GPU parallelism is doubling every year allowing many to exploit the use of these devices, now available within commodity hardware, for optimizing general purpose applications by means of running thousands of cores concurrently(See Figure 1.1-1). Not only do they have many cores, GPUs have memory bandwidth which is orders of magnitude greater than that of modern CPUs. This makes such devices well suited for low-latency analytics.

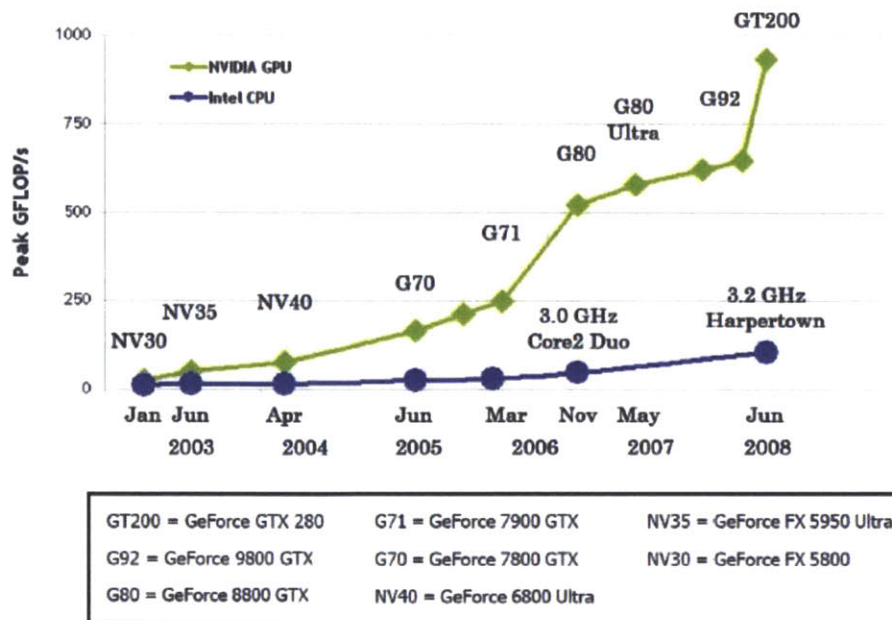


Figure 1.1-1 - GPU vs CPU performance over time

Source: [1]

In addition, the memory (RAM) size for such devices is rapidly increasing with advancing semiconductor technology. Soon, we will be able to place large datasets in a cluster of these devices' memory. This can simplify the design of the database system, and in particular, its memory management. Furthermore, this allows for new optimizations techniques that were not easily achieved with limited RAM.

Unfortunately, data management software platforms that use GPUs have been slow to develop due to some limitations. First, not every algorithm is easily parallelizable on the GPUs. In many cases, algorithms have to be re-written in a completely different way than when written for sequential CPU application or even parallel CPU application. Second, writing parallel code on GPU requires the knowledge of unconventional languages like CUDA or OpenCL. Third, the GPU memory is limited compared to the CPU. For instance, state of art GPUs might have 4GB or 6GB of RAM, while modern CPU servers routinely have 256 GB or more of DDR3 RAM. Fourth, the communication bus between the GPU and CPU, called the PCI bus, is a bottleneck in many applications since it transmits data at orders of magnitude less than the GPU scan rate. For

example, Nvidia's GeForce GTX 780 has memory bandwidth of about 288 GB/sec when the PCI transfer rate is only about 12 GB / sec.

Since the emergence of the general purpose GPUs, some research has been done with regards to optimizing database operators on GPUs [2] [3]. Other work describes database indexes and parallelized partitions on GPUs [1]. In addition, some previous work focuses on optimizing and parallelizing machine learning libraries on the GPUs as well [4]. Yet, there is almost no work describing a full hybrid multi-CPU/multi-GPU architecture of a general database system. Due to the increasing availability of parallelizable hardware devices like MICs and GPUs, there is a greater need for the design of general database architecture which leverages multi-GPUs/MICs along with multi-nodes.

MapD was designed and implemented as a reaction to this need. MapD is a massively parallel database which exploits multi-node and multi-GPU architectures for querying, analysis, and visualizations of large datasets. MapD tackles the problems and limitations described earlier in various innovative ways, namely by means of an efficient architecture to allow for massive parallelization with ultra-low latency in order of milliseconds.

Database server systems like MapD may receive many concurrent queries. At the moment, these queries are served one at a time. However, a significant fraction of these queries involve the scan of the same data chunks whether on the CPU or on the GPU. Therefore, in this thesis, I present a new architecture for MapD version 2 (MapD2), optimized for shared scans, allowing concurrent queries to be executed concurrently.

1.2 Massively Parallel Database (MapD)

MapD [5], designed and implemented by Todd Mostak, is database server which uses a hybrid of multi-CPU/multi-GPU architecture for query execution and analysis. MapD's front-end comprises a map that allows users to visualize large geospatial datasets, on the order of hundreds of millions of records, in milliseconds.(See Figure 1.2-1).

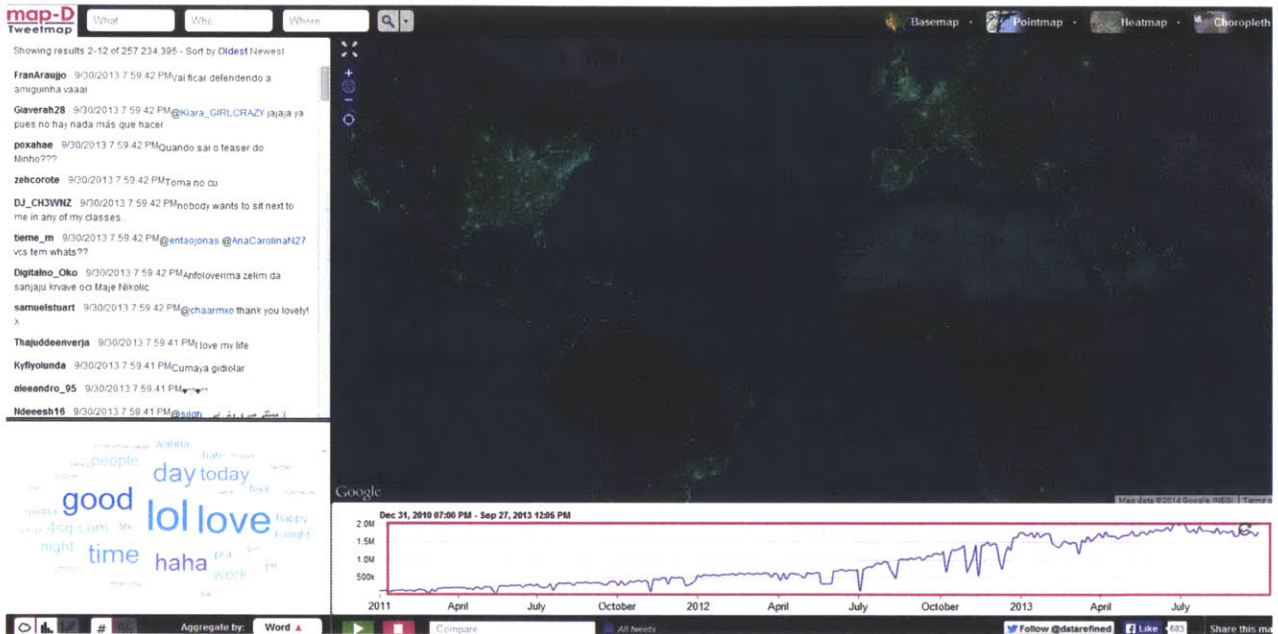


Figure 1.2-1 - MapD Web User Interface

Source: www.map-d.com, or <http://mapd.csail.mit.edu/tweetmap-desktop/>, on 4/7/2014

MapD’s architecture allows for massive parallelization of querying, analysis, and visualization of big datasets. It exploits multi-CPU and multi-GPU architectures systematically. The GPU-generated visualizations, the compressed bitmaps communicated from GPU to CPU, and the parallel architecture allow for real-time exploration of very large datasets where the processing time is in the order of milliseconds, as opposed to seconds, minutes, or hours using other modern database models including distributed DBMS or distributed frameworks like MapReduce or Giraph. [5]

MapD was designed as a Main Memory Database System (MMDB). With cheaper semiconductor memory and higher chip densities it is becoming more feasible to store large datasets in memory. MapD utilizes the GPU memory to keep most frequently accessed data in GPU memory while less frequently accessed data in RAM. The GPU memory bandwidth is greater than that of the CPU.

MapD is column-store database. This implies that the column data is stored contiguously in memory rather than the traditional DBMS approach of row-store, where complete tuples are

stored contiguously. The main advantage of column stores is that they make it possible to read/store, in RAM or GPU memory, blocks of data that contain just the fields (columns) of interest, rather than having to read blocks consisting of many unrelated attributes.

1.2.1 Database Table Structure

Each table is divided into *chunks* where each chunk represents a column of a table. The *chunk* structure simply defines a pointer in memory to contiguous data of some type on GPU or CPU main memory. If a table is partitioned across multiple nodes, say N, then column of that table will have N chunks, each residing on a separate node.

Chunk_0	Chunk_1	Chunk_2	Chunk_3	Chunk_4
tweet_id	user_name	longitude	latitude	tweet_text
1	@saher	-71.104	42.365	map-d2 has a new buffer pool!
2	@sam	-9.133	38.716	Postgres was written in Lisp!
3	@tmostak	18.064	59.3325	checkout map-d vizualizations
222001890	@abed	17.045	59.332	Just took my flu shot...
222001891	@sameeh	13.088	58.332	The exam wasn't too bad #great

Figure 1.2-2 - Database Table Structure Example - MapD - Tweets Table

Figure 1.2-2 shows a simple example of one table, representing geocoded tweets, with 5 columns. Each column is stored in a contiguous chunk of memory. For simplicity, the last column (tweet_text) is shown to store actual string although MapD uses dictionary-encoding of words as means of compression of text columns and in order to optimize equality joins on such data.

1.2.2 MapD Architecture Overview

MapD implements a shared-nothing database design and architecture. This implies that each processor has its own private memory as opposed to a shared global memory. Each process has its private one or more disks as opposed to having processors having direct access to all disks. This implies that the data needs to be partitioned across the existent nodes. MapD partitions data by means of two methods. The first method is a simple round-robin partitioning of the chunks described. The second method involves explicit clustering of data by values of one or more fields (columns) of a table.

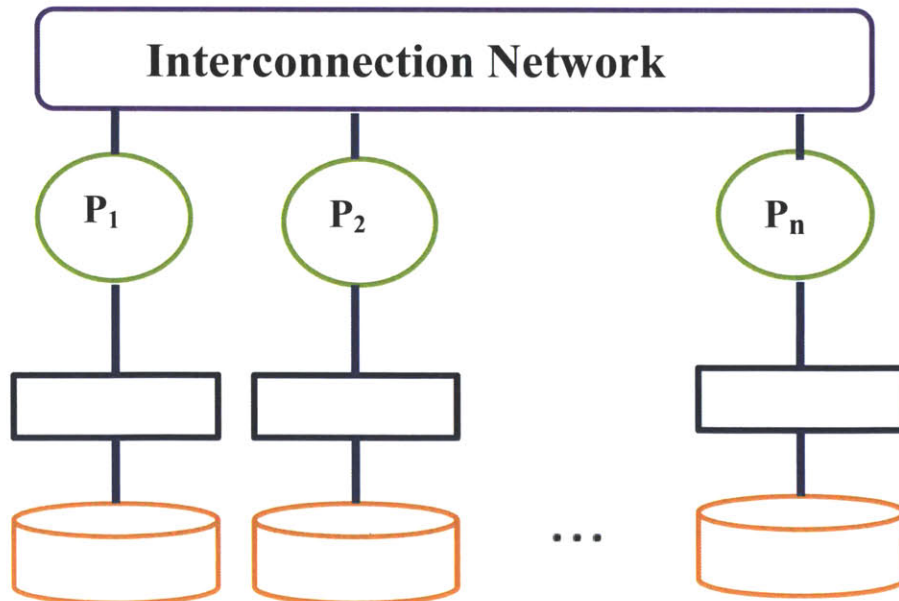


Figure 1.2-3 - Basic Shared-Nothing Design

The main innovative features of MapD are listed as follows:

1. Partitioning data physically over multiple GPUs allow querying over datasets larger than GPU memory.
2. MapD returns compressed bitmaps or indexes of results from GPU to CPU rather than returning the rows themselves. This helps overcome the high bandwidth costs of transferring data over the slow PCI bus.
3. An algorithm to conduct fast spatial joins.
4. Using dictionary encoding for text data (storing text as product of prime numbers).

1.3 MapD Index and Partitioning Opportunity

Indexes are data structures that allow faster retrieval of data from database tables. Famous database indexes are B+-trees, hash indexes, and spatial indexes. In general, indexes achieve speedup by directing the query to the relevant table fragments for scan based on the search key(s).

In the [previous section](#), we have seen that MapD doesn't implement an index of any sort, but partitions the data across multiple GPUs and scans all table fragments. As discussed earlier, MapD's current major use case is for geospatial data tables. With the growing size of datasets, it becomes very costly to scan all data fragments when a user is only concerned with data within a certain geographical bounds. For instance, a user might want to analyze geocoded tweets coming from the state of Massachusetts. It will be very wasteful to scan fragments containing all the tweets in the world to obtain the result of his query. This is where a spatial index can be utilized to optimize such use cases.

Of course, MapD's use cases also include non-geospatial datasets and queries. For example, torrent downloads data (torrent title, number of seeders/peers, download time) may need an index on the torrent category and title for efficient title and category searches, and

maybe another index on the download time field for queries that animate number of seeders and peers for torrents over certain time range.

Therefore, we saw that MapD needs a general purpose partitioning architecture. We call this the “**Partition-Scheme**” data structure. The architecture should allow adding indexes as necessary in the future while maintaining the main interface of inserting database tuples and querying fragments. The key idea is that the database table structure should maintain a pointer to a Partition Scheme data structure. Partition Scheme should be an abstract class where a particular implementation of that scheme implements the details of partitioning fragments and bookkeeping needed for the index.

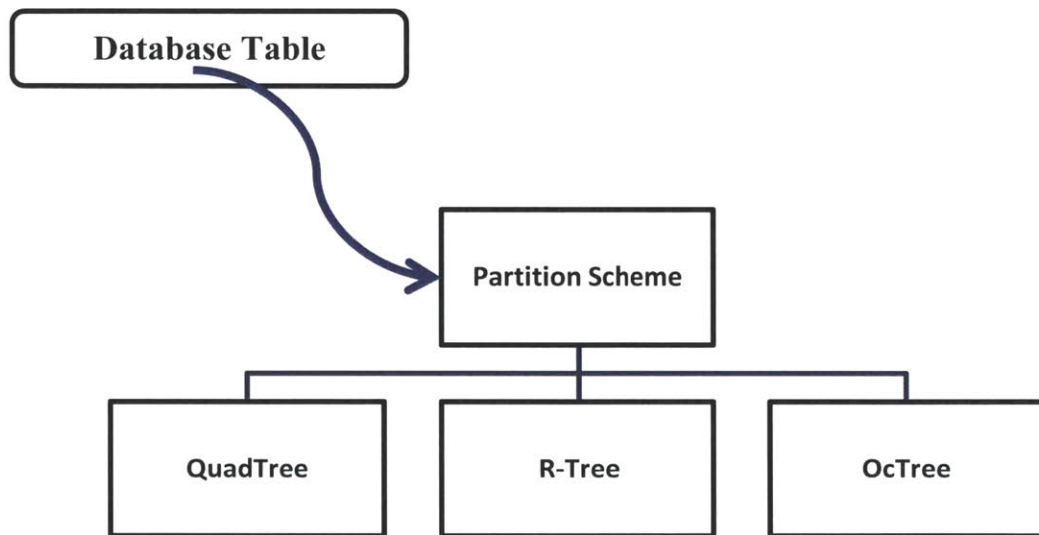


Figure 1.3-1 – Partition Scheme: Abstract Class. Three example implementations of Partition Scheme

For this thesis project, we designed and built the described **Partition Scheme** architecture. Furthermore, we implemented a QuadTree [6] [7] partitioning scheme (See Figure 7) which implements the **Partition Scheme** interface and allows the partitioning of fragments according to the longitude and latitude values. The idea behind a QuadTree is that once a leaf

node, representing a rectangular bound on map, reaches maximum capacity of values; it splits into four leaves representing: northeast, northwest, southeast, and southwest. (See Figure 1.3-2).

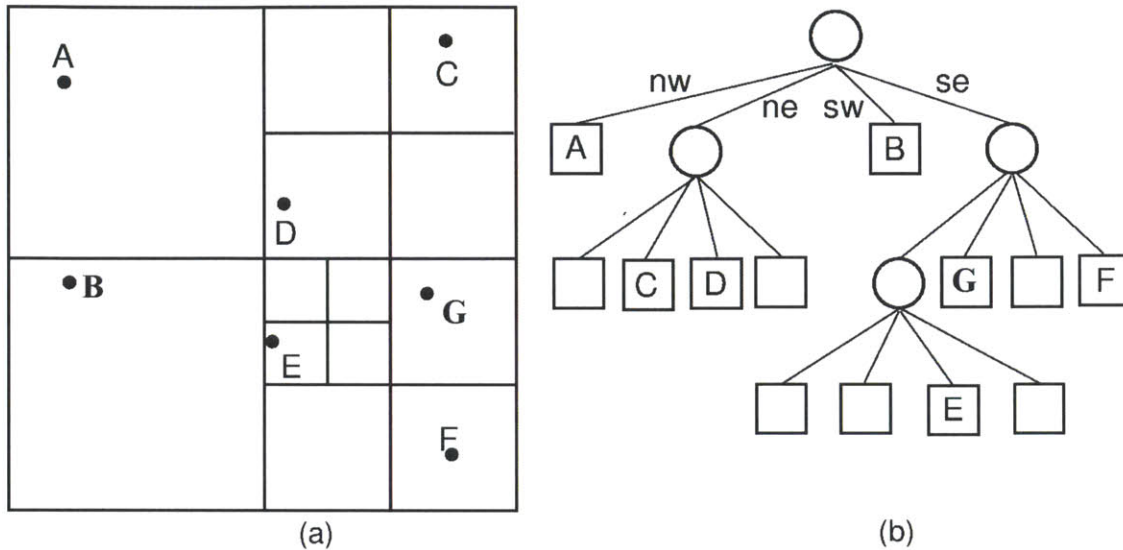


Figure 1.3-2 - Sample QuadTree partition scheme.

1.4 Memory Pool Management and Replacement Policy

MapD was initially implemented as an in-memory database. Yet, for real world datasets, we cannot assume that all the data should fit in CPU/GPU memory. The underlying operating system implements memory caches, but fetching memory blocks involves overhead. Therefore, many real-world DMBSs implement DBMS-managed buffer pool in user-space in order to reduce such overhead.

The buffer pool acts as a memory cache. When the database *Executor*¹ requests a page in memory, the buffer pool will return a pointer to that memory chunk if it exists. Otherwise, the

¹ The Database Executor is the module responsible for interpreting the abstract syntax tree (AST) and executes the query. It may perform optimizations or act on the query plan of the optimizer to execute it.

buffer pool needs to evict a page in memory and replace it with the requested page from disk. The choice of the page to evict/swap is determined by means of an eviction policy.

The eviction policy algorithm used by most commercial systems is LRU (Least Recently Used). While it might be the best for general random access patterns, it does not perform well in a database environment. A general-purpose database server involves the combination of the following access patterns: [8]

- 1) Sequential access to chunks which will not be referenced.
- 2) Sequential access to chunks which will be cyclically referenced.
- 3) Random access to chunks which will not be referenced again.
- 4) Random access to chunks which will be referenced again with some probability greater than zero.

For cases (1) and (2), LRU eviction policy will perform worst unless all the data fits in the buffer pool. For case (2), MRU (most recently used) eviction policy is known to perform the best. [9] While cases (3) and (4) are common as operating systems access patterns, this is not the case for MapD.

For the majority of MapD visualization queries, memory fragments are accessed sequentially and are to be referenced again for other queries. For example, let A be the query of visualizing tweets in New York state that have the word “taxi” while query B is analyzing tweets in New York state with the words “rain” and “sick”. Let query C be the query that animates over time range the visualization of the tweets that have the word “love” in New York City. A , B , and C queries require the scan of the fragments that hold the geo-located tweets in the state of New York. Such queries fall under cases (1) and (2) above.

While sequential access patterns dominate in MapD for now, the buffer pool manager needs to have a modular implementation to allow for various eviction policy implementations that are efficient. This, in turn, allows MapD to become more general purpose database with not many assumptions about access patterns.

Memory pages in the buffer pool are multi-sized. This is because the atomic unit in memory in MapD is a partition of a column, a *chunk*.² (See [new chunk definition](#)). The maximum elements a chunk can hold may differ from table to another. Moreover, a chunk that can hold up to N float values has half the size of one that can hold the same number (N) of double values from the same table.³ The eviction policy implementation for this thesis project, therefore, takes into account the multi-sized pages during replacement.

For this thesis project, we implemented a buffer pool manager for CPU and GPU memory management. It supports multi-sized pages and offers the flexibility of implementing different eviction policies. The details of the Buffer Pool Manager are explained in [the next chapter](#).

1.5 The Deadline Scheduling Problem

The query logs on MapD server reveal that many concurrent requests are arriving. Many of these queries target data of the same table, in particular MapD's tweet table. This implies that these queries share the same fragments in common. For example, if we get N requests from users viewing tweets in the US, they all share the same fragments retrieved by the QuadTree index that cover the US bounds in the world map. This provides an opportunity to share the scan of these fragments on the GPU and produce the output of such queries in one scan.

The design of shared scan architecture involves some challenges:

1. We will have a limited space for output buffer which will limit the number of queries we can scan concurrently.
2. The number of fragments required by each query hitting MapD varies. Consider a query Q_A requiring the fragments $\langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$, and query Q_B , requiring the fragment $\langle 8 \rangle$. In this case, executing Q_B separately and before Q_A could be better than sharing the scan, since the user initiating Q_B will have to wait

² The new definition of a chunk is a partition of a column.

³ Each float value is represented in 4 bytes while a double is 8 bytes.

until Q_A starts to scan fragment 8. On the other hand, one can imagine an approach of reordering the fragments optimally to minimize latencies.

3. Animated queries may have to be treated in a special way to ensure smooth display of frames for the client.

These challenges suggest that MapD can be likened to a real-time system. In real-time systems, the performance is not dependent on the correctness of task results only, but upon tasks meeting their deadlines. [10] While in many real-time systems, missing deadlines is fatal, it is not the case with MapD. This gives us flexibility in the design of shared scan architecture to implement a greedy approach and try our best to meet the assigned deadlines.

In this thesis project, we implement a pipeline to support sharing scan of queries. The first stage of the pipeline handles the query, estimates its deadline, and passes it along to the next stage. The second stage places the query in a priority queue while keeping track of the output buffer size. A separate thread, running continuously, is the scheduler which determines which fragment it should scan next and which queries can share it. Our query scheduler was implemented under the inspiration of the different implementations of OS schedulers like EDF (Earliest Deadline First scheduler) and LST (Least Slack Time scheduler).

1.6 Thesis Road Map

In Chapter 2 we will discuss in detail the design and architecture of MapD2's storage and indexing. In Chapter 3, we explain the details of the memory management in MapD2 and the functionality of the Buffer Pool. Later, we introduce the design of shared scan architecture and the scheduling algorithm for concurrent queries in Chapter 4. We present the results of experiments performed to evaluate our implementation of indexing and shared scans in Chapter 5. Later, we summarize related work in Chapter 6. Then, we propose future work opportunities and conclude in Chapter 7 and Chapter 8 respectively.

Chapter 2 – Database Storage and Indexing

In this chapter, I introduce the design and architecture of the new GPU-optimized database system we have built, MapD2. First, I describe the database table structure including how a table in MapD2 is stored in memory. Next, I describe the architecture of our indexing or partitioning system, **Partition Scheme**, and describe two specific indexes, the **Linear Scheme** and the **Quad-Tree Scheme**. I conclude this chapter by defining the **Catalog Manager**, a necessary component which maintains tables and indexes metadata.

2.1 Database Table Structure

MapD2 decouples the physical partitioning of table data from the logical partitioning. The two main entities that constitute a database table are *fragments* and *chunks*.

Fragments

Every database table in MapD2 consists of a set of logical sub-tables, or *fragments*. A fragment is a table partition with a maximum number of records allowed. Each fragment consists of a set of *chunks*, one or more per table column.

Chunks

Each column is stored on disk as a set of contiguous chunks. The chunk file stores data for a particular column contiguously. Chunks are the atomic units of memory management in MapD2. Each chunk allows for a user-defined maximum number of records. This maximum size is the same as that of the fragment that the chunk belongs to. If a chunk is taken from a column that allows null values, it will have a corresponding bitmap to indicate if value at index is null or not. All column chunks that correspond to a particular set of rows are combined into a **fragment**.

(See Figure 2.1-1.)

The user-defined maximum fragment size introduces a tradeoff. As the sizes of chunks gets smaller, the buffer pool memory cache is able to store more chunks with smaller level of granularity. Yet, with smaller chunks, we get slower transfer of chunks from CPU to GPU across the PCI bus as well as slower reading of chunks from disk due to more I/O operations.

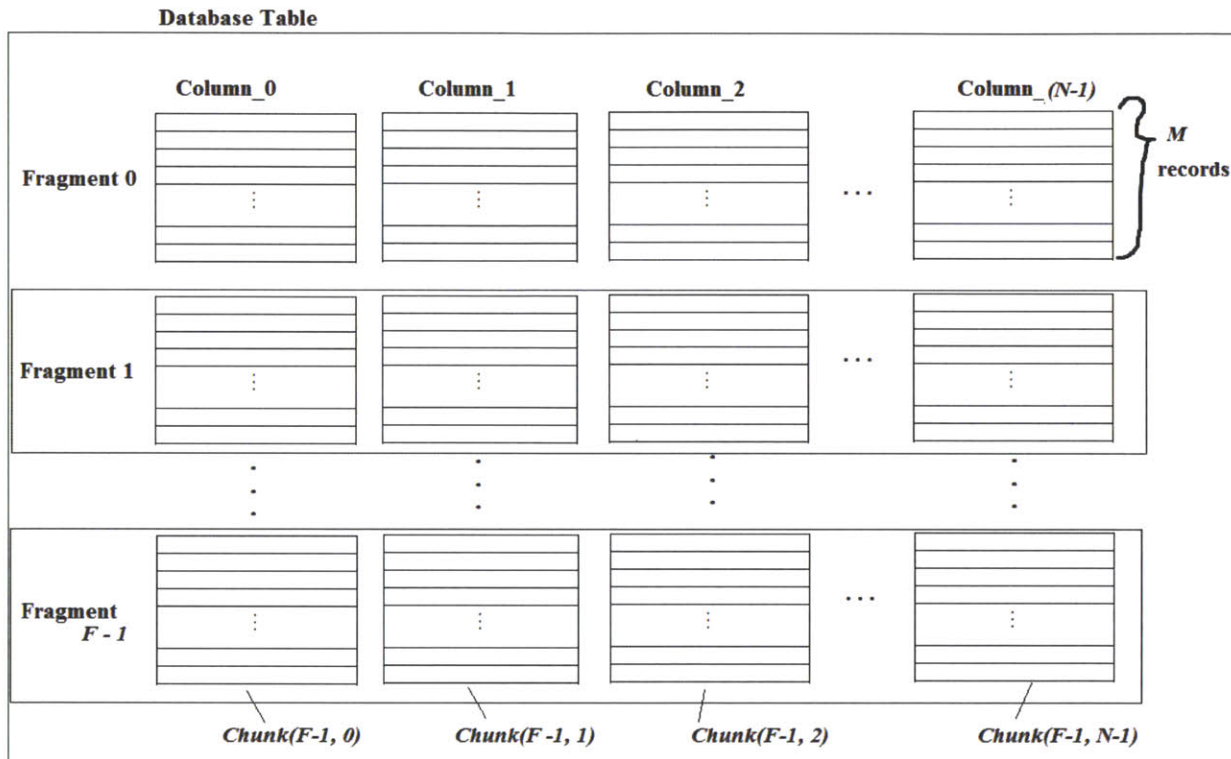


Figure 2.1-1 - MapD2 Database Table Structure⁴

As a major enhancement to MapD, the new design delegates the control of partitioning and handling of fragments to the **Partition Scheme** rather than the table representation. (See Figure 1.3-1). The main table interface defines two procedures. The first procedure inserts a batch of tuples into the table. The second procedure takes in a query and returns a list of fragment IDs that are relevant to the query. (See Figure 2.1-2.)

⁴ The figure describes a table with N columns and F fragments. Each fragment has a maximum size of M elements.

```

Class DbTable {
    PartitionScheme * partScheme_;

    void insertTuples( List tuples ) {
        partScheme_ -> insertTuples( tuples );
    }

    List <int> getFragmentIds (QueryBounds queryBounds) {
        partScheme_ -> getFragmentIds( queryBounds );
    }
}

```

Figure 2.1-2 - Database table pseudo code.

QueryBounds is an abstract class that allows the partition scheme of choice to return the relevant fragments to a specific query. An implementation of *QueryBounds* class captures the attributes that the data is partitioned on. If the partition scheme chosen was some kind of a spatial index, a query bound would be a rectangular bound of coordinates. On the other hand, if the partition scheme was some kind of a hash index on one or more columns, the query bound may comprise the prefixes of the columns used to generate the hash bins. The bottom line is that any *QueryBounds* implementation associated with a particular **Partition Scheme** should keep all necessary attributes to decide on the fragments needed to be scanned to answer queries.

2.2 Physical Partitioning Scheme Design

Now that we've described the basics of how data is laid out in MapD2, we describe the Partition Scheme API and two particular implementations of that interface.

2.2.1 Partition Scheme API

The **PartitionScheme** module is an abstract class that allows MapD2 to be extended with an arbitrary physical partitioning of the table data. The following describes briefly the API of this class.⁵

- **Init** (int tableId, ..., QueryBounds bounds, vector<int> colsPartitioned).
 - This initializes the partition scheme of choice.
 - As described earlier, QueryBounds is another abstract class which its implementation should capture the required metadata for the partition scheme. For example, a spatial index would have a QueryBounds which that a rectangular bounds of the world map. Some partitioning strategies will not require a specific implementation of QueryBounds, and thus pass an empty instance.
 - colsPartitioned is a list of indexes of the columns that the partitioning strategy is needs to perform its job. A 2D spatial index needs the column indexes that hold the longitude and latitude values. A hash index needs the column index/ indices of the keys for the hash buckets.
- **InsertTuples** (vector<string> colNames, vector<void*> data, int numTuples)
 - This inserts a batch of *numTuples* to the table using the partitioning scheme strategy.
- **getFragmentIds** (QueryBounds bounds)
 - Returns the IDs of the fragments specific to the query bounds.

⁵ Only relevant argument to the API description will be shown for readability reasons.

- `Chunk * getChunk (int fragmentId, int chunkId)`
 - Returns pointer to a chunk represented by `chunkId` (column number) and the `fragmentId`.
- `Fragment * getFragment (int fragmentId)`
 - Returns pointer to `Fragment` with `fragmentId`.

2.2.2 Linear Scheme

A simple subclass of the **PartitionScheme** is the **LinearScheme**. Once the batch of tuples/rows are read in memory, MapD2 starts inserting the data chunks contiguously in the fragment until the maximum capacity is reached, where a new fragment is created and so on.

When a query comes in, the `getFragmentIds` of this scheme can only return all the fragments of the table. This is because the **LinearScheme** does not partition the data on any particular attribute.

This scheme was implemented for three main reasons:

1. It serves as a reference point when compared to a partitioning strategy. With regards to the batch insert of tuples, **LinearScheme** is very efficient. Comparing insertion performance of newly implemented schemes to that of **LinearScheme** will help apply optimizations as necessary to the insertion pipeline.
2. The Catalog Manager reuses the Database Table structure to save two tables that keep MapD2 metadata about the tables and columns respectively. These tables use **LinearScheme** as default partition scheme.
3. A MapD2 table created without any partitioning scheme will use **LinearScheme** by default.

2.2.3 QuadTree Scheme

A spatial-index implementation for MapD2 is QuadTree Scheme. It is another subclass of the **PartitionScheme** class.

2.2.3.1 Overview

A QuadTree is a spatial data structure which performs disjoint regular partitioning of the space. This means that when a partitioning needs to be applied, the node in the tree divides the space into mutually disjoint regions of equal shape and size. A QuadTree may operate on d dimensions where $d > 1$. When a division occurs on a leaf node, the space is divided into 2^d regions.

For MapD2, a QuadTree Scheme for 2-dimensional spatial data was implemented to partition on x and y coordinates representing *longitude* and *latitude* values respectively. However, the modular implementation allows for increasing the dimensions with minimal changes. This is a lightweight in-memory tree structure initialized with the following:

- `bounds`: an instance of `RectBounds` a subclass of `QueryBounds`, to describe the bounding rectangle of the root node of the tree.
- `maxFragSize`: maximum size of the fragment representing the node.

2.2.3.2 Tuple Batch Insertion

Tuples to be inserted are loaded from disk onto the CPU main memory as complete column chunks. The QuadTree scheme inserts this batch applying the following steps:

1. **Pre-Subdivide Phase**: pass through all the tuples of the columns involved in the partitioning (x and y coordinates), adding the points $\langle x_i, y_i \rangle$ to the tree and subdividing as necessary. (See Figure 2.2-2 to see how the current implementation splits and assigns fragment IDs.)

2. **Fragment Grouping Phase:** Create an index mapping fragment IDs to indices of tuples. The resulting structure looks like the one in the following table

Table 2.2-1 - Fragment Grouping Phase - map

Fragment ID	Tuple Indices ⁶ - Row numbers
2	6, 7, 10
3	1, 5
4	2, 4
5	3, 8, 9

3. **Pre-flush Phase:** iterate through the index from the previous phase and figure out which tuples need to go to which fragment. Perform disk writes without explicit flushes to disk to optimize performance.
4. **Flush:** flush all the chunk files to disk at the very end.

The main design principle we followed here was driven by the need to to eliminate I/O bottlenecks. The initial phase described above guarantees that, for a single insert, we do not have to write fragments to disk and then discover that this data needs to redistribute among other fragments due to a split.

The in-memory implementation of the batch insertion of tuples in the QuadTree outperforms I/O bounded QuadTree construction and insertion. With this approach, we do not have to implement bulk loading and z-order sorting [11] since we don't implement an I/O bound algorithm.

However, what happens if the data to be inserted does not fit into main memory? We solve this by splitting the insertion into multiple batches. This implies we need to keep track of fragments that get subdivided after we have synced with disk. We do so during the Pre-

⁶ Index here means the position of the tuple or its row number.

Subdivide Phase and end up with a list of fragments that were subdivided after the insertion of the new batch.

As a result, at a stage prior to the second stage (Fragment Grouping Phase), we read the fragment **chunk** data from disk for fragments that will subdivide and append those to the batch we are about to insert. Then, we advance with the steps 2, 3 and 4 as before.

2.2.3.3 Recovery

We need to recover the in-memory representation of the QuadTree for two main reasons. First, when the database server fails and restarts, we want to recover the state of the tree before the failure occurred. Second, to save CPU memory, we allow loading/ unloading of the QuadTree Scheme upon need. Therefore, when we need to insert a batch of values to a previously unloaded table with a QuadTree Scheme, we need to recover the previous state of the tree.

The current scheme implements means of recovery by storing two files on disk: *points file* and the *fragment sizes file*. The points file (Figure 2.2-1 (a)), stores contiguous x and y coordinates of all data inserted so far in the table. The fragment sizes file stores contiguous pairs of the form $\langle \text{fragment size}, \text{maximum fragment size}^7 \rangle$ (Figure 2.2-1 (b)). The points file is used to load the QuadTree Scheme and re-construct the tree so it is ready for new batches. We use the second file to efficiently determine the fragments that were subdivided and so are not needed to be kept track of in memory.

⁷ While the current implementation and benchmarks apply unique maximum fragment size per table, we allow for variable maximum sizes of fragments.

x_1	y_1	x_2	y_2						
-------	-------	-------	-------	--	--	--	--	--	--

(a) Points file – for recovery

F_1	M_1	F_2	M_2						
-------	-------	-------	-------	--	--	--	--	--	--

(b) Fragment Sizes file – for recovery

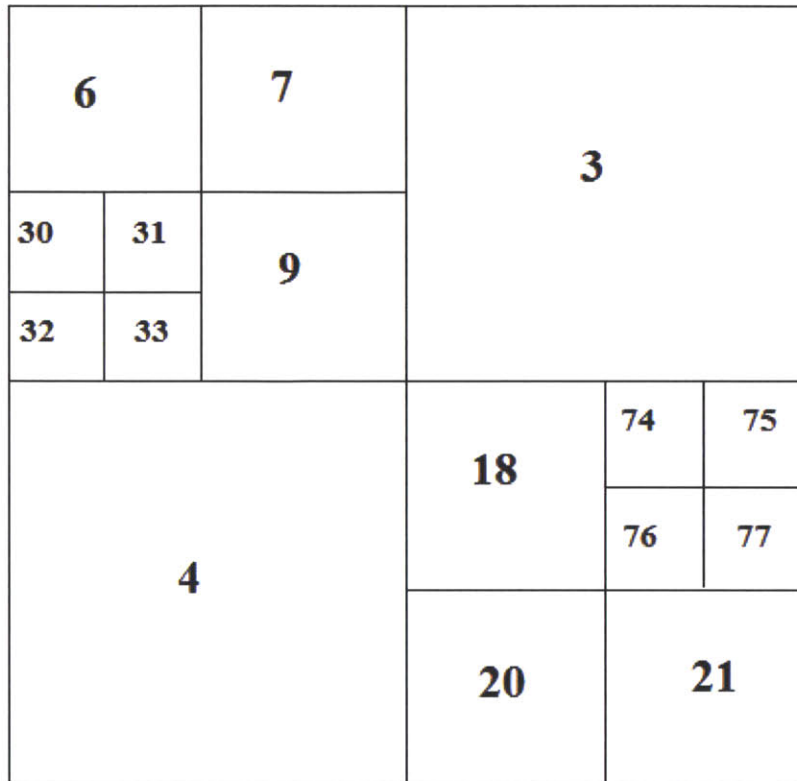
Figure 2.2-1 - Points file and Fragment-Sizes file - for recovery⁸

2.2.3.4 Retrieve relevant fragments

By means of the `getFragmentIds` function defined earlier in PartitionScheme API, we retrieve the relevant fragments in the QuadTree. A recursive call is applied and we prune as necessary the tree nodes with spatial rectangular bounds that do not intersect the query bounds. Depending on the query performed, these fragments returned need to be scanned to return the matching tuples.

⁸ F_i = size of Fragment i . M_i = maximum size of Fragment i .

(a) Example World Map with Resultant Fragments - QuadTree



(b) The QuadTree Representation which Maps to the Resultant Fragments Split Above in (a)

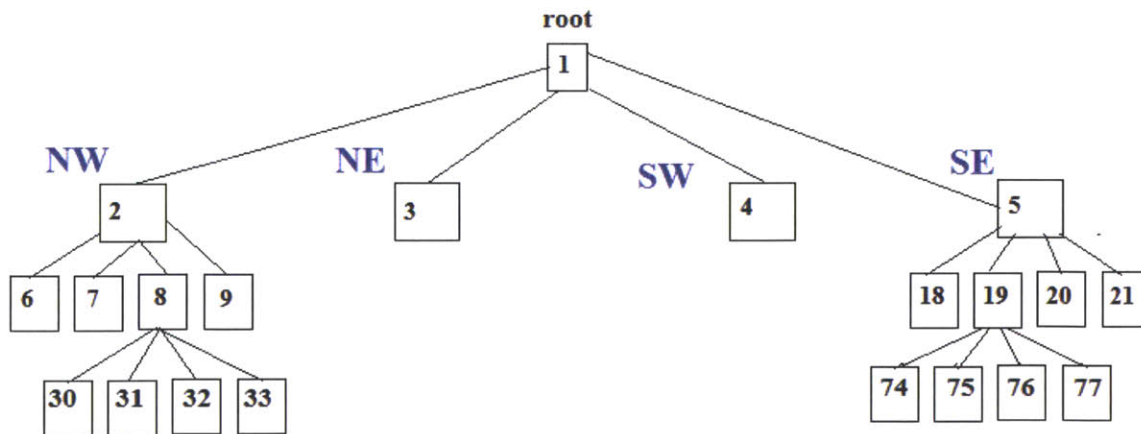


Figure 2.2-2 - QuadTree Scheme Example - Fragment ID assignments - (NW, NE, SW, SE)

2.3 Catalog Manager

The catalog manager, as in most database systems [12], stores the metadata about the tables and their columns in the database system. In MapD2, the Catalog Manager keeps information of the tables, columns and indexes. It reuses the DbTable structure to store the following two tables:

1. `tableTable`: This is a table of tables. It has the following columns / fields
 - **Table name.** Variable sized string representation of table name.
 - **Table identifier.** Auto incremented table ID.
 - **Partition scheme type.** Variable-sized string representation of the partition scheme.
 - **Maximum Fragment Size.** Integer to describe the maximum size of the fragment.

The current implementation of `tableTable` allows for one partition scheme per table. This can be easily changed by adding two tables, one to define all partition schemes we have in the system, and another to map table identifiers to partition scheme identifiers. The latter table defines a many-to-many relationship. For the sake of this thesis project, it was not necessary to partition the same table by means of more than one scheme. (See Figure 2.3-1 for how the catalog can easily support multiple partitioning schemes).

tableTable	
Table Name	Table ID
tweetTable	5
torrentTable	6
musicDataTable	7

PartitionSchemeTable	
Table Name	Table ID
LinearScheme	1
QuadTree Scheme	2
OcTree Scheme	3

Table_partitionScheme_mapTable		
Partition Scheme ID	Table ID	Maximum Fragment Size
2	5	1000000
1	6	2000000
2	6	500000
3	7	3000000

Figure 2.3-1 - Catalog metadata tables for multi-partitioning schemes support

2. `columnsTable`: This is a table of columns. It keeps track of the columns of all tables.
 - **Column name.** Variable-sized string representation of column name.
 - **Table Identifier.** Integer identifier of the table the column belongs to. (Foreign key).
 - **Type.** Variable size string describing the type of data (BigInteger, date, integer, char, text, datetime, long).
 - **Column Identifier.**
 - **Variable Length Flag.** Boolean flag to indicate if the data in the column are variable sized (varchar column, text column ...) or fixed (int column, double column ...).
 - **Is-Partitioned-On Flag.** Boolean flag indicating if the column is involved in the partitioning or not. For example, if a spatial partitioning is used and this

column describes latitude or longitude values, then this flag should be set to true.

- **Null Allowed Flag.** Indicates if we allow nulls in this column or not.

To support tables with more than one partitioning scheme in the future, the `Is-Partitioned-On` flag may be factored out. A new table which maps partition scheme identifiers to column identifiers will suffice to keep track of which columns are partitioned on in what particular partitioning scheme.

Figure 2.3-2 shows an example of metadata in the catalog when the tweet table uses both QuadTree and OcTree partitioning. The QuadTree partitions on longitude and latitude, while the OcTree partitions on longitude, latitude and date.

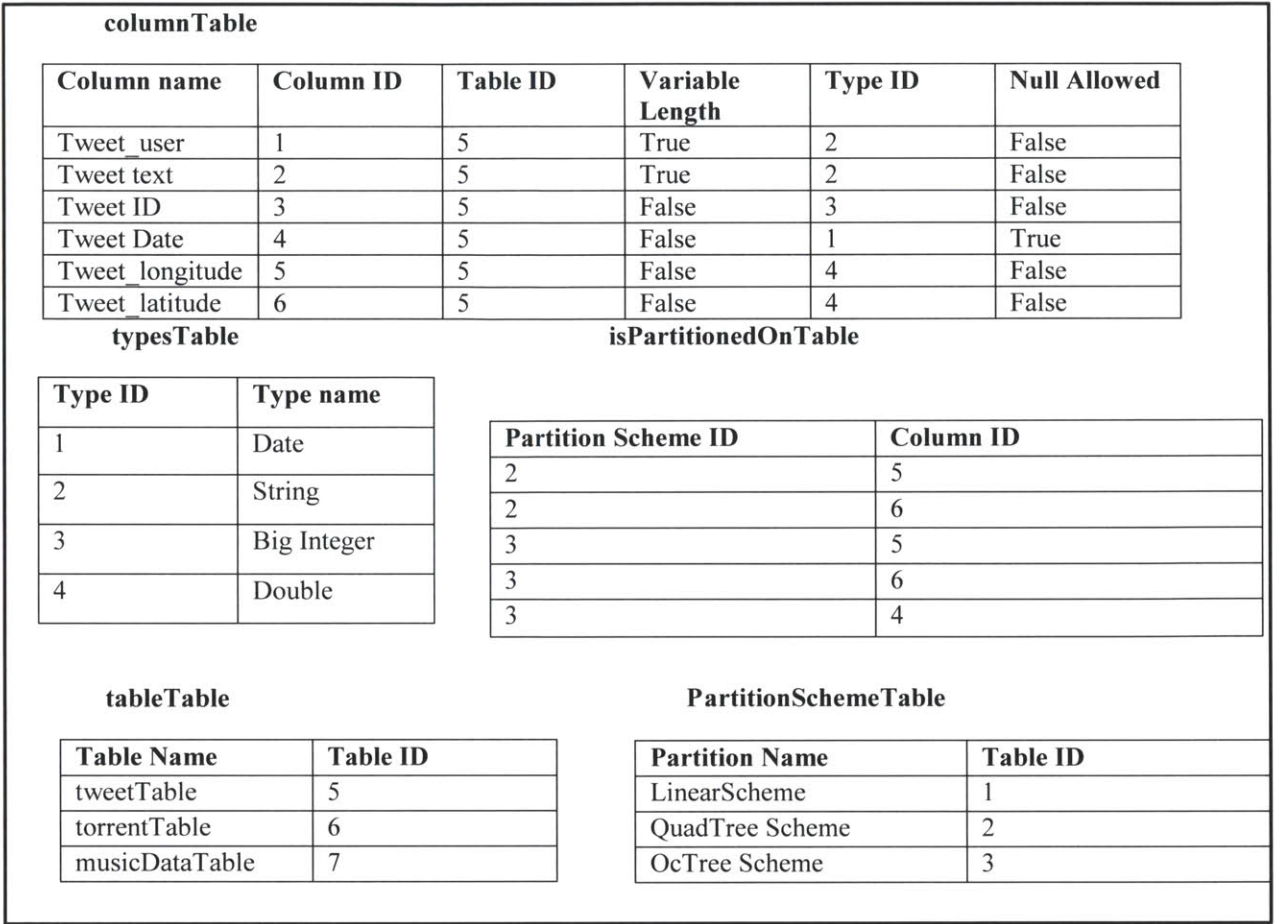


Figure 2.3-2 - Catalog metadata – support of multiple partition schemes

When the MapD2 server is initialized, the catalog reads the metadata tables. Instead of placing the metadata in the memory buffer pool, it is stored in memory data structures like hash sets, hash maps, lists, and vectors. This is more efficient than having the chunks stored in buffer pool memory. This is acceptable since this data is not to be used by the query executor for query processing. This data is to be used mostly by the query parser and query optimizer only.

Briefly, the main API calls of the Catalog Manager are:

- **addTable**
 - Adds a table to MapD2 and updates the catalog metadata.

- Client should specify partitioning scheme choice, the columns partitioned on, the description of the columns (name, data types ...), and the table name.
- **loadTableFromMySQL**
 - Similar to *addTable*, the client should specify all the required fields. The main difference is that this API call imports an existing table from MySQL DB server into MapD2.
 - The Catalog Manager keeps track of connectors to allow importing tables from other databases like MySQL.
- **getTableByName**
 - Retrieve a pointer to the DbTable structure of the specified table by name.
- **getColumnDesc**
 - Given a column name and table name, retrieves a pointer to a structure that describes the specified column. This is mainly column metadata.

2.4 Summary

In this chapter, we learned about the table structure of MapD2 and how the table data is stored in memory. Moreover, we learned the details of the partitioning scheme implementation and how the QuadTree partitioning strategy works. We also learned how MapD2 maintains table and column metadata through the Catalog Manager. Now, we move on to discuss the implementation of the database cache, also known as the buffer pool.

Chapter 3 – Database Memory Management

Now that we've described how MapD2 stores tables and allows various indexing schemes, we need to discuss how it manages table chunks in memory. We've discussed in chapter 1 that many DMBSs implement their own caching mechanism by means of a buffer pool. Therefore, in this chapter, I first discuss the design and architecture of the **Buffer Pool Manager** for MapD2 in a distributed multi-node setting. Then, I describe in detail the data structures of the Buffer Pool, their maintenance, and the Buffer Pool API. Finally, I describe how MapD2 supports multiple **Cache Eviction Policies**.

3.1 Buffer Pool Manager Overview

In this section, I describe the design and implementation of the Buffer Pool. This serves as an implementation of memory cache in user-space to accommodate for database access patterns and thus improve the cache hit rate.

3.1.1 Three-level Memory Hierarchy

Unlike many databases systems, MapD2 does not use a dual-level memory model, where data moves between the CPU and hard disk. Instead, the MapD2 architecture implements a three-level model of memory and data synchronization. These three levels are arranged in a pyramid like arrangement where each level is slower computationally but larger in size than the previous level. The three levels from fastest to slowest are: GPU, CPU, and the hard disk. Furthermore, every level is a mirrored subset of the previous level, where the most frequently accessed data remains in the faster memory level. (See Figure 3.1-1.).

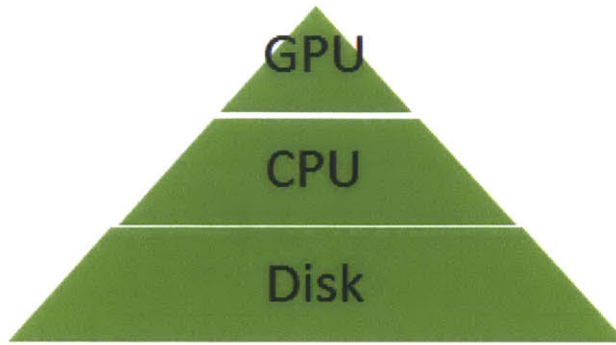


Figure 3.1-1 MapD2 Buffer Pool Memory Hierarchy

3.1.1 Architecture

MapD2 implements a shared-nothing architecture described earlier. Each processor is a GPU card with thousands of cores. Every GPU owns its private memory. Furthermore, every GPU owns private host (CPU) memory and private disk.

This architecture simplifies the design and architecture of the buffer pool. Shared-nothing design implies we can design independent buffer pools, one per GPU. The figure on the next page illustrates the high-level design.

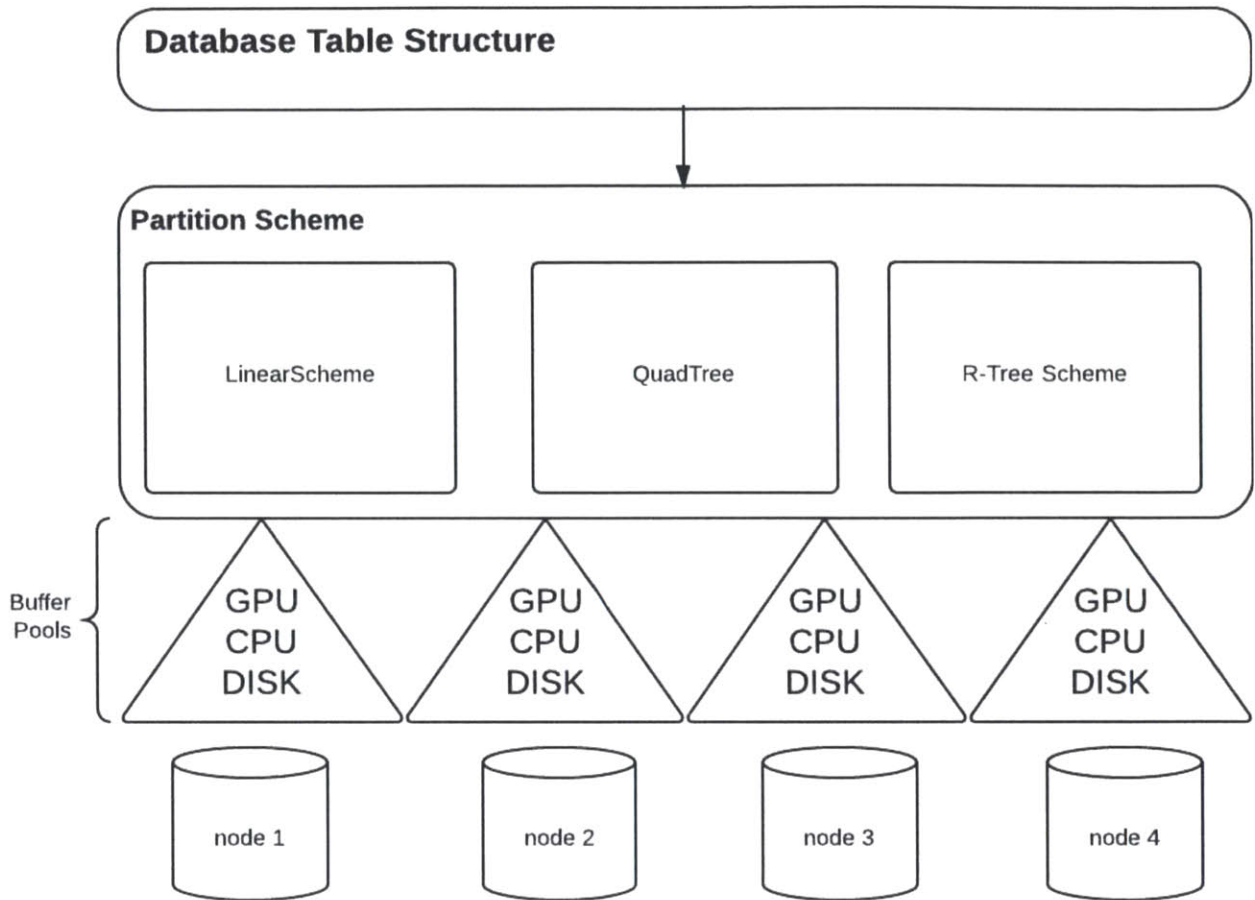


Figure 3.1-2 MapD2 architecture – 4 nodes example

3.1.2 Data Structure Design and Maintenance

Now, we discuss the data structures that the buffer pool maintains for efficient lookup of table chunks and efficient eviction of multi-sized memory pages.

3.1.2.1 Buffer Pool Partitioning

For technical reasons of GPU memory allocation, we allocate pages in memory as integer multiples of 512 bytes. Hence, in the following discussion, 1 page = 512 bytes.

The buffer pool implementation allows the partitioning of the cache into page segments based on the chunk sizes. This partitioning is done based on two parameters: number of buffers (B), and the page dividing size (D). If $B = 1$, D does not really matter and we have no partitioning. For example, if $B = 3$ and $D = 5$, then our buffer is partitioned into 3-sub buffers, one holding chunks of size in range $[1,5)$, the second $[5, 10)$, and the last will hold any chunks of size 10 or greater. (See Figure 3.1-3).⁹

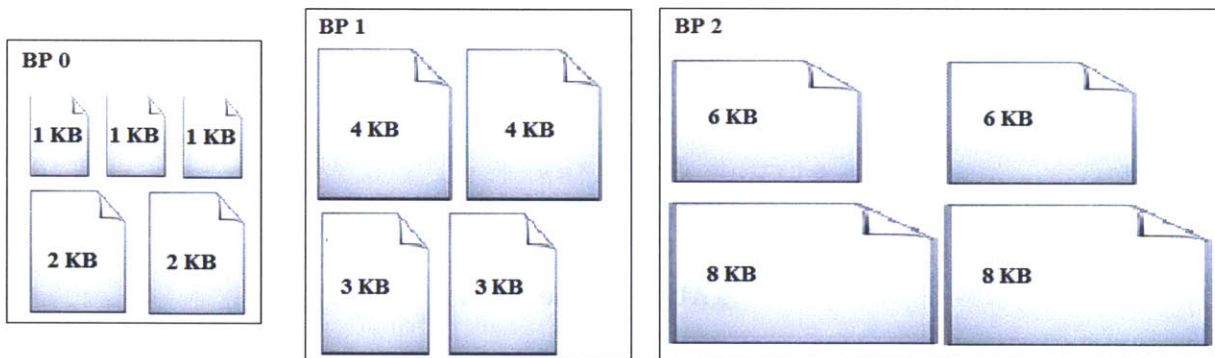


Figure 3.1-3 - Buffer Pool Partitioning Example. $B = 3$, $D = 5$.

The sizes of the buffer pool partitions are decided upon database initialization. It is the responsibility of the database administrator to decide on the B and D parameters and the sizes for each partition. These parameters and partition sizes can be different for each node in the distributed MapD2 architecture.

⁹ Remember that 1 KB = 2 pages in our definition.

3.1.2.2 Data Structures

Memory Segments

The *ChunkInfo* structure describes a memory segment in the buffer pool. It stores metadata of table chunks or other memory chunks stored in the cache. See the figure below.

```
struct ChunkInfo {  
  
    PageBounds pageBounds;  
    MemStatus memStatus;  
    bool dirty;  
    ChunkKey chunkKey;  
  
};  
  
struct ChunkKey {  
    int tableId;  
    int fragmentId;  
    int chunkId;  
  
};
```

Figure 3.1-4 - ChunkInfo (Memory Segment) and ChunkKey Definitions

Every memory segment has `pageBounds` which is simply a pair of page values. The first value describes at what page the chunk data starts, where the second value is where it ends. The `chunkKey` is a unique identifier of a table chunk in MapD2.

The `memoryStatus` is an enumeration value. It can be one of the following:

- **FREE:** Indicates the page is free and available for use.
- **PINNED:** Indicates the page is pinned in memory and will not be evicted. Such pages may be allocated to keep track of metadata tables from catalog or even histograms to be used for query optimizers in the future. Moreover, we use pinned pages for the output buffers when implementing the shared scan architecture described in the next section.
- **TEMP:** Indicates the page is temporary and may not be evicted. The main purpose of such page is to keep intermediate results of queries.
- **USED:** Indicates memory page is used for some table chunk. Such page may be evicted if necessary.

Each buffer pool partition, whether managing GPU or CPU memory uses a linked list, called *MemSegs*, to maintain the order in which memory segments are accessed. Many buffer pool eviction policy implementations need to keep track of access history of memory segments and/or frequency. Therefore, it made sense to use a linked list representation where the head/tail pointer points to least recently used page and the tail/head pointer points to the most recently used page depending on the eviction policy used. If an MRU (most-recently-used) or LRU (least-recently-used) eviction policy is used, then finding the page to be evicted is $O(1)$ since it is either at the head or tail of the linked list depending on the implementation.

Chunk Page Map

This is a simple hash table implementation mapping chunk keys to the iterator of the memory segment. This data structure is maintained to achieve $O(1)$ time operation of accessing the cache if the memory chunk resides in it. An alternative will be iterating over the linked list which is $O(n)$ in the worst case where n is the number of memory segments in the buffer pool partition accessed.

3.1.3 API summary

The following are the main API calls to the buffer pool:

- `allocateSpace(size, gpuFlag, memStatus)`.
 - Allocates space of specified page size on GPU if the `gpuFlag` is set, otherwise it is allocated on the CPU.
 - Set the memory status of the page. Can be only TEMP or PINNED.
- `getHostChunk(chunkKey)`
 - Returns a pointer to the data in CPU RAM for the `chunkKey` specified. It follows this simple algorithm:
 - If the chunk exists, return pointer in memory.
 - Otherwise, look for free space
 - ❖ If there is free space, read the chunk from disk allocate space in buffer pool, update the data structures, and return memory pointer.
 - ❖ If there is no free space we need to evict some memory segment(s), copy the chunk over from disk in place of evicted segment(s), update data structures, and return pointer to the data.
- `getDeviceChunk(chunkKey)`
 - Returns a pointer to the data in device/ GPU memory for the specified chunk key. It follows the following algorithm:
 - If chunk exists in GPU memory, return pointer
 - Otherwise, look for free space
 - ❖ If free space found, call `getHostChunk` on the same `chunkKey`, allocate space on device, update data structures, and copy data over to the GPU, and then return pointer to data.
 - ❖ Otherwise, we need to evict one or more memory segments from the GPU, use `getHostChunk` to get the chunk data,

update data structures, copy over to the GPU, and finally return pointer to data on device.

3.1.4 Eviction Policy Design

Now, we discuss how our buffer pool implementation allows for the implementation of different cache eviction policies.

3.1.4.1 Overview

This discussion of eviction policy design applies to any buffer pool partition whether it manages GPU or CPU memory. The eviction policy is governed by two main functions / methods in the buffer pool manager:

- **updateAccess(MemSegs::iterator it)**
 - When a page is accessed by the buffer pool by means of the API calls defined above, this method is called on the new iterator of the memory segment accessed.
 - This method moves the element to a different position in the linked list in constant time. The destination position of the segment depends on the policy of choice. For example, if an MRU policy is applied, we place the accessed segment in front of linked list. This means that the list goes in order of most accessed to least accessed memory segments. This implies that MRU will evict always the first segment of the linked list since it is the most recently used.
- **getMemSegToEvict**

- This function iterates over the linked list according to policy of choice and returns an iterator of the memory segment to be evicted.
- This function makes sure it does not choose TEMP or PINNED pages for eviction.

3.1.4.2 Eviction Cases

When a memory segment is chosen for eviction according to the policy implemented, we need to deal with the following cases:

1. The segment evicted is **equal** in size of pages to that replacing it.

This case simply evicts the memory segment and places the new segment perfectly.

2. The segment evicted is **greater** in size than that of replacing page.

In this case, the replacing page utilizes the space needed, and we split the rest creating a new segment with a FREE memory status. We place that FREE segment at the beginning of the linked list to speed up the lookup for free segments.

3. The segment evicted is **smaller** in pages size than that of replacing page.

This case is more complicated. It requires us to evict more than one page until the combined size of evicted pages is enough for the page coming in. Since we evict in order governed by the policy, we are not guaranteed contiguous space in memory. Therefore, we implement a coalescing of a copy-in-place approach to de-fragment the cache and reorder the segments. (See Figure 3.1-6). The coalescing only happens when it is needed by the buffer pool since it is an expensive operation.

Figure 3.1-6 shows an example where we need to evict E1, E2, and E3 memory segments (order depends on policy) from a buffer pool partition of 2 million pages – 1GB. The figure shows the physical location in the buffer and how coalescing is done to gain a free segment to allocate our new segment.

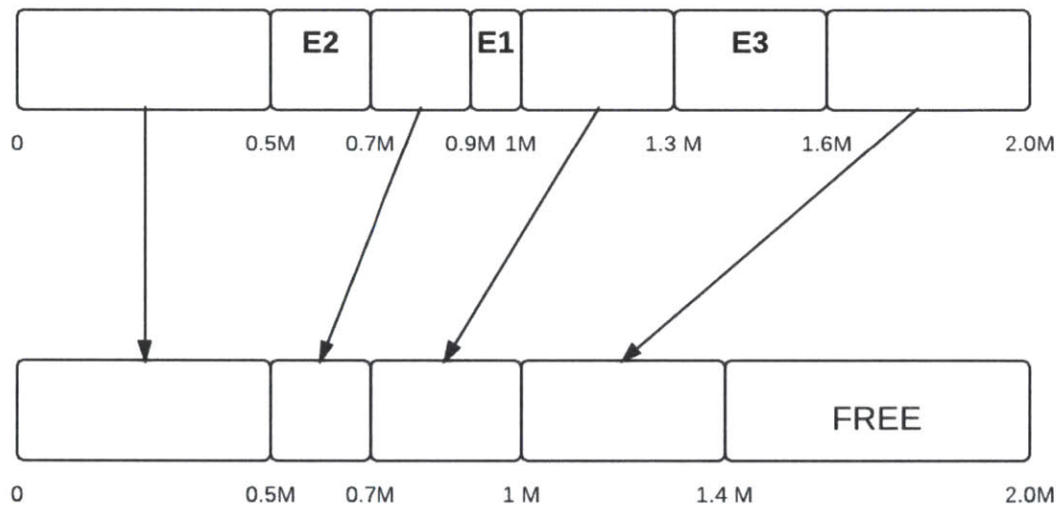


Figure 3.1-5 - Eviction Case 3 - Buffer Pool Coalescing

3.2 Summary

In this chapter, we discussed how MapD2 manages memory by means of a three-level buffer pool per node. We explained the data structures and API calls supported by the buffer pool. Then, we explained the partitioning of the buffer pool based on page sizes. Finally, we described how our implementation allows for the implementation of reference-aware cache page-replacement policies. Now, we are ready to discuss the design of the shared scans architecture which allows MapD2 to handle many concurrent queries.

Chapter 4 - Shared Scan Design

In this chapter, I start by introducing the scheduling problem in real-time systems and show how it relates to the problem we are trying to solve with shared scans. Then, I describe the high level architecture and the pipeline for shared scan. Finally, I describe the detailed design and implementation of the components involved including the **Task Wrapper**, the **Query Information Estimator**, and the **EDF Shared Scheduler**.

4.1 Overview

In order to understand the need for an asynchronous query scheduler for MapD2, we need to discuss general real-time systems, scheduling models, and task parameters. Then, we can map the relevant portions to our application, MapD2.

A real-time system consists of a set of tasks $T = \{\tau_1, \tau_2, \dots, \tau_n\}$ and associated deadlines $D = \{D_1, D_2, \dots, D_n\}$.

Usually, tasks have the following timing properties: [10]

- **Release time (or ready time):** Time at which the task is ready for processing.
- **Deadline:** Time by which execution of the task should be completed, after the task is released.
- **Minimum delay:** Minimum amount of time that must elapse before the execution of the task is started, after the task is released.
- **Maximum delay:** Maximum permitted amount of time that elapses before the execution of the task is started, after the task is released.
- **Worst case execution time:** Maximum time taken to complete the task, after the task is released. The worst case execution time is also referred to as the *worst case response time*.

- **Run time:** Time taken without interruption to complete the task, after the task is released.
- **Weight (or priority):** Relative urgency of the task.

Furthermore, a task in real-time systems has the following types of properties:

1. **Hard/ Soft real-time:** a hard task τ_i needs to be completed before its absolute deadline D_i . Failing to meet a deadline is fatal for the system. On the other hand, soft task τ_i gets penalized if delayed beyond specified deadline D_i . In other words; it is not fatal for such task to miss its deadline.
2. **Periodic, aperiodic, and/or sporadic.** Periodic tasks are activated regularly every fixed period of time. Aperiodic tasks happen unpredictably at no fixed period. Yet, sporadic tasks fall in between periodic and aperiodic, since these tasks are expected to be activated within some bounds or time range.
3. **Preemptive/ Non-Preemptive.** A task is preempted to allow a higher priority task to use the resources in a preemptive system. Otherwise, once the task starts, it needs to complete before a higher priority task starts.

In the context of MapD2, tasks are the queries originated by the clients. We view queries as **soft real-time** tasks since we cannot guarantee that all tasks received at server finish by their assigned deadlines. Furthermore, these tasks are **aperiodic** in nature and appear at arbitrary times.

In order to implement a sub-optimal priority driven dynamic scheduling algorithm, we need the queries to be **preemptive**. Some real-life schedulers [13] define pre-emption points for the tasks. These are points at which preemption can occur and higher priority tasks will take over the execution. With our definition of a task as a query, we place the preemption points at the end of the scan of a fragment. (See Figure 4.2-1). Note that, at the moment, we deal with queries that consist of fragments from the same table.

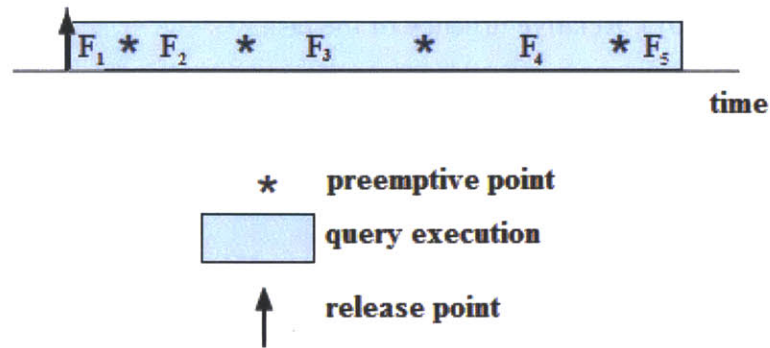


Figure 4.1-1 Example Task with Preemptive Points

4.2 Architecture of Shared Scan Pipeline

At the high level, MapD2's shared scan pipeline is as shown in Figure 4.2-1. The clients send a query request to the server, which then passes through the first stage, the Task Wrapper. As the name suggests, this stage decodes the query information and wraps it into an object, **QueryInfo**, with attributes describing the query. This information is then passed onto the next stage, the Query Info Estimator.

At the second stage, we analyze the query for important attributes relevant to the scheduler. One important property is the size of the query output. Depending on the query and its type, we expect different size of query output where some can be calculated exactly while others need to be estimated. If the query, Q_A , generates a 2D histogram of $1600 * 900$ pixels and counts, then we expect $1600 * 900 * 4 = 5760000$ bytes = 5.76 MB of output. (The count at each pixel is a 32-bit / 4 bytes integer). On the other hand, a different query, Q_B , which groups unique (GROUP-BY) words and counts occurrences may need 12 bytes for each row of output (8 bytes for word identifier if we index words by long primes, and 4 bytes for integer count). An estimate of the number of unique words, hence the number of rows, is needed for Q_B in order to predict the size of its' output buffer.

Moreover, at this stage of the pipeline we estimate the *relative deadline* of the query. This procedure is of key importance to the performance of the scheduler. Any implementation of the Query Information Estimator may use the **QueryInfo** attributes to estimate the deadlines.

Finally, after estimating other attributes required by the scheduler, we add this new information to the **QueryInfo** structure and send this to the final stage, the Query Scheduler.

The next stage, the Query Scheduler, is implemented in a producer-consumer design pattern. In this scenario, the producer is the function inserting the query information to the internal data structures of the scheduler while the consumer is the asynchronous function running endlessly picking fragments for shared scanning among queries.

We define the parameter **MAX_OUTPUT_BUFFER_SIZE** which is the maximum amount of output memory needed, in bytes, for concurrent queries. All queries processed by the scheduler need to have their output buffer allocated prior to the scheduling phase. We use the **PINNED** memory feature of the buffer pool to ensure the output buffers are not evicted. The queries that cannot be processed, because we might exceed the maximum output buffer size we can allocate, are kept in the *waiting queue*. The query scheduler keeps track of the memory amounts allocated so far for output and will only pull out a new query from the waiting FIFO / queue if the total sum of the output buffer sizes does not exceed **MAX_OUTPUT_BUFFER_SIZE**.

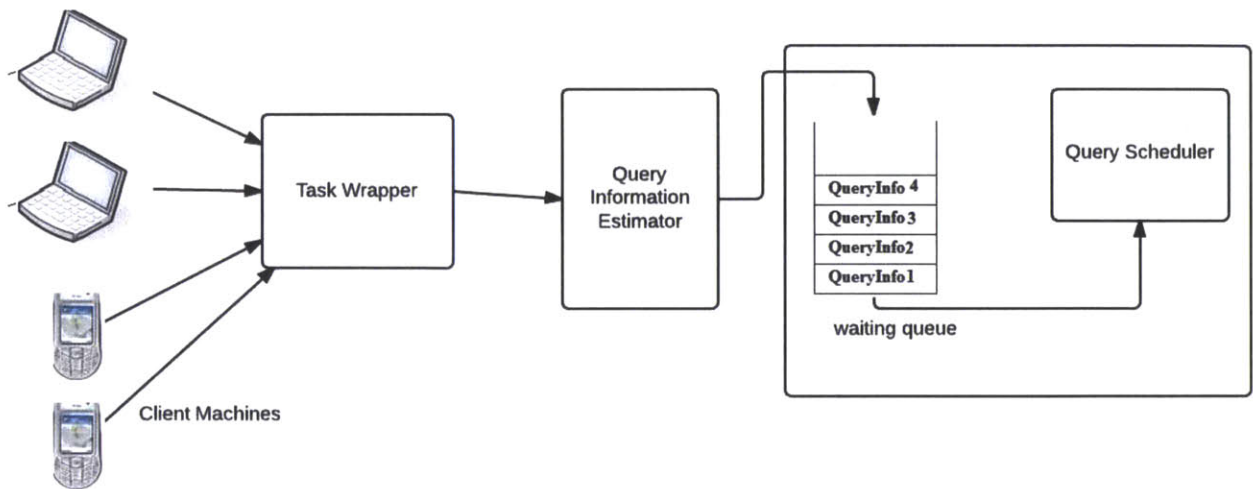


Figure 4.2-1 - High Level Pipeline for Shared Scan

4.2.1 Task Wrapper

Once a query request initiates from the client, the Task Wrapper's job is to classify the type of the query and identify what table(s) it needs. On the basis of this classification and the partitioning strategy of the table(s) required, this module finds the fragment information of all fragments needed to answer the query. This includes the identifiers of fragments, their sizes, and the column/chunk identifiers needed by the query. The module, as its name suggests, wraps all this information in **QueryInfo** structure and passes it to the next stage.

4.2.2 Query Information Estimator

This is an abstract module which takes in the **QueryInfo** structure and uses information to estimate deadline of the query. Depending on the implementation, example deadline (D) estimation implementations include one or a mixture of the following:

- Fixed Deadline $D = d$.
- $D(n)$: as a function of the number of fragments n
- $D(f_1, f_2, \dots, f_n)$: as a function to the fragment information.
- $D(query_type)$: as a function of query type.
- $D(\mathbf{QueryBounds}, partition\ scheme)$: as a function to the query bounds and the partitioning strategy.

Furthermore, this module estimates the output buffer sizes for the queries as mentioned earlier. The current implementation deals with **HISTOGRAM_TYPE** queries. Such queries generate a 2D histogram of pixels, where each pixel maintains a count. These histograms are used to generate PNG / JPEG images to show geo-located data on the map. Moreover, we deal with animated **HISTOGRAM_TYPE** queries. Based on two parameters, one describing the frame width, and the other describing the number of frames, such query produces histograms describing different time ranges. At the client side, this is an animation on the map of geo-located data over time.

4.2.3 Query Scheduler

This is the final and the most critical stage in the pipeline. The Query Scheduler is an abstract class with the interface defined as follows:

- **addQuery(QueryInfo &qInfo)**
 - This adds the query to the internal scheduler structure. It involves bookkeeping of necessary information about the query including fragment information and deadlines to aid the scheduler pick the next fragment to be scanned.

- **consumeAndScan ()**
 - This function runs asynchronously in a separate thread. As the names suggests, it is consumes available queries and applies sharing scan of fragments among queries.

4.2.3.1 EDF Shared Scheduler

For this thesis, we implemented a dynamic priority driven scheduling. One algorithm that stands out is EDF (Earliest Deadline First) scheduling. It is a preemptive scheduling algorithm that allows jobs with the earliest deadline to run first. This algorithm is used in real-time operating systems by placing processes in a priority queue. Furthermore, EDF is known to be optimal for a schedulable set of tasks. It is optimal in the sense that it minimizes the maximum lateness for n independent tasks arriving at arbitrary times. [14]

In MapD2's EDF shared scheduler, we maintain an invariant such that the sum of output buffer sizes of all queries in the priority queue is less than **MAX_OUTPUT_BUFFER_SIZE** . We achieve that by placing queries we cannot schedule in a waiting queue. The waiting queue processes queries in order of arrival or in FIFO order. We chose this particular implementation for two main reasons:

1. The invariant maintained simplifies the scheduling and sharing implementation. We know that any query in the priority queue must already have output buffer space allocated. On the other hand, if we didn't maintain such invariant and let any query be placed directly in the priority queue, we will end up facing a deadlock. Such deadlock arises when there is not enough output buffer space left and the next fragment to scan is needed by at least one query with no output buffer space allocated. Avoiding such deadlock requires extra bookkeeping and involves a more complex implementation of the scheduler. With such implementation, the choice of next fragment to scan will need to take into account the number of queries requiring the scan of such fragment that have allocated output buffer space.

2. The waiting queue processes queries in arrival order and not by earliest deadline to avoid starvation. Starvation can happen if many users are sending query request with very short deadlines and always getting preference of execution by EDF algorithm. By having a FIFO waiting queue, we can guarantee that queries with longer deadlines end up being executed.

Now that we've argued for our choice of implementation, we describe the details of how the **addQuery** and **consumeAndScan** interface implementations work for **EDFSharedScheduler**.

The **addQuery** API call simply adds queries to the waiting queue. The **consumeAndScan** API call is responsible for two main jobs:

1. As long as the waiting queue is not empty, pull as many queries as possible and place them in the priority queue as long as the invariant is maintained. For each query placed in the priority queue, we allocate PINNED memory in the buffer pool so that these pages are not evicted.
2. Pick the query with the earliest deadline from the priority queue. Once this is done, the scheduler picks an un-scanned fragment from this query that can be shared amongst most of the queries. A GPU kernel runs the scan allowing the queries sharing the fragment to fill results of their output buffer in parallel. When fragment scan is done, we mark this fragment as scanned. When all of the fragments of a query have been scanned, we run a separate thread to deliver the query result to the client, which then signals to the scheduler to release the output memory buffer the query had to allow for newer queries to be processed.

4.3 Summary

In this chapter, we presented the architecture of the shared scan design. We explained a greedy EDF algorithm and described how our implementation chooses the fragment it needs to scan next in order to share amongst as many queries as possible. Now, we move on to discuss the experiments we have performed for evaluating our work in the next chapter.

Chapter 5 – Experiments and Results

In this chapter, we present experiments on MapD2 to evaluate and analyze the performance of the **QuadTree** partitioning index and the **EDF Shared Scheduler**.

5.1 Datasets

We experimented with three real-life datasets:

1. Small-sized geocoded tweet data ~ 14.2 million tweets.
2. Medium-sized geocoded torrent data ~ 45.3 million IP addresses of torrent seeders / peers worldwide. This data was collected by means of a periodic scraper implemented to download torrent files, connect to trackers, and extract IP addresses using the Bit-Torrent protocol.
3. Large-sized geocoded tweet data ~ 200 million tweets.

In each of these datasets, we have at least the following three fields (columns):

1. **Latitude**: a double (8-byte) value representation of latitude / x – coordinate.
2. **Longitude**: a double (8-byte) value representation of longitude / y – coordinate.
3. **Time**: a 4-byte representation of time values. In the context of tweets, it is when a tweet was posted, while in the context of torrents, it is the time at which the IP address was received by the tracker.

5.2 Benchmark Query

Throughout the experiments to follow, we use the **generate2D_Histogram** query for analysis and evaluation of geo-coded data. This result of this query is a 2-dimensional histogram of integer values that represent counts of records for a given bounding box in space in a specified time range.. This query can be thought of as a simple GROUP-BY pixels query. The result is therefore, a count of records of data found at each pixel.

This query needs the following parameters:

- **Bounding Box:** represents a rectangular bounding box in space on the world map. At the client side, these values are obtained from the zoom level the user does on the map.
 - xMin: minimum x coordinate / latitude
 - xMax: maximum x coordinate / latitude.
 - yMin: minimum y coordinate / longitude.
 - yMax: maximum y coordinate / longitude.
- **Screen Resolution (pixels)**
 - Width: integer number of pixels for width of screen
 - Height: integer number of pixels for height of the screen.
- **Time range**
 - tMin: the minimum time.
 - tMax: the maximum time.

3	6	77	233	7	777	900	102	1
19	13	17	29	177	233	212	424	2
2	24	88	99	911	911	212	33	4
33	3	99	10	12	16	2	12	8
44	1	122	98	990	133	1334	113	9
13	19	21	112	111	991	335	556	987
5	2	3	33	229	221	33	110	19
6	22	21	44	55	333	928	113	44
5	13	39	122	78	799	909	133	179
224	133	212	787	909	133	222	898	13

Figure 5.2-1 An example histogram result of 9 * 10 pixels

5.3 Partitioning Scheme Experiments

In the following sections, we explain and show results of experiments performed to evaluate the performance of the QuadTree and the Shared Scans architecture. All of these experiments are run on a server with 256 GB of CPU RAM and having 8 Nvidia Tesla K40m GPU cards. The GPU name is GK110B and has 12.288 GB of GDDR5 memory with bandwidth of 288 GB/second.

5.3.1 QuadTree Index vs Linear Scheme

As mentioned in chapter 1, this thesis project implementation is geared towards minimizing the query latency to support fast analytics and visualizations. Our QuadTree partitioning strategy / index enables MapD2 to scan the necessary fragments only and not all fragments in a table.

Our performance metric for evaluating the QuadTree partitioning scheme is simply the CPU/GPU scan time of the **generate2d_Histogram** query. We vary the **selectivity** of the data to obtain measurements. Selectivity, $S(q)$ for query q is defined as the result of dividing the number of records after applying the filter $\sigma(q)$ on table T by the total number of records in the table.

Equation 5.3-1 Selectivity S

$$S(q) = \frac{|\sigma(q)|}{|T|}$$

We use the **LinearScheme** as a point of reference to compare the performance of the query for a table partitioned on. In following experiment, we insert two batches of about 14.2 million records resulting in about 28 million. The parameters are defined in table below.

Table 5.3-1 QuadTree experiment parameters – small dataset

Data Size	28×10^6
Maximum Fragment Size	3×10^6
GPU buffer pool size	3 GB
CPU buffer pool size	16 GB
GPU bandwidth	280 GB / sec
Width * Height (pixels)	1366×768

Figure 5.3-1 and shows a plot of query time against selectivity for running the histogram generation query on the GPU. Figure 5.3-2 makes the same plot, but only runs the query on the CPU.

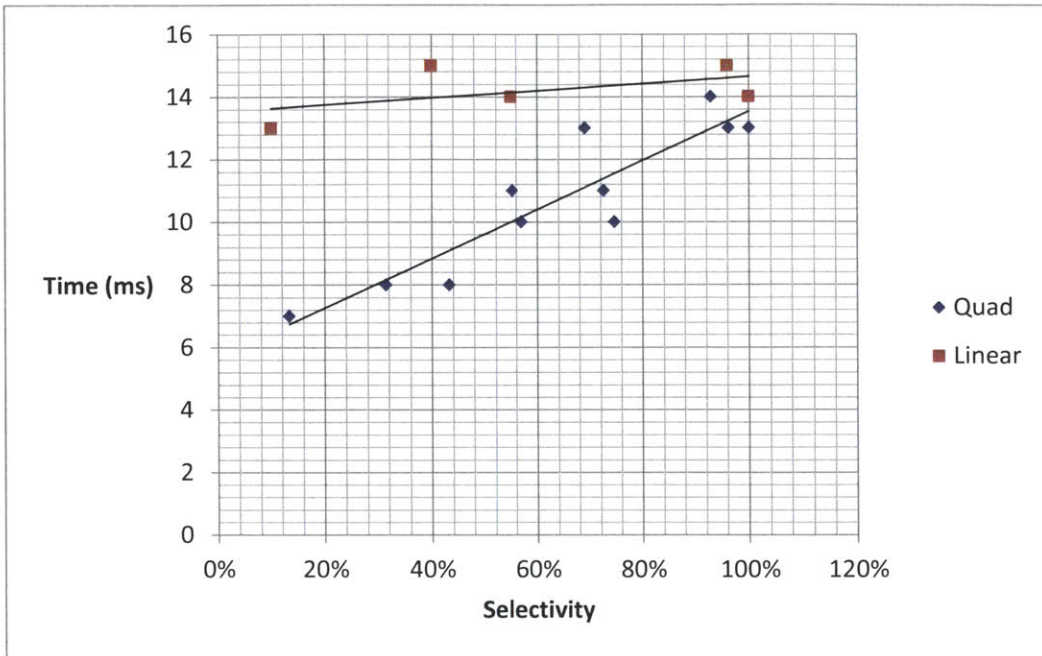


Figure 5.3-1 QuadTree vs. LinearScheme on GPU – Small Data

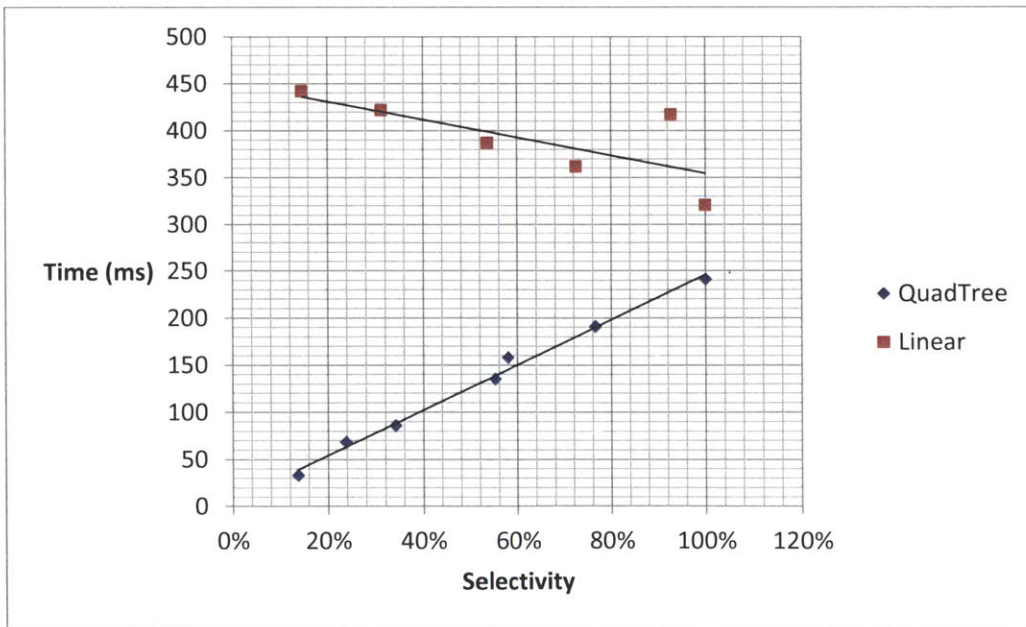


Figure 5.3-2 QuadTree vs. LinearScheme on CPU – Small Data

We see how the QuadTree partitioning provides us with lower query latency with less selectivity. This is mainly because we need to scan less number of fragments, whereas the LinearScheme will always scan all fragments in the table. We find it interesting that the

QuadTree achieves slightly lower latency even for 100% selectivity and this is because of increase in cache hit rate when writing to the histogram. When scanning a fragment in a QuadTree, we are more likely to write to pixels that are close by in the histogram due to the geo-spatial partitioning the QuadTree does, and this increases the probability that the memory pages holding these pixels are in cache.

Notice how the LinearScheme shows almost constant performance on the GPU for variable selectivity. The slightly positive slope is mainly due the increase in GPU writes to the resulting histogram. While theoretically this should be the same on the CPU, it shows a slightly negative slope there mainly because of the CPU caching which we cannot control with such experiment.

We now run the same experiment on a large data set of 200 million tweets. The following table lists all of the constant parameters of the following experiment.

Table 5.3-2 QuadTree experiment parameters – large dataset

Data Size	200×10^6
Maximum Fragment Size	35×10^6
GPU buffer pool size	10.2 GB
CPU buffer pool size	16 GB
GPU bandwidth	280 GB / sec
Width × Height (pixels)	1366 * 768

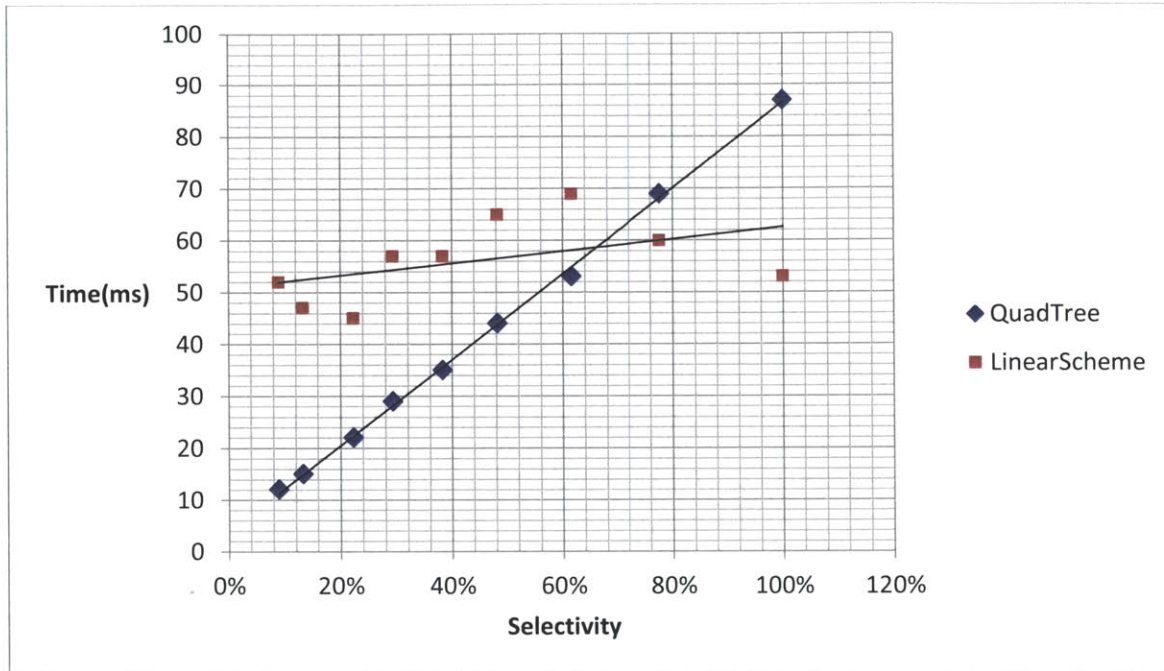


Figure 5.3-3 QuadTree vs. LinearScheme on GPU – Large Dataset

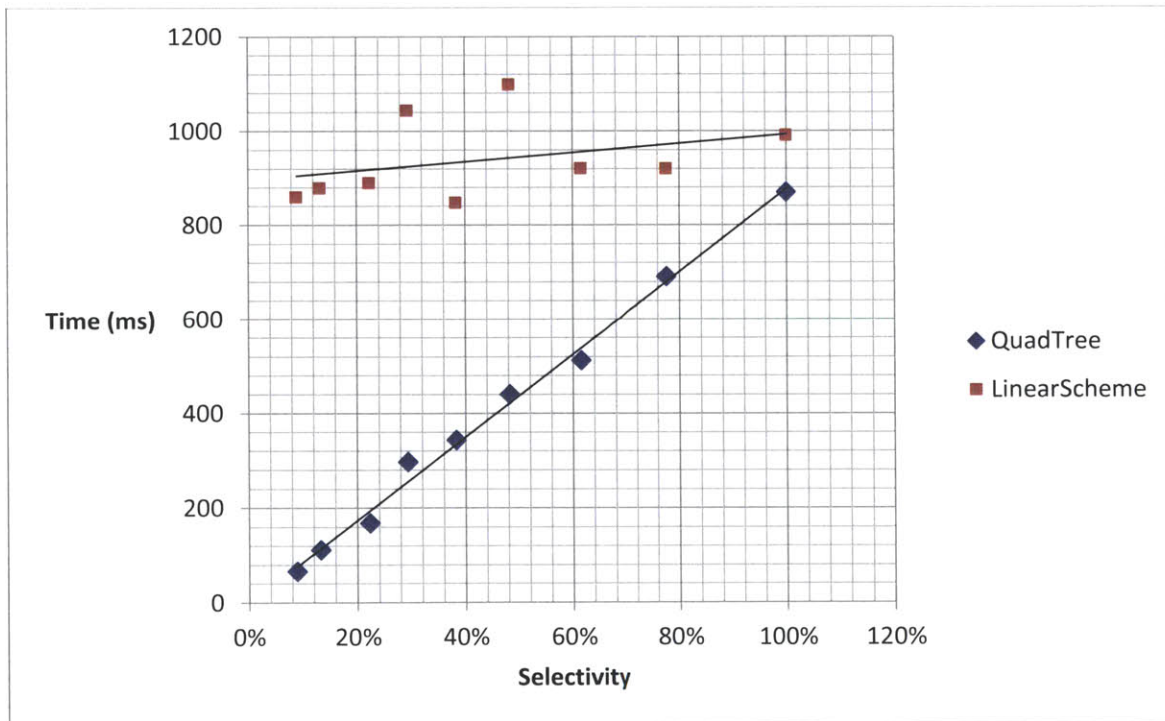


Figure 5.3-4 QuadTree vs. LinearScheme on CPU - Large Dataset

With the large dataset, the CPU result is more like what we expect. The LinearScheme graph with a low positive slope, and the QuadTree graph showing latency decreasing linearly with selectivity. For the GPU, however, at a selectivity of a little less than 70% and upwards, the LinearScheme performs better. The reason is due to the unequal number of fragments for both schemes. The QuadTree at this point has many more fragments needed for scan than the LinearScheme causing it to perform worse.

5.4 Shared Scan Evaluation and Analysis

The following experiments aim to evaluate the performance of the shared scan and the EDF scheduling algorithm. First, however, we need to define the performance metrics for shared scan.

The performance metrics we use for this project are *Query Throughput*, *QMD* (Queries Miss Deadline), and *Average Lateness*.

1. **Query Throughput:** (Queries / second) number of queries per second.
2. **QMD:** percentage of queries that missed their deadline.
3. **Average Lateness (L):** The average lateness value over all queries. Lateness for query Q_j is defined as the difference between the task absolute completion time (C_j) and the absolute deadline (D_j). A positive value of lateness indicates that the query has missed its deadline. For an experiment with N queries, the following is the definition of average lateness.

Equation 5.4-1 Average Lateness for N Queries

$$L = \frac{\sum_{i=0}^N (C_i - D_i)}{N}$$

5.4.1 Overlap

In this experiment, we investigate the effect of the degree of overlap among the queries on the performance of shared scans. We define *Overlap* as the number of fragments shared among *all* queries divided by the total number of fragments in the table.

The constant parameters for the following experiment are as follows:

Table 5.4-1 Overlap Experiment Parameters

Data Size	200×10^6
Maximum Fragment Size	35×10^6
Number of queries (N)	1000
Deadline (fixed)	100 ms
Output Buffer Pool Size	380 MB
Width \times Height (pixels)	1366 * 768

The table below demonstrates that the shared scan, in this particular case, provided an average of 300x speedup over serial execution. We don't get a 1000x speedup (number of queries) because the output buffer of 380MB will only allow 90 queries at a time.

Table 5.4-2 Shared Scan vs. Serial Execution Performance

Overlap	GPU time – Serial Execution (ms)	GPU time – Shared Execution (ms)	Approximate (Serial Exec. / Shared Exec.) Ratio
100%	86000	225	382
84.60%	77500	260	298
61%	69500	250	278
38%	60000	198	303
15%	54500	186	293

Next, we have two charts, one showing *QMD* vs. *Overlap* while the other demonstrating *Average Lateness* vs. *Overlap*. *QMD* starts out about constant with high overlap and then starts to fall down. While we define a constant deadline for queries of 100ms, our implementation when choosing a query will pick a fragment that is most shared among other queries, thus queries with less number of fragments to scan will finish earlier. As the overlap decreases, the number of queries with less number of fragments increases. This is due to our definition of *Overlap*.

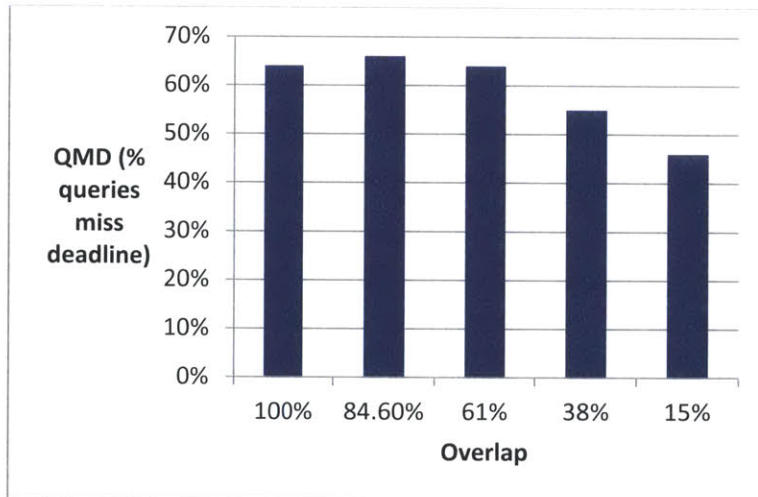


Figure 5.4-1 QMD vs. Overlap

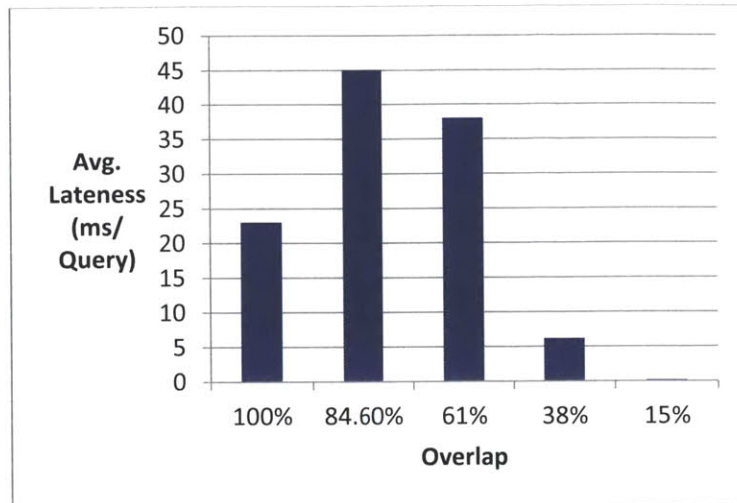


Figure 5.4-2 Avg. Lateness vs. Overlap

5.4.2 Deadlines

In this section, we investigate the effect of varying the deadlines values on the performance of shared scans.

5.4.2.1 Fixed

This section investigates the shared scan performance for different fixed deadline values. By fixed deadline, we mean that the value of relative deadline is some constant d regardless of query type or any attribute of **QueryInfo** described earlier. The following experiment parameters are found in the table below.

Table 5.4-3 Deadline Experiments Parameters

Data Size	200×10^6
Maximum Fragment Size	35×10^6
Number of queries (N)	3600
Output Buffer Pool Size	512 MB

Width × Height (pixels)	1366 * 768
Query Bounds	Represent 12 unique rectangular bounds in space uniformly distributed among queries.

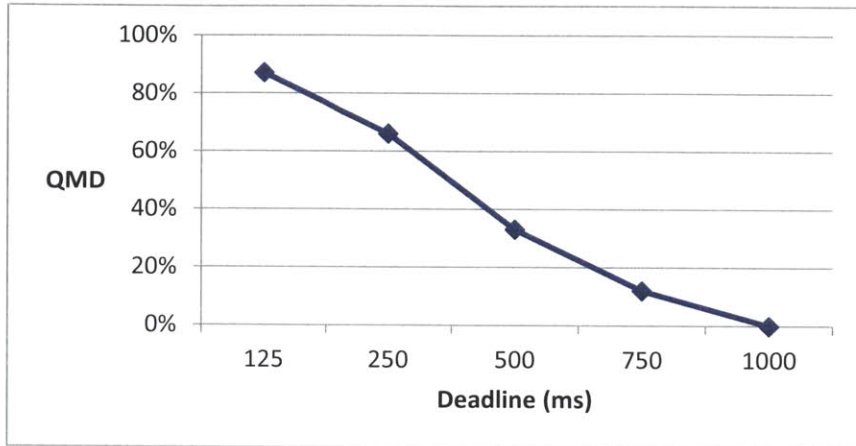


Figure 5.4-3 QMD vs. Deadline

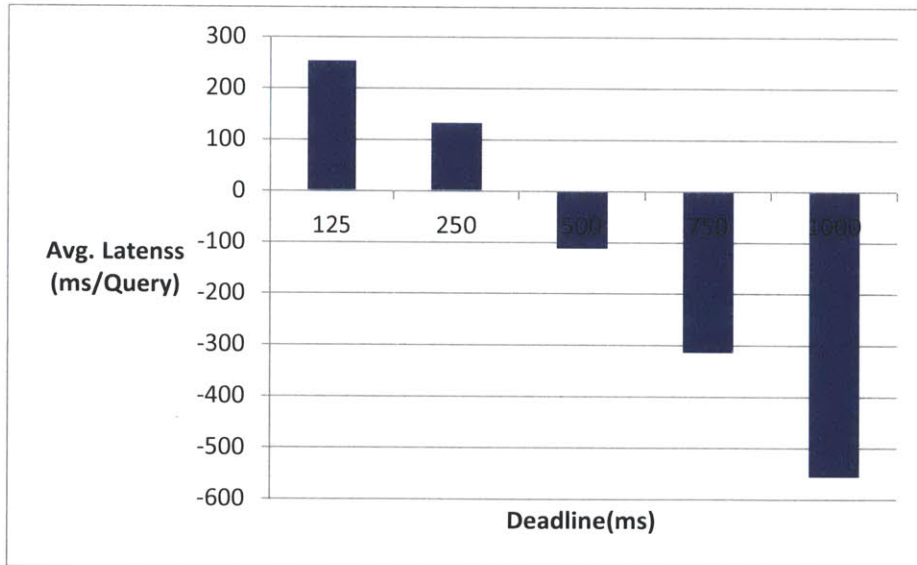


Figure 5.4-4 Avg. Lateness vs. Deadline

Figure 5.4-3 shows almost a linear relationship between *QMD* and *deadline* values. As the fixed deadline set for the queries increases, the *QMD* or the percentage of queries that miss the deadline drops. Figure 5.4-4 shows the *Average Lateness* plot against *deadline*. At a deadline of 500 ms we start to see that on average, queries are being early rather than late.

5.4.2.2 Dependent on Number of Fragments

For skewed deadlines depending on the fragments, we leave it for the last section for evaluating EDF algorithm.

5.4.3 Number of Queries

In the following experiment, we investigate the effect of the number of concurrent queries N on the performance metrics of our shared scan implementation.

Table 5.4-4 Parameters for Number of Queries Experiments

Data Size	200×10^6
Maximum Fragment Size	35×10^6
Output Buffer Pool Size	256 MB
Deadline	60 ms
Width \times Height (pixels)	1366 * 768
Query Bounds	All queries are rectangular bounds of the whole world map – 100% Overlap.

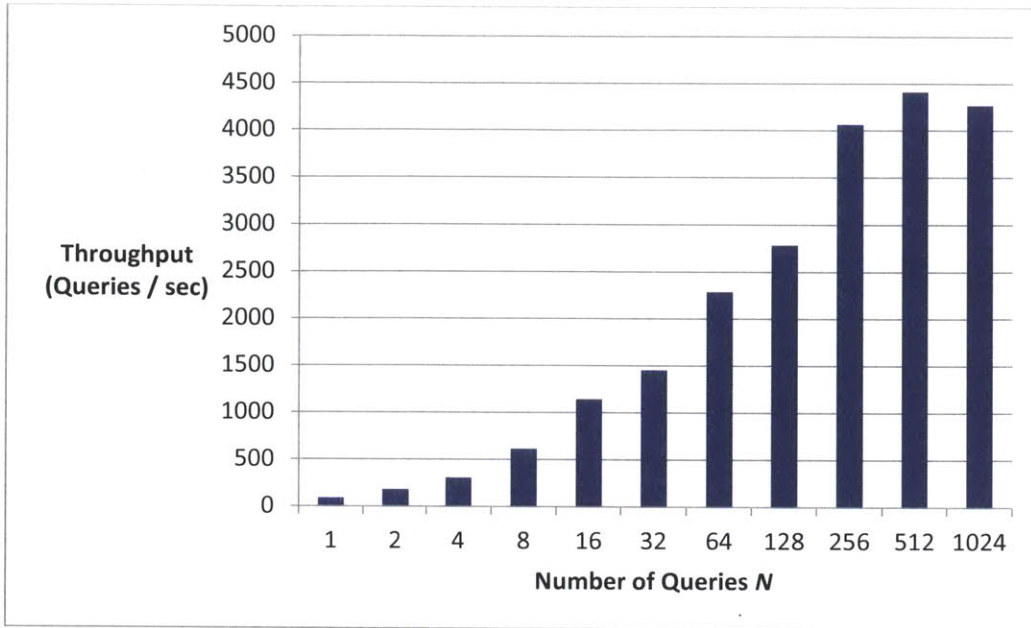


Figure 5.4-5 Throughput vs. Number of Queries

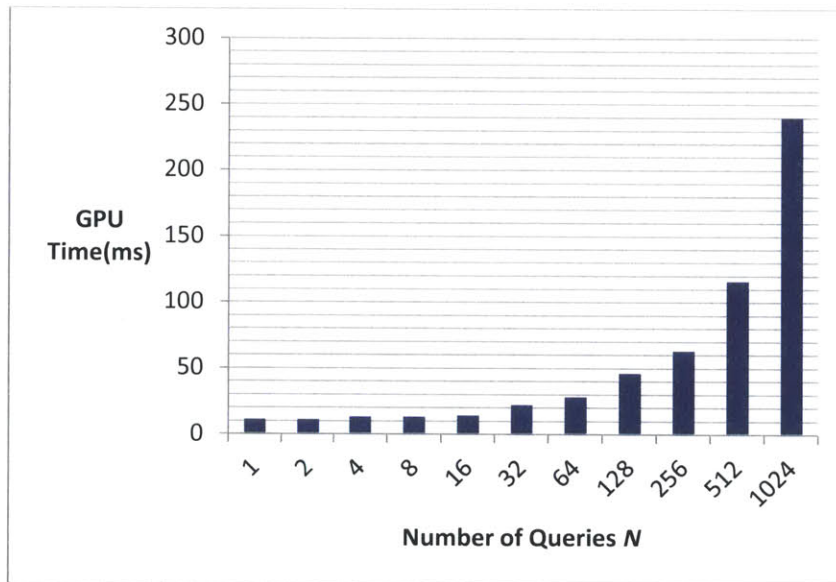


Figure 5.4-6 Time vs. Number of Queries

Figure 5.4-5 and Figure 5.4-6 show us how we are able to gain throughput with increasing number of concurrent queries. Yet, for more than 512 queries, the throughput starts to decline due to the limited output buffer size and overhead of sharing many queries where the cost of writing the resulting histograms dominates the overall cost.

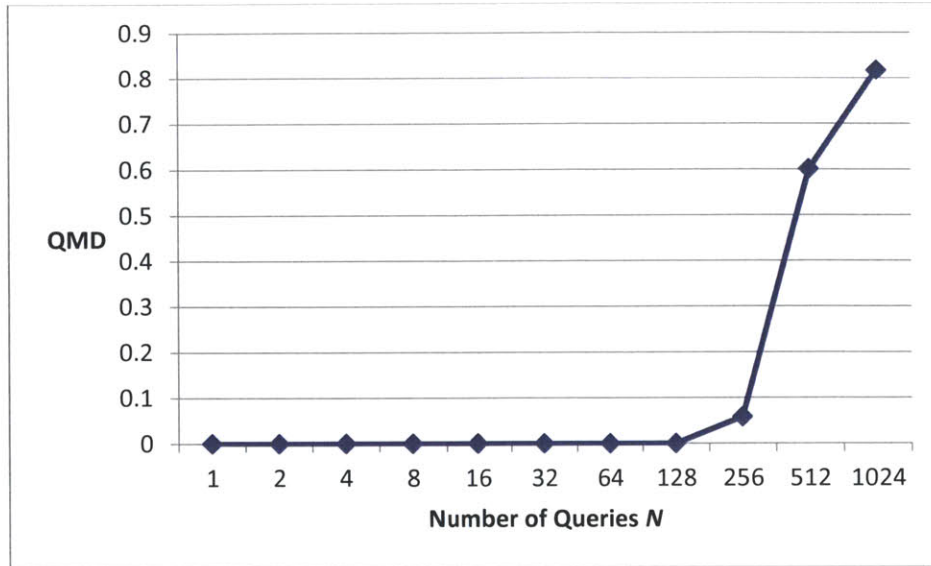


Figure 5.4-7 QMD vs. Number of Queries

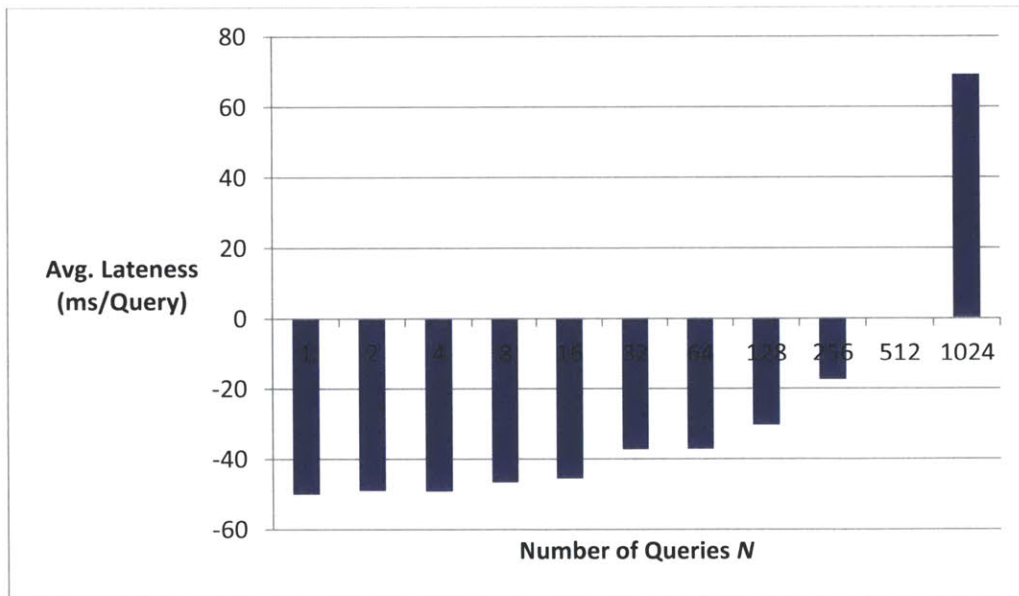


Figure 5.4-8 Avg. Lateness vs. Number of Queries

It was interesting to see that with the choice of deadline (60 ms), we only started to miss deadlines when the number of queries was about 256. Yet, the average lateness became positive only near 512 concurrent queries. (See Figure 5.4-7 and Figure 5.4-8).

5.4.4 Output Buffer Size

This experiment varies the maximum size of the output buffer used, or `MAX_OUTPUT_BUFFER_SIZE` introduced in the previous chapter, to save the answers of the queries. We expect better performance with more output buffer size since we are able to keep more queries in the priority queue.

Table 5.4-5 Parameters for Output Buffer Size Experiment

Data Size	200×10^6
Maximum Fragment Size	35×10^6
Number of Queries N	600
Deadline	<i>min (number of fragments \times 10, 70)</i>
Width \times Height (pixels)	1366 * 768
Query Bounds	Queries have 12 unique rectangular bounds in space.

Figure 5.4-9 shows the increase in query throughput with increasing the maximum output buffer size. Figure 5.4-10 plots the QMD vs. output buffer size and shows a best fit line. This clearly shows how percentage of queries that miss the deadline is linearly related with output buffer size. Finally, Figure 5.4-11 plots average lateness against output buffer pool. A similar linear relationship is observed.

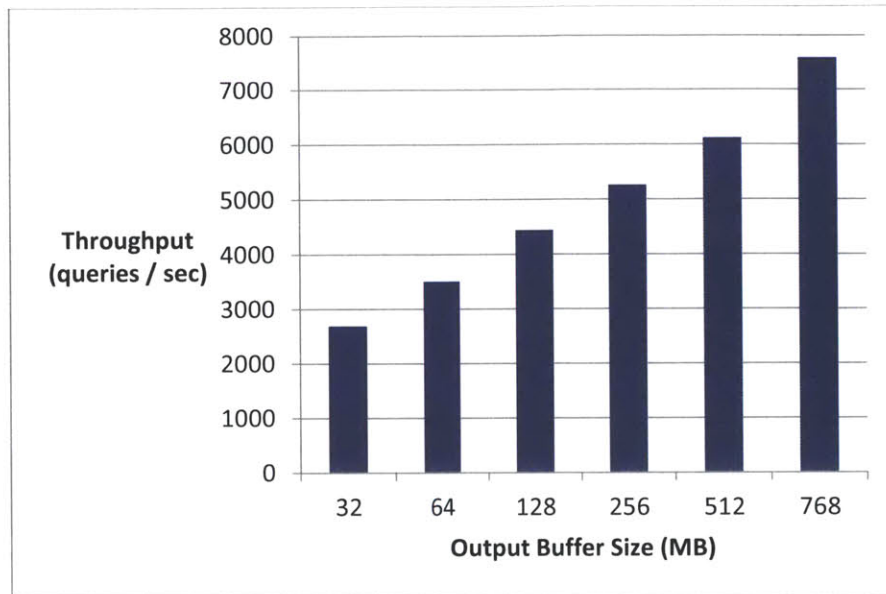


Figure 5.4-9 Throughput vs. Output Buffer Size

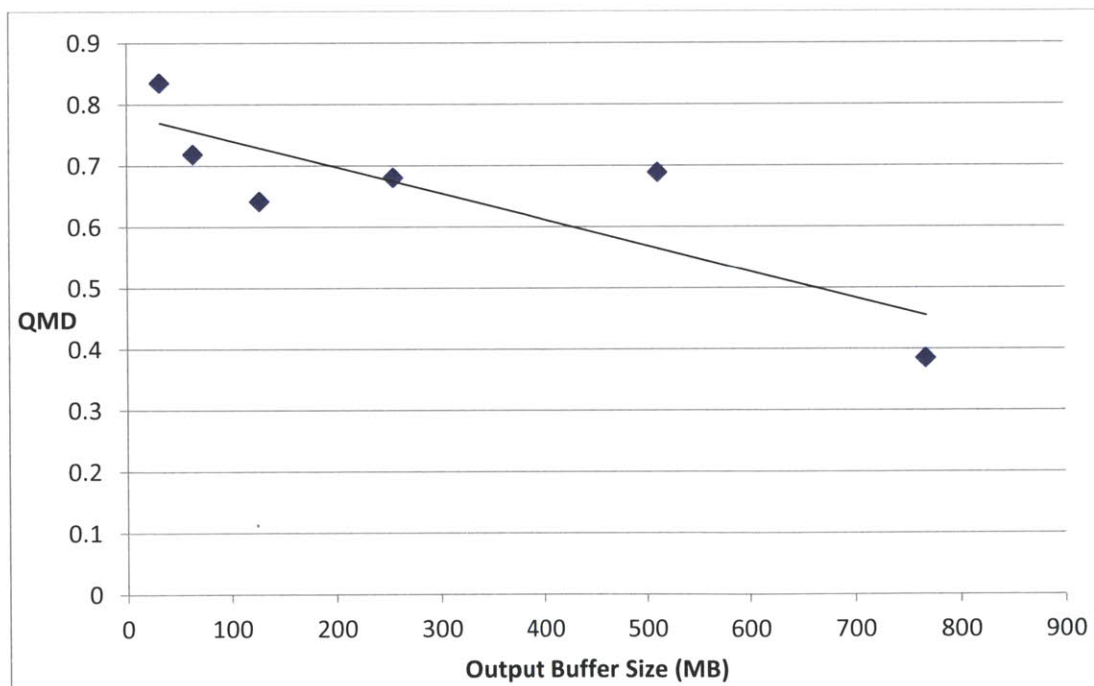


Figure 5.4-10 QMD vs. Output Buffer Size

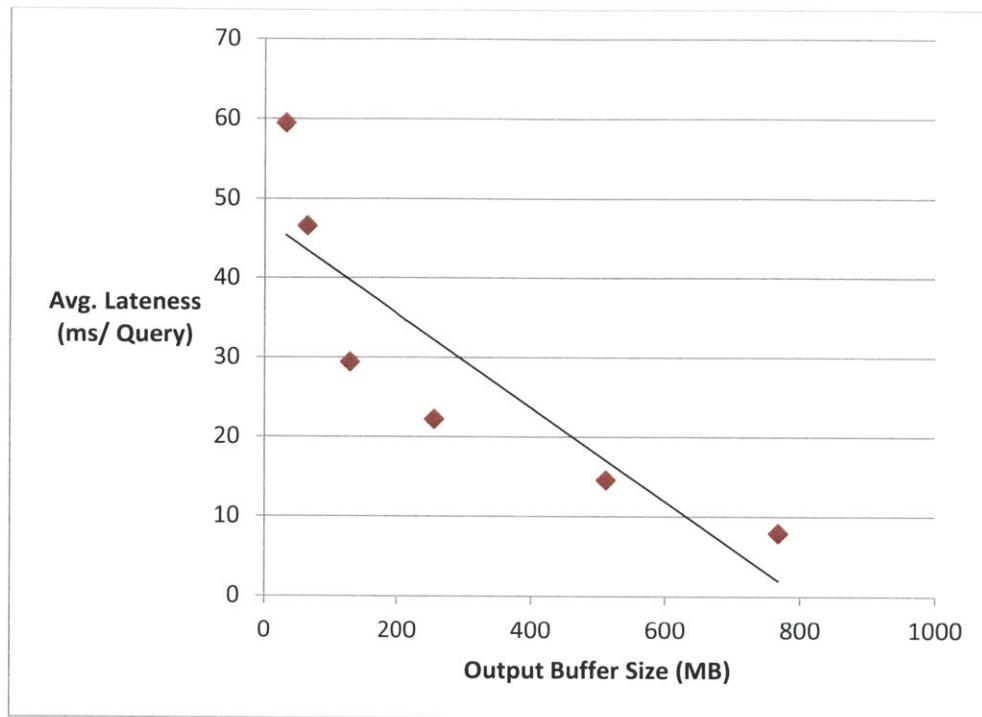


Figure 5.4-11 Average Lateness vs. Output Buffer Size

5.4.5 Scheduling (EDF vs. FIFO)

In this section, we evaluate the scheduling aspect of our shared scan implementation. By scheduling we mean the EDF algorithm for the choice of the next query to scan. We compare EDF to FIFO (queue order: first in, first out) scheduling. In both cases, however, after a query is picked, the fragment chosen to scan is *still* the one shared amongst the highest number of queries in the priority queue.

A. Small-Sized Output Buffer

We start by experimenting with a small value for the maximum output buffer size. This means that the priority queue will not be able to hold all the queries at the same time and perform scans to answer all of them.

Table 5.4-6 Parameters for EDF vs. FIFO Experiment - Small Buffer

Data Size	200×10^6
Maximum Fragment Size	35×10^6
Output Buffer Size	512 MB
Deadline(ms)	<i>min (number of fragments \times 10, 100)</i>
Width \times Height (pixels)	1366 * 768
Query Bounds	Queries have 12 unique rectangular bounds in space, uniformly distributed among queries.

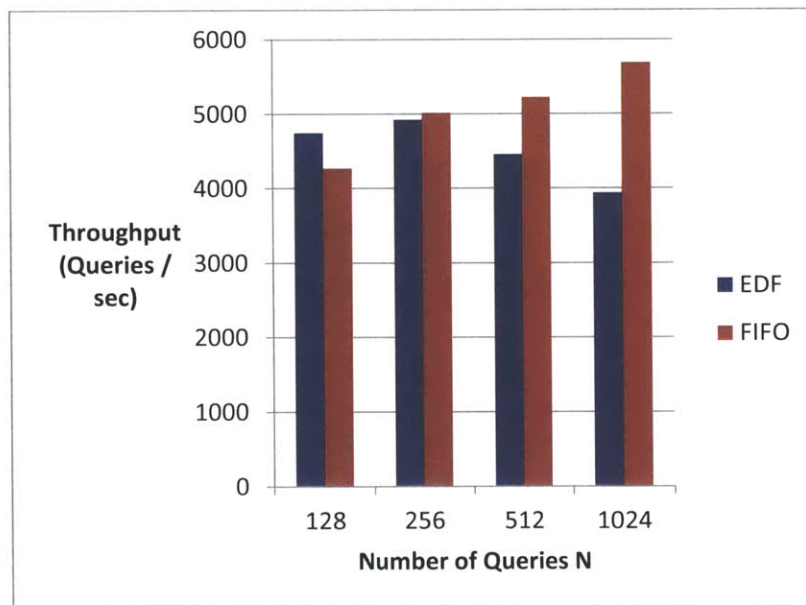


Figure 5.4-12 EDF-FIFO. Throughput vs. N –Small Buffer

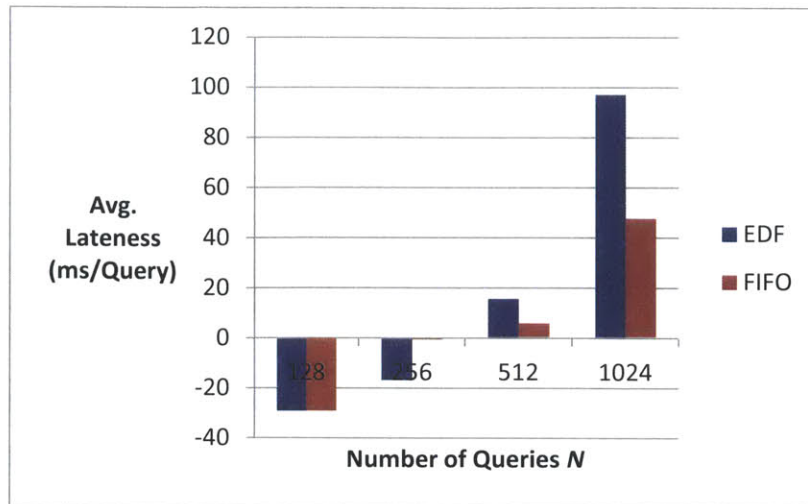


Figure 5.4-13 EDF-FIFO. Avg. Lateness vs. N - Small Buffer

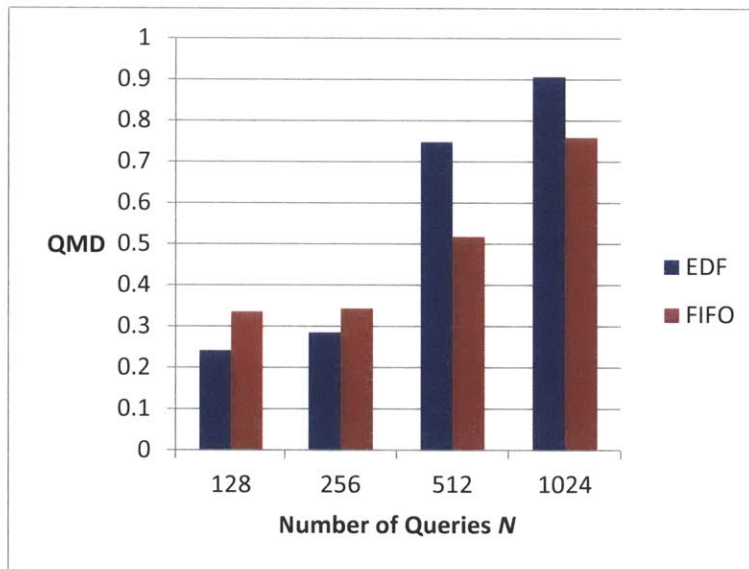


Figure 5.4-14 EDF-FIFO. QMD vs. N - Small Buffer

Figure 5.4-12 shows how EDF starts performing better than FIFO while it performs worse with higher number of queries. This is mainly because of output buffer limit and the invariant we maintain. Only the queries in the priority queue, which already have been allocated space for their output, are valid choices for EDF algorithm. This implies that the waiting queue holds queries with earlier deadlines that have to wait for enough space to be freed when other queries

finish. We see this worse performance of EDF vs. FIFO in the other figures too (Figure 5.4-13, and 5.4-14).

B. Large-Size Output Buffer

For this experiment, we alter the parameters such that all queries end up being able to be present in the priority queue. In other words, all queries will have output buffer space allocated.

Table 5.4-7 Parameters for EDF vs. FIFO Experiment - Large Buffer

Data Size	200×10^6
Maximum Fragment Size	35×10^6
Output Buffer Size	512 MB
Deadline (ms)	<i>min (number of fragments \times 10, 100)</i>
Width \times Height (pixels)	1366 * 768
Query Bounds	Queries have 12 unique rectangular bounds in space. Equally represented in each set.

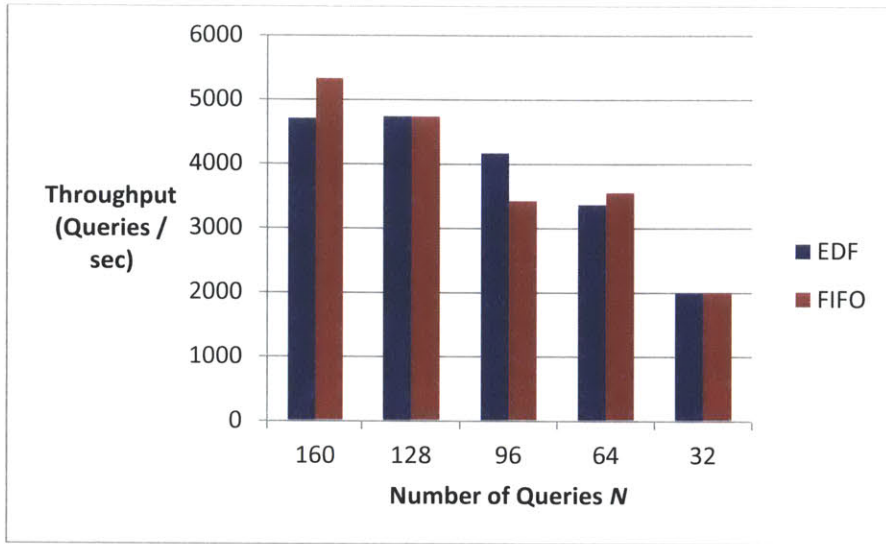


Figure 5.4-15 EDF-FIFO. Throughput vs. N - Large Buffer

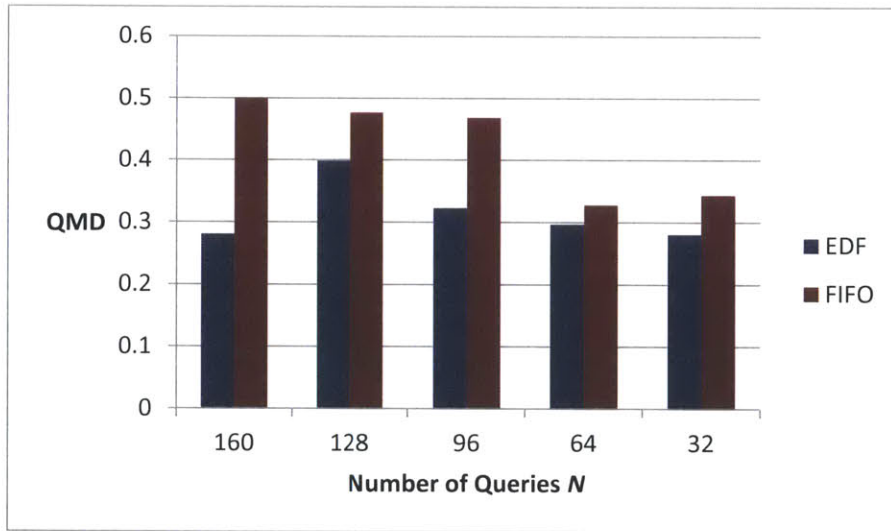


Figure 5.4-16 EDF-FIFO. QMD vs. N - Large Buffer

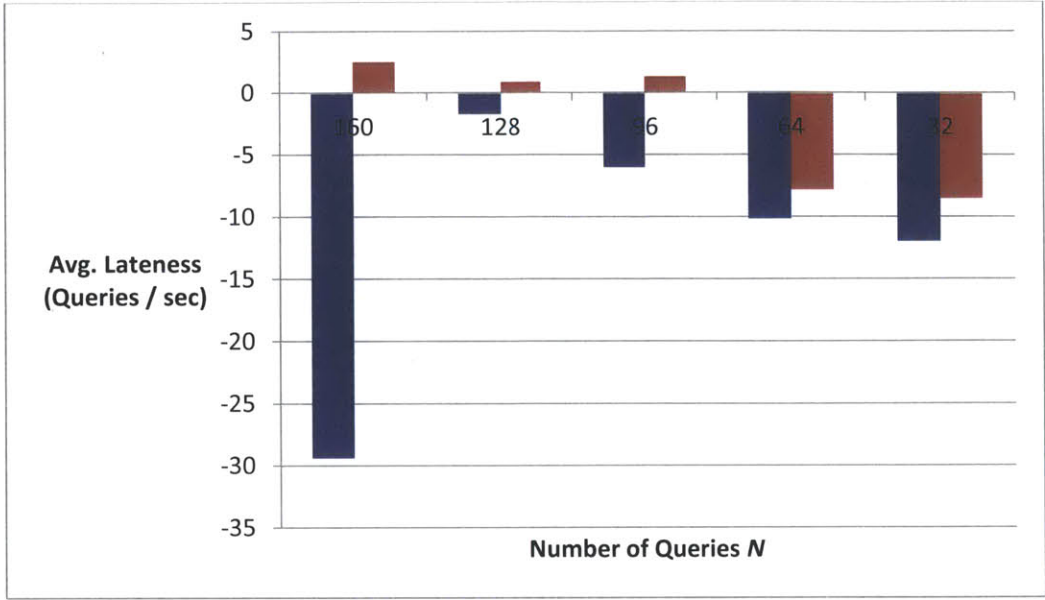


Figure 5.4-17 EDF-FIFO. Avg. Lateness vs. N - Large Buffer

While the throughput plot shows EDF and FIFO perform about the same, we see the difference in QMD and Avg. Lateness (Figure 5.4-16 and Figure 5.4-17). In both of these cases EDF performs better than FIFO. This makes sense since EDF chooses the query that is closest to its deadline.

C. Skewed Deadlines

We now experiment with varying deadlines and comparing EDF's to FIFO's performance. We start with the experiment with the following fixed parameters.

Table 5.4-8 Parameters for Skewed Deadlines Experiment

Data Size	200×10^6
Maximum Fragment Size	35×10^6
Output Buffer Size	768 MB
Number of Queries (N)	150
Width \times Height (pixels)	1366 * 768
Query Bounds	Queries have 12 unique rectangular bounds in space that are equally represented.

We start by experimenting with the following deadlines: (F is the number of fragments needed for scan by the query)

1. $\min(F \times 20, 200)$
2. $\min(F \times 15, 100)$
3. $\min(F \times 10, 50)$
4. $\min(F \times 5, 40)$

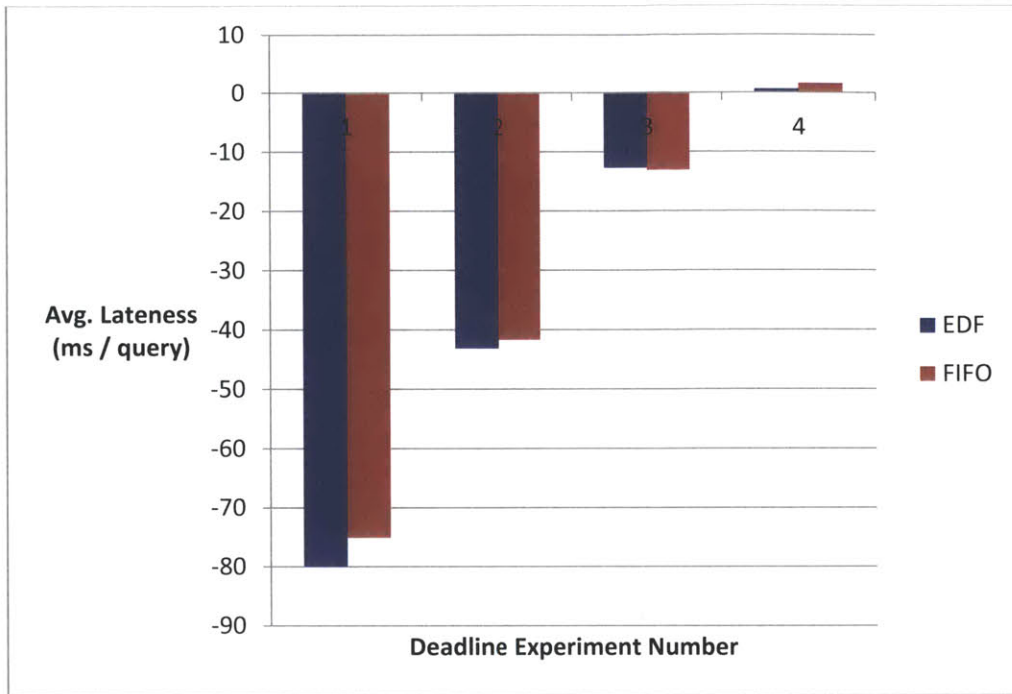


Figure 5.4-18 Skewed Deadlines. Avg. Lateness vs. Deadline

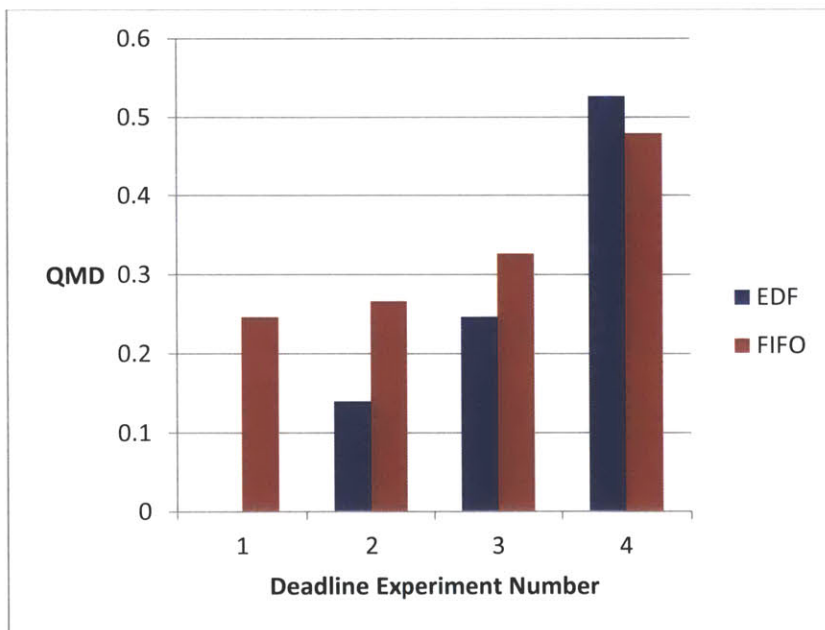


Figure 5.4-19 Skewed Deadlines QMD vs. Deadline

Figure 5.4-18 and Figure 5.4-19 show how EDF performs better than FIFO. In particular, the QMD values are much higher for FIFO since FIFO does not take query deadline into account. Now we perform a similar experiment on the following deadline values (in ms - F is the number of fragments the query needs to scan).

1. $F \times 20$
2. $F \times 15$
3. $F \times 10$
4. $F \times 5$

The results are shown in Figure 5.4-20 and Figure 5.4-21. A similar performs win for EDF is observed in these two figures. This gives us confidence that EDF is the right approach, but the limited output buffer size remains a challenge.

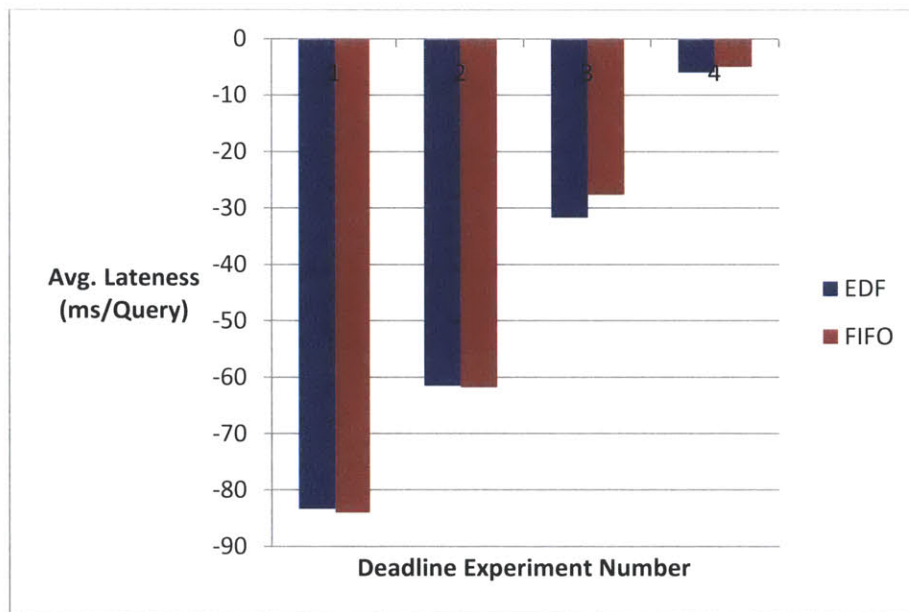


Figure 5.4-20 Skewed Deadlines. Avg. Lateness vs. Deadline

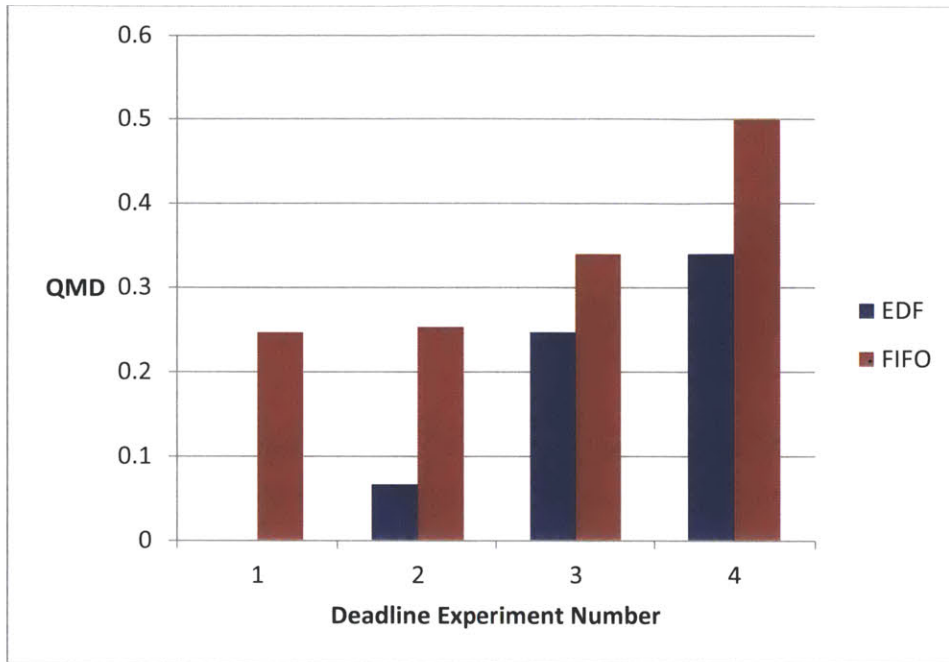


Figure 5.4-21 Skewed Deadlines. QMD vs. Deadline

Chapter 6 - Related Work

In this project, we tackle two main challenges in implementing a database system architecture optimized for low latency analytics. The first is the managing and indexing of the data in hybrid CPU-GPU architecture. The second is the scheduling of a large number of concurrent queries by means of a shared concurrent scan.

6.1 Partitioning Schemes and QuadTree Indexes

Using tree based filters/indexes for spatial data is not a novel idea. Kothuri et al [15] describe and compare the use of QuadTree and R-tree filters or indexes for Oracle Spatial. Furthermore in [15], they implement an indexing framework which allows for the implementation of domain-specific indexes. This is similar to what we do with regards to the **Partition Scheme** described earlier. Our **Partition Scheme** acts as an indexing framework allowing the implementation of spatial or non-spatial domain specific indexes.

Samet et al in [16] and [11] describe a fast IO-bound algorithm for construction of QuadTree even if main memory size is less than that of the resulting tree. They use z-ordering of the spatial points to minimize the I/O operations in the QuadTree construction. Since new computer architectures are now based on multi-core devices with more RAM, the bottleneck is now the memory bandwidth, replacing the IO bottleneck. This is why our QuadTree construction occurs in main memory with the assumption that main memory can hold all the points in the QuadTree. With hardware getting cheaper, and considering that our design is for GPU-based systems with limited amounts of RAM, this is a safe assumption to make.

Other related work to the QuadTree indexing is found here [1] where a hybrid CPU-GPU parallel sorting-based algorithm is used to increase the speed of the construction of the QuadTree index.

6.2 Buffer Pool Partitioning and Replacement Policy

With regards to the implementation of the memory buffer pool, there has been a great deal of related work on managing memory in database systems. In [17], IBM demonstrated the implementation of buffer pool partitioned based on page size. This is similar to the buffer pool partitioning we implement except that we handle multi-sized pages. DeWitt et. al [18] compare various eviction policies on different workloads. They mainly present DBMIN, an algorithm to manage the buffer pool for relational database systems by predicting future reference behavior. We on the other hand focus on implementing a general purpose buffer pool which is extendible with eviction policies proven to work well with different workloads according to the database literature (LRU-2 / LRU-K [19] [20], LRU-MRU or LRU2-MRU [21]).

6.3 Shared Scans and Scheduling

There hasn't been much prior work addressing the specific problem of scheduling shared scans for concurrent queries arriving at a database server. Yet, the research work by Agrawal et. al [22] is very similar to what we implement. They study how to schedule tasks that can share scans over a common set of large input files. The main goal is to amortize the expensive file scans across as many jobs as possible. They do so by implementing a simple stochastic model of job arrivals from each input file. Their system takes into account anticipated future jobs when attempting to schedule the jobs in the input queue. Stochastic modelling of job arrivals for MapD2 may be promising for the future, yet we focused on a simple Earliest Deadline First (EDF) implementation. They do not estimate deadlines for tasks but define a PWT (average perceived waiting time) performance metric which is the average of difference between the system's response time in handling the task and the minimum possible response time. Their work shows clearly that shortest-job-first (SJF) scheduling policy doesn't perform well in the presence of opportunity for job sharing.

On a different line of work, Qiao et. al [23] address the bottleneck of memory bandwidth of CPU-bound scans by implementing a novel scan sharing technique for main memory query processing. They attach many queries to the same scan, amortizing the overhead of data scanning

from main memory into the CPU's cache. By exploiting data skew and statistical metrics, they group queries in a way to avoid thrashing. While they claim that in main memory DBMS, the cache takes over the buffer pool, this is not quite the case when GPUs are involved. Shared memory and constant memory on GPUs (equivalents to cache memory on CPU) are orders of magnitude smaller than CPU caches. Therefore, our shared scan implementation focuses on global GPU memory. Furthermore, with GPU memory bandwidth being orders of magnitude higher than that of CPU, it does not represent a bottleneck at the moment. In the near future though, GPU cards will have greater cache sizes allowing for optimization opportunities similar to the work done by Qiao et al.

We are not aware of any prior work that addresses scheduling queries for shared scan such that a limited memory is allocated for output memory buffers for the queries and where deadlines are estimated for use in the scheduling policy. In particular our work is unique in terms of defining a software pipeline and architecture that is extendible for online dynamic scheduling of concurrent queries.

Chapter 7 - Future Work

There are many opportunities for interesting projects aimed at systems that are optimized for low-latency, like MapD. The most interesting future projects, however, is that related to improving the shared scans performance and the EDF scheduling algorithm.

7.1 Partition Scheme Architecture

Future research may focus on the **PartitionScheme** architecture. One can compare the performance of different indexes on the same table and come up with a strategy to choose which index to use based the client's query. Allowing an automated choice of a partitioning scheme / index to access when a query arrives helps improve the performance of shared scans as well.

With regards to the QuadTree implementation, an interesting future work would extend the QuadTree with Bloom Filters to allow for pruning on text. This provides opportunities for shared scans of queries involving filters on both text and space.

At the moment, each QuadTree index allows for one maximum fragment size at which the split occurs. If we relax this condition and allow the QuadTree to split based on variable maximum fragment sizes, by maintaining statistics of the current data, we may end up with more balanced fragments. This can lead to better estimation of deadlines for many concurrent queries, and thus an improved shared scans throughput.

An interesting future research can evaluate partitioning the spatial data across multiple nodes, each with multiple GPUs. One approach is by delegating to each node a spatial partition (e.g. NW, NE, SW, SE for 4-node system), where each node has its' own QuadTree index. This can benefit shared scans of skewed queries as we can determine the node(s) to send the query to. However, this approach does not balance the load of data. Realistic spatial data sets are not distributed uniformly in space, and therefore, a round robin or hash partitioning across many nodes may work better. It will be interesting to evaluate shared scans with these two approaches

and evaluate the overhead of combining the result from multiple nodes to obtain the overall query result.

7.2 Buffer Pool Manager

The buffer pool design provides opportunities for future work as well. A reasonable optimization is the implementation of a functionality that, when the system is idle, looks for free/ un-used non-contiguous pages and combines them into one. This will increase the likelihood of finding free pages of desired size or greater, thus avoiding eviction in many cases.

Moreover, future work will evaluate different combinations of replacement policies and research the possibilities of automating the choice of policy depending on statistics based on history referencing of memory chunks and the history of query workload. We know that a better eviction policy implies better cache hit rate.

7.3 Shared Scans and EDF

A good optimization to the current shared scans implementation involves sharing output buffers. This includes using less space to compute the output of more queries. For example, we can use *xor* optimizations where outputs of queries are *xor*-ed to save output space. Furthermore, sparse array representation of the histogram result can reduce the output space per query, and this is due to the fact that the output histogram can be a sparse matrix in many cases (many values are zero). Reducing the size of the output buffer space per query is definitely a prioritized future work. We've seen in Chapter 5 how the limited output buffer caused the EDF implementation to perform poorly in terms of average query lateness.

Another interesting future work relates to devising policies to optimize for the percentage of queries missing the deadline (*QMD*) and average lateness when sharing scans of many concurrent queries. This includes formulating a cost model as a function of query parameters and other factors (e.g buffer pool partitions, memory size, maximum fragment size, ..etc). We

believe this will greatly help in optimizing shared scans and achieving a higher quality of service.

Finally, interesting research involves the prediction of future queries based on historical data of previous requests. This can benefit animated queries in particular. If we predict that the client will perform an animation on some table, we can insert the queries that produce future frames, with their future deadlines, into the priority queue and thus be able to have the frames ready when the client needs them.

Chapter 8 - Conclusion

In this thesis, we tackled two key challenges that arise when implementing software architecture optimized for low latency analytics. The first is the managing and the partitioning of data sets across multiple nodes with multiple CPUs and GPUs. The second is the processing of many queries (perhaps issued concurrently) where queries have very fast latency constraints, e.g, for visualization. We implemented an indexing framework or a partition scheme which enables scanning the relevant partitions of the data only as opposed to the whole dataset. We also implemented a three-level memory pool hierarchy to serve as a database cache with support of various replacement policies. Finally, we implement an algorithm “EDF Shared Scan” which allows for scheduling concurrent client query requests to share processing of many queries in a single pass through the data.

We found that the QuadTree partitioning strategy provides lower latency for queries with lower selectivity. Yet, we perceived that with large datasets, at a high selectivity rate, the cost of scanning more fragments due to the QuadTree leaves subdivision overcomes that of scanning less number of fragments. As discussed in the previous section, implementing a new QuadTree that splits on various maximum fragment sizes will result in a more balanced partitioning. This can speed up the queries with high selectivity on large datasets partitioned by a QuadTree. Another approach may be simply implementing another spatial index like an R-Tree.

Furthermore, our results and experiments section shows how shared scans improve throughput (queries / second). Furthermore, we saw that when the maximum size of the output buffer is enough to process all concurrent queries, our EDF algorithm, choosing the query with earliest deadline from the priority queue, performs very well as opposed to FIFO implementation. Additionally, we observed how EDF achieves less QMD rate and less average lateness than FIFO for queries with skewed relative deadlines. While a limited output buffer size caused EDF to perform poorly compared to FIFO, we are sure that reducing the output buffer space needed per query, as discussed in the previous section will improve EDF performance and decrease the rate at which queries miss their deadline.

Bibliography

- [1] M. Kelly and A. Breslow, "Quadtree Construction on the GPU: A Hybrid CPU-GPU Approach," Swarthmore College, 2011.
- [2] P. B. e. a. Volk, "GPU-Based Speculative Query Processing for Database Operations," September 2010.
- [3] A. F. S. J. A. U. G. P. Units, "Lieberman, Michael D. et al.," University of Maryland, 2008.
- [4] A. K. Sujeeth, H. Lee and K. J. e. a. Brown, "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning," Stanford University, 353 Serra St., Stanford, CA 94305 USA, 2011.
- [5] T. Mostak, "An Overview of MapD (Massively Parallel Database)," CSAIL, MIT, 2012.
- [6] H. Samet, Applications of spatial data structures: Computer graphics, image processing, and GIS, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 1990.
- [7] H. Samet, The design and analysis of spatial data structures., Reading, Mass., USA: Addison-Wesley, 1990.
- [8] M. Stonebraker, "Operating System Support for Database Management," *Computing Practices*, vol. 24, no. 7, pp. 412-414, 1976.
- [9] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," *Algorithmica*, vol. 1, no. 1-4, pp. 311- 336, 1986.
- [10] A. Mohammadi and S. G. Akl, "Scheduling Algorithms for Real-Time Systems," Queen's University School of Computing, Kingston, Ontario, July 15, 2005.
- [11] H. Samet and G. Hjaltason, "Speeding up construction of PMR quadtree-based spatial indexes," *The VLDB Journal*, vol. 11, no. 2, pp. 109-137, 2002.
- [12] J. M. Hellerstein and M. Stonebraker, "Architecture of a Database System," *Foundations and Trends in Databases*, vol. 1, no. 2, pp. 141-259, 2007.
- [13] J. Erickson, G. Coombe and J. Anderson, "Soft Real-Time Scheduling in Google Earth," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE*

18th, April 2012.

- [14] R. Pelanek, "Aperiodic Task Scheduling," Masaryk University, [Online]. Available: <http://www.fi.muni.cz/~xpelanek/IA158/slides/aperiodic.pdf>. [Accessed 3 3 2014].
- [15] R. K. V. Kothuri, S. Ravada and D. Abugov, "Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data," in *SIGMOD '02 Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, New York, 2002.
- [16] G. R. Hjaltason and H. Samet, "Improved Bulk-Loading Algorithms for QuadTrees," in *Proc. of the 7th Intl. Symposium on GIS (ACM GIS '99)*, Kansas City, MO, 1999.
- [17] IBM, "IBM Knowledge Center," IBM, [Online]. Available: http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=%2Fcom.ibm.db2.z10.doc.intro%2Fsrc%2Ftpc%2Fdb2z_bufferpoolsanddatacaching.htm. [Accessed 1 2 2014].
- [18] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," *Algorithmica*, vol. 1, pp. 311-336, 1986.
- [19] E. J. O'Neil, P. E. O'Neil and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," pp. 297--306, 1993.
- [20] J. Boyar, M. Ehmsen, J. S. Kohort and K. S. Larsen, "A theoretical comparison of LRU and LRU-K," *Acta Informatica*, vol. 47, pp. 359-374, 2010.
- [21] S. Ding and Y. Li, "LRU2-MRU Collaborative Cache Replacement Algorithm on Multi-core System," in *Computer Science and Automation Engineering (CSAE), 2012 IEEE International Conference*, Zhangjiajie, China, 2012.
- [22] P. Agrawal, D. Kifer and C. Olston, "Scheduling shared scans for large data files," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 958 - 969, 2008.
- [23] L. Qiao, V. Raman, F. Reiss, P. J. Hass and G. M. Lohman, "Main-Memory Scan Sharing for Multi-Core CPUs," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 610 - 621, 2008.
- [24] Q. Kuang and L. Zhao, "A Practical GPU Based KNN Algorithm," Soochow University, Suzhou, 2009.
- [25] P. e. a. Bakkum, "Accelerating SQL Database Operations on a GPU with CUDA," University of Virginia, Charlottesville, VA, 2010.

- [26] N. K. e. a. Govindaraju, "Fast Computation of Database Operations using Graphics Processors," University of North Carolina, 2004.
- [27] T. e. a. Kaldewey, "GPU Join Processing Revisited," IBM Almaden Research, San Jose, CA, 2012.
- [28] B. e. a. He, "Relational Joins on Graphics Processors," Microsoft Corporation, 2008.
- [29] C. e. a. Bohm, "Index-supported Similarity Join on Graphics Processors," BTW, 2009.
- [30] S. Madden, *Visualizing Twitter Slides Presentation*, CSAIL, MIT, 2013.
- [31] K. Devine, "<http://www.cs.sandia.gov/~kddevin/LB/quadtrees.html>," 1997.
- [32] C. S. L. Notes, "Extending Linear Regression: Weighted Least Squares, Heteroskedasticity, Local Polynomial Regression," CMU, 2009.
- [33] J. Frost, "Regression Smackdown: Stepwise versus Best Subsets!," 2012.
- [34] K. Gouda, M. Hassan and M. J. Zaki, "Prism: An effective approach for frequent sequence mining via prime-block encoding," *Journal of Computer and System Sciences*, vol. 76, no. 1, pp. 88-102, February 2010.