

Resource Utilization for Raw Data Query Processing

Optimizing Required Resources & Maximizing Utilization of Existing Resources

by

MAYANKKUMAR LALABHAI PATEL
201521012

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

to

DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY



March, 2023

Declaration

I hereby declare that

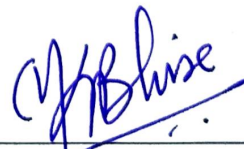
- i) the thesis comprises of my original work towards the degree of Doctor of Philosophy at Dhirubhai Ambani Institute of Information and Communication Technology and has not been submitted elsewhere for a degree,
- ii) due acknowledgment has been made in the text to all the reference material used.



Mayankkumar Lalabhai Patel

Certificate

This is to certify that the thesis work entitled RESOURCE UTILIZATION FOR RAW DATA QUERY PROCESSING: OPTIMIZING REQUIRED RESOURCES & MAXIMIZING UTILIZATION OF EXISTING RESOURCES has been carried out by MAYANKKUMAR LALABHAI PATEL for the degree of Doctor of Philosophy at *Dhirubhai Ambani Institute of Information and Communication Technology* under my supervision.



Prof. Minal Bhise
Thesis Supervisor

Acknowledgments

I would like to express my sincere gratitude to my Ph.D. advisor Prof. Minal Bhise for her continuous guidance and support throughout my Ph.D. work. I am indebted to my advisor for her constant motivation during my tough times. I sincerely appreciate her kindness, patience, and constant encouragement that kept me going.

I am grateful to my RPS (Research Progress Seminar) & Synopsis committee members: Prof. Sanjay Srivastava, Prof. Sourish Dasgupta, Prof. P. M. Jat, Prof. Saurabh Tiwari, and Prof Pankaj Kumar, for their inputs and suggestions, which helped me improve my work. I am also thankful to DA-IICT for providing a supportive environment, lab and library resources.

I sincerely thank all my colleagues at Distributed Database Research Group, DA-IICT. I thank Dr. Trupti, Nitish, Parth, Utsav, Kalgi, and Ami. I express my heartfelt gratitude to Pranav and Dr. Jignesh and for their constant assurance and support. I thank all my friends, Harshendu, Jayesh, Dhairya & Jignesh, for their best wishes and constant encouragement.

Finally, I am heartily grateful to my parents and grandparents, who raised me with utmost love and care. They have given me the freedom to choose my path and supported my every decision. Because of their constant support and motivation, I have been able to chase my ambitions. My Ph.D. became possible because of the invariable support of my soulmate Dimpal and son Malay, who constantly motivated me and helped me deal with situations assertively.

Contents

Abstract	xi
List of Principal Symbols and Acronyms	xiv
List of Tables	xv
List of Figures	xvi
1 Introduction	1
1.1 Raw Data Query Processing	4
1.1.1 Raw Data	4
1.1.2 Raw Data Query Processing	5
1.2 Resource Utilization	6
1.3 Motivation	6
1.4 Objectives	8
1.5 Contributions	8
1.6 Structure of the Thesis	9
1.6.1 Thesis Work	9
1.6.2 Thesis Organization	10
2 Background Information	13
2.1 Raw Data	13
2.2 Raw Data Query Processing	14
2.2.1 Traditional Approach	14
2.2.2 In-situ Approach	15
2.2.3 Hybrid Approach	16

2.3	Resource Utilization	17
2.4	Use Cases	17
2.5	Raw Data Processing Architecture	21
2.5.1	Maturity Levels	21
2.5.2	Raw Data Maturity & Resources	25
3	Literature Survey	26
3.1	In-situ Processing	26
3.1.1	No Loading	27
3.1.2	Main Memory Caching	28
3.2	Raw Data Loading	29
3.2.1	Incremental Loading	30
3.2.2	Partial Loading	31
3.3	Interactive Data Exploration	32
3.4	Heterogeneous Raw Data	33
3.5	Analytics on Raw Data	34
3.6	Machine Learning Techniques for Raw Data	36
3.7	Resource Monitoring	36
3.7.1	Framework	37
3.7.2	Resource Monitoring Tools	38
3.8	Optimizing Required Resources	40
3.9	Maximizing Resource Utilization	41
3.10	Cloud based Systems	42
3.11	Summary of Research Issues	43
3.11.1	In-situ Processing	43
3.11.2	Raw Data Loading	44
3.11.3	Interactive Data Exploration	44
3.11.4	Heterogeneous Raw Data	44
3.11.5	Analytics on Raw Data	45
3.11.6	Machine Learning Techniques for Raw Data	45
3.11.7	Resource Monitoring	45
3.11.8	Optimizing Required Resources	45

3.11.9	Maximizing Resource Utilization	46
3.11.10	Cloud based Systems	46
3.12	Open Research Issues	50
4	Thesis Overview: Resource Utilization for Raw Data Query Processing	55
4.1	Thesis idea	55
4.2	Thesis Approach	57
4.2.1	Hybrid System	57
4.2.2	Partitioning Approaches for Hybrid System	58
4.2.3	Heuristic approach	58
4.3	Thesis Objectives	61
5	Resource Monitoring Framework for Raw Data Query Processing	64
5.1	Phase-I Raw Data Query Processing (RQP)	64
5.1.1	Hybrid systems: In-situ engine & DBMS	65
5.1.2	Raw Data Query Processing Framework	66
5.1.2.1	Functional Requirements of Hybrid systems	66
5.1.2.2	Proposed General Framework	67
5.2	Phase-II Resource Monitoring (RM)	70
5.2.1	Role of Resources	70
5.2.2	Resource Monitoring Framework	72
5.2.3	Integration of Resource Monitoring Framework	72
5.2.4	Algorithms & Data structures	73
5.2.4.1	Data Structures	74
5.2.4.2	Algorithms	75
6	Resource Availability and Workload aware Hybrid Framework (RAW-HF): Optimizing Required Resources	78
6.1	Resource Availability and Workload aware Hybrid Framework (RAW-HF)	78
6.2	RAW-HF: Optimizing Required Resources	81
6.2.1	Query Complexity Aware (QCA) Partitioning Technique	83
6.2.1.1	QCA: Data structures	83

6.2.1.2	QCA: Algorithm	84
6.2.1.3	Data Distribution Cases	87
6.2.2	Workload and Storage aware Cost-based Technique (WSAC)	88
6.2.2.1	WSAC: Data Structures	88
6.2.2.2	WSAC: Algorithms	89
7	Resource Availability and Workload aware Hybrid Framework (RAW-HF): Maximizing Utilization of Existing Resources	95
7.1	Existing Resource Maximization Techniques	95
7.1.1	CPU Resource Maximization	96
7.1.2	RAM Resource Maximization	96
7.1.3	IO Resource Maximization	97
7.2	Maximizing Utilization of Available Resources (MUAR)	97
7.2.1	MUAR: Framework	98
7.2.2	MUAR: Resource Maximization Module	98
7.2.3	MUAR: Data structures	100
7.2.4	MUAR: Algorithm	101
8	Experimental Setup	104
8.1	Hardware & Software Setup	105
8.1.1	Implementation Block Diagram	106
8.2	Dataset & Query Set	107
8.2.1	SDSS	107
8.2.2	LOD	107
8.2.3	Comparison of Datasets	108
8.2.4	Datasets used in Experiments	109
8.3	Evaluation Parameters	110
8.3.1	Input Parameters	110
8.3.1.1	Workload Parameters	110
8.3.1.2	Resource Utilization Parameters	111
8.3.2	Output Parameters	111
8.3.2.1	Query Performance Parameters	111

8.3.3	Resource Utilization Parameters	112
8.4	Experiment Flow	112
8.4.1	Resource Monitoring Framework for Raw Data Query Processing	112
8.4.1.1	Raw data Query Processing (RQP) module	113
8.4.1.2	Resource Monitoring (RM) Module	114
8.4.1.3	Experiment Flow	116
8.4.2	RAW-HF: Optimizing Required Resources	116
8.4.2.1	QCA: Experiment Flow	116
8.4.2.2	WSAC: Experiment Flow	118
8.4.3	RAW-HF: Maximizing Utilization of Existing Resources	120
8.4.3.1	Single Resource Maximization	120
8.4.3.2	Multiple Resource Maximization	122
8.4.3.3	MUAR Technique	122
9	Results & Discussion	125
9.1	Raw Data Query Processing and Resource Monitoring framework	125
9.1.1	Workload Execution on Raw	126
9.1.2	Query Processing for Scaled Data	126
9.2	Resource Monitoring Tools Configuration	127
9.3	Resource Monitoring	129
9.3.1	Resource Utilization	129
9.3.2	Average Resource Utilization	131
9.3.3	Resource Utilization for Scaled Data	132
9.3.3.1	DBMS	132
9.3.3.2	NoDB	134
9.4	Query Classification	136
9.4.1	Query Execution Time	136
9.4.2	Resource Utilization	139
9.4.2.1	Data Caching	139
9.5	Summary of Raw Data Query Processing and Resource Monitoring framework Results	141

9.6	RAW-HF: Optimizing Required Resources	142
9.6.1	Query Complexity Aware (QCA) Partial Loading Technique	142
9.6.1.1	QCA: Partial Loading	142
9.6.1.2	QCA: Resource Utilization	144
9.6.2	Workload and Storage Aware Cost-based (WSAC) Partial Loading Technique	145
9.6.2.1	WSAC: Storage Resource Utilization	145
9.6.2.2	WSAC: Partial Loading	146
9.7	RAW-HF: Maximizing Utilization of Existing Resources	148
9.7.1	MUAR for SDSS Dataset	148
9.7.2	MUAR for LOD Dataset	149
9.7.2.1	Single Resource Maximization without MUAR . .	150
9.7.2.2	Multiple Resources Maximization without MUAR	152
9.7.2.3	MUAR: WET	154
9.7.2.4	MUAR: Resource Utilization	154
9.7.2.5	Comparison of MUAR with State-of-the-art tech- niques	157
9.7.3	MUAR for Different Datasets	159
9.8	RAW-HF: Optimizing Required Resources & Maximizing Utiliza- tion of Existing Resources	161
9.8.1	RAW-HF: Workload Execution Time	161
9.8.2	RAW-HF: Resource Utilization	162
9.8.3	RAW-HF: Processing Capacity Estimation	164
9.9	RAW-HF for Different Datasets	166
9.10	Characterization of RAW-HF	168
9.10.1	AET	168
9.10.2	WSAC: Algorithm Optimization	169
9.10.3	RAW-HF AET for Different Datasets	170
9.11	Comparison with State-of-the-art	171

10 Conclusions and Future work **176**

References	178
Appendix A List of SDSS Queries	194
Appendix B List of LOD Queries	197
Appendix C List of Publications	208

Abstract

Scientific experiments and modern applications generate large amount of data every day. Many such applications store the data in raw format initially, as the schema is not known. The traditional database management system (DBMS) requires the entire dataset to be loaded before querying it. Data loading requires a significant amount of time and resources, which increases application latency and running costs. In-situ engines eliminate the data loading requirement, thereby reducing upfront resource utilization. However, they suffer from high query execution time (QET) and reparsing. It has been observed that state-of-the-art in-situ and DBMS do not utilize available resources efficiently.

This thesis proposes Resource Availability and Workload aware Hybrid Framework (RAW-HF) to tackle underutilization of resources. It optimizes required resources (ORR) and maximizes utilization of existing resources (MUER) for resource efficient processing of raw datasets. It is a hybrid system consisting of an in-situ engine and DBMS. The in-situ engine reduces data to query time while DBMS moderates the raw data reparsing. Hybrid framework for raw data query processing and resource monitoring is developed during the initial phase. Analysis of resource monitoring indicated substantial underutilization of resources. The optimization of required resources is done using Query Complexity Aware (QCA) and Workload and Storage Aware Cost-based (WSAC) algorithms. QCA and WSAC also improved workload execution time (WET). Further resource utilization is improved by Maximizing Utilization of Available Resources (MUAR) algorithm.

RAW-HF is demonstrated using scientific experiment datasets like Sloan Digital Sky Survey (SDSS) and Linked Observation Data (LOD). RAW-HF query and

resource performances are compared with state-of-the-art techniques. The state-of-the-art techniques which allocate resources accurately based on historical resource consumption data do not address ad-hoc queries and multi-format joins. On the other hand, RAW-HF addresses ad-hoc queries and also supports multi-format joins. The ORR phase of RAW-HF reduced the WET by 26% compared to the state-of-the-art Partial Loading technique. MUAR component of RAW-HF is capable of estimating work memory value with 15-20% error required to achieve the best query performance with only single query run data. A comparison of MUAR with machine learning based techniques like PCC and AutoToken is also presented. The overall CPU, RAM, and IO resource utilization has been improved by 61-91% over traditional database management systems. Although the Partial loading technique requires 33% lesser RAM than RAW-HF, it needs 24% more IO. The improvement in dataset size processing capacity is also estimated for SDSS dataset. The estimation proposes that RAW-HF framework can be used to process large application datasets efficiently using existing resources.

List of Principal Symbols and Acronyms

AET Algorithm Execution Time

APS Accessed Partition Size

B Storage Budget

CPU Central Processing Unit

CSV Comma Separated Values

DBMS Database Management Systems

DLT Data Loading Time

DWAHP Workload-Aware Hybrid Partitioning and Distribution

FAA Fraction of Attributes Accessed

FAL Fraction of Attributes Loaded

GB GigaByte

HDD Hard Disk Drive

HP Horizontally Partitioned Table

HTAP Hybrid Transactional/Analytical Processing

IO Input Output Devices - Data Storage Devices

IO_wait Percentage of CPU time wasted in waiting for IO

J_C Join Count

JSON JavaScript Object Notation

LHC Large Hadron Collider

M Million Records

MB MegaByte

MF Multi-Format

MMDB Main Memory Database

MUAR Maximizing Utilization of Available Resources

NVMe Non-volatile memory express

ODBMS Object Database Management System

OLAP Online Analytical Processing

OLTP Online Transaction Processing

PgRAW PostgresRAW

PgSQL PostgreSQL

QCA Query Complexity Aware

QET Query Execution Time

QT Query Type

RAM Random Access Memory

RAW-HF Resource Availability and Workload aware Hybrid Partitioning

RDF Resource Description Framework

RUA Resource Utilization Aware

SSD Solid State Drive

w_l Workload List

WA Workload Aware Partitioning Techniques

WET Workload Execution Time

WSAC Workload and Storage Aware Cost-based

XML eXtensive Markup Language

List of Tables

2.1	Raw Data Maturity	24
3.1	Resource Utilization Tool Details	39
3.2	Summary of Research Issues	46
3.3	Comparison of Existing Tools and Techniques	51
3.4	Summary of Open Research Issues	52
5.1	Workload list (w_l)	74
5.2	Result List	74
6.1	Query Type Dictionary (QT)	84
6.2	WSAC: Schema Dictionary	89
6.3	WSAC: Nested Dictionary	89
6.4	Complexity of WSAC Algorithms	94
7.1	MUAR: Workload list (w_l)	100
8.1	Dataset Details	108
8.2	Query Classification	109
8.3	Dataset used in Phases	109
9.1	WET: Comparison with static resource allocation techniques	157
9.2	Comparison with State-of-the-art Resource Allocation Techniques	158
9.3	ORR: Fraction of Attributes Accessed (FAA) and Loaded (FAL)	167
9.4	RAW-HF Technique Comparison with State-of-the-art	172
9.5	RAW-HF Performance Parameters Comparison	174

List of Figures

1.1	Scientific Dataset Size Growth	2
1.2	NoDB Performance Comparison	7
1.3	CPU Utilization in DBMSs	7
2.1	Data to Query flow in In-situ Engines & DBMS	15
2.2	Raw Data Use cases	18
2.3	Raw Data Processing Architecture	22
4.1	RAW-HF: Initial Architecture	59
5.1	Raw Data Query Processing Framework	68
5.2	Resources	71
5.3	Resource Monitoring Framework for Raw Data Query Processing .	73
6.1	RAW-HF: Block Diagram	80
6.2	RAW-HF Architecture: Optimizing Required Resources	82
7.1	RAW-HF Architecture: Maximizing Utilization of Existing Resources	99
8.1	Experimental Setup: Block Diagram	104
8.2	Implementation Setup Diagram	106
8.3	<i>top</i> Tool Output: CPU & RAM Resource Monitoring	115
8.4	<i>iostat</i> Tool Output: IO (Disk) Resource Monitoring	115
8.5	Experiment flow: Phase-I & II Resource Monitoring for Raw Data Query Processing	117
8.6	Experiment Flow: QCA Technique	119
8.7	Experiment Flow: WSAC Technique	121

8.8	Experiment Flow: MUAR Technique	123
9.1	Workload Execution Time	126
9.2	Scaled Data: Workload Execution Time	127
9.3	RM Tools: Overhead	128
9.4	DBMS (PgSQL): Resource Utilization:	130
9.5	NoDB: Resource Utilization	131
9.6	Average Resource Utilization	131
9.7	DBMS (PgSQL): Resource Utilization	133
9.8	DBMS (PgSQL): Total IO Utilization	133
9.9	Storage Space Utilization: Database (PgSQL) vs. Raw (NoDB) . . .	134
9.10	NoDB: Resource Utilization	135
9.11	NoDB: Total IO Utilization	136
9.12	SDSS: QET comparison	137
9.13	SDSS: Query Classification based on Join Count	138
9.14	LOD: Query Classification based on Join Count	138
9.15	NoDB: Simple Queries using Caching	140
9.16	Resource Utilization of Simple Queries	141
9.17	QCA: Data Distribution CASEs	143
9.18	QCA: Resource Utilization	144
9.19	WSAC: Storage Utilization	146
9.20	WSAC: Partial Loading WET	147
9.21	RAW-HF: Maximizing Utilization of Existing Resources	149
9.22	Single Resource Maximization: DLT	150
9.23	Single Resource Maximization: QET	151
9.24	Multiple Resource Maximization: DLT	152
9.25	Multiple Resource Maximization: QET	153
9.26	MUAR: QET	155
9.27	MUAR: Average Resource Utilization	156
9.28	MUAR: Disk Writes	156
9.29	MUAR: Comparison with state of the art	158
9.30	Impact of MUAR on WET for LOD & SDSS datasets	160

9.31 RAW-HF: WET Comparison	162
9.32 RAW-HF: Resource Utilization	163
9.33 Raw Data Processing Capacity Estimation	165
9.34 Impact of RAW-HF on WET for LOD & SDSS datasets	167
9.35 AET of RAW-HF Algorithms for SDSS	168
9.36 WSAC: Algorithm Optimization	169
9.37 RAW-HF AET for SDSS and LOD	170
9.38 Scaled Data: AET for SDSS and LOD	171

CHAPTER 1

Introduction

The scientific, smart city, Internet of Things (IoT), and modern applications have generated massive amounts of data in last decade. Their data generation speeds have significantly increased with improvements in sensors, cheaper smart devices, and users. 4G and 5G internet services have also impacted data generation, and consumption rates [10]. IoT devices like health monitoring, location sharing, smart meters, and other smart devices were estimated to double from 7B in 2018 to 13.5B in 2022 [75]. However, the latest estimation suggests that the number of IoT devices may reach 14.4B by the end of 2022 [90]. For IoT applications, real-time processing of sensor data is challenging due to the high velocity and no downtime. For example, IoT applications like e-bike [59] can continuously produce more than 6GB of data every second if applied to just 1% of vehicles around the globe [9]. Scientific experiments on the Large Hadron Collider (LHC) generated 2.6PB data during its initial runs in 2009. The amount of generated data increased by 25 times to 90PB during experiments done in 2018 [23]. The future runs at CERN are expected to generate ten times more data due to its improved particle accelerator chain High Luminosity LHC [13].

The volume of Astronomy datasets like the Sloan Digital Sky Survey (SDSS) has increased by 233 times when comparing the first version (DR-1) released in 2003 to the most recent version (DR-17) released in 2021 [17]. NASA's Earth Observing System (EOS) collects more than 3.3TB of data every day from more than 30 polar-orbiting and low inclination satellites [61]. These satellites perform long-term global observations of the land surface, biosphere, atmosphere, and oceans to monitor & predict changes in the earth's environment. Figure 1.1 compares the

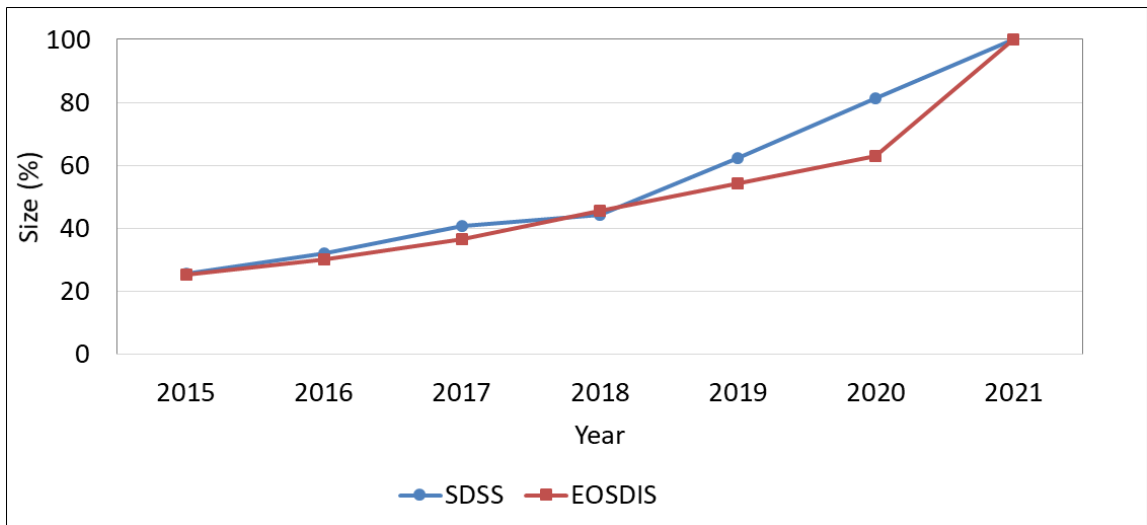


Figure 1.1: Scientific Dataset Size Growth

growth rate of SDSS and EOS scientific datasets. The X-axis in the graph shows the dataset size observation year and Y-axis shows the dataset size in percentage, considering 2021 as 100%. It can be seen that both datasets have nearly quadrupled in size in just 7 years, from 2015 to 2021 [17]. Earth Observing System Data and Information System (EOSDIS) repository in the cloud reached the size of 59PB in September 2021 from 15PB in 2015 [21]. Similarly, the SDSS dataset size has also increased nearly four times, from 166TB in 2015 to 652TB in 2021.

To query large datasets, researchers have developed traditional disk based row stores [22], column stores [116], main memory based systems [60], stream engines [80, 68], approximate query processing (AQP) [44], in-situ [33, 94], and hybrid systems [29, 53]. Stream data processing engines could only process a limited amount of data without loading. In contrast, traditional row store, column store, and main memory based database management systems (DBMS) must load the entire dataset into DBMS specific formats before executing queries. In-situ engines can process large datasets stored in raw format without loading them. Main memory-based database systems (MMDBs) have higher throughput than disk-based DBMSs, but are more vulnerable to failures [84]. Therefore, MMDBs like Zen+ propose to use non-volatile main memory (NVMe) to build log-free systems for higher throughput [81]. The RAM and NVMe storage like Intel Optane are getting cheaper, but they are still 20-40 times more costly than SSDs & HDDs.

The stream processing engines and AQPs provide inaccurate results as they do not process the entire dataset to answer queries. The thesis work focuses on getting accurate results by efficiently processing large datasets using limited resource systems. Therefore, MMDBs, data stream processing engines, and AQP systems have not been considered.

Traditional row or column store DBMSs require significant CPU, RAM, and IO hardware resources to load the entire dataset into DBMS. Because row or column store DBMSs require to load entire dataset before they start query execution. Irrespective of how much from this loaded data is actually required for processing that query. This fact is visible in the results of the experiments conducted by A. Jain [67]: “only 8% of data can answer 64% of workload queries”. That means traditional DBMSs are using 92% resources for the remaining 36% workload. Querying large amounts of fresh and historical data in real time requires the deployment of datasets in distributed or cloud environments. The distributed environments keep multiple copies of data to make the system fault-tolerant and provide faster query responses. The cloud based systems offer pay-as-you-go costing models, which bill users based on the cloud resources utilized. The requirement for more hardware and electricity resources increases application running costs. A work estimates that cloud and other organization data centers will consume 20% of the world’s energy by 2025, increasing carbon footprint by 5-10 times [38].

Applications data with high volume, variety, and velocity conditions are stored in raw formats to reduce the data storage time and upfront resource requirements [59, 31, 62]. The in-situ engines can query raw data directly by eliminating data loading steps. These in-situ engines suffer from high query execution time (QET) as they need to execute queries on raw data. Additionally, in-situ engines discard the processed data after completing query execution, which introduces the reparsing. The processed data needs to be retained to improve the QET of future queries and reduce redundant utilization of resources. Hybrid systems containing in-situ engines and DBMS have been developed to process only required partitions of the dataset and retain processed data for future queries [29, 53]. The amount of resources required during query execution depends on the dataset size, number

of joins, data filtering, and other aggregation operations. Most DBMS are tuned to allocate a fixed amount of resources considering the average resource requirements of Online Transaction Processing (OLTP), and Online Analytical Processing (OLAP) queries. Resources required by query in the past have been considered to allocate appropriate resources to repetitive queries [106, 103]. However, it is challenging to provide resources accurately to new or ad-hoc queries.

The thesis work tries to find answers to the following questions.

- Which data processing tools to use for resource efficient processing of data?
- How to partition a dataset to optimize resource utilization?
- How to schedule and allocate appropriate resources to each workload task for faster execution?

1.1 Raw Data Query Processing

This section defines raw data and briefly explains how traditional DBMSs, in-situ engines, and hybrid systems process the application data to obtain results.

1.1.1 Raw Data

The human readable unprocessed format of data is known as raw data. Raw data is the collection of records stored as strings. The strings can be stored in unstructured or semi-structured formats. A speech stored in a text file is an example of an unstructured raw data format. The comma-separated values (CSV) file storage is the semi-structured storage format. The attribute values are separated using a specific delimiter for storage in CSV format. CSV, JSON, XML are semi-structured formats used to store raw data. Storing data in raw format does not require data to be converted into a fixed format or schema. Therefore, raw data formats can easily store data generated by users, applications, sensors, or external sources as a collection of string records in a file. IoT, Scientific experiments, and other applications store the generated data in raw format due to its low storage time than traditional database management systems DBMS [31, 59]. Additionally, the

schema of the generated experimental data, error logs, event logs, and external data sources may not be known beforehand. More use cases of raw data storage are discussed in Section 2.4.

1.1.2 Raw Data Query Processing

This section explains how data stored in raw format gets processed by traditional row store or column store DBMSs, in-situ engines, and hybrid systems.

Traditional Approach: The traditional way of extracting knowledge or performing analytical queries requires the entire dataset to be loaded into a row store or column store DBMS. The data loading and indexing steps are time-consuming tasks that improve the execution time of future queries. However, the row stores are optimized for OLTP queries, while OLAP queries can be executed faster using column stores.

In-situ Approach: In-situ or raw data query processing engines directly work on raw data files, eliminating the data loading time and vendor lock issues. However, the query execution time (QET) is high for in-situ engines as required raw data undergoes parsing, conversion, and tokenization steps during query execution. These in-situ engines do not collect or store the processed data for future queries, which means the resources might be used to process the same raw data recurrently for future queries. Therefore, it can be determined that neither traditional systems nor in-situ engines efficiently utilize available resources [125, 33].

Hybrid Approach: Hybrid Transactional/Analytical Processing (HTAP) systems containing row store and column store DBMSs have been developed to improve OLTP and OLAP query execution time. These systems load data into row and column store DBMS and redirect OLTP and OLAP queries to relevant DBMS for faster response times. Therefore, HTAP systems suffer from high data loading time and require more resources [115]. To eliminate the need to load entire dataset into DBMS, hybrid systems containing in-situ engines and traditional DBMS have been proposed. Such hybrid systems can query raw data directly using in-situ engine, reducing data generation to query time. The DBMS component of hybrid systems loads the processed data into DBMS to resolve reparsing and improve

QET for future queries.

1.2 Resource Utilization

Executing data processing operations on raw data requires CPU, RAM, and IO resources. The traditional way of loading data into DBMS requires significant resources in advance compared to incremental loading and partial loading techniques [29, 53, 125]. However, executing queries on preprocessed data reduces the time and resources required by all future queries during query execution. On the other hand, raw data storage saves time, and upfront resources requirement by eliminating data loading steps. In-situ engines require more time and resources than DBMSs during query execution because they have to process the raw data. Techniques like partitioning, caching, and partial loading have been developed to optimize resource utilization and improve query execution on raw data. At the same time, task scheduling and allocating appropriate resources to each query task are necessary steps to utilize existing resources completely. The literature survey of the thesis has identified that all the proposed techniques fundamentally optimize & maximize the utilization of available resources by carefully organizing the data and data processing operations.

1.3 Motivation

The traditional way of processing large datasets requires an entire dataset in the loaded format increasing data to result time. While raw engines can answer several queries before data loading gets completed, as shown in Figure 1.2 [33]. However, NoDB (PostgresRAW) caches the entire dataset to reduce the QET of future queries, which increases main memory utilization. The data cached by NoDB gets cleared from the main memory to accommodate new data increasing reparsing. Increasing main memory size is not practical due to continuously increasing dataset size and higher RAM hardware costs compared to hard disk drives.

On the other hand, it has been observed that existing DBMSs are not utilizing

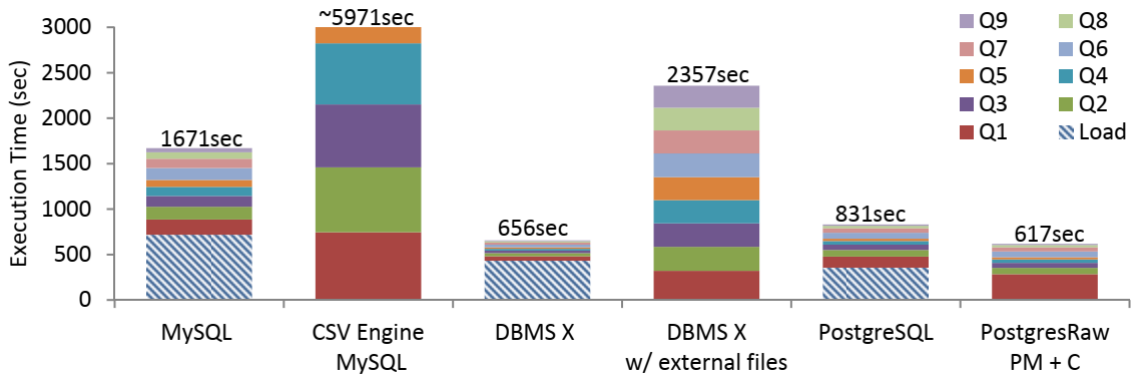


Figure 1.2: NoDB Performance Comparison [33]

all available resources. Figure 1.3 displays the percentage of CPU capacity utilized by different DBMSs like Shore-MT, DBMS D, VoltDB, and HyPer [32]. Most DBMSs utilize only 25-50% of CPU resources, while only 12% of CPUs are utilized at data centers [32]. The future of data management requires processing large datasets with minimal resources to reduce application running costs [89]. Therefore, finding ways to process collected data efficiently is a crucial open issue for all modern and scientific applications.

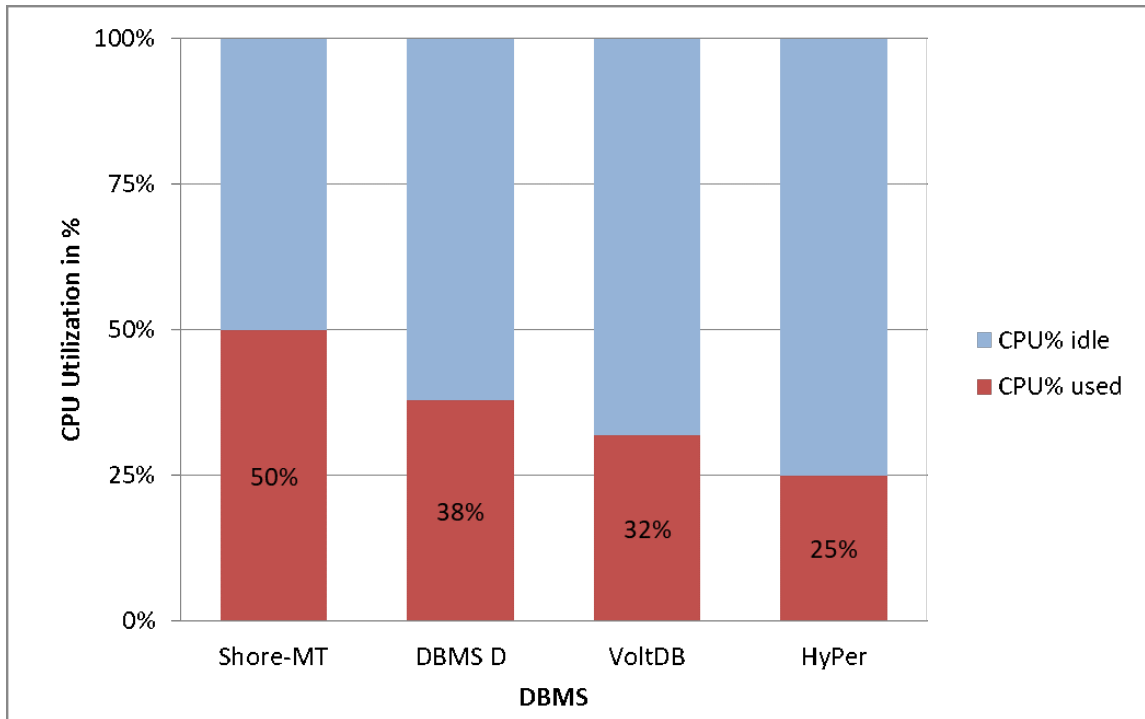


Figure 1.3: CPU Utilization in DBMSs [32]

1.4 Objectives

The data generated by most applications is stored in raw format initially due to low storage time. Keeping raw data always in raw format requires reparsing of raw data, increasing application running costs and QET. At the same time, DBMS are optimized to query preprocessed data faster, reducing query execution time and resource utilization. However, the traditional way of processing data requires substantial time and resources upfront to load the entire dataset into DBMS.

- The objective of the research work is to find a resource efficient way of processing raw data while reducing total workload execution time (WET).
 - Combine the best features of in-situ engines and DBMSs to reduce WET while utilizing available resources efficiently.
 - Identify resource efficient ways of partitioning, distributing, and querying raw data to optimize resource utilization.
 - Improve the utilization of available resources to reduce WET and application costs.

1.5 Contributions

The summary of thesis contributions is listed below.

- This thesis proposes a Resource Availability and Workload aware Hybrid Framework (RAW-HF) to query raw data efficiently.
- The work proposes Query Complexity Aware (QCA) and Workload and Storage Aware Cost-based (WSAC) partial loading techniques to optimize the resources required to execute given workload tasks.
- Developed a lightweight task scheduling and resource allocation technique MUAR (Maximizing Utilization of Available Resources) for faster workload execution.

- MUAR considers the real-time availability of resources and allocates more resources to complex queries (CQ) to improve WET.
- The proposed resource optimization and maximization techniques have been demonstrated using the real-world dataset Sloan Digital Sky Survey (SDSS) and Linked Observation Data (LOD) [30].
 - Results analysis has shown that ORR techniques works better for broad table datasets like SDSS, while MUAR is capable of improving WET for workload with complex multi-join queries like LOD.
- The results of the proposed techniques have been compared with the state-of-the-art in-situ engine [33], row store DBMS [22], workload aware Partial Loading technique [125], PCC[103], Elastic[106], and AutoToken[109].

1.6 Structure of the Thesis

This section briefs the thesis work and thesis organization for a better understanding of the research work discussed in this thesis.

1.6.1 Thesis Work

The thesis work proposes a Resource Availability and Workload aware Hybrid Framework (RAW-HF). The components of RAW-HF have been discussed in four phases for better understanding. The first two phases collect valuable information used by Phase III to optimize required resources and maximize the utilization of available resources in Phase IV. The combination of all four phases as RAW-HF helps in achieving proposed objectives.

Phase-I Raw Data Query Processing Framework – The first phase develops and implements a raw data query processing framework, which allows the execution of queries on raw data and data loaded into database tables. The framework is implemented using an in-situ engine NoDB (PostgresRAW), and PostgreSQL (PgSQL) DBMS, which can execute join queries on multi-format data.

Phase-II Monitoring Resources – This phase updates the phase-I framework to monitor CPU, RAM, and IO resources utilized by DBMS & In-situ engine during execution of workload tasks.

Phase-III Optimizing Required Resources – This phase developed Query Complexity Aware (QCA) and Workload and Storage Aware Cost-based (WSAC) partitioning techniques to optimize resource utilization. This phase reduces resource utilization during query execution by processing dataset partitions required by workload queries.

Phase-IV Maximizing Utilization of Existing Resources – The maximization phase combines multiple resource maximization techniques to utilize idle resources. This phase proposes a MUAR (Maximizing Utilization of Available Resources) technique to maximize the utilization of available resources to reduce WET.

1.6.2 Thesis Organization

The thesis contains the chapters listed below. Each chapter is discussed in brief for a better understanding of thesis organization.

Chapter-2 Background Information – This chapter explains basic details of the raw data query processing domain. The chapter explains the definition of raw data, its use cases, and how query processing is done on raw data. A raw data processing architecture explains the life cycle of raw data. Raw data maturity levels have been defined to represent the number of operations performed on raw data.

Chapter-3 Literature Survey – This chapter surveys vital raw data query processing techniques that give insight into raw data query processing issues and their proposed solutions. The literature survey has been extended to cover resource monitoring, optimizing, and maximizing techniques to find resource efficient ways of processing raw data.

Chapter-4 Thesis Overview: Resource Utilization for Raw Data Query Processing – This chapter discusses the thesis approach, core idea, and objective of thesis work. The chapter briefly discusses the tasks of each thesis phase.

Chapter-5 Resource Monitoring Framework for Raw Data Query Processing– This chapter proposes a Raw Data Query Processing Framework (RQP) with resource monitoring capabilities. Phase-I developed a general raw data query processing framework, while Phase-II integrated a resource monitoring module into RQP.

Phase-I Raw Data Query Processing –This subsection proposes a raw data processing framework. The required framework features have been identified by studying key papers necessary for resource efficient processing of raw data. The proposed framework has been implemented using state-of-the-art raw engine (NoDB) and row store DBMS (PgSQL).

Phase-II Resource Monitoring – Monitoring resources required by raw data processing tasks can provide insights into optimizing resource utilization. Therefore, this section proposes a resource monitoring module and integrates it into RQP.

Chapter-6 RAW-HF: Optimizing Required Resources – This chapter discusses two workload aware vertical partitioning techniques developed in Phase-III to optimize resources required to process raw data efficiently: 1) Query Complexity Aware (QCA) technique, 2) Workload and Storage Aware Cost-based partial loading technique (WSAC).

Chapter-7 RAW-HF: Maximizing Utilization of Existing Resources– Phase-II results showed that the state-of-the-art in-situ engine (NoDB) and DBMS (PgSQL) could not utilize all available resources. Therefore, this chapter surveys CPU, RAM, and IO maximization techniques to develop a resource utilization aware task scheduling algorithm to maximize the utilization of existing resources. A Maximizing Utilization of Available Resources (MUAR) technique is proposed to automate resource allocation and task scheduling processes for a given workload.

Chapter-8 Experimental Setup – This chapter discusses the experimental setup, dataset, query set, and experiment flow for all the thesis phase experiments. Two real-world datasets have been used for experiments to analyze effective techniques for broad datasets and complex workloads.

Chapter-9 Results & Discussion – This chapter contains the experiment results of all the thesis phases. The result analysis of phase-I & II shed light on

some important patterns, which helped develop WSAC, QCA, and MUAR techniques. The performance of the proposed algorithms has been compared with state-of-the-art raw engine NoDB [33], row store DBMS PostgreSQL [22], Partial Loading technique [125], PCC[103], Elastic[106], and AutoToken[109].

Chapter-10 Conclusion & Future Work – The final chapter of the thesis work concludes the research work. It discusses how the proposed algorithm performs better than the existing state-of-the-art data processing tools and partitioning algorithm backed by experimental result analysis. This chapter also lists promising advances that can be investigated to improve the proposed work.

The lists of References, Publications, and Queries have been added at the end.

CHAPTER 2

Background Information

This section discusses the background information on raw data and how to query raw data. The raw data section defines which data is considered raw data in the context of thesis work. The following subsections discuss how raw data gets processed utilizing hardware resources. The raw data processing architecture shows the relationship between tools, raw data, and maturity levels. The last section of raw data maturity and resources sheds light on the relationship between maturity levels and resources.

2.1 Raw Data

What is raw data? The unprocessed form of data is referred to as raw data. For example, a temperature sensor observes the surrounding temperature every second and provides it as the output data stream. The data stream is a continuous string of values in unprocessed format at its source. This data stream can be stored at the sensing source as text, which may be unstructured. These unstructured strings of data have been defined as raw data. Researchers also consider semi-structured data formats like CSV, JSON, XML, and others as raw data because the data needs to be parsed, tokenized, and converted to specific data types at the destination. The Big Data definition includes unstructured data characteristics of raw data, meaning Big Data applications use raw formats to store data with high volume, velocity, and variety [62].

The following section discusses the use cases where generated data is stored in raw format initially. The query processing section discusses the sequence of oper-

ations performed on raw data to get results based on the tools used. The images and audio-video files can be considered a form of raw data storage. Images can be represented as Flexible Image Transport System FITS can be stored as human readable raw data files [125].

2.2 Raw Data Query Processing

This section discusses the traditional and in-situ approaches to process raw data. Different applications have different query processing needs on stored data. For example, many IoT applications need to automate tasks based on their reading from sensors in real time. Stream engines handle this task, which processes only current data collected during a fixed time frame or dataset size [112]. However, applications may need to analyze historical data to make accurate decisions requiring analysis of the entire dataset. The row store or column store DBMS & in-situ engines are used to execute queries on the entire dataset.

2.2.1 Traditional Approach

The traditional way requires the entire dataset to be loaded into DBMS, requiring substantial time and resources before executing any query. The data loading process parses, tokenizes, and converts the raw data into required data types and stores the processed data in a DBMS specific format. The future queries access processed data stored in databases eliminating reparsing of raw data. Analytical query processing on traditional row store DBMSs is slow due to high IO costs from reading the entire records compared to column stores. At the same time, transactional queries are faster in row stores. Indexes can help in getting data faster. However, index generation as a part of data loading tasks can substantially increase data loading time (DLT).

Indexes become impractical compared to data scanning as the data selectivity increases [72]. Therefore, column store DBMSs have been developed to reduce the IO cost of reading entire columns [27, 26, 116]. The column stores introduced sorting and compression of columns to minimize the data IO costs at query time.

However, the data loading time in column stores is higher due to storage of individual attribute values at different disk locations with additional sorting and compression steps. M. Boissier et al. [45], M. Stonebraker et al. [102], and T. Padiya et al. [96], and have proposed incrementally shorting, partitioning, and refining indexes in later stages as they gather workload information. The delayed partitioning and index creation may also fail when the workload changes. This demonstrates that it is challenging to organize the data beforehand efficiently, while query processing on the well organized preprocessed data can reduce QET.

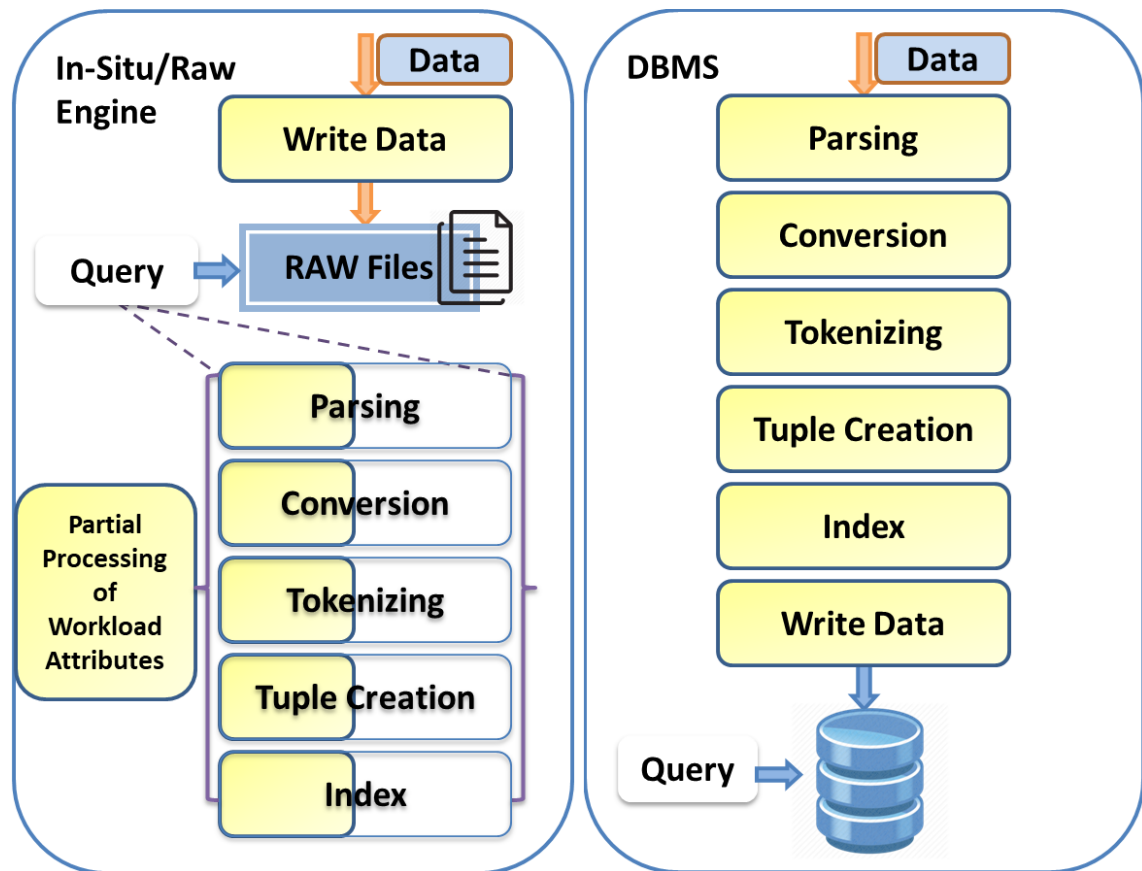


Figure 2.1: Data to Query flow in In-situ Engines & DBMS

2.2.2 In-situ Approach

In-situ engines allow the execution of queries on the entire dataset without loading it. Figure 2.1 shows that raw data is immediately available for executing queries using in-situ or raw engines, while the traditional way requires the entire dataset to be loaded into DBMS. The in-situ engines delay the processing

of raw data until the arrival of the query. This delay enables us to identify the data required by the query. The data processing steps of tokenizing, parsing, and datatype conversions can be optimized by knowing the exact query requirements for raw engines compared to traditional systems. If the query requires only a part of the data, the partial processing method can save a considerable amount of data to result time for larger datasets [97]. However, the delayed processing increases the QET time of initial queries compared to traditional row or column store DBMS [33]. The raw engines proposed caching of processed data to reduce the QET time for future queries [33]. However, the cached data gets discarded to process new data, introducing the reparsing. Additionally, the online analytical query processing (OLAP) queries needs to access the whole raw data file to extract single column data, which is slower than compressed columnar access in column stores. Hence, permanently keeping data in an unprocessed raw format is not ideal when parts of the dataset are frequently required.

2.2.3 Hybrid Approach

Hybrid systems containing in-situ engines and DBMS allow execution of queries on raw datasets immediately. At the same time, the DBMS component is used to save the data processed by in-situ engines for future queries [29]. SCANRAW proposes incrementally loading the raw data into DBMS whenever resources are idle [53]. These hybrid systems use multiple software tools and need a framework to handle the query execution and data loading tasks. The in-situ and DBMS used to build a hybrid system may not be compatible. In such cases, DBMS or In-situ engines cannot access data processed by other systems necessitating data required by a query in either raw or database format. Another workaround would require combining results from multiple tools for each query. R. Chaiken et al. [51], A. Ailamaki et al. [33, 70], and Y. Cheng et al. [53] are working on developing systems with in-situ engines as an extension to existing DBMS to solve this issue, allowing queries to directly access data stored in raw and database formats easily.

2.3 Resource Utilization

Performing data processing tasks requires three main hardware resources CPU, RAM, and IO. Executing raw data parsing, tokenizing, and data type conversation requires CPU resources. RAM resource caches the data required by the CPU for processing. IO resources are the storage devices that save the raw or processed data for future use. The traditional way of loading raw data into DBMS requires significant amount of resources and time before queries can be executed on the raw data. The traditional way processes the raw data and stores it in a specific database format. Query processing on the loaded data requires less time and resources. However, most application queries do not access the entire dataset to obtain the required results [76, 124, 67]. This means the resources are used in processing the data, which may never get used by workload queries. Thus, it can be derived that resources are used in processing unnecessary data. Moreover, large dataset sizes increase QET because queries need to access more data to generate query results.

On the other hand, in-situ engines work directly on the raw data, reducing the upfront requirement of resources to load data. However, in-situ engines require more resources in the long term because they have to process the raw data every time. This repetitive processing of raw data issue is also known as raw data reparsing. This means DBMS and In-situ data processing tools are not optimized to utilize minimum required resources.

2.4 Use Cases

This section lists the real world applications where the generated data is stored in the raw format. The application datasets used for experiments have also been listed for reference. Figure 2.2 presents raw data generating applications with their raw data formats.

Scientific: The scientific applications include raw data from scientific experiments, simulation data, and sensor observations [113]. The generated data needs

to be stored in raw format initially due to their unknown schema and velocity. Scientific applications like LHC generate 1TB of sensor data every hour [29, 31]. The streaming sensor data is filtered in real time before storing in raw format to accommodate high data velocity. Parts of the LHC experiment dataset are available to users at CERN's open access data portal CERN [14]. The LHC dataset is too large for most traditional systems to be processed in a centralized model. Therefore CERN provides access to these datasets using Jupyter from Cloud deployed datasets.

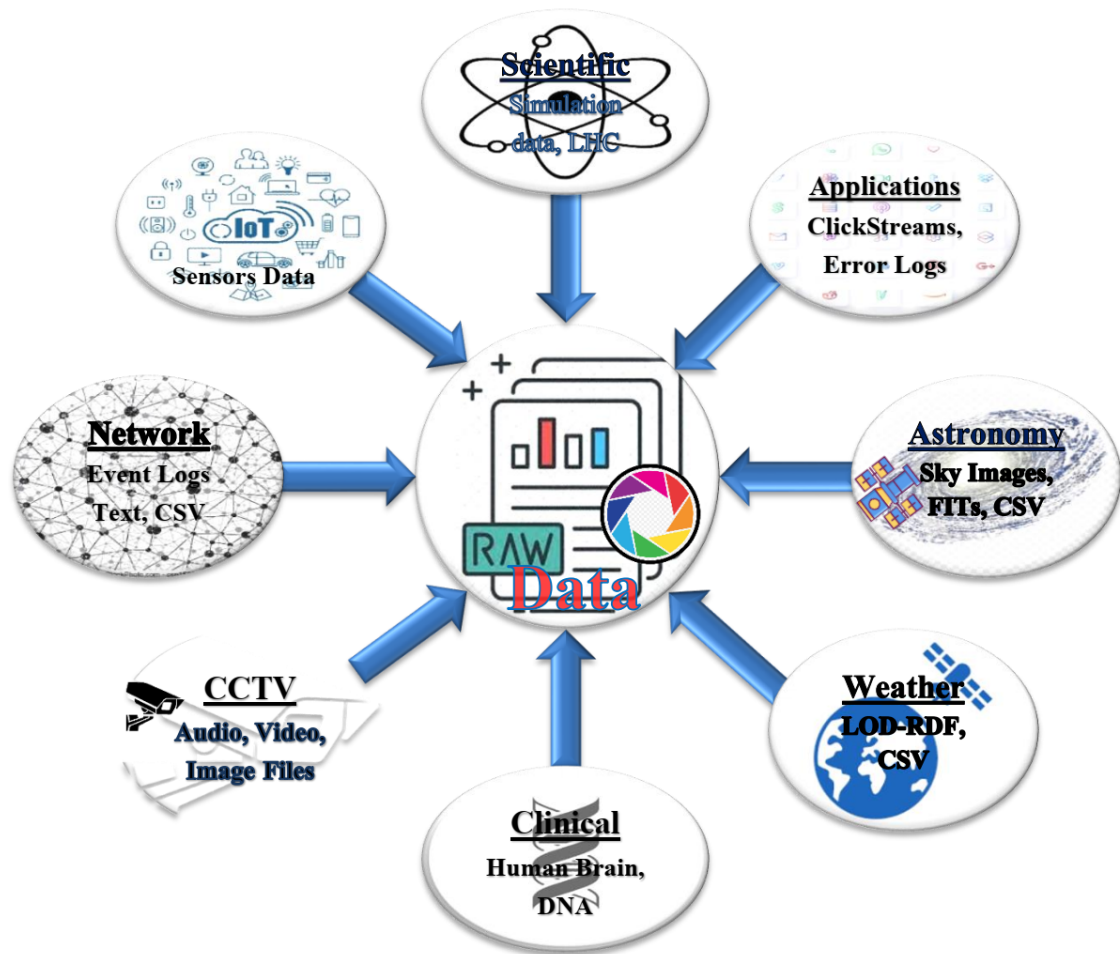


Figure 2.2: Raw Data Use cases

Astronomy: Astronomy is a field of science that studies celestial objects and phenomena. The majority of data observed by telescopes is image data. These images are converted to FITs and CSV formats for better analysis using relational DBMS. Scientists have deployed many telescopes worldwide, including the Hub-

ble telescope launched into space in 1990. Open source astronomical observation datasets like Sloan Digital Sky Survey (SDSS) have generated images, FITs, and CSV format data reaching 652TB [28, 125]. The entire astronomy dataset SDSS can be downloaded from the SkyServer website using SQL queries.

Web applications: The web applications record user clicks, pages visited, errors, and other event data in raw format to avoid additional load on the servers [29, 33, 71]. The data is analyzed later to find user access patterns and improve the application features. Most modern applications use WEB APIs to consume data from other sources. The binary file sending is not practical due to different DBMS tools used at source and destination. Therefore, data is transferred in XML, JSON, RDF, and other human-readable formats, which are then converted to their required formats at the destination [42, 59, 105]. TPC provides benchmark datasets like TPC-H, and TPC-IoTx for research and benchmarking purposes [25].

Clinical: The patient details, DNA sequences, diseases, viruses, and vaccination details are stored at hospitals, clinics, and research facilities. The data may be in physical files, excel sheets on a machine, or in-house servers. These data are protected in many countries by the law. The law restricts the data movement outside the source location, limiting the available resources to process that data. Projects like Human Brain Project kept the different data in their raw format to avoid data transformation to a specific format, and vendor-lock [70]. The DNA data can be stored as a sequence of DNA bases A, C, G, T in string format. M. Castillo has been working on storing digital data in DNA because research suggests that 1g of single-stranded DNA can hold 455 EB of data [50].

Weather: The earth is observed by satellites from space, airborne, and ground sensors which record global images, temperature, humidity, wind speed, and other environmental aspects. According to Committee on Earth Observation Satellites (CEOS), more than 514 Earth observation satellites were launched from 1962 to 2012 [61]. The 30 satellites of NASA from the Earth Observing System (EOS) project generates more than 3.3 TB of data per day. The volume of EOS project data is estimated to increase by 16 times, from 15PB in 2015 to 246.6PB by 2025 [21]. The size of these datasets and streaming nature require stream processing

of current data using stream engines and storing them in raw format as historical data for later analysis. The EOS datasets are available on the NASA website [2]. Linked Observation datasets which contained rainfall, humidity, temperature, and other observations, have been used by researchers for experimental purposes [67, 96].

CCTV: The CCTV images and video format data are also considered one variety of raw data as without processing it, knowledge cannot be extracted. Many smart applications apply machine learning techniques to identify people, animals, or other objects to provide automated notifications and triggered tasks [91, 121]. Machine learning applications use raw video footage to automate traffic lights and generate challans [119].

Internet of Things IoT: Most modern applications like Smart Homes, Smart Cities, Smart Devices like smartphones, e-bikes, and self-driving cars are equipped with multiple sensors that generate different varieties of data at configurable speeds [42, 57, 59]. Developers of e-bike application stored the streaming sensor data in raw format and loaded data later to reduce the DLT [59]. The modern IoT applications also consume data generated by other sensors or sources to automate tasks at houses, factories, and cities [93, 104]. The smart city dataset of CityPulse is available in CSV, and Resource Description Format (RDF) formats [1].

Network: The speed of data transfer has increased worldwide with the introduction of 5G. The package logs, error logs, and other networking events logs are stored in raw formats to avoid additional load on the servers [29]. These logs are analyzed to distribute internet traffic, find the fastest routes for packets, and find faults in the network lines or nodes to improve network throughput and reliability.

Google big query provides access to 300 plus datasets, including New York Trips, Weather, and other IoT datasets, with limited 30 days of free access [19]. The linked open data of 1301 datasets are also available on the website, which provides datasets in RDF format [16].

2.5 Raw Data Processing Architecture

As discussed in Section 2.2, raw data processing steps may differ based on the tool used. The in-situ engines do not store the processed data, while database management systems create a copy of the dataset in their specific database formats to answer the queries. Different partitioning techniques allow different partitions to be stored in multiple formats and storage levels. Data residing in a specific format and storage medium may answer queries faster than un-processed data or data stored in a slower storage medium. The raw data maturity levels have been defined in this paper to identify which combinations are faster.

Figure 2.3 shows the raw data processing architecture to demonstrate how raw data gets processed to reduce QET. The diagram shows the lifecycle of raw data beginning from raw data generating sources to caching of frequently accessed data in main memory. The generated data is initially stored in raw format, which can be queried using query processing tools listed on the left. Traditional database management tools require raw data to be loaded into a database format. The architecture displays the type of data processed by the tools in the middle with data characteristics. The data characteristics show the format of the data. For example, text format is unstructured, CSV & JSON are semi-structured, while database formats are completely structured. The tools that process those different data formats have been displayed on the same level, i.e., stream processing or in-situ engines work on raw data, while DBMSs work on databases. The last column on the right shows the defined maturity level of raw data for that combination.

2.5.1 Maturity Levels

This section discusses the Maturity levels of raw data. These levels have been defined based on operations executed on the data, format of data, and the tools used at that level. The raw data passed through multiple operations have a higher maturity level. These operations include data parsing, data type conversion, partitioning, caching, and others. These operations allow data processing tools to execute queries faster using required partitions. At level 0, data has not gone

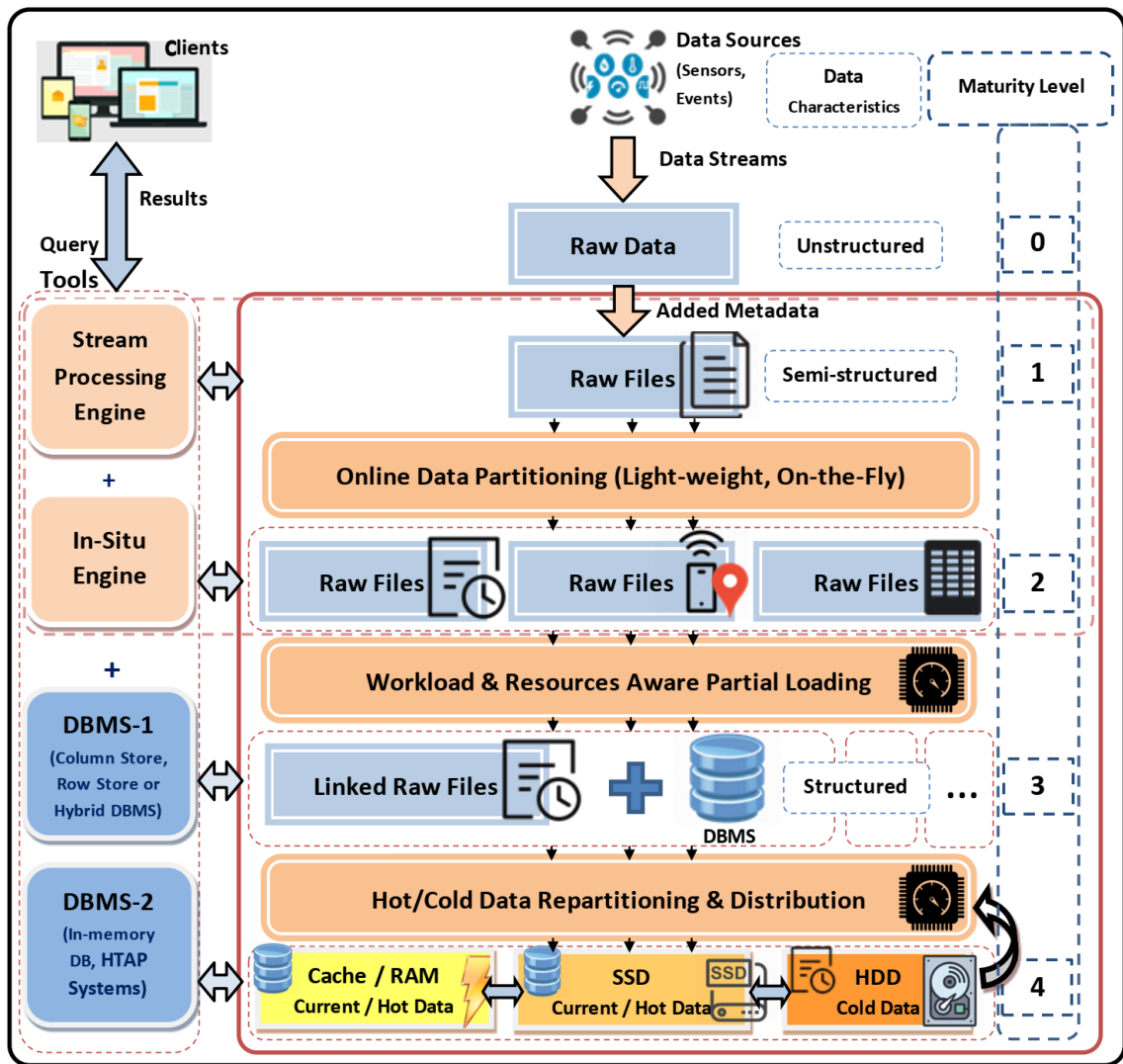


Figure 2.3: Raw Data Processing Architecture

through any processing steps.

Level 1 data is processed to relate it with metadata, making the raw data queryable. Data residing at Level 2 is partitioned, allowing queries to process only required parts of raw data. Levels 3 & 4 allow access to preprocessed data, reducing QET compared to raw data accesses. A detailed explanation of different operations performed at each level has been discussed here.

Level 0: At the 0th level of data maturity, raw data has no meaning, or no knowledge can be gained. One cannot identify what the data stands for by seeing only the observed values. It is impossible to identify a value as temperature, age, or another number. The values need additional information like observation unit in Celsius or Fahrenheit, observation location, date, and time of observation. This

additional information that gives the values their meaning is often called metadata.

Level 1: The metadata is added based on sensor details and application domain knowledge to provide meaning to data. The data with metadata is then stored as a comma or other delimiter-separated text format in files. These text files have been called raw data files or raw files in short. It is the most common form of data storage as it's easy to store and human-readable. Most applications discussed in the use cases section used raw format to store the streaming data. The stream processing engines or in-situ engines work at this level, which process this semi-structured data to obtain query results.

Level 2: This level partitions raw data streams or original raw dataset files using one or more lightweight partitioning techniques to reduce data processing time for future queries. These basic partitioning techniques use domain knowledge, schema, time, location, or sensor id to partition the dataset into smaller files [82]. Smaller files reduce the IO cost for in-situ or raw engines, as most queries require only part of the entire dataset [67]. For example, suppose a weather dataset is partitioned based on location. In that case, queries like "finding the average temperature of a particular location" only need to process that specific location covering files.

Level 3: At Level 3, the spatial-temporal, workload, and resource aware partitioning techniques identify the hot data for loading into DBMS [29, 53]. While cold or unused data can be kept in raw format [125]. The OLAP queries execute faster on loaded data than the previous levels accessing semi-structured raw data. The data stored in a DBMS has a proprietary data storage structure, which other DBMSs or external tools cannot access directly. It provides data security, but a single database structure is generally optimized for a few specific types of queries. For example, data stored in row stores allow faster online transaction processing (OLTP) queries, while column stores allow faster analysis of single column data. Therefore, at Levels 3 & 4, Hybrid Transactional/Analytical Processing (HTAP) systems execute transactional queries on row stores and analytical queries on column stores. However, such systems suffer from high loading time caused by

storing copies of data in multiple DBMSs [115].

Level 4: The 4th level of maturity is gained when frequently used data is further partitioned into hot and cold. The partitioning techniques use workload information, data access logs, available resources, or other application requirements to distribute the hot data among faster storage mediums like RAM or SSD while cold data to HDD [49, 67, 74]. The most frequent or current data is cached in RAM to reduce QET. The hot data changes when workload changes or shifts from old to newer data, requiring re-partitioning and rearrangement. There are no additional levels of data maturity except caching results, refreshing query results, and using summaries to answer the frequent queries [74, 111].

A summary of all the maturity levels is shown in Table 2.1. It shows the details of data & file format, partitioning technique, storage location of partitions, data to query time, faster analytics, and query processing tools used at every level.

Table 2.1: Raw Data Maturity

Mat- urity Lvl.	Data & File Format	Partition Type	Storage Location	Data To Query Time	Faster Analyt- ics	Data Processing Tools
0	Raw – Un- structured (Text, CSV)	None	Device Level	Low	No	NA
1	Raw – Semi- structured (CSV, JSON, XML)	None	HDD	Low	No	Stream Processing & In-situ Engines [33]
2	Raw – Semi- structured (CSV, JSON, XML)	Time, Loca- tion, Schematic [82]	HDD	Low	No	In-situ Engines [33, 94]
3	Raw & Database (CSV, JSON, XML, Binary)	Workload Aware [125]	HDD	Moderate	Yes	DBMS + In-situ Engines [29, 53]
4	Database \newline (Binary)	Hot/ Cold [67, 76]	HDD, Main Memory	High	Yes	DBMS-1 + DBMS-2 (HTAP [115])

2.5.2 Raw Data Maturity & Resources

The raw data must pass through many resource consuming operational steps to increase its maturity level. Higher maturity level data reduces the resource requirements during query execution. At Level 1 of maturity, the semi-structured data requires more resources and time to process the entire dataset to answer queries. Level 2 reduces resource utilization by processing partitions required by queries. At Level 3, the required data is preprocessed and stored in a binary format, i.e., database. The query processing on a database requires fewer resources due to indexes, data blocking, compression, and other data preprocessing techniques used by DBMS. Level 3 & 4 data maturity has to deal with changing workloads where data is classified as hot and cold based on detailed workload analysis to moderate resource utilization. The raw data query processing techniques try to achieve these different levels of data maturity by optimizing the required operations. The required operation may differ from application to application based on their data and workload characteristics. Therefore following section discusses techniques that try to process raw data efficiently to get results faster. These techniques try to achieve higher maturity levels for required data partitions while minimizing the resource utilization, time, or operations to reach that level.

CHAPTER 3

Literature Survey

The chapter surveys research papers that provide insights into raw data query processing and resource monitoring to find open research issues. The papers have been categorized based on the research issues they address. The first subsection discusses techniques and tools that propose solving issues faced while executing queries on raw data. It was observed that the raw data query processing techniques performed better due to the efficient utilization of available resources. Therefore, the subsequent subsections discuss the techniques that focus on resource monitoring, optimizing required resources, and maximizing utilization of available resources. Subsection 3.11 discusses a summary of research issues of the literature survey to identify open research issues. The discovered open issues have been briefed in Section 3.12, which researchers can investigate and develop innovative techniques to solve those issues.

3.1 In-situ Processing

The traditional way of querying raw files was to write a code to open that file and read lines one by one. This section discusses the evolution of raw data query processing from writing file-specific codes to using standard query language SQL to answer queries over raw files. The main challenge faced while querying raw data is the reparsing. Main memory caching is one way to reduce raw data reparsing. This section is divided into two subsections that categorize the techniques based on the location of processed data to moderate the reparsing.

3.1.1 No Loading

This section discusses raw data query processing tools and techniques which do not load data into DBMS. Loading data into DBMS creates copies of original raw data files as databases which require more storage space. It is possible to delete the raw data files loaded into a DBMS. However, organizations keep the raw data because data loaded into databases is converted into a DBMS specific binary file structure. Other tools or DBMSs cannot process these binary files. It is also known as the vendor lock-in problem. Therefore, raw files are retained to keep the data available for processing by other tools and applications.

Initially, the motivation for querying raw data files was to gain basic information and not perform analytical queries. Developers wrote only the necessary C or Java code to open the raw files and filter the required information. However, this coding method requires writing long and complex codes, which becomes inefficient for performing complex analytical operations on raw data. Therefore, Just-in-time operators generate specialized codes to efficiently process raw data based on query requirements [70]. Systems like Hadoop and Bigtable have been developed, which use Map-Reduce jobs to query large raw data files using multiple CPU cores or distributed environments. These Map-Reduce jobs are highly parallel and can divide the tasks into small subtasks to multiple CPU cores or data processing nodes for processing. These systems also merge results of subtasks to generate final results. In Map-Reduce jobs, Mapper code assigns key-value relations to data, and Reducer code performs the required operations on the data. These codes are written specifically for each query. The issue with Map-Reduce codes is that they are less reusable, which increases the efforts required to query the raw data. To overcome the issue, R. Chaiken et al. at Microsoft have developed declarative and extensible scripting language Structured Computations Optimized for Parallel Execution (SCOPE) to query raw data directly [51]. SCOPE has a SQL-like structure that is standard and easy to use. Users only need to replace the relational table names with the actual raw file names to query the raw data. MySQL and Oracle also supported querying raw files using their CSV engine, and external table features [107, 122]. Mison [77] and speculative parsing

[58] approaches extract records faster from raw files. Mison is used to parse JSON files, while speculative parsing techniques have been applied to CSV files.

The tools and techniques discussed in this section do not increase the maturity level of raw data because they do not retain processed data for future queries. All the queries have to parse, tokenize and convert required raw data accessing raw files every time. This issue is known as raw data reparsing. Resources are used multiple times to do the same work, increasing overall data processing costs. The techniques discussed in the following sections aim to retain the parsed data to moderate the reparsing.

3.1.2 Main Memory Caching

Main memory caching and indexing strategies have been developed to moderate the reparsing without loading raw data. Initially, developed tools process raw data files and cached the processed data into the main memory [52]. Later, raw data processing tools like NoDB proposed caching and indexing the raw data for faster query processing [33]. The NoDB is one of the first few raw data query processing tools with DBMS-like features. Like SCOPE, MySQL CSV engines, and Oracle External Tables, NoDB also allows executing SQL queries on CSV files. Slalom extended the caching and index creation by generating partition specific indexes for logically partitioned raw data [33]. Slalom created best suited index for each partition on the fly based on query access patterns. Data vaults cached raw data into arrays to improve analytical queries [66]. The main memory caching of data increases the maturity level of data to the final 4th level of maturity. However, the raw data reparsing cannot be eliminated as the processed data is not stored permanently.

Different caching strategies have been developed to handle different combinations of raw formats. ReCache proposes to cache processed JSON or CSV raw data, which improves overall query processing time by 19-75% [39]. As the new data arrives or workload changes, the cached data needs to be replaced. An adaptive cache mode selection algorithm Acme considers the dataset, main memory size, and workload to choose the optimal cache mode at run time to accommodate

workload changes [40]. Data Vaults cached column data into array structures to reduce analytical query processing time [66]. However, caching entire columns consumes a large amount of main memory space. Therefore, a work proposes to cache only one column completely and filter the remaining column rows based on the query condition to reduce memory consumption [97]. Another issue that leads to removal of cached data is updation of raw files. When data existing in raw files is changed, the processed raw data cached in main memory needs to be removed and re-cached to reflect changes. Alpine is a prototype that logically partitions the raw files and keeps track of the updates executed on the raw file [37]. If any file is updated, Alpine triggers watchdog and adds a log entry to process before or during query execution to provide updated results. Alpine only updates the binary cache, indexes, and positional maps for the updated logical partition, reducing recaching and reparsing overhead.

Main memory space is a limited resource. Large datasets cannot be cached entirely in the main memory. Caching only frequently accessed data requires less memory than caching an entire dataset. It enables faster processing of larger datasets with available resources. However, changing workloads and addition of new data require processed data to be removed from the memory. Therefore, main memory caching techniques can not completely eliminate the reparsing.

3.2 Raw Data Loading

This section surveys techniques that retain processed data for future queries to eliminate reparsing. Hybrid systems consisting of in-situ engine and row or column store DBMS avoid raw data reparsing and improve QET for future queries. The core idea behind hybrid systems is that the in-situ engine executes queries on raw data, while the data processed by in-situ engines is stored in DBMS for future queries. However, it is challenging to identify which dataset partitions should be loaded into DBMS to improve WET.

A. Dziejic et al. [56], and T. Muhlbauer et al. [92] have tried to reduce data loading time by developing faster bulk data loading methods for magnetic disks,

solid state drives, non-volatile main memory NVMe, and random access memory file systems RAMFS. S. Kim et al. have also proposed to remove costly steps like sorting and redistribution while streamlining the conversion process to improve data loading in array-based DBMS SciDB [73]. However, loading time is substantial even with best bulk loading technique COPY [56]. Therefore, invisible loading [29], and speculative loading [53] techniques have been developed to incrementally load data to reduce upfront resource requirements. Incremental loading techniques gradually load horizontal or vertical data partitions into DBMS whenever resources are available. In contrast, partial loading techniques avoid loading entire datasets to reduce overall cost or meet resource usage limitations [125]. The techniques discussed in this section increase the data maturity to Level 3 or higher as dataset partitions required by workload queries are loaded into DBMS.

3.2.1 Incremental Loading

Incremental loading techniques divide the dataset into smaller horizontal chunks or vertical column partitions for gradual loading of the dataset based on workload requirements or resource availability. In 2013, J. Abadi's group developed an invisible loading technique to store the data processed by Map-Reduce jobs into column store MonetDB [29]. The technique keeps the processed data in cache until the main memory is full. Once the main memory is completely utilized, the processed data is loaded into DBMS for future queries. Data Vaults tool caches the processed data as arrays in main memory to improve analytical query execution on scientific data [66]. Data Vaults incrementally loads these processed column arrays into MonetDB to reduce upfront utilization of resources and time required to load data into DBMS. It has been observed that the IO resource utilization gets reduced once required data is cached in the main memory. SCANRAW loads raw data in parallel to raw data query processing, which utilizes idle resources efficiently [53]. SCANRAW uses current and historical resource utilization data to adapt to workload changes and switch between data loading and query processing tasks. It also uses min-max summaries of chunks to skip unrelated chunks

during query execution. All the incremental loading techniques are bound to load the entire dataset into a DBMS if the workload queries access all attributes at least once. Incremental loading techniques reduce upfront data loading time. However, the overall time and resources required in incrementally loading data are higher than loading entire dataset at once [29]. It has been observed that the query execution time increases as the dataset size increases. Therefore, limiting the amount of data loaded into DBMS is proposed to maintain faster query execution time.

3.2.2 Partial Loading

This section discusses the techniques that propose loading only a specific part of the dataset into DBMS while keeping the rest in raw format. It has been observed that most application workloads frequently access only a small part of the dataset [76, 49, 67]. A. Jain et al. have observed that 63% of query workload can be answered using only 8% of data [67]. Moreover, the smaller partition size reduced query execution time by 83%. Workload analysis is required to find frequently accessed data, and changing workload requires repetitive analysis of new workload queries. Partial Loading [125] proposes to vertically partition the dataset into two parts. The first partition represents the frequently accessed dataset which should be loaded into DBMS, while the second partition is kept in raw format due to storage budget limitations. The cost function developed by W. Zhao et al. compares the total time spent loading and querying required columns using traditional DBMS with the time spent querying raw data to find which attributes should be loaded. The cost function finds attributes that can cover maximum workload queries for a limited storage budget.

Finding k -attribute cover of the workload to reduce total workload execution time is an NP-hard problem [125]. Most cost-based algorithms require correct metadata of dataset and workload queries to make informed decisions. The process of collecting accurate metadata using the original data increases algorithm execution time (AET). W. Zhao et al. [125], SCANRAW [53] and others have tried to approximate the query execution time, attribute size in loaded format, data

loading time, and other necessary metadata. However, finding accurate metadata without performing these operations on the entire application dataset and query set is challenging.

3.3 Interactive Data Exploration

Interactive data exploration techniques are similar to sampling techniques [78, 54, 83]. However, the users guide the sampling processes rather than choosing random chunks and shifting or changing window sizes to estimate approximate answers. Approximate Query Processing (AQP) provides approximate answers by processing sample chunks of dataset [83]. A. Alvarez-Ayllon et al. have done a detailed survey on AQP, and interactive data exploration techniques [35]. The interactive techniques continuously take inputs from users to process only relevant data to produce the result. An interactive visual exploration tool RawVis executes sampling queries on raw data based on user inputs [44, 15]. RawVis incrementally builds lightweight indexes in the main memory to keep track of accessed partitions and improve group by operations [86]. It also adapts to workload changes by refining indexes based on GUI interactions of users to keep indexes relevant to future queries. The latest update on the RawVis added the functionality of building indexes based on resource availability [87]. N. Bikakis et al. have developed an ExDRa tool to query federated raw data by solving the statistical heterogeneity of raw data stored in different formats [43].

The interactive data exploration techniques save a lot of time and resources by not processing the entire dataset. However, this also makes the results approximate and inaccurate. The sampling and interactive data exploration techniques are best suited in cases where results are needed fast, and approximate results are adequate. However, most applications may not work with approximate results. Therefore, the remaining sections are focused on techniques that produce accurate results using entire raw files or processing all relevant partitions.

3.4 Heterogeneous Raw Data

Modern applications need to collect data from multiple external and internal sources to provide specialized features to the users. These sources have various data formats like JSON, CSV, XML, text, or HDF5. For example, user click events and other logs are kept in CSV files to avoid high loading time. The data fetched from web APIs use XML or JSON formats. The complex queries might need to join various types of data from different sources to gain knowledge from data. The processing of such heterogeneous data in real-time is crucial for many applications.

The traditional way of processing heterogeneous raw data is complex and resource consuming. Conventionally, developers write extract, transform, and load ETL codes to process all types of raw data files into a single database format. The ETL operations increased the data to first query time [33]. The delays might not be acceptable for applications that need real-time query processing on streaming and existing historical data. Sometimes, the applications need to access new sources with no fixed formats. The existing data source formats might change over time. Such cases necessitate modification of the ETL codes for the smooth running of applications. Therefore accessing and processing heterogeneous data from various sources is challenging. The techniques discussed here try to achieve higher maturity levels for raw formats, which are costly to process.

The in-situ engines should access most formats and perform queries without loading the data into a single database format. Hybrid systems consisting of in-situ engines and DBMS have been proposed to handle raw data & load the processed data into databases [29]. However, using multiple systems complicates the handling of data. Therefore, hybrid systems having in-situ engines as extensions have been developed to handle raw and binary data using the standard query language SQL [29, 33, 122]. The multi-format data partitioning and distribution strategies have been proposed for hybrid systems [125]. M. Karpathiotakis et al. proposed using just-in-time operators to execute queries on data sources having different data formats like CSV, JSON, XML, and RDBMS [70]. Raw data from

different sensors or sources have multiple formats and data generation speeds. Different data sizes pose a statistical heterogeneity issue for storage in relational databases. ExDRa optimizes access to these federated raw data sources [43].

Recache technique discussed caching mechanisms for JSON & CSV data considering their reading, parsing, and caching costs [39]. The efficient caching management helped reduce query time by 19-75%. Few research papers demonstrated partitioning techniques on CSV, JSON, FITS, or binary data formats to distribute data among database and raw formats to improve query execution [125]. Most query engines are built for general query processing tasks, which are not optimized to process different types of queries on heterogeneous datasets. Proteus query engine uses code generation techniques to write specialized code for faster execution of each query on CSV, JSON, and database formats [69]. The online sampling and aggregation technique on CSV and FITS format data is discussed in a paper [54]. The Hierarchical Data Format version 5 (HDF5) is an open-source file format that stores large, complex, heterogeneous raw data. A work demonstrated querying raw data of scientific simulations stored in HDF5 and XDMF formats [113].

3.5 Analytics on Raw Data

Analytical query processing is a core part of most applications to gain knowledge from massive data. Analytical queries need to access entire columns to provide results of sum, average, and other aggregation queries. In-situ engines and traditional row store DBMS need to access the complete file or dataset to extract the required columns, increasing query time [122, 107]. On the other hand, the response time of online analytical processing (OLAP) queries is much faster in column stores DBMS. However, column stores DBMS take more time to load data due to the presence of sorting and compression operations [27]. Moreover, tuple reconstruction time is high due to its columnar storage architecture [47]. Traditionally, most applications use hybrid HTAP systems to answer OLTP and OLAP queries faster. However, these HTAP systems require additional time to load data

in multiple database systems [115]. This section discusses techniques that aim to find efficient ways of processing raw data required by analytical queries. These techniques load and cache required data columns, increasing the data maturity level to 3 or higher.

Column stores are faster because they store column data together on disks. The similarity of data allows better compression, which reduces the size. Due to data compression, the main memory usage and IO accesses are much less in column stores. Column stores also use multiple techniques like vectorized query processing, invisible joins, and late materialization to answer queries faster [27, 26, 116]. On the other hand, the in-situ engines have to parse each record line from the raw file and perform tokenizing and data type conversion tasks. A technique proposed to reduce the data parsing time by filtering unwanted data based on the data selection of the query [97]. Main-memory caching and indexing of required data columns can improve response times for future analytical queries [39, 93]. However, large datasets cannot fit entirely in the main memory. The reparsing persists for most main memory dependent techniques. Therefore, data needs to be loaded into column stores for faster processing, as multiple OLAP queries might require the same data columns. The loading process is more time consuming in column stores than in row stores. Therefore, incremental and partial loading techniques have been developed to reduce upfront data loading time [29, 53, 64, 66, 125]. The techniques have already been discussed in Sections 3.2.1 & 3.2.2. The Data Vaults cached columns in array structures to process OLAP queries faster in main memory, and incremental loading of columns in MonetDB [66]. R. Borovica-Gajic et al. have proposed a Skipper framework for latency insensitive analytical query processing over historical data stored in cold storage devices [49]. Cold storage devices are not always online to save energy. The technique proposed grouping queries and scheduling cold storage devices to reduce wait time.

3.6 Machine Learning Techniques for Raw Data

This section covers machine learning and other techniques proposed to address sampling, summarizing, and deleting stored data without losing the collected knowledge and other issues.

ML techniques have been applied to aggregation, sampling, query approximation, resource management, and other tasks to process stored raw data [54, 110]. ML techniques have been applied to heterogeneous federated raw data for cleaning, preprocessing, and solving statistical heterogeneity challenges [43]. The sampling techniques process only a chunk of raw data files to approximate query results with error prospect [79]. The approximate query processing AQP systems process only sampled chunks from the original dataset, reducing hardware resource requirements. The query execution time and result accuracy can be controlled by changing the window size of samples. Y. Cheng et al. have proposed keeping track of accessed samples from files by building bi-level indexes to store summaries and avoid repetitive conversion [54]. Q. Ma et al. estimated density and aggregation-attribute values using ML models to build a light, accurate, and fast AQP system [83]. This work does not focus on the sampling and visual exploration techniques that work directly on raw data because a majority of techniques were covered in the recent survey paper [35]. The automatic summarization and deletion of raw data while retaining extracted knowledge techniques are in their initial stages [89]. D. Saxena and A. K. Singh have proposed identifying resource requirements of workload queries using a neural network based technique to allocate minimal Physical Machines (PMs) to complete the query processing tasks [108].

3.7 Resource Monitoring

This section discusses research papers that show the importance of monitoring resource utilization to process raw data efficiently. The techniques discussed in earlier sections have tried to reduce the operations on data or the amount of data

to improve WET and reduce resource utilization. For example, loading data into a database reduces repetitive operations on raw dataset because future queries access preprocessed data stored in a database. Y. Cheng et al. [53], and A. Dziedzic et al. [56] have been monitoring resource utilization during raw data processing steps to find meaningful patterns that can optimize the entire data to result process. SCANRAW tool has been developed to efficiently utilize available resources based on the observed resource utilization patterns [53]. The following section discusses the framework and tools researchers have used to monitor and analyze resources used by raw engines and DBMS during raw data processing operations.

3.7.1 Framework

A. Dziedzic et al. monitored resource utilization patterns of different DBMSs to find bottlenecks in the raw data loading process [56]. This DBMS data loading work analyzed the CPU and IO resource utilization of four DBMSs and explored the possibility of improving data loading time. The analysis established that slower IO storage devices are the main bottleneck. The experiments conducted with different IO devices showed that changing IO devices from hard disk HDD to solid state drive SSD or main memory file system RAMFS can significantly improve the data loading time. The *sar*, *iostat*, and *iosnoop* tools have been used to monitor CPU, RAM, and IO resource utilization during the data loading process. SCANRAW proposed to find idle CPU and IO resource time to load data into DBMS in parallel to raw data query processing [53]. Y Cheng and F. Rusu have observed that IO bandwidth is underutilized when CPU is processing raw data fetched from disk. SCANRAW proposes to take advantage of this available IO bandwidth and idle CPU cores. SCANRAW identifies CPU and IO resource utilization patterns to load the small chunks of raw data in parallel, increasing resource utilization.

Main memory caching techniques discussed earlier in Section 3.1.2 also require monitoring RAM resource utilization to configure existing data eviction policies or develop new ones [40, 117]. A work proposed to utilize 50% RAM to cache summaries of dataset partitions [117]. The literature survey shows that monitor-

ing resource utilization is crucial in understanding raw data query processing and developing novel techniques.

3.7.2 Resource Monitoring Tools

This subsection discusses frequently used resource monitoring tools to identify resource utilization during raw data processing. Operating systems like Linux and Windows provide *System Monitor* and *Task Manager* tools having detailed GUI interfaces to display utilization of all essential hardware resources like CPU, RAM, and IO. However, an analysis of the GUI needs to be done by humans due to limited data export features. Therefore, *top*, *iostat*, *sar*, *iosnoop* tools are used to monitor and analyze the resource utilization information in textual format [56, 48, 46]. The textual format also allows easy filtering and faster storage in CSV format. The filtering process limits resource monitoring output size and facilitates easy analysis.

Table 3.1 shows some of the frequently used resource monitoring tools. Operating systems like Linux and Windows provide *System Monitor* [3] and *Task Manager* [24] applications with advanced graphical user interfaces GUI for visual analysis of resources utilized by individual processes and total utilization. However, these tools required 2-10% of CPU resources to run the application. The *top* [7] & *htop* [4] tools track CPU and RAM resources utilized by individual processes. These tools also provide the percentage of CPU time wasted waiting for IO resources. The *iostat* tool provides data read/write the information of individual processes and overall data read/write [5]. Tools like *VmStat* tool provides total utilization of all resources in a single tool [8]. However, it does not provide resources utilized by processes. *Sar* tool provides CPU resource utilization by default [6]. However, advanced options allow tracking queue length, network statistics, disk blocks, page faults, and other memory statistics in pages per sec unit.

None of the above mentioned techniques provided open source tools or standard procedures for monitoring resources for data processing tasks. The data processing tasks are threads or sub-processes running inside a primary DBMS process. Therefore, existing tools cannot track the resources used by each data

Table 3.1: Resource Utilization Tool Details

#	Tool Name	Tool Type and OS	Resources Monitored			Tracks Individual Process	Overhead (CPU%)	Interface
			CPU	RAM	IO			
1	<i>System Monitor</i> [3]	Linux Application	Yes	Yes	Yes	Yes	2-10%	GUI
2	<i>Task Manager</i> [24]	Windows Application	Yes	Yes	Yes	Yes	2-10%	GUI
3	<i>top</i> [7], <i>htop</i> [4]	Terminal cmd - Linux	Yes	Yes	No	Yes	<1%	Text
4	<i>Iotop</i> [5]	Terminal cmd - Linux	No	No	Yes	Yes	<1%	Text
5	<i>VmStat</i> [8]	Terminal cmd - Linux	Yes	Yes	Yes	No	<1%	Text
6	<i>Sar</i> [6]	Terminal cmd - Linux	Yes	No	No	No	<1%	Text

processing task without a proper framework. The following sections discuss techniques that optimize required resources and maximize utilization of resources by analyzing resource utilization patterns of different workload tasks.

3.8 Optimizing Required Resources

This section classifies techniques that optimize the resources required to execute application workload queries. It is essential to understand how different tools process the raw data to optimize resource utilization. It is known that the traditional way requires processing and storing of entire raw dataset in a DBMS specific format to answer queries. The conversion process requires CPU resources, and additional storage space to save processed data as a database. However, the reuse of processed data helps in reducing CPU, RAM, and IO resource utilization during query execution for DBMSs. On the other hand, in-situ engines utilize resources in doing repetitive work because processed data is not saved for future use. Therefore, this section focuses on the research work that utilizes optimal resources to execute the entire workload.

Traditional and modern database management systems store the data in their specific binary formats to execute queries faster and utilize optimal resources. However, most of the workload execution time is spent processing the raw data and creating database files for large datasets. A. Dziejic et al. have carried out a series of experiments to study resource utilization patterns [56]. It helped in understanding role of modern hardware resources in the data loading process of DBMSs. The study stated that CPU resources stay idle due to the slower speed of IO devices. The parallel processing of raw data cannot improve CPU or IO resource utilization. The work concluded that sequential access to disk storage devices is the resource-efficient way of loading data into DBMS. At the same time, loading data in faster IO devices can utilize the CPU better, reducing data loading time. Another way of reducing data loading time is to load only required partitions of the dataset into DBMS based on workload analysis. The incremental loading Section 3.2.1 and partial loading Section 3.2.2 discuss such workload

aware techniques that benefit from loading required data partitions [29, 53, 125].

3.9 Maximizing Resource Utilization

This section discusses resource maximization techniques. Most maximization techniques try to maximize the utilization of one or more hardware resources. Traditional DBMS systems are optimized to utilize maximum IO bandwidth in the data loading process [56]. In other words, IO bandwidth is the bottleneck in the data loading process. The parallel loading of data cannot reduce the DLT for disk based storage devices. The work also stated that changing slower IO devices with faster SSD, NVME, or main memory can reduce the DLT. Faster data loading techniques for main memory databases have been developed that utilize faster IO devices and multiple CPU cores to improve DLT [92]. SCANRAW uses idle IO bandwidth during the query execution process to load data into DBMS [53]. It proposed monitoring the IO and CPU resource utilization and scheduling raw data loading tasks to maximize utilization of available resources.

Main memory caching techniques discussed in Section 3.1.2 increase the main memory utilization to reduce WET [39]. PDC-Query (PDC) proposed to cache the summaries of data partitions utilizing 50% main memory [117]. The cached summaries identified which partitions are useful for query and executed sub-queries on all relevant nodes to reduce the QET of a single query. However, partitioning a single query on multiple nodes requires merging intermediate results to produce final results [117]. Therefore, most techniques execute multiple queries on multiple CPU cores to reduce overall WET [113]. A work distributed join operations of a query to reduce QET and efficiently utilize multiple cores of CPU [120, 85]. The techniques used distributed index join algorithm and a novel distributed learned index to improve join performance for sorted and unsorted partitions. For a multi-node setup, resource allocation strategies try to balance workload evenly to maximize utilization of all available resources [108, 98]. The two-pass Mison and speculative parsing techniques to parallelize file parsing using multi-core CPU systems [77, 58]. D. Saxena et al. have been using machine learning techniques

to predict the resources required by workload and assign minimum hardware resources where their utilization can reach 100%, reducing application running costs [108]. Similarly, machine learning techniques were proposed to allocate maximum RAM resources to frequent queries to reduce QET [99, 103].

The limited number of techniques consider a combination of resource optimization and maximization techniques [125]. Moreover, many resource optimization and maximization techniques cannot be used in conjunction. For example, DBMS data loading experiments show that parallel execution of a well-optimized data loading technique COPY to maximize CPU resource utilization cannot improve DLT [56]. This work tries to find techniques that can be used in conjunction to reduce WET significantly.

3.10 Cloud based Systems

Cloud service providers like Amazon, Google, and Microsoft provide cloud services with various pay-as-you-use models. The two most relevant models are serverless, and server resource reservation. The serverless query execution services act like in-situ processing on raw files. The Amazon web services like Athena and Redshift provide services to query raw files using SQL [18, 20]. Google big query also allows executing SQL queries on datasets with the main memory caching option [19]. The bill generation is based on the queries executed by users on the dataset. This service does not provide any dedicated infrastructure to users. Therefore, this service is also known as a serverless query service.

Modern applications need to store data, install specific software, and run business logic code, which serverless services cannot fulfill. Therefore, cloud service providers allow renting servers where organizations or companies can deploy their applications. The companies use cloud resources to reduce the initial hardware investment costs. Cloud service providers propose that the cloud is cheaper than managing in-house servers when maintenance costs are considered. Cloud services calculate costs based on resource utilization preferences selected by users. For example, reserving dedicated or private server resources costs 2.4x more than

a shared resources model. In addition, cloud services provide virtually unlimited scaling facilities to handle peak loads. The cost of additional resources gets calculated based on the time and amount of resources used. Therefore, researchers have been trying to reduce the application running costs for cloud based systems. The primary difference between cloud and in-house resources is that in-house resources are limited and cannot be increased in real time. However, most optimal resource allocation techniques can be applied to cloud based and in-house systems with minor changes.

C. Wu et al. have observed that the default resource allocation strategies are not cost effective [123]. H. S. Patel et al. proposed to provide sufficient resources to frequent queries considering its earlier performance while allocating minimum resources to new queries to reduce overall cost and WET [99]. A. Pimpley proposed to create a performance characteristic curve (PCC) for each query which relates resources and performance [103]. This PCC curve is then analyzed to create a cost efficient resource allocation plan to execute queries. D. Saxena et al. tried to estimate the number of resources required to complete the given workload and allocated minimum possible virtual machines by training ML models [108]. A. Raza et al. have also considered distributing available resources between OLAP and OLTP tasks to modern overall workload execution time. The developed technique reduced OLAP QET by 50% by maintaining a small and controlled drop in OLTP throughput [106].

3.11 Summary of Research Issues

This section summarizes the literature survey to find important open research areas that do not have a proven correct solution. Table 3.2 summarizes the literature survey to describe each category's solved and open research issues.

3.11.1 In-situ Processing

Raw data query processing techniques and tools can query raw data without loading [70, 51, 107, 122]. However, the reparsing persists as all queries access un-

processed raw data from raw data files. Main memory caching techniques can improve QET of frequent queries by caching hot partitions. However, they suffer from reparsing for datasets larger than the main memory size.

3.11.2 Raw Data Loading

The incremental loading technique eases the upfront resource requirement to load data by distributing loading tasks to later times. However, incremental loading techniques require more time and resources to load the same amount of data than bulk loading methods. Incremental loading techniques are also prone to load the entire dataset into DBMS if an attribute is used at least once. Partial loading of frequently accessed data limits the size of loaded data. However, a work stated that choosing attributes that cover maximum queries is an NP-hard problem [125]. It reduced the partitioning problem to the finding k-attribute cover problem to solve it using a heuristic approach.

3.11.3 Interactive Data Exploration

A. Alvarez-Ayllon et al. surveyed 242 research papers to find interactive data exploration tools and techniques that satisfy three requirements: accessing raw data files, accessing files distributed on multiple nodes, and responding within a few seconds [35]. The study stated that only one paper could satisfy three requirements of interactive data exploration. Therefore, extensive work needs to be done in this area to develop interactive tools satisfying various application requirements, like improving result accuracy using limited resources.

3.11.4 Heterogeneous Raw Data

S. Baunsgaard et al. [43] & M. Karpathiotakis et al. [70] have developed several tools that handle CSV, JSON, XML, and HDF5 formats. However, raw data processing tools that can process all types of datasets and raw formats while solving statistical heterogeneity are scarce or developed for research only [43, 70].

3.11.5 Analytics on Raw Data

A. Abouzied et al. [29] and S. Kim et al. [73] have proposed to load raw data incrementally into column store DBMSs like monteDB and caching data in arrays to improve the performance of OLAP queries. Researchers can work on finding cost-efficient ways of executing OLAP queries.

3.11.6 Machine Learning Techniques for Raw Data

Applying machine learning ML techniques to different raw data query processing problems is an emerging area. Data aggregation, deletion, knowledge retention, mining, and solving statistical heterogeneity are some of the topics where researchers have been applying ML techniques.

3.11.7 Resource Monitoring

Most DBMS and in-situ engines do not provide resource monitoring capabilities due to the inherent overhead of monitoring and analysis. L. Viswanathan et al. have proposed that integrating resource information with query planning can improve query performance and reduce application running costs [118]. However, there are no standard resource monitoring frameworks nor tools exist that can provide resources utilized by data processing tasks like data loading, query processing, sorting, and indexing.

3.11.8 Optimizing Required Resources

Utilizing optimal resources to complete data processing tasks is an NP-hard problem [85]. The raw data query processing requires a hybrid data management tool to utilize optimal resources. The partitioning of application datasets into raw files and database tables is an NP-hard problem [125].

3.11.9 Maximizing Resource Utilization

Maximizing resource utilization problems include NP-hard problems like resource allocation and task scheduling [114]. Y. Cheng et al. [53], H. Tang et al. [117] and W. Zhao [126] have been monitoring resource utilization to maximize CPU, RAM, or IO utilization to reduce workload execution time. However, techniques maximizing utilization of all available resources are scarce.

3.11.10 Cloud based Systems

Reducing resource utilization to reduce costs is crucial for applications deployed on the cloud [118]. H. S. Patel et al. have proposed allocating required resources to frequent queries while limited resources to ad-hock queries to reduce costs [99]. Elastic resource scheduling proposes trading off OLTP throughput to reduce execution time of OLAP queries by 50% [106]. It is known that task scheduling is also an NP-hard problem [114]. Therefore, scheduling workload tasks to utilize minimal or maximum resources to reduce total WET is a promising research area for researchers. The summary of research issues helped propose open research issues discussed in the following section.

Table 3.2: Summary of Research Issues

#	Research Issues	Objective	Techniques	Advantages / Solved Issues	Remaining Issues
1	Raw Data Query Processing	Query raw data without loading.	No Loading No Loading [70, 51, 107, 122]	No data loading, No replication – requires less storage, Reduce data to result time.	Reparsing.

			Main Mem-ory Caching [52, 33, 66, 39, 40, 97, 37]	Moderated reparsing & improve QET by caching processed data.	Efficient caching of multi-format data.
2	Raw Data Loading	Store pro-cessed data to reduce the reparsing.	Incremental Loading [66, 29, 53]	Incrementally loading of data required by workload. Eliminate Reparsing. Improve QET.	Prone to load entire dataset. Requires more time & re-sources in long run.
			Partial loading [125]	Prevent loading of entire dataset while reducing WET. Low DLT. Improve QET for frequent queries.	Partitioning dataset for hybrid systems. NP-hard [125]
3	Interactive Data Ex-ploration	Explore parts of the dataset needed by work-load/users.	Visual ex-ploration of raw data [44, 43]	Incremental ex-ploration based on user inputs. Faster re-sponses at the cost of accuracy.	Approximate Answers. Results are not 100% accurate.

4	Heterogeneous Raw Data	Query different raw data formats without loading.	Code generation. Main memory caching. [70, 33, 39, 43, 125]	Developed in-situ engines that can query one or more raw formats like CSV, JSON, XML, text, or HDF5.	Fewer open-source tools that can query all kinds of raw formats.
5	Analytics	Faster analytics on raw data.	Caching and processing attributes using column stores or arrays [66, 73, 29, 53]	Store required data into column stores. Improves QET. Reparsing eliminated or reduced.	Identifying hot partitions for loading and caching is challenging. Minimizing the cost of performing analytics.
6	ML on Raw	Apply ML to solve aggregation, sampling, and other issues.	Machine Learning [43, 54, 110]	Aggregation, Sampling Query approximation Resource management	Data aging & deletion. Knowledge retention.
7	Resource Monitoring	Monitor resources of individual workload tasks.	Integration of resource monitoring tools. [56, 48, 46]	Find the relationship between processes and resources. Obtain useful patterns. Explore tools.	No inbuilt resource monitoring features in most DBMS.

8	Optimizing Required Resources	Utilize minimal resources to complete given workload tasks.	Optimizing operations on raw data by loading only required data & managing processed data. [56, 29, 53, 125].	Process only required data. Reduce resource utilization. Reduce WET.	Distribution of hot-cold data in single or Hybrid systems. Multi-format data management.
9	Maximizing Resource Utilization	Maximize utilization of available resources to reduce WET.	CPU, RAM & IO resource maximization. [39, 53, 113, 117, 98, 103]	Reduce WET. Efficient utilization of available resources.	Task scheduling. Resource Allocation. [114]
10	Cloud based Systems	Cost effective scheduling and resource allocation.	Serverless [108, 99, 103, 106] Cost aware Resource Allocation	Reduce application running costs. No hardware cost. Serverless – pay per query. Pay as per resources used.	Resource monitoring & allocation of individual tasks. No inbuilt features to configure resources for each query.

3.12 Open Research Issues

This section discusses open research issues based on the classification, existing tools, and literature survey. Table 3.3 compares the existing tools and techniques developed by researchers to process raw data. Table 3.4 lists implementation problems and algorithm complexity of identified open research issues based on existing tools and literature survey.

The existing tools and techniques have been compared in Table 3.3 based on features like data storage format, tool type, partitioning technique, data loading, replication, caching, and limitations faced while processing raw data. SCOPE [51], NoDB [33], Slalom [94], PDC [117], PCC [103] processes raw or semi-structured data. Slalom is an improvement over NoDB with logical partitioning, but it is not open source. Invisible loading [29], SCANRAW [53] and Partial Loading [125] techniques are examples of hybrid systems with DBMS data loading operations to reduce reparsing. PCC allocates specific resources to each query at run time. This feature is not provided by most existing systems. However, PCC lacks partial loading and caching features. PCC, PDC and SCOPE are distributed systems with the ability to process raw data stored on multiple sources or cloud storage. - However, most systems do not include important features like resource monitoring, partitioning, or query specific resource allocation required to process raw data efficiently.

Research Silos: Current solutions lack a single raw data management system having all the features. Researchers are working in silos and keeping their work private. The developed techniques are suitable only for specific hardware, DBMS, or Hybrid system combinations. Improving such work is very challenging. Everyone needs to combine their efforts and develop a system that automatically selects the best technique to avoid reparsing based on application requirements. The automated analysis of application requirements and changing the data arrangements or storage structures using the best possible technique is still in its initial stages [65]. Cloudera combined multiple open-source data management systems. It provides most features, including raw data query processing and

Table 3.3: Comparison of Existing Tools and Techniques

#	Tools/ Technique	Data Storage	Tool Type	Partiti- oning	Data Load- ing	Repl- icati- on	Cach- ing	Limitation
1	SCOPE [51]	Distribu- ted Raw	In-situ	Yes	No	Yes	No	Reparsing
2	NoDB [33]	Raw	In-situ	None	No	No	Yes	Reparsing
3	Slalom [94]	Raw	In-situ	Yes, Logi- cal	No	No	Yes	Reparsing, Not Open source.
4	Invisible Load- ing [29]	DB, Raw	Hybrid, (In-situ, DBMS)	None	Yes, Incre- mental	Yes	No	Prone to load entire dataset.
5	SCAN RAW [53]	DB, Raw	Hybrid, (In-situ, DBMS)	None	Yes, Incre- mental	Yes	No	Prone to load entire dataset. Not Open source.
6	Partial Load- ing [125]	DB, Raw	Hybrid, (In-situ, DBMS)	Yes, VP	Yes, Partial	Yes	No	High AET of Cost calculation
7	PDC [117]	Object Data	ODMS	Yes, HP	Yes, Meta- data	Yes	Yes	Underutilized Resources.
8	HTAP [106]	DB	Row & Col. store DBMS	No	Yes	Yes	Yes	High DLT – 100% replication.
9	PCC [103]	Distribu- ted Raw	In-situ	Yes	No	Yes	No	Reparsing.

arranging data for different workload requirements. However, the Cloudera is bulky and requires more resources than traditional DBMS and In-situ engines. It also lacks the ability to select the best data management systems or a combination of systems for the given workload. There is a need for a lightweight system that can efficiently handle raw data without human intervention.

Table 3.4: Summary of Open Research Issues

#	Open Research Category	Open Issues	Algorithm Complexity
1	Research Silos	No single tool has all the required features.	-
2	Resource Utilization	No inbuilt features to monitor resources used by workload tasks in most DBMS. Lack of standard resource monitoring frameworks. Resource-efficient task scheduling.	Task Scheduling, NP-hard [114]
3	Multi-level Partitioning	Manage raw data for multiple tools and maturity levels. Resource aware database partitioning & distribution. Task scheduling. No reliable open source tools.	Data partitioning, NP-hard [125, 36]
4	Data Aging & Deletion	No standard open source tools. Hard to confirm which data is safe to delete [89]. Lossless aggregation and summarization of data.	Data aggregation, NP-hard [63]
5	ML on Raw Data	Approximation, optimization, resource allocation, and many other problems can be solved using machine learning. Few DBMS & In-situ engines use ML to solve complex issues.	Approximation, Optimization, Resource allocation, NP-hard [114]

Resource Utilization: Techniques considering real-time resource availability for cost effective processing of application workload are exceptional.

Optimization & Maximization: Incremental loading, indexing, or caching techniques have been developed to elevate upfront utilization of resources and repetitive resource utilization while processing raw data [53]. The partitioning and sampling methods reduce the amount of data to be processed, reducing resource utilization [125]. However, it has been observed that most traditional and modern DBMS systems cannot utilize all available resources [32]. The query performance and resource utilization cost have been considered to reduce data processing costs [103]. Therefore, systems capable of optimizing and maximizing resource utiliza-

tion that considers real-time resource availability needs to be developed to reduce workload processing costs.

Cost estimation: Different applications have different dataset sizes and query types that require different amounts of resources based on the data processing tools used. It is challenging to efficiently utilize available resources while distributing data among in-situ engines and DBMS to reduce total workload execution time. L.Viswanathan et al. [118], and W. Zhao et al. [125] have proposed to use cost functions that balance resource utilization and workload execution time to reduce data processing costs for cloud based systems. Cost estimation is another challenging open issue where cost functions can be improved using multiple parameters like data format, distribution of the partitions, resource utilization, and operational costs.

Multi-format Partitioning: The partitioning data stored in multiple formats increases the complexity of partitioning algorithms which traditionally require all data in a single database format. W. Zhao et al. have proposed partitioning data between DBMS and Raw files at disk level [125]. However, the partitioning techniques that consider the distribution of hot data among high-speed storage devices like RAM and SSD in database format, while cold data on cold storage devices in raw format considering availability of resources are limited [49]. Repartitioning and distributing hot and cold data is challenging due to increased storage layers and data formats to adapt to workload changes.

Data Aging & Deletion: The data generation speeds have increased tremendously in the last decade. Earlier, it was possible to store and maintain generated data without incurring high data maintenance costs due to smaller dataset sizes. The present data age is focused on keeping the data in permanent storage devices and analyzing it later to gain more knowledge. However, processing and maintaining historical data generated by applications is becoming challenging due to increased maintenance requirements in terms of resources and costs for large datasets [89]. Most traditional and modern tools do not provide data summarization or compression features with automatic deletion of unwanted data while preserving knowledge.

ML on Raw Data: S. Baunsgaard et al. have applied ML models to clean and preprocess the raw data. However, applying ML models to every raw data query processing step is an open issue [43]. Researchers can apply ML techniques to process raw data files considering multiple features, like sampling size, error estimation, partitioning, and available memory to identify frequently accessed raw data for partial loading or caching in main memory. Applying ML to solve statistical heterogeneity, estimating query resource requirements, and scheduling allocation of resources to reduce raw data processing costs are hot research topics [103].

CHAPTER 4

Thesis Overview: Resource Utilization for Raw Data Query Processing

This chapter discusses the thesis idea and objective of the work. The primary objective of the thesis work includes NP-hard problems like partitioning, optimizing required resources, task scheduling, and resource allocation for efficient workload processing. The thesis approach section discusses our approach to process the given workload by keeping resource utilization optimal and utilizing existing resources effectively. The thesis objective section discusses tasks to complete in each phase to reach an effective solution.

4.1 Thesis idea

Researchers have proposed indexing, sorting, data compression, partitioning, blocking, and summarization techniques to process application workload faster. The task scheduling and resource allocation techniques try to utilize existing resources efficiently to reduce the WET. The core idea behind all existing techniques can be categorized into three approaches; 1) Reduce operations on data, 2) Reduce the amount of data, and 3) Efficient utilization of resources to perform required operations.

The first category conveys reducing operations required to generate results. This category contains data partitioning, data blocking, indexing, and sorting strategies [94, 125]. These strategies reduce the operations during query execution time by utilizing preprocessed or prearranged data resulting in faster query

processing. However, the entire dataset needs to be preprocessed or loaded into DBMS at least once. Additionally, the data preprocessing tasks are resource and time consuming. The second approach reduces the amount of data being processed during data loading, and query execution time to provide approximate results [83], [35]. These applications use sampling approaches to approximate query results. Therefore, the results are not always accurate. The third category tries to utilize available resources efficiently to process data faster to reduce WET. The task scheduling and resource allocation techniques that optimize or maximize the utilization of available resources can be added to this category [106, 53]. The incremental loading, indexing, partitioning, and sorting techniques used to reduce the upfront utilization of resources are also efficient resource utilization techniques [94, 29, 101].

Modern applications require handling a huge amount of data coming from multiple sources. It is clear from the literature survey that storing data in raw format is the most efficient way of storage. The in-situ/raw engine can execute queries on raw data directly to reduce data to result time. On the other hand, traditional row or column store DBMSs are optimized to answer queries faster on loaded data. Researchers have been trying to reduce data loading time using incremental loading techniques, but it is still costly. It can be derived that in-situ engines or traditional DBMSs alone cannot process the raw data efficiently. Therefore, hybrid systems have been developed to reduce data to query time QET and improve QET for repeating queries [29, 53]. The hybrid approach uses in-situ engines to query raw data directly to reduce the initial data to query time while incrementally loading data into DBMS to eliminate reparsing and improve QET.

Researchers have proposed several efficient data processing techniques by identifying functional patterns of the data processing operations and their relation with resources. For example, in-situ engines do not access IO once all the required data is cached in the main memory [53]. It allowed researchers to propose speculative loading techniques that utilized idle resources to load data in parallel. Thus, monitoring resources and analyzing experimental data can shed light on many hidden patterns which can be used to develop efficient raw data query

processing techniques. Therefore, the proposed thesis work focuses on observing monitoring resources for hybrid systems to develop resource efficient techniques to process the data.

4.2 Thesis Approach

Managing resource utilization is one of the most important topics because most proposed techniques desire to efficiently utilize CPU, RAM, and IO to improve WET. The efficient utilization of resources also reduces the application running costs. The thesis proposes to build a resource and workload aware framework to find resource-efficient way of processing raw data. This section summarizes the thesis approach based on the literature survey. Subsection 4.2.1 justifies why a hybrid system needs to be considered to utilize optimal resources to process a given dataset. The proposed system plans to partition the dataset using appropriate partitioning methods so that only relevant parts of the dataset can be accessed. Subsection 4.2.2 discusses partitioning approaches suitable for the proposed hybrid system. The chosen approach and thesis phases are discussed in Subsection 4.2.3.

4.2.1 Hybrid System

The traditional way of processing data requires the entire dataset to be loaded into DBMS, increasing resource utilization and data-to-result time. While, in-situ engines allow querying data without loading it, saving upfront resource requirements. However, query execution time is much higher in in-situ engines than in DBMS due to raw data access causing reparsing. Literature survey discovered that hybrid systems containing an in-situ engine and traditional DBMS could provide the best features of both. The in-situ part can facilitate querying raw data to reduce data to query time, while the DBMS part of the hybrid system can retain the processed data for future queries. The literature survey lists many research works that developed novel data processing flow to benefit from both in-situ and DBMS approaches. For example, SCANRAW proposes to query raw data using

in-situ engines to reduce data to query time and load data into DBMS in parallel whenever resources are available to improve QET of future queries [53]. This work utilized idle resource time to process the dataset efficiently, which can reduce additional hardware requirements. Therefore, it is important to monitor and analyze resources utilized by workload tasks in real-time for cost-effective management of workload.

4.2.2 Partitioning Approaches for Hybrid System

Partitioning a given dataset to utilize optimal resources for the hybrid systems is an NP-hard problem [125]. Similar to partitioning, other optimization problems in data management like task scheduling, frequent item set mining, and resource allocation do not have polynomial time solutions [114, 36, 63]. M. T. Ozsu and P. Valduriez have developed greedy and heuristic algorithms to solve problems faster to approximate near-optimal solutions [95]. Greedy algorithms choose the local best option at the decision time and hope to achieve a globally optimal solution. For example, incremental loading algorithms load all the attributes observed in workload analysis to reduce QET [29]. However, the time required to load attribute columns into the database and execute queries may exceed answering those queries using in-situ engines. Heuristic algorithms employ strategies derived from previous experiences to find optimal solutions. For example, a heuristic algorithm compares the cost of loading attributes plus executing queries on the database with the cost of answering queries using raw files to find an improved solution [125].

4.2.3 Heuristic approach

This thesis chooses a heuristic approach to partition, schedule tasks, and allocate resources for resource efficient processing of a given workload. The heuristic approach includes learning from past experiences to group or split schemas to obtain near optimal solutions to the partitioning problem in pseudo polynomial time [95].

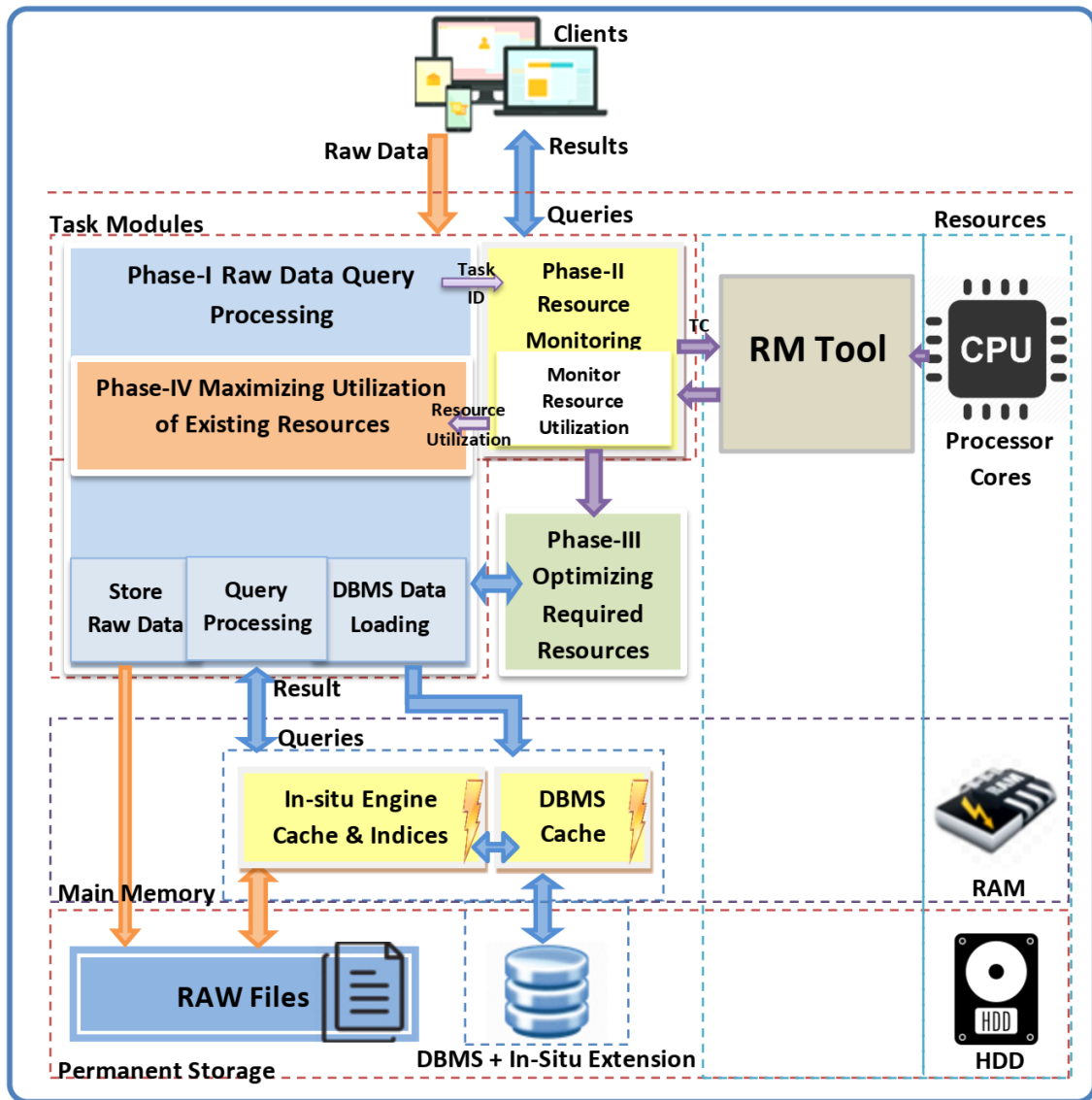


Figure 4.1: RAW-HF: Initial Architecture

The thesis work has been divided into four phases. The first two phases develop a raw data query processing and resource monitoring framework to quantify the time and resources required by in-situ engine and DBMS to complete given workload tasks. Analyzing the initial experiment result might find valuable facts and resource utilization patterns that can be used to optimize resource utilization. Phase-III optimizes the resources required to process the given workload. This phase partitions the dataset to process only the required data relevant to the workload. The fourth phase maximizes the utilization of existing resources, thereby taking care of any possible underutilization of resources. The Phase-III and IV algorithms follow heuristic approaches to optimize required resources and maximize utilization of existing resources, considering the knowledge gained from experiments performed during Phase-I & II of the thesis. Figure 4.1 shows an initial architecture of the Resource Availability and Workload aware Hybrid Framework (RAW-HF) that we propose to build.

RAW-HF has been developed incrementally. Each phase discussed below has been integrated into RAW-HF as a module. It can be seen in Figure 4.1 that each module is connected with the other. The Phase-I module executes the workload and communicates the currently running tasks ID with the Phase-II Resource Monitoring module. This allows the Resource Monitoring module to relate the resource utilization information to workload tasks in real time. Phase-III & IV modules use the resource utilization values obtained from the Phase-I & II modules to develop the resource optimization algorithms or decide on how to schedule tasks and allocate resources to maximize utilization of existing resources. The goals achieved in each phase of the thesis are summarized here.

- **Phase-I Raw Data Query Processing (RQP):** Phase-I develops a raw data query processing (RQP) framework. It is a hybrid framework consisting of an in-situ engine and DBMS.
- **Phase-II Resource Monitoring (RM):** Phase-II develops the resource monitoring framework to monitor resources used by workload tasks. Further, the resource monitoring framework is integrated with the raw data query processing framework developed in Phase-I.

- **Phase-III Optimizing Required Resources (ORR):** This phase optimizes required resources using partitioning and distribution strategies to handle raw data efficiently for the hybrid framework. The algorithms developed in this Phase-III are based on Phase-I & II result analysis and resource utilization patterns.
- **Phase-IV Maximizing Utilization of Existing Resources (MUER):** This phase maximizes the utilization of existing resources for the given workload. The technique developed in this Phase-IV addresses the issue of any possible underutilization of available resources.

4.3 Thesis Objectives

- “Find resource efficient way of processing raw data while improving total workload execution time WET utilizing available resources.”

The primary objective of the thesis work is to reduce workload execution time by utilizing resources efficiently. Efficient utilization of available resources is important because it directly affects application running costs. For applications deployed in in-house servers, the hardware resources are limited. On the other hand, cloud deployed applications can have virtually unlimited resources, increasing application running costs. Therefore, efficient resource utilization is also crucial for cloud-based systems. The thesis work needs to find a sequence of data processing steps, architecture, data partitioning, data distribution strategies, and many essential details to complete the given tasks utilizing optimal resources.

Finding the best partitions, scheduling tasks, and allocating resources for efficient processing are well-known NP-hard problems [114, 36, 63]. The NP-hard problems can be solved by using the heuristic approach in polynomial time. The solutions obtained using the heuristics approach are not the best, but applying factual knowledge of past experiences can improve the accuracy of the solution. Therefore, basic experiments must be performed using appropriate data processing tools to obtain valuable facts and achieve near-optimal solutions faster.

The thesis work proposed a Resource Availability and Workload aware Hybrid Framework (RAW-HF) to process given workload on raw data efficiently. RAW-HF development has been divided into four phases for incremental development and sub-tasks accomplished in each phase. The sub-tasks assigned to each phase have been listed below to understand the flow of thesis work. The first two Phases-I & II need to identify experimental facts about how raw data get processed and resources utilized by in-situ engine and DBMS. The optimization and maximization phases use the knowledge gained from experimental result analysis of the first two phases to develop efficient raw data partitioning, task scheduling, and resource allocation strategies.

- **Phase-I Raw Data Query Processing:** Develop a raw data query processing framework to query raw data using state-of-the-art in-situ engine and DBMS tools to identify the total time required to execute the given workload.
 - Query raw data using the in-situ engine.
 - Load data into DBMS and execute queries using the database.
 - Build a raw data query processing framework to perform the given workload tasks automatically using required data processing tools, i.e., in-situ engine or DBMS.
- **Phase-II Resource Monitoring:** Monitor the resources required by the in-situ engine and DBMS to execute the given workload.
 - Find and implement resource monitoring tools.
 - Integrate resource utilization information with the workload task IDs for better analysis.
 - Update the raw data query processing framework to automatically record the resources used by workload tasks for different data processing tools.
- **Phase-III Resource Availability and Workload aware Hybrid Framework (RAW-HF) - Optimizing Required Resources:** Find best data partitioning and distribution strategies to utilize optimal resources to complete workload tasks.

- Identify hot and cold data.
 - Find optimal partitioning and distribution strategy to minimize WET.
 - Distribute dataset between in-situ engine and DBMS for efficient processing.
- **Phase-IV Resource Availability and Workload aware Hybrid Framework (RAW-HF) - Maximizing Utilization of Existing Resources:** Maximize the utilization of available resources to reduce WET.
 - Identify the effect of each resource maximization technique on data loading and query execution operations.
 - Find best strategies to schedule workload tasks by assigning maximum available resources for faster completion of workload.
 - Automate task scheduling and resource allocation.

CHAPTER 5

Resource Monitoring Framework for Raw Data Query Processing

This chapter covers Phase-I and II of the thesis. It explains the building blocks of the resource monitoring framework for raw data query processing. The first phase proposes developing a raw data query processing framework to query raw data and data loaded into DBMS. The framework developed in the first phase recorded the time required by in-situ engine and DBMS to complete the given workload. It could not identify the CPU, RAM, and IO resources used by the workload tasks. Most data processing systems do not provide resource monitoring capabilities because they do not require resource information to complete most general tasks. Phase-II proposes to develop a resource monitoring framework that can record resources used by all the processes running in a system. The recorded readings need to be associated with the workload tasks for better analysis. Therefore, the resource monitoring framework has been integrated with the raw data query processing framework to develop a complete framework. This combined framework is called Resource Monitoring Framework for Raw Data Query Processing.

5.1 Phase-I Raw Data Query Processing (RQP)

This phase proposes to develop a general purpose raw data processing framework. The raw data query processing framework combines the functionalities of in-situ engine and traditional DBMS. This framework uses multiple systems therefore, it is called Hybrid framework. Section 5.1.1 discusses existing hybrid

systems to justify why a hybrid system consisting in-situ engine and DBMS is used.

5.1.1 Hybrid systems: In-situ engine & DBMS

The traditional way of data processing requires the entire dataset to be loaded into a database. The time required to load data into a database is known as data loading time (DLT). Most DBMS systems can be categorized into these three stores; 1) Row store – stores each record’s data together, 2) Column store – stores attributes or one column’s data together, and 3) Hybrid stores – these stores either use two DBMS system like HTAP systems, or Hybrid based on application workload requirements [60]. Database management systems have existed for half a century [41]. They have evolved to answer queries faster. Earlier data generation speed was not high, so high DLT was an affordable setback. Nowadays, advanced sensors, smart devices, and an increased number of people and machines generate huge amounts of data that traditional DBMSs cannot handle with traditional bulk data loading methods. Researchers have developed distributed systems that utilize the resources of multiple machines to handle large volumes and high-speed streaming data. However, more resources increase application running costs. To reduce the DLT, researchers proposed storing data in raw format.

In-situ or raw engines have been developed to query raw data directly to eliminate data loading requirements. These engines eliminate the need to load data into DBMS. However, advanced in-situ engines with DBMS like features have existed for less than 10-15 years [33]. The in-situ engines require significant time to query raw data because they need to extract, tokenize, and parse the raw data to answer the queries increasing query execution time QET. NoDB [33] and Slalom [94] have developed efficient caching, partitioning, and indexing strategies to cache the processed data to reduce the QET for future queries. However, the reparsing persists for datasets larger than the main memory size of a system. Therefore, hybrid systems have been developed that load data processed by in-situ engine into DBMS to eliminate the reparsing. Such hybrid systems can reduce upfront resource requirements, reduces data to query time, and improves

QET gradually. Therefore, a hybrid system consisting of in-situ and DBMS is chosen, because the positive features of both systems can help in finding resource efficient approach to processing raw data. For the remaining chapters, hybrid system term will refer to the system consisting in-situ engine and DBMS.

5.1.2 Raw Data Query Processing Framework

This section discusses the functional requirements and components of the raw data query processing framework. Basic query execution tasks can be done directly using an in-situ engine or DBMS. However, building a framework allows us to change the underlying DBMS or in-situ systems without changing functionalities or algorithm logic. Additionally, functionality like recording QET and DLT can be added. The framework executes list of workload queries automatically, rather than executing them manually one by one. This allows us to perform more experiments easily.

Some examples of hybrid systems include Invisible loading [29], SCANRAW [53]. Invisible loading technique used Hadoop with map-reduce jobs as an in-situ engine and MonetDB as a traditional column-store DBMS to process raw data [29]. R. Borovica-Gajic et al. and Y Cheng et al. also implemented in-situ extensions into prototype systems like DataPath [49] and SCANRAW [53]. The NoDB philosophy has been implemented in the traditional database management system PostgreSQL (PgSQL) and created PostgresRAW [33]. PostgresRAW is one of the few complete raw engines providing all DBMS features, including executing SQL queries directly on raw CSV files. PostgresRAW is open source and available on GitHub [11].

5.1.2.1 Functional Requirements of Hybrid systems

Modern applications reduce DLT by storing streaming data in raw format and loading them into DBMS later to improve QET [29, 33, 53, 55, 88, 99, 125]. The raw data query processing framework should be able to query raw data while retaining processed data for future queries. The processed data can be retained using two methods. 1) Caching processed raw data in the main memory. 2) Load

required data into databases. Main memory is a limited resource. Large datasets cannot be cached entirely into main memory leading to raw data reparsing. Therefore, the framework must be capable of handling basic tasks like loading raw data files into DBMS and querying loaded data. The in-situ engine can be an extension to DBMS or two separate tools that allows query processing on raw files with processed data caching and DBMS data loading features. Some hybrid systems may allow executing queries on data existing in multiple formats i.e. raw formats (CSV, JSON, XML), and databases. Another feature allows executing join queries on data existing in different formats. This feature is referred to as Multi-Format (MF) join in Table 9.4.

5.1.2.2 Proposed General Framework

This section explains the components of the proposed raw data query processing framework. Figure 5.1 shows the basic framework to handle raw data files and databases. The framework should collect data from clients, sensors or smart devices and store it in a CSV file format. The raw data collected in CSV files can be loaded into the database faster using COPY method compared to bulk load, transaction, and prepared SQL statement methods [56]. The framework requires a hybrid system consisting of an in-situ engine and a traditional DBMS or DBMS with an in-situ extension to execute SQL queries on CSV files and databases. A hybrid system capable of executing join queries on data existing in raw files and DBMS is best to address this job. However, it is possible to implement the proposed framework using two separate data processing tools. The issue with using separate tools is that the queries will need all the required data in a single format that needs data replication. The raw data query processing module is responsible for handling raw data and query execution tasks on the implemented hybrid system. Each component of the proposed framework is discussed below to understand its role in processing raw data.

Raw Data Query Processing: This part of the system is the central part of the framework. This part must handle multiple tasks smoothly to allow storage and query execution on raw data and databases using a hybrid system. This compo-

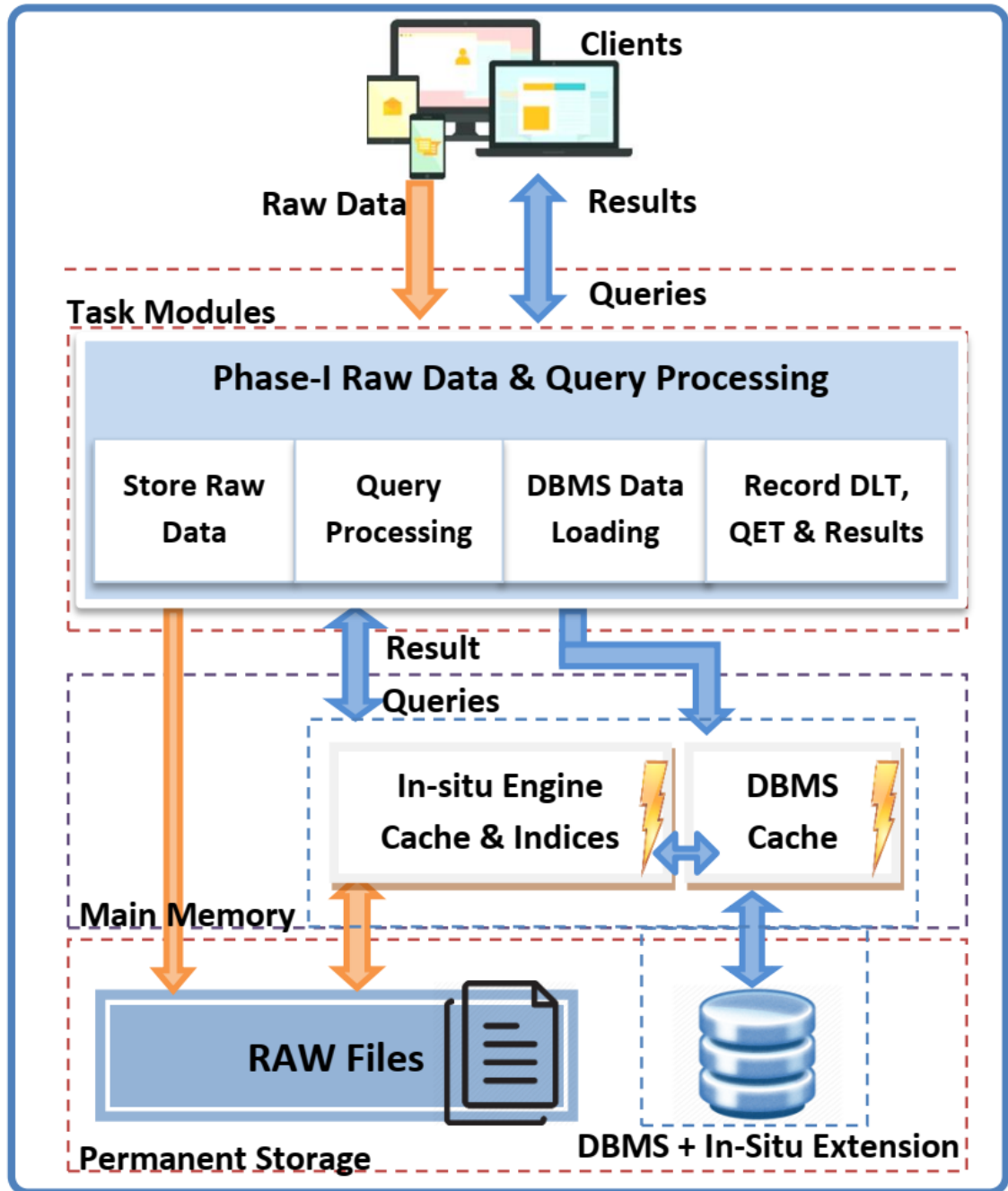


Figure 5.1: Raw Data Query Processing Framework

ment stores the streaming data from sensors into CSV files directly. The in-situ query engine uses the generated CSV raw files to answer the workload queries. This component must be able to handle data loading and query execution operations using the in-situ engine and/or database system when partial data is loaded into DBMS. The bulk loading number decides how many records are stored in buffer storage before saving them into a raw file or loading into a database. This bulk number can be set according to the streaming data velocity or query frequency .

Hybrid system: A state-of-the-art in-situ query engine and database management system can be used for this framework component. This component is responsible for executing queries on raw files and/or databases using a hybrid system. The in-situ engine executes the SQL queries directly on the stored raw data files. At the same time, the raw data can be loaded into the database parallel to raw data query processing. The proposed framework can be implemented using a hybrid system consisting in-situ engine and traditional DBMS. The hybrid system should be able to execute join queries on data existing in raw files and DBMS to avoid data replication. The experimental Section 8 discusses the implementation details of a hybrid system.

Main Memory: The main memory of the system is used to cache the entire raw files for faster processing. The in-situ query engine can parse, tokenize and filter the cached raw files faster. Some in-situ engines cache the processed data to improve the QET of future queries [39]. NoDB and Slalom have also proposed to build in-memory indexes to reduce QET [33, 94]. Most DBMS systems also cache the required database files to answer queries faster. The main memory can also be used as buffer storage to gather data for bulk loading.

Database & Raw Files: The raw data query processing module stores the application data in raw formats for faster storage. The raw data can be stored in semi-structured formats like CSV, JSON, or unstructured string raw formats. The raw data stored in the CSV files can be loaded into DBMS using faster data loading methods like COPY. The raw data query processing module is responsible for storing data in raw files and loading them into DBMS by executing COPY com-

mands.

5.2 Phase-II Resource Monitoring (RM)

Monitoring resources utilization of each workload task helps in understanding data processing steps and resources required by different raw data processing tools in depth. Identifying CPU, RAM, and IO resource utilization patterns can help optimize or maximize resource utilization. This section explains the role of resources in raw data processing, followed by the updated raw data query processing framework.

5.2.1 Role of Resources

Processing raw data needs all three core resources, CPU, RAM, and data storage devices (IO). Processing raw data using in-situ or raw engines requires storage resources to read stored data to answer queries. CPU resource performs data parsing, conversion, and tokenizing tasks. The main memory caches the processed data required by the CPU for faster access. The traditional DBMS stores the processed data back to disk for future queries. When a query arrives, a query plan is made, which might require fetching data from the disk. If the required data is already cached in RAM, it can be processed faster. The time of storing or reading data from the permanent storage is directly affected by the IO speed of the storage hardware resource. The RAM resource provides the cached data to the CPU for processing. The time required to process query operations and generate results depends on the CPU processing speed. If the required data is not in RAM, the CPU has to wait for data to be fetched from the disk, known as a cache miss. The CPU cycles are wasted in waiting for IO completion.

Figure 5.2 presents the latency and cost relationship of different resources. CPU contains L1-L4 caches inside it. The speed of L1-L4 is very high, although the size of L1-L4 in range from 64 KBs to 8 MBs. Double Data Rate Synchronous Dynamic Random-Access Memory (DDR SDRAM) is main memory where all the data is cache for faster processing. DDR SDRAM is referred to as RAM or main

memory in following sections. Slow storage speed and smaller Cache and RAM size increases cache misses, and CPU has to wait for data from Solid State Drive (SSD) or Hard Disk Drives (HDD) referred as IO. The percentage of time waited by CPU for IO is referred as IO_wait. The higher the IO_wait, higher data to result time. The speed of processing large dataset usually depends on IO speeds as they are the primary reason for bottlenecks [56]. L1-L1 caches have not been monitored due to their smaller sizes. Network speed is not important for a single-machine setup because application data do not travel through the network. Cloud storage is also slower than single node setup storage devices as it adds time required to transfer data over network. GPU resource is similar to CPU but mostly used for processing graphical data. Therefore, resource monitoring framework proposed in this work only considered CPU, RAM and IO resources. Monitoring these resources allowed us to understand resource utilization patterns and propose an efficient way of processing the large datasets.

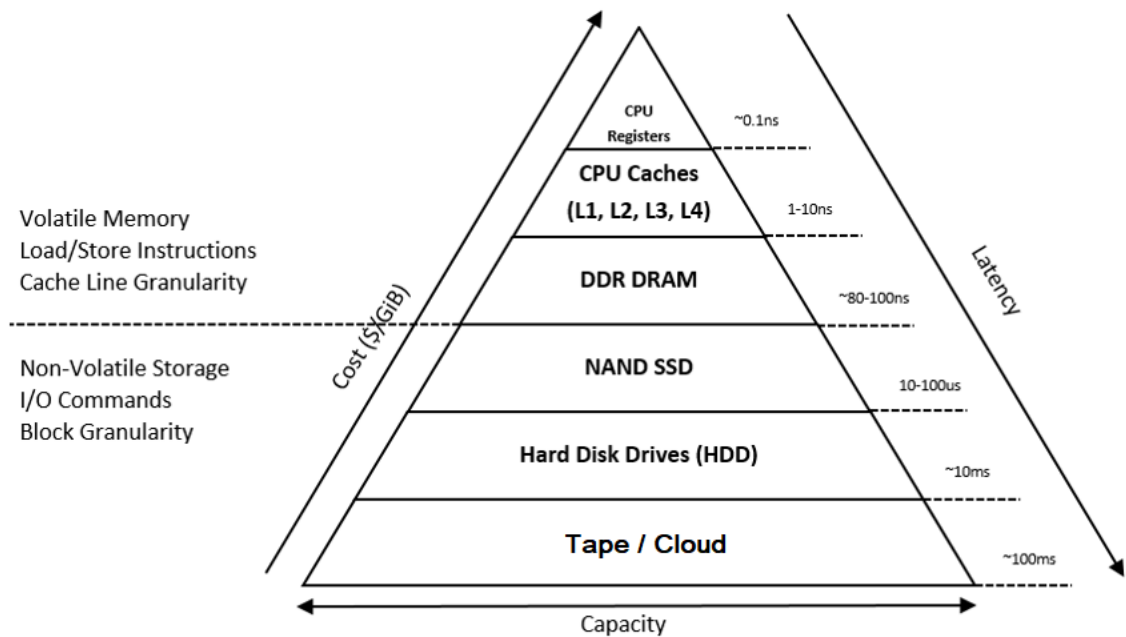


Figure 5.2: Resources

5.2.2 Resource Monitoring Framework

Resource Monitoring Framework consists of Resource Monitoring (RM) module and Resource Monitoring (RM) tools. RM module accesses the resource utilization data from system monitor, task manager, SAR, top, htop, iotop, or other tools by sending appropriate RM commands. RM tools can provide resource utilization details in GUI or text-based interfaces. The RM tools that provide text outputs should be chosen because the output will require less space, and the collected data can be easily filtered. RM data should be stored in CSV format for easy filtering using basic data processing tools like excel. Therefore, RM module filters and converts the resource utilization data received in string format to CSV. The RM module can send CPU, RAM, and IO resource monitoring commands to RM tools and collect output for further analysis. However, the monitored resource utilization records show only the process ids of the running processes. They cannot identify which workload task was running at that time. Therefore, the data processing tasks need to be associated with RM output. Therefore, the resource monitoring module must be integrated with an earlier proposed raw data query processing framework to achieve the correlation.

5.2.3 Integration of Resource Monitoring Framework

This section discusses integrating the resource monitoring framework and raw data query processing framework. The goal is to identify which workload tasks were running and associate resource utilization data to that particular task. The raw data query processing module of RQP framework has access to task IDs, while the RM module of the RM Framework collects the resource monitoring data. The RM module is comparatively smaller to integrate with RQP framework, so the RM module is added to RQP framework to achieve the goal.

The updated raw data query processing framework with a resource monitoring module can be seen in Figure 5.3. It can be observed that the RM modules send commands to the tools and receive the output. The received output is filtered to save monitored resources of the raw engine, DBMS, and framework processes.

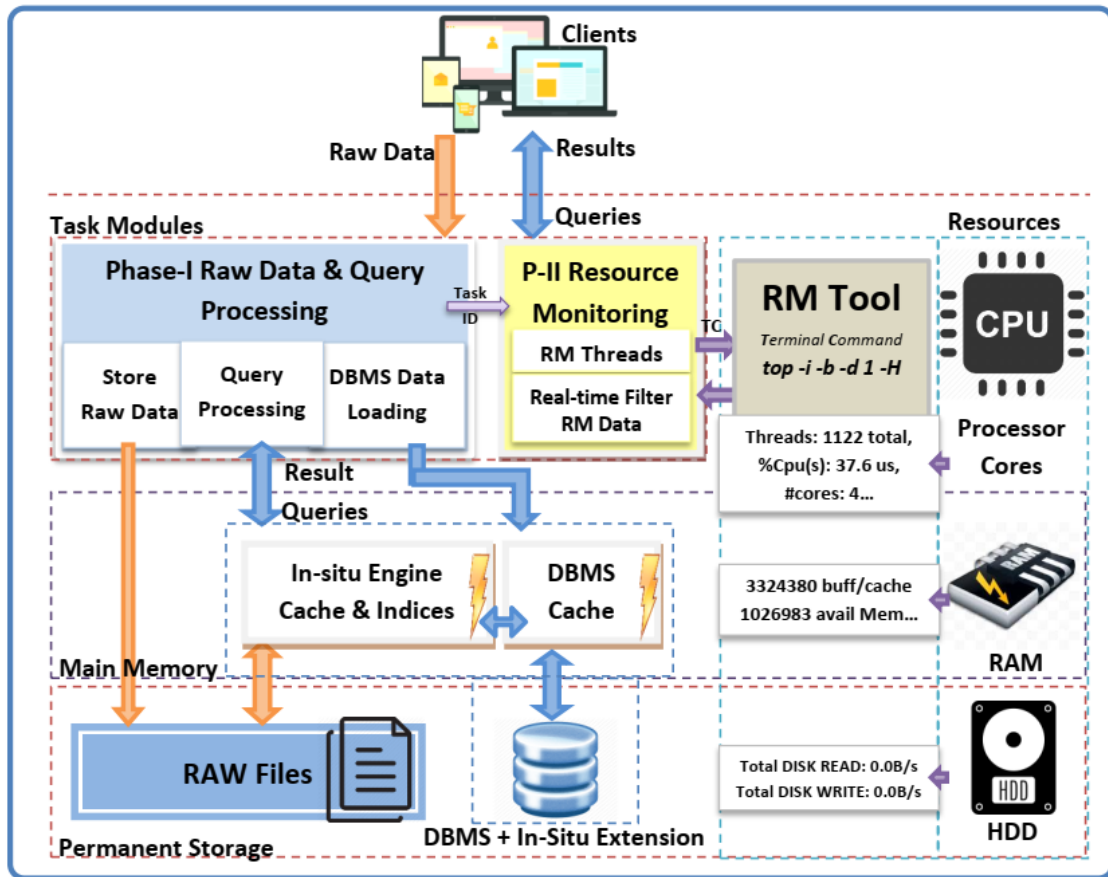


Figure 5.3: Resource Monitoring Framework for Raw Data Query Processing

The module also saves total resource utilization but filters out details of other processes. At the same time, the RM modules get task details in real-time from the streaming data and query processing module. The task details and monitoring results are then combined in a list of comma-separated strings format and saved to the output result file. The output is not loaded to a database to avoid using additional resources, a common practice used by developers. However, real-time decision making algorithms may require real-time analysis of recorded resource utilization.

5.2.4 Algorithms & Data structures

This section discusses experiment flow for resource monitoring experiments followed by pseudo-code.

5.2.4.1 Data Structures

The resource monitoring algorithm uses simple data structures like arrays and lists. The input workload file contents associate the task ID with task statements. Table 5.1 shows an example of a workload list (w_l) containing task ID and task statements. The workload file is read and kept in a `List<String[]>` and read one by one by the query processing functions. The output data is stored in a `List<String[]>` before writing to a CSV result file. Table 5.2 shows a sample output list. Figures 8.3 and 8.4 highlights all the important data collected from external RM tools. The list contains CPU, RAM, and IO utilization data of system and important processes. Basic metadata like data unit, read, or write details have been added to output list to identify the data later while analyzing output CSV file. It can be seen in Table 5.2 that the output list contains time to complete a task, CPU, RAM, and IO utilization monitored during a TRUNCATE Table (TRUN) task. The *Postgres* process CPU utilization is a single CPU core utilization, while memory utilization percentage is out of 100.

Table 5.1: Workload list (w_l)

0[T_ID]	1[Statement]
TRUN	"TRUNCATE TABLE PhotoPrimary;"
COPY	"COPY PhotoPrimary FROM '/.../PhotoPrimary.csv' (DELIMITER ',');"
Q0	"Select count(objid) from PhotoPrimary;"
Q1	"SELECT objID, ra ,dec FROM PhotoPrimary WHERE ra >185 and ra <185.1 AND dec >56.2 and dec <56.3 limit 100;"

Table 5.2: Result List

0	1	2	3	4	5	6	7	...
Q_ID	TRUN	Time	444	ms	#Rec.	0	star	
CR_T	TRUN	Used	3.9	Free	92.1	I/O	1.9	
CR_P	Postgres	1core	79.4	Mem	0.2			
IO_T	TRUN	Read	0.59	Write	0.02			
IO_P	Postgres	Read	0.50	Write	0.01	I/O	0.2	

5.2.4.2 Algorithms

This section explains the flow of raw data query processing and resource monitoring algorithms. These algorithms run in parallel and share important information with each other to relate and record output data for better analysis. Pseudo-codes of Algorithm 1 Raw Data Query Processing and Algorithm 2 Resource Monitoring are also presented in this subsection.

Algorithm 1 Raw Data Query Processing (RQP): The raw data query processing algorithm is responsible for performing data loading and query processing tasks. This process sets the input and output file paths. It starts resource monitoring (RM) threads. The query processing thread reads workload tasks from given file and starts executing them on a hybrid system. This process sets and updates the currently running data processing task details in shared variables. The RM threads read those task details and incorporate them with RM output. Once the assigned task gets completed the algorithm stores the task ID and time required to complete that task in a result output file.

Algorithm 2 Resource Monitoring (RM): The resource monitoring algorithm executes in parallel using a new thread. This RM thread is responsible of executing resource monitoring commands on external RM tools. The RM tool commands are executed on the interface provided by OS terminal or external tools. The resource utilization output stream is read by these threads using buffer reader. The RM threads filter the received resource monitoring output line by line to find specific resource utilization details and save it with task information in a CSV file. The filter parameters are set to find lines that contain data processing tools and framework processes to record their resource utilization. Here, *Postgres* and *Java* processes have been filtered. The *Postgres* process represents NoDB (PostgresRAW) and PostgreSQL (PgSQL) data processing tasks, while *Java* processes are the framework tasks.

The RM output is not immediately stored on disk to avoid continuous usage of IO. The threads collect the RM information in the result list residing in the main memory before flushing to the disk. The list collects the data until a predefined threshold value is met, i.e., the number of records filtered. The final result

Algorithm 1 Raw Data Query Processing

Data: w_p = workload file path,
 r_p = result file path,
 $IsRM$ = Monitoring threads flag,
 $RMType$ = CPU & RAM or I/O RM types,
 $RMcommand$ = Monitoring command for tool,
 RM_F = Monitoring Freq.,
 $Task_ID$ = data loading or Query ID

Result: DLT, QET, Total WET & Result Count in CSV file

1. **RawDataQP** (w_p , $IsRM$, RM_F , r_p)
2. Set parameter values w_p , r_p
3. Set $RMcommand = top -I -b -d RM_F -H$
4. **If** ($IsRM == True$) then
 - #Initialize. RM Threads A
5. RM_Thread ($RMType$, $RMcommand$, r_p)
6. $RM_Thread.start()$
7. $DB.Connect()$
 - #start workload execution
8. **For** each task T in w_p do
9. Set static $Task_ID = T.T_ID$
10. Results = $T.Statement.Execute()$
11. Save time & Result count for each task
12. **End**
13. $RM_Thread.interrupt()$
14. **Return;**

Algorithm 2 Resource Monitoring

Data: *RM_OutPut* = stores RM tool output read from buffer
r_p = result file path,
RMType = CPU & RAM or I/O RM types,
RMcommand = Monitoring command for tool,
Result: Resource monitoring data with task ID

```
1. RM_Thread (RMType, RMcommand, r_p) ,
2.   Set local parameter values T_ID,
      #Execute RM tool command
3.   Execute RMcommand on external tools,
4.   List RM_OutPut = read resource monitoring output
5.   While RM_OutPut.readline is NOT NULL
6.     For each line in RM_OutPut
7.       Filter RM data from line
8.       RM_Info = Set T_ID to RM monitoring;
      #save data
9.       WriteRM(RM_Info, r_p);
10.    Run till interrupt from RawDataQP
11.    End
12. End
```

file contains the overall CPU, RAM, and IO utilization observed during raw data processing. The resulting file also contains *Postgres* and *Java* process resource utilization information. The raw data query processing process interrupts both RM threads when the data processing tasks are completed. A few seconds of delay can be imposed to store collected data on the disk.

CHAPTER 6

Resource Availability and Workload aware Hybrid Framework (RAW-HF): Optimizing Required Resources

This chapter discusses Resource Availability and Workload aware Hybrid Framework (RAW-HF) to optimize required resources. Section 6.1 discusses the motivation behind developing a hybrid framework and components of RAW-HF. The partitioning techniques used by RAW-HF to optimize resource utilization are explained in Section 6.2.

6.1 Resource Availability and Workload aware Hybrid Framework (RAW-HF)

Resource Availability and Workload aware Hybrid Framework (RAW-HF) uses a hybrid system containing an in-situ engine and traditional DBMS. The motivation behind choosing a hybrid system for RAW-HF is already discussed in Section 4.2, which was based on the literature survey. The important point of choosing a hybrid system is to reduce data-to-result time using in-situ engines and retain processed data using DBMS to eliminate raw data reparsing. Most existing systems propose to load raw data into DBMS to improve the QET of future queries [29, 53]. However, partitioning raw datasets for hybrid systems is an NP-hard problem [125]. W. Zhao et al. have proposed to load attributes based on a cost function that considers attribute loading time and time required to access data

from DBMS and raw files. However, the cost function was complex and costly to implement. The cost function did not consider the improved QET time of in-situ engines, which could cache the processed data for future queries.

Phase-I & II explored and implemented an open source hybrid system consisting of PostgreSQL (PgSQL) [22] and NoDB (PostgresRAW) [33]. The experiments performed using a real-world scientific dataset prove that traditional DBMS required significant time to load the entire dataset into DBMS. In comparison, the in-situ engine could answer many queries before DBMS could complete the data loading. However, the total workload execution time (WET) of in-situ engine was very high compared to DBMS. A detailed analysis of the time each workload query took showed that some queries had low QET in in-situ engines compared to DBMS. This led to the below given key discoveries that inspired partitioning techniques discussed in Section 6.2.

1. Simple queries can be executed using in-situ engines on partitioned raw files to reduce high QET of initial queries. While complex queries must be executed using DBMS because they are well optimized to handle such tasks.
2. In-situ engine and DBMS could not utilize the available resources completely with the default resource allocation settings.

The literature survey and Phase-I & II results showed that existing in-situ engines and DBMS utilize resources in processing data that workload queries never use. Therefore, a query workload analysis became a necessary part of the partitioning algorithm. Additionally, the proposed partitioning techniques utilize the knowledge gained from Phase-I & II results that discovered simple queries had low QET compared to DBMS when the in-situ engine had cached the required data in the main memory. The RAW-HF proposes to keep the simple query (SQ) partition always in raw format and improve QET for initial cold queries by accessing smaller raw partitions. Figure 6.1 displays a block diagram of RAW-HF to understand how proposed QCA (Query Complexity Aware) and WSAC (Workload and Storage Aware Cost-based partitioning) techniques partition the given dataset in Phase-III. The QCA partitions the dataset based on query complexity

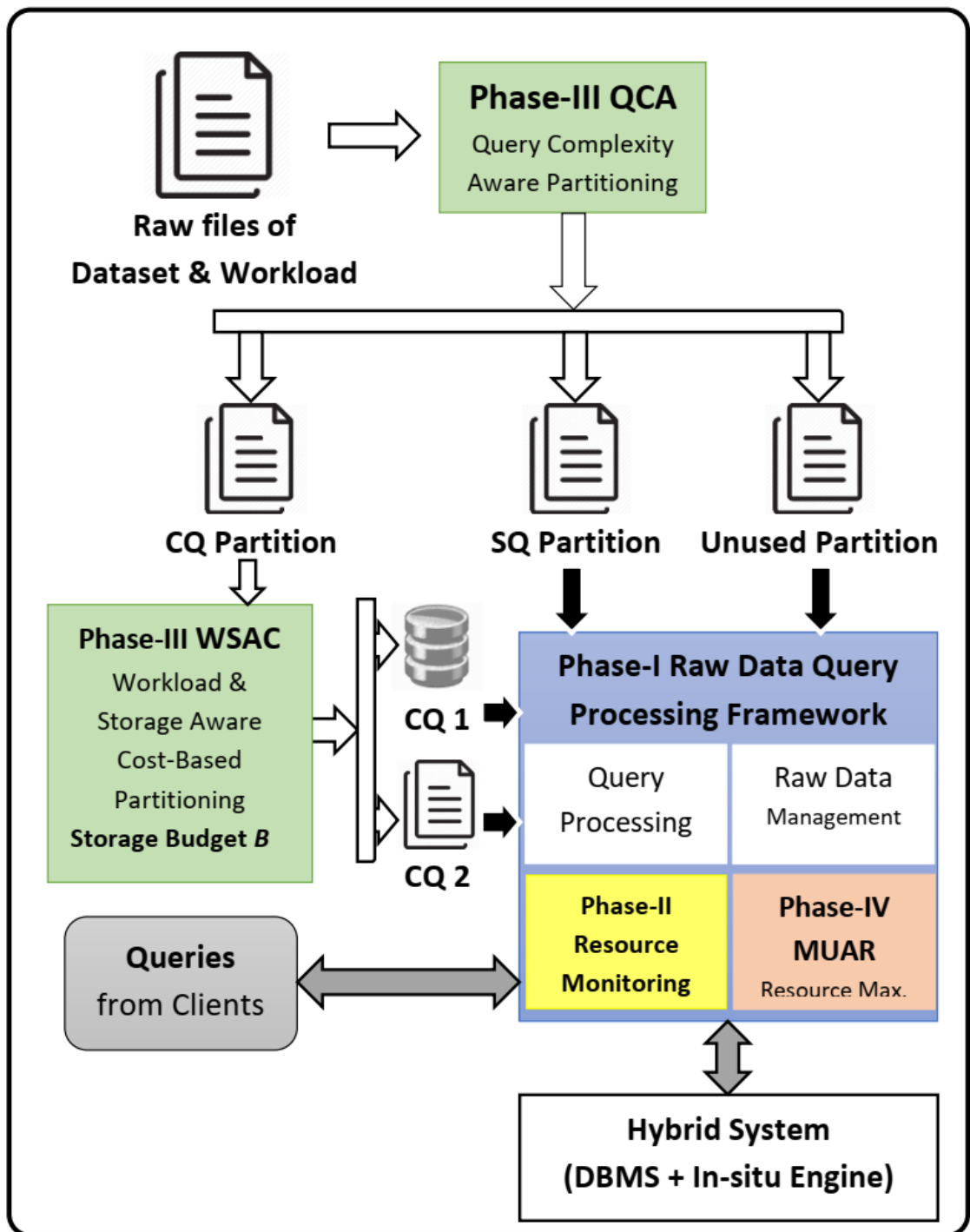


Figure 6.1: RAW-HF: Block Diagram

to create three vertical partitions: a simple query partition (SQ), a complex query partition (CQ), and an unused attributes partition. WSAC is used to refine the complex query partition when enough storage resources or main memory is unavailable to load all complex query attributes in DBMS or MMDB. The black arrows show the final partitions sent to resource monitoring and raw data query processing framework. More details on QCA and WSAC are discussed in Section 6.2.

The Maximizing Utilization of Available Resources (MUAR) technique developed in Phase-IV handles the task scheduling and resource allocation tasks. The Phase-I & II result analysis showed that CPU, RAM, and IO resources are underutilized. The MUAR maximizes the utilization of existing resources by executing workload tasks in parallel and allocating enough RAM resources to different types of workload queries at runtime. The lightweight algorithm of MUAR considers the query complexity to fine-tune the dynamic allocation of available resources for each query. MUAR receives the real-time availability of resources from the Resource Monitoring (RM) module. The RM module is also modified to filter the RM data required by MUAR in real-time. The MUAR technique is further discussed in Chapter 7.

6.2 RAW-HF: Optimizing Required Resources

The thesis work proposes two techniques to partition and distribute the application dataset to process the workload efficiently for hybrid systems.

1. QCA – Query Complexity Aware partitioning technique identifies the Simple Queries (SQ) and Complex Queries (CQ) using a lightweight algorithm to decide which attributes to load and which can be kept in raw format.
2. WSAC – Workload and Storage Aware Cost-based partitioning technique decides which attributes to load in the database when the attributes selected by QCA require more space than a given storage budget B.

The final partitioned table schemas are given to the raw data query processing module to create relational tables in DBMS and in-situ. The CSV file of original

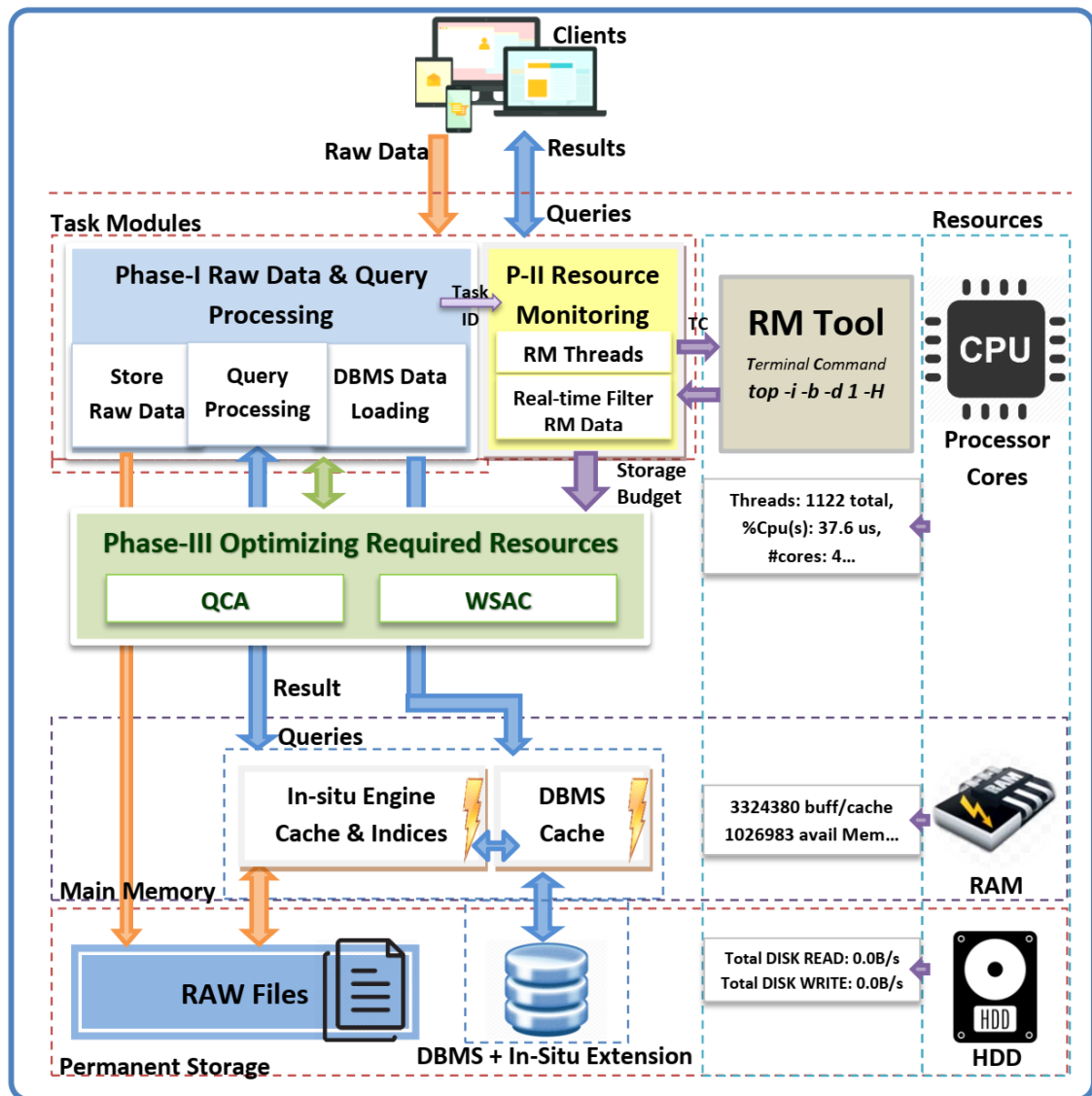


Figure 6.2: RAW-HF Architecture: Optimizing Required Resources

dataset is partitioned to create smaller CSV files for loading into DBMS and access using in-situ engine. The framework links the raw partitions with the in-situ engines to execute SQL queries. Figure 6.2 displays a detailed architecture of RAW-HF. Each phase is shown in different colors for easier identification of phases and where they fit in the entire framework. The optimization phase is shown in green. It provides the partition schemas to the raw data query processing module. The raw data query processing module of RAW-HF then loads the final complex query partitions into DBMS and links the remaining partitions to the in-situ engine. The hybrid system supports the execution of join queries on databases and raw files to avoid replication. However, all the replication cases are discussed in Section

6.2.1.3 of QCA partitioning technique Section 6.2.1.

6.2.1 Query Complexity Aware (QCA) Partitioning Technique

Row store databases and raw data processing engines need to read entire records stored inside a database or a file to answer the queries which require those specific records. Data partitioning techniques can improve the QET time by reducing the accessed data. For traditional databases, partitioning and re-partitioning are very costly operations. Whereas partitioning raw data is relatively less costly when considering frequent changes in the workload [94]. Therefore, RAW-HF proposes to partition the raw dataset files without loading them. This section discusses the Query Complexity Aware (QCA) technique to partition the raw dataset by analyzing only workload query statements. QCA tries to partition the dataset by considering the fact that simple queries (SQ) had low QET in in-situ engines. Therefore, QCA proposes partitioning the application dataset to keep the partition access by simple queries in raw format. In contrast, the complex query (CQ) partition should be loaded into DBMS for efficient workload processing for the hybrid system. If CQ partition is larger than the storage budget, the CQ queries are further given to WSAC to refine the database partition. The following sections discuss data structure and algorithm logic used by QCA.

6.2.1.1 QCA: Data structures

QCA algorithm uses simple lists and key-value dictionaries to store the query attributes, query complexity, and relevant partitions. QCA uses a schema dictionary, workload list, workload attributes, and query dictionaries generated using schema and workload extraction functions similar to WSAC. Table 5.1 shows workload list (*w_l*) populated by reading workload file. The Query Type (QT) dictionary stores query types identified by the query complexity identification steps. QT values are shown in Table 6.1. The value 0 means the query type is simple, and 1 for the complex type. *QT_P0* and *QT_P1* store list of attributes coming in simple and complex queries. Common attributes for both partitions are stored in the common attributes partition CAP list. *PC_Q0* and *PC_Q1* lists keep track of

partially covered queries based on CAP list.

Table 6.1: Query Type Dictionary (QT)

Key (Q_ID)	1	2	3	4	5	6	7	9	10	11	12
Value(Query Type)	1	0	1	0	1	0	0	1	0	1	1

6.2.1.2 QCA: Algorithm

The idea behind the QCA technique is to partition the dataset and distribute the workload in such a way that queries performing faster on a given tool can be allocated to that tool. Contrary to the HTAP systems, QCA tries to achieve faster query execution times with minimal replication and loaded data partitions.

The proposed workload and query complexity aware algorithm uses lightweight query identification and partitioning steps to reduce algorithm execution time (AET) compared to other cost based techniques [125]. QCA algorithm can be divided into three parts, 1) Query Complexity Identification (QCI), 2) Grouping of Attributes based on query classification (GRA), and 3) Identification of Partially covered Queries (PCQ). QCA algorithm first identifies the type of queries best suited for a given tool using initial results. The initial experiment results have been plotted in Figure 9.13. The analysis has shown that zero-join queries perform faster in raw engines than in traditional DBMS. While queries having multiple joins are slow in raw engines. Therefore, the technique classified the query workload into two query types. The first type is for Simple Queries (SQ), which contains zero join. The second query type includes the remaining one or more join queries. This second category of queries is called Complex Queries (CQ) in this paper.

The proposed QCA technique uses general attribute and table name extraction functions from schema and query workload to populate workload list, schema dictionary, and query dictionary. The QCA algorithm first identifies the SQ and CQ type queries stored in the workload list. The algorithm uses the simple logic of counting no. of tables present in the query statement. The query is classified as a complex query; if two or more table instances are found in the query statement. The technique also considers self-join queries as complex queries. Table

Algorithm 3 QCA Partitioning

Data: w_l = Workload List
 QT = Query Types Dictionary
 q_l = Query List
 que_d = Dictionary of Queries
 s_d = Schema Dictionary
Result: SQ-Raw, CQ-DB & CAP Partitions

```
# Query Complexity Identification
1. def QCI( $w_l$ ,  $que_d$ ,  $s_d$ ):
2.     For each task T in  $w_l$  do
3.         If T.Statement has multiple tables
4.              $QT[T.Q\_ID] = 1$ 
5.         Else
6.              $QT[T.Q\_ID] = 0$ 
7.         End
8.     Get  $QT\_P0$ ,  $QT\_P1 = \mathbf{GRA}(que_d, QT)$ 
9.      $CAP = QT\_P0 \quad QT\_P1$  #Common Attributes
10.     $QT2 = \mathbf{PCQ}(que_d, QT\_P0-CAP, w_l)$ 
11.     $QT3 = \mathbf{PCQ}(que_d, QT\_P1-CAP, w_l)$ 
12.    Repeat steps 8 to 12 For  $QT2$ ,  $QT3$ .
13.    Return  $QT^*$ ; #Return all QT partitions
```

```
#Grouping of Attributes
14. def GRA( $que_d$ , QT)
15.     For each query i in  $que_d$ :
16.         For each attribute j in  $que_d[i]$ :
17.             If  $QT[i] == 0$ 
18.                 Add j in  $QT\_P0$ 
19.             Else
20.                 Add j in  $QT\_P1$ 
21.         End
22.     End
23.     Return  $QT\_P0$ ,  $QT\_P1$ 
```

```
# Identification: Part. covered Queries
24. def PCQ( $que_d$ ,  $QT\_P$ ,  $w_l$ )
25.     $QT = [0]*w_l.length$ ; #Assign 0s
26.    For each query i in  $que_d$ :
27.        For each attribute j in  $que_d[i]$ :
28.            If j not in  $QT\_P$  list
29.                 $QT[i] = 1$  #New QT list
30.        End
31.    End
32.    Return QT;
```

6.1 shows the query ID and query complexity type updated in QT as key-value pair. The single table instance queries are classified as simple queries (SQ). The GRA function groups the SQ and CQ attributes in two different lists (QT_P0) and (QT_P1). The intersection of these two lists provides the list of common attributes partition CAP. The SQ partition is (QT_P0), and the CQ partition is (QT_P1) after the first round of QCA partitioning. The SQ partition can be stored in raw format, while the CQ partition needs to be loaded in DBMS.

The QCA algorithm can further refine the partitions based on output from partially covered queries PCQ list as new query type QT input. Steps 8-12 can be repeated until all workload queries get covered in QT2 union QT3 to further partition raw and database partitions. These steps reduce partition size and find new groups of queries covered by smaller partitions. The partition refinement benefits broad table datasets with a large number of distinct queries in the workload. For most cases, the first round of partitioning might be enough to partition a single table into the raw format for SQ queries (QT_P0) and database format for CQ queries (QT_P1). Here, CAP gets replicated in both partitions. The following section discusses no replication and replication data distribution cases for CAP partition.

QCA – Complexity: The complexity of the proposed QCA algorithm is $O(j^*(w_l))$. It can be seen from the pseudo code that the complexity of steps 2 to 7 is $O((w_l))$ because it loops for each query in the workload list. The complexity of GRA and PQC functions is $O(j^*(w_l))$. The j is the number of attributes in each query statement, and (w_l) is the count of queries. The remaining steps execute only one time $O(1)$. The total complexity of the algorithm is $O(j^*(w_l))$ which shows that the algorithm does not depend on the size of the dataset. The (w_l) count is the dominating factor in the equation because $j \ll (w_l)$. The (w_l) count can be reduced by clubbing together identical structure queries. The $O(j^*(w_l))$ complexity concludes that the QCA algorithm runs in polynomial time even in worst-case scenarios.

6.2.1.3 Data Distribution Cases

All the partitions received from the QCA technique having zero common attributes can be kept in their respective raw or database formats. For example, if a (QT_{P0}) raw partition covers 4 out of 5 simple queries with no attributes of the fifth query, it can be kept in raw format. This can also be interpreted as all attributes required by the fifth query would be in one or more common attribute partitions CAPs. For simplicity, consider that there is only one CAP partition. The CAP and remaining partitions can be arranged in five different ways. This section discusses all five cases with their advantage and disadvantages based on the location of CAP.

- **No Replication cases:** The cases discussed in this section do not replicate common attributes among raw or database formats. Therefore, a data processing tool capable of executing join queries on multi-format data is required.
 - *Case-I:* Keep the (QT_{P1}) which includes CAP in the loaded format. The remaining partition (QT_{P0})-CAP can be stored in raw format. This ensures that complex queries have the best query response times. However, simple queries requiring data from CAP will have to perform join with database partition (QT_{P1}), which might increase QET for partially covered queries.
 - *Case-II:* Keep the (QT_{P0}) which includes CAP in the raw format. This arrangement allows simple queries to execute without any joins. However, complex queries will require joining (QT_{P1})-CAP partition stored in database format with raw partition (QT_{P0}). This case has minimal attributes in the loaded format. Therefore, CQ queries may suffer from high QET.
 - *Case-III & IV:* In this arrangement, the (QT_{P0})-CAP stays as a raw partition, and the (QT_{P1})-CAP partition is loaded in the database. The difference between Case-III and Case-IV is the location of the CAP partition. The CAP partition can be either loaded into the database or kept in raw format. The benefit of this case is that all fully covered queries

that do not require the CAP partition may execute a little faster due to the smaller partition size. However, all partially covered queries PCQ have to be joined with CAP partition. The additional JOIN operation may increase WET time due to the high number of PCQ queries in Case-III & IV compared to other cases.

- **Replication Case:** The HTAP systems and most query workload balancing techniques require replication of data on all nodes. QCA technique limits the replication to only CAP partitions. Replication of CAP with raw and database partitions provides freedom to choose different tools for different format partitions which can not join different format partitions.
 - *Case-V:* The (*QT_P0*) and (*QT_P1*) partitions that cover all simple and complex queries have to be used in this case. Both partitions include common attributes CAP, which allows the execution of queries using a single partition with no additional joins. This case also eliminates internode communication in a distributed setup.

6.2.2 Workload and Storage aware Cost-based Technique (WSAC)

This section discusses the proposed Workload and Storage aware Cost-based technique (WSAC), which considers the query workload, read-write costs, and storage parameters to decide the optimal partitions of raw data files. Most algorithms provide two partitions of a table to distribute among database and raw engines [125]. The first partition is to be loaded in a database, and the other one is to be kept in raw format. However, the proposed technique suggests three partitions for big tables. The third partition is a list of attributes that do not get used by the workload and should be kept raw.

6.2.2.1 WSAC: Data Structures

Most techniques require the basic arrays, and list data structures. The proposed technique required 2D or more complex array data structures. We have used Python dictionary and nested dictionary data structure during implementation.

Python uses hash functions to map keys to values for dictionaries. Table 6.2 & Table 6.3 show data structures that are used in sub algorithms. The Schema Dictionary (*s_d*) shown in Table 6.2 stores list of attributes associated with tables. The instance of *PhotoPrimary* table from the SDSS dataset has been displayed for understanding. Table 6.3 represents (*que_d*), the nested dictionary of workload queries that contain tables and attributes used in each query.

Table 6.2: WSAC: Schema Dictionary

#s_d	Table_list	Attr_list
0	photoprimary	objid
		run,
		ra
		rerun,
		camcol
		field
		obj
...		

Table 6.3: WSAC: Nested Dictionary

#que_d	Q_ID	Table_list	Attr_list
0	1	photoprimary	objid
			run
			rerun
			...
1	2	photoprimary	objid
			type
			flags_r
			...

6.2.2.2 WSAC: Algorithms

This section describes sub-functions used by WSAC technique; 1) Attributes & Entities Extraction, 2) Cost Function, 3) Query Coverage (QC), and 4) Attribute Usage Frequency (AUF). After extracting attributes, the cost function finds the access costs of workload attributes. The QC sub-algorithm tries to cover all attributes of frequent queries for a given storage budget, while AUF tries to fill up the remaining budget with the most frequent attributes.

Attributes & Entities Extraction: This function extracts table names, attribute names used in the workload queries, and the original dataset schema. The database schema file is provided in a data definition language DDL format. After removing additional words and data types, a dictionary s_d of entities and their attributes gets generated. From the provided workload file, (que_d) gets populated, which lists tables and attributes used by each query. Comparing both the dictionary lists provides us with two raw data partitions wa_l , and $s_d - wa_l$. The wa_l is the attributes used by workload, and the $s_d - wa_l$ lists the unused attributes not used by the given workload.

Algorithm 4 Cost Function

Data: s_d = Schema Dictionary;
 wa_l = Workload Attributes List;
 que_d = Dictionary of Queries;
 $cost_d$ = read/write storage costs of attributes;

```

1. def CostCalculation( $s_d$ ,  $wa_l$ ,  $que_d$ , dataset CSV):
2.   For query  $i$  in  $que_d$ :
3.     For each attribute  $j$  in  $que_d[i]$ :
4.       For each attribute  $A$  in  $wa_l$ :
5.         If  $j = A$ 
6.           Find index  $k$  of  $A$  in  $s_d$ :
7.            $cost_d[A].r = \mathbf{readfromraw}(j, k, \text{dataset CSV})$ 
8.            $cost_d[A].l = \mathbf{loadattribute}(j, k.lower())$ 
9.            $cost_d[A].S = \text{size of } A \text{ in loaded format}$ 
10.        End
11.     End
12.   End
13. return  $cost_d$ ;
```

Cost Function: The attribute & entities extraction part provides the list of attributes and tables used by the workload queries. The WSAC calculates the cost of each attribute used by query workload listed in wa_l by performing operations on the actual raw data. The cost function extracts each attribute from the raw data file and loads it into the given database system to find the attribute's extraction time, load time, and DB size. The values get recorded in the $cost_d$ dictionary list. The cost of an i th attribute is:

$$Cost_{d_i} = Read_Extract_Time(r_i) + Load_Time(l_i) \quad (6.1)$$

The above equation calculates the cost of i th attribute. Read-Extract time represents the time required to read the raw file and extract the given attribute. The load time represents the time taken to load the attribute data into a database as a single-column table. The size of attribute S_i is queried from the database by checking the actual size of the single-column table on the storage medium, which can be HDD, SSD, or RAM.

Storage budget: When processing a big amount of data, the word big can be related to many parameters of data processing requirements like the size of data, type of operations that need to be performed, and maximum wait time affordable to get results for that application. The algorithm uses the parameters storage budget B , which indicates cache size or database size, which a machine can process efficiently. WSAC tries to reduce the total workload execution time defined in Equation 6.2 by covering attributes of queries that can fit in the available storage space B . The total Workload Execution Time (WET) is:

$$WET = \sum_{i=1}^m DLT(A_i) + \sum_{j=1}^n QET(Q_j) \quad (6.2)$$

A research paper discussing a storage-aware partitioning algorithm directly assumed budget B as some x number of attributes [125]. While the proposed algorithm first takes an actual storage budget B in MB and finds out how many attributes can be covered in the given budget. In the earlier case [125], if the storage budget is given as $B=3$ attributes. Then the algorithm may cover three attributes having data type $char(100)$ or three Boolean attributes. It means the storage budget size may differ significantly in each case for covered attributes. On the other hand, the proposed algorithm checks the actual storage size used by the attributes considering the actual data type and data values, this allows us to fill the available storage budget more accurately.

Workload awareness: The WSAC uses two sub-algorithms, Query Coverage (QC) and Attribute Usage Frequency (AUF), to integrate workload awareness.

Algorithm 5 Query Coverage – QC

Data: B = Storage budget B in MB;
 ca_l = List of covered Attributes;
 cq_l = List of covered Queries;

```
1. def QueryCoverage( $que\_d$ ,  $cost\_d$ ,  $B$ ):
2.    $ca\_l=0, cq\_l=0$  #list of covered attributes & Queries
3.   For each query  $q$  from freq. query set  $que\_d$ 
4.     if (SUM(size of attributes of  $q[i]$ )) <  $B$  :
5.       for each attribute  $A$  in  $q[i]$ 
6.         If  $A$  is not in  $ca\_l$ 
7.           if size of  $A$  < remaining Budget  $B$ 
8.             Add  $A$  in  $ca\_l$  list update  $B$ 
9.             Add  $q$  in  $cq\_l$  if all attributes are in  $ca\_l$ 
10.    Else
11.      Query  $q$  cannot be covered.
12. return  $ca\_l$ ,  $cq\_l$ ;
```

Query Coverage (QC): The function uses a modified query coverage algorithm [125] to find the queries that can fit inside the given storage budget B . The function uses the storage cost values of each attribute recorded in $cost_d$ for each query (que_d) and adds them. The algorithm tries to cover most queries from the frequent queries set based on the storage budget. The list of queries gets filtered based on their required storage budget requirements. The queries requiring less budget than given B are considered further. The first query is chosen based on storage cost from the most frequent query set. The attributes of the covered query get added in the covered attributes list (ca_l) and completely covered queries in the cq_l list. Now, the remaining queries having covered attributes will need a lower storage budget to get covered. This way, each query is checked to see if it can be covered within a given storage budget until no other query can be covered.

Attribute Usage Frequency (AUF): This sub-function attempts to fill up the remaining storage budget using the remaining attributes used by uncovered queries. The list of remaining attributes gets extracted from the remaining queries with the frequency and removing covered attributes. The attribute coming in most queries will be at the top of the list. This way, the algorithm tries to reduce the query execution time for partially covered queries. If two attributes have the same frequency, then the attribute having a high cost is considered. The final list of cov-

Algorithm 6 Attribute Usage Frequency - AUF

wa_rl = list of remaining non-covered attributes

```
1. def AUF (ca_l, cost_d, B, s_d, wa_l):
2.   wa_rl = wa_l - ca_l
3.   wa_rl = Sort(wa_rl) based on Attribute Freq.
4.   For each attribute A from wa_rl
5.     if (Cost_d[A].S) < B :
6.       if (A.Freq. == (A+1).Freq.)
7.         if Cost_d[A].r + Cost_d[A].l >
           Cost_d[A+1].r + Cost_d[A+1].l
8.           Add A in ca_l list update B
           # Cover Attributes with high Read/Load Cost First
9.           Else if (Size of (A+1)) < B :
10.            Add (A+1) in ca_l list update B
11.   Else
12.     Attribute cannot be covered.
13.   return partitions ca_l, (wa_l - ca_l), (s_d - wa_l);
```

ered attributes (*ca_l*) for each table decides the remaining two partitions. All attributes covered in the given storage budget will get loaded into a database, so all the covered queries (*cq_l*) can be answered directly from a database. The raw partition is created for attributes used in the workload, but the algorithm could not cover them due to storage limitations.

The final output of WSAC consists of three vertical partitions for the given table; 1) Memory budget partition (*ca_l*), 2) Remaining workload partition (*wa_l* - *ca_l*), 3) Unused data partition (*s_d* - *wa_l*). If any attributes exist in the list (*wa_l* - *ca_l*), then it means some data needs to be fetched from raw partitions to answer partially covered or non-covered queries. If the storage budget is sufficient to store all the attributes existing in list *wa_l*, then there will be only two partitions *ca_l*, and (*s_d* - *wa_l*). The loaded data will cover all the queries.

WSAC – Complexity: The complexity of all the WSAC algorithms is shown in Table 6.4. It can be seen that the complexity of addition of all the algorithms $O(j^2 * (w_l))$. The *j* is the number of attributes in each query statement, and (*w_l*) is the count of queries. The total complexity of all WSAC algorithms never exceeds $O(j^2 * (w_l))$. The cost calculation lines in the cost function shown in steps 7 & 8 find the time required to read the attribute column from the raw file and load it

Table 6.4: Complexity of WSAC Algorithms

Algorithm	Complexity	Remarks
Cost Function	$O(A*j*w_l) \Rightarrow O(j*j*w_l)$	<i>que_d count is equal to w_l. A is a subset of j. In the worst case A=j.</i>
QC	$O(A*w_l) \Rightarrow O(j*w_l)$	<i>que_d count is equal to w_l. A is a subset of j. In the worst case A=j.</i>
AUF	$O(wa_{rl} \log wa_{rl}) \Rightarrow O(w_l \log w_l)$	<i>Sort (textitwa_rl) = O(n*logn) $\Rightarrow O(wa_{rl} \log wa_{rl})$, wa_rl is a subset of w_l. In the worst case, wa_rl = w_l.</i>
WSAC Total	$O(j*w_l) + O(j*j*w_l) + O(w_l \log w_l) \Rightarrow O(j^2*w_l)$	Algorithm runs in Polynomial time.

into DBMS. The read and load time depend on the size of the dataset. However, to eliminate that dependency, only a fixed sample size partition of the dataset can be taken. Therefore, the complexity of the 7 & 8th steps is $O(1)$. The cost function executes steps 7-9 for each attribute used by workload queries $O(A)$, where A is a subset of j . $A = j$ when workload queries access all dataset attributes. Therefore, the complexity of cost function will be $O(j^2*(w_l))$. The QC and AUF use values generated and saved by the cost function. Therefore, Cost function is the dominating factor in WSAC complexity calculations. However, the total complexity $O(j^2*(w_l))$ concludes that WSAC algorithm will run in Polynomial time even in worst-case scenarios.

CHAPTER 7

Resource Availability and Workload aware Hybrid Framework (RAW-HF): Maximizing Utilization of Existing Resources

This chapter explains a dynamic real-time resource allocation and task scheduling algorithm used by RAW-HF to maximize the utilization of available resources. The proposed lightweight algorithm allocates appropriate resources considering the complexity of workload tasks and real-time resource utilization readings. The existing resource maximization techniques section explains how to increase the utilization of each hardware resource independently. The experiments performed using existing resource maximization techniques concluded that maximizing RAM and CPU resources had the highest impact on total workload execution time (WET). Therefore, this work proposes Maximizing Utilization of Available Resources (MUAR) algorithm to maximize utilization of idle CPU and RAM resources in real-time.

7.1 Existing Resource Maximization Techniques

There are three main resources CPU, RAM, and Storage devices (IO). The CPU is responsible for performing data processing operations on data. RAM caches the data to be processed by the CPU. CPU writes the processed data to RAM before storing it on permanent storage devices. The storage or data input-output (IO) devices can be hard disk HDD, solid state drives SSD, Non-Volatile Memory Express

NVMe, or Cloud storage. The data read and write speeds or IO speeds, represent how fast data can be fetched for processing. Slower IO devices increase the wait time keeping the CPU idle even after a task is assigned. The following sub-section discusses relevant resource maximization techniques that may improve data loading and query execution time to reduce total workload execution time (WET).

$$WET = DLT + QET \quad (7.1)$$

Total workload execution time WET is calculated by summing up data loading time (DLT) and query execution time (QET) for a given workload. Therefore, reducing DLT and QET reduces WET.

7.1.1 CPU Resource Maximization

The primary step to maximize CPU resource utilization is creating multiple task threads to be executed in parallel using multiple cores. It is known that loading data on disk-based storage mediums need to be sequential [56]. Therefore, parallel loading cannot improve DLT for disk-based storage. However, most DBMS cache the required data in the main memory. The parallel access to cached data can reduce the time required to execute queries. Although, the actual QET time of each query does not improve. The dataset & query need to be partitioned and processed in parallel to reduce the QET of a query [85, 117]. However, it has a high overhead of partitioning and joining intermediate results to produce the final answer. Therefore, executing workload queries in parallel is the best CPU maximization technique to complete workload tasks of various applications faster.

7.1.2 RAM Resource Maximization

Main memory storage space and size have increased significantly. NoDB [33], Slalom [94], and PDC [117] have used main memory to cache processed data, indexes, and summaries for faster query execution. RAM is a random access memory. It can provide data faster during parallel access. Therefore, data cached in main memory can improve parallel execution of data loading and query execu-

tion operations. Most DBMS cache the databases or indexes in main memory by default. Most operating systems automatically cache the frequently accessed files in the main memory. However, the amount of cached data can be configured in OS, and DBMS. A. Pimpley et al. have been analyzing CPU and RAM resource requirements to train resource allocation models to provide precise resources for faster query execution [103]. Therefore, the main memory configuration settings must be tuned to improve WET for a given workload or individual query.

7.1.3 IO Resource Maximization

There are multiple types of storage devices. Each storage type has its own limitation and advantages. The most used permanent storage type is magnetic hard disk HDD due to its low cost. On the other hand, HDD is slower than SSD or NVMe storage types due to several moving parts. A. Dziejic et al. have observed that to improve DLT time; the IO device needs to be changed from HDD to SSD or RAM [56]. Parallel access to HDD cannot improve DLT due to high seek time. Therefore, HDD needs to be accessed sequentially to achieve maximum IO speeds. Additionally, SSD or RAM storage options can be explored to improve WET.

7.2 Maximizing Utilization of Available Resources (MUAR)

This section discusses a dynamic real-time task scheduling and resource allocation technique MUAR to maximize the utilization of available resources. MUAR has been integrated into a raw data query processing framework to automate the process of maximizing utilization of available resources in real time during workload execution. Most DBMS and Machine Learning (ML) based resource allocation techniques do not consider the real-time availability of resources. Additionally, the estimated resource requirements might not be applicable at run time due to insufficient resources. The resource allocations estimated using historical records

might not be accurate as resource requirements of complex queries (CQ) change exponentially with dataset size. The MUAR proposes a lightweight resource allocation & task scheduling technique that considers the complexity of queries and the real-time availability of resources. More details on the working of MUAR are explained in the algorithm section with pseudo code.

7.2.1 MUAR: Framework

The earlier discussed resource maximization techniques must be applied to an application workload multiple times to find WET reduction. Therefore, this section proposes to update a raw data query processing framework to maximize the utilization of available resources. Figure 7.1 displays a resource maximization framework for applying different resource maximization techniques to a given workload. The framework has four main modules that help complete required operations using a given DBMS or in-situ engine. The raw data query processing module and resource monitoring module have already been discussed in Chapter 5. The optimization module is discussed in Chapter 6. Therefore, only basic tasks handled by existing framework modules are briefed. The raw data query processing module handles the query execution and data loading tasks. The resource monitoring module records the resource utilization information and filters important resource information in real-time to be used by the resource maximization module. The resource maximization modules implement CPU and RAM resource maximization techniques. The following sub-section explains the tasks performed by RAM and CPU resource maximization components in detail.

7.2.2 MUAR: Resource Maximization Module

The CPU and RAM resource utilization can be controlled dynamically at runtime to maximize their utilization. Changing IO to RAMFS is a static setting and cannot be configured at runtime. Additionally, results of existing resource maximization techniques showed that combining CPU and RAM resources maximization techniques can significantly reduce WET. Therefore, MUAR uses only CPU and

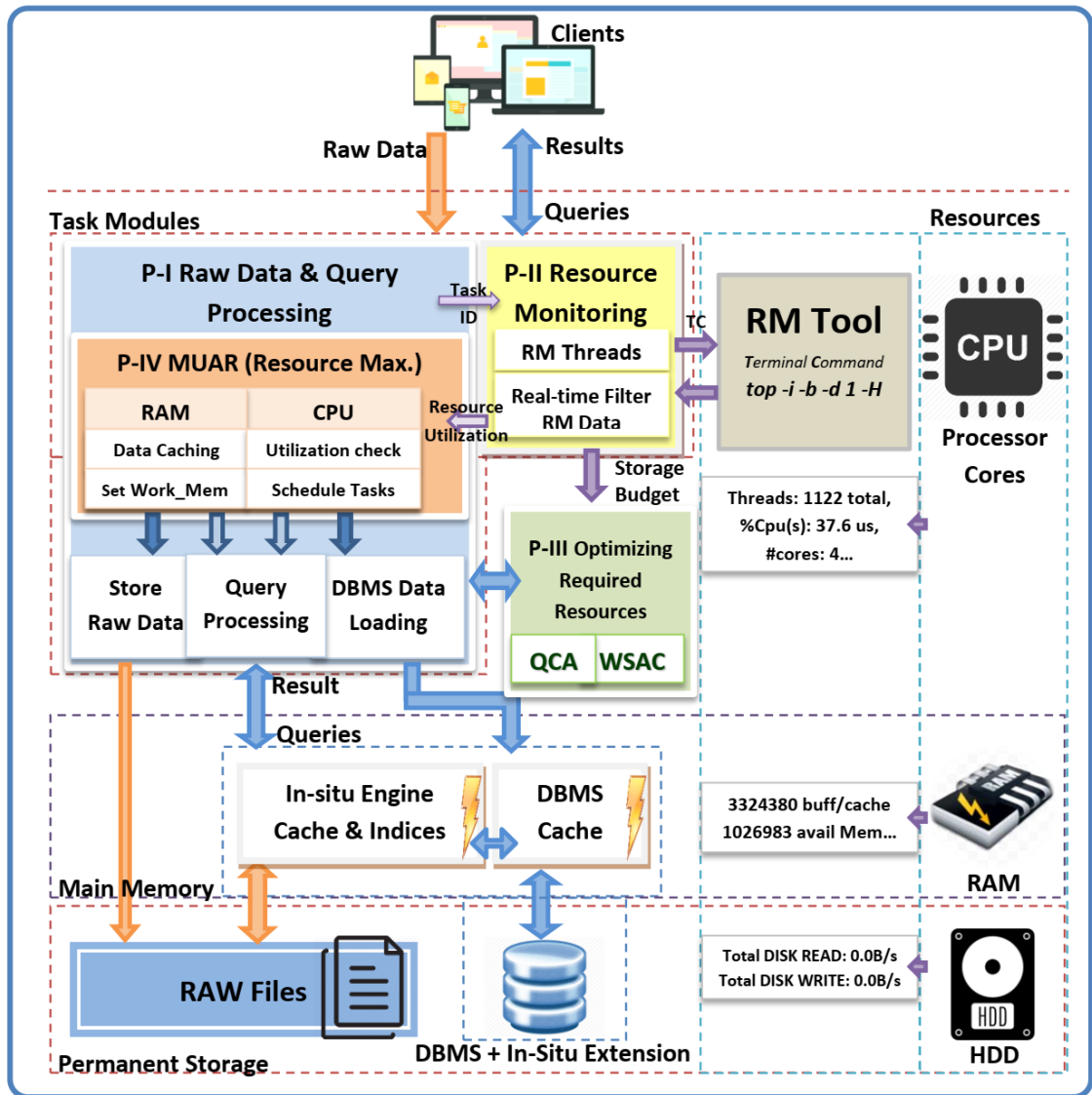


Figure 7.1: RAW-HF Architecture: Maximizing Utilization of Existing Resources

RAM maximization components to increase their utilization at runtime, considering real-time utilization.

CPU Maximization: The basic step to maximizing utilization of CPU resources is to add multiple threads to be executed in parallel. It is known that data loading on disk-based storage mediums needs to be sequential. Therefore, this component checks the number of CPU cores and creates an equal number of query threads for parallel execution. The task scheduler component checks for the hardware specifications like total CPU cores, total RAM, and availability of those resources based on real-time resource utilization observed by the resource monitoring module. The scheduler adds a new query processing thread to be executed in parallel

if the resources are available more than a specific threshold value obtained by considering the minimal resource required by a query.

RAM Maximization: RAM resource maximization function tries to increase the utilization of available RAM by caching required data and setting work memory for the queries. Caching required data is done automatically by most DBMS systems and in-situ engines. Therefore, this component focuses on setting work memory parameters to improve QET considering free RAM. Work memory is the amount of RAM given to each query process to store intermediate results. The data is written to disk when intermediate results grow larger than provided memory, increasing QET. One approach is to check RAM utilization and CPU cores to divide available RAM equally between query processing tasks by setting *work_mem* parameters. Another option is to set *work_mem* values based on RAM requirements for each query separately [103]. MUAR proposes to provide more *work_mem* to complex queries to improve overall WET.

7.2.3 MUAR: Data structures

MUAR uses simple lists, custom data structures, and other basic variables to perform algorithm operations. Table 7.1 shows the workload list, which stores list of workload query statements and their task IDs. The task ID of a query relates time and resource utilization readings recorded by query processing and resource monitoring modules. MUAR uses shared global structures to pass Available Resources (*RM_AR*) values of CPU, RAM, and IO between resource monitoring and resource maximization modules of MUAR.

Table 7.1: MUAR: Workload list (*w_l*)

[T_ID]	1[Statement]
TRUN	"TRUNCATE TABLE PhotoPrimary;"
COPY	"COPY PhotoPrimary FROM '/...SDSS/PhotoPrimary.csv' (DELIMITER ',');" "
Q0	"Select count(objID) from PhotoPrimary;"
Q4	"SELECT objID, ra ,dec FROM PhotoPrimary WHERE ra >185 and ra<185.1 AND dec >56.2 and dec <56.3 limit 100;"

7.2.4 MUAR: Algorithm

The resource monitoring module extracts the RAM and CPU utilization values in real time. The extracted values are input to the CPU and RAM resource maximization modules of MUAR. The goal of the proposed algorithm or task scheduler is to utilize CPU and other resources to a given threshold value of 90-99%. If resources are utilized 100% by workload tasks, many systems processes might fail, causing system failure. Most existing static resource allocation techniques do not consider resource availability in real-time, which can cause over or under allocation of resources. The MUAR algorithm proposes to predict the resource requirements of queries before starting them to provide appropriate resources based on query complexity. In addition, MUAR also tries to utilize maximum available resources up to the given threshold value time while keeping some resources for the system and other processes to complete in between if they get initialized.

The pseudo code of MUAR task scheduling and resource amount allocation is written below. MUAR considers real-time resource monitoring values of CPU, RAM, and IO resource utilization stored in the global structure RM_AR . The RM_AR has three float datatype variables to store CPU, RAM, and IO utilization values. The MUAR adds a new task for processing if all three values of RM_AR are greater than the minimum required CPU, RAM, and IO resources set in global variable Minimum Required Resources (Min_RR). Before executing a query from (w_l), the WM_Query function is called to set the work memory parameter for the query to increase RAM utilization. The WM_Query function first counts the number of joins used in the given query and stores the count in J_C . The WM_Value in line 13 calculates the work memory value based on available memory $RM_AR.RAM$, process count P_C , total RAM TR, and join count J_C . The first part ($RM_AR.RAM/P_C$) of the equation divides the available RAM between the maximum processes the available CPU can handle. The second part (TR/CPU_C) defines the maximum RAM that can be assigned to a thread, while the third part ($J_C/4.0$) helps in increasing allocating more RAM to complex queries considering join count J_C . MUAR also tries to estimate required work memory considering previous $work_mem$, disk writes, current & past record count ratio for frequent queries to achieve the

Algorithm 7 MUAR Algorithm

Data: w_l = Workload List;
 RM_AR = Real-Time availability of CPU, RAM, & IO resources in %;
 TR = Total RAM;
 Min_RR = Minimum resources required to schedule a task;
 J_C = Join Count of a query q ;
 P_C = Process count that free CPU can handle;
 WM_value = Work memory value;
 CPU_C = CPU cores count of experiment machine;
 P_WMq = Work_Mem assigned to query q during previous run;
 P_DWq = Disk Writes required by query q during previous run;
 C_RC = Current Record Count;
 P_RCq = Record Count during previous run of query q ;

```
1. def MUAR( $w\_l$ ,  $RM\_AR$ ,  $Min\_AR$ ):
2.   For each query  $q$  in  $w\_l$ :
3.     While resources are not available- $RM\_AR < Min\_AR$ :
4.       sleep 0.1sec;
5.     if minimum resources are available- $RM\_AR > Min\_AR$ :
6.       Set work_memory =  $WM\_Query$  ( $RM\_AR$ ,  $q$ );
7.       Add a new query thread in parallel;
8.   End for
9. Exit;
```

```
10. def  $WM\_Query$  ( $RM\_AR$ ,  $q$ )
11.    $J\_C$  = Count Joins in a query  $q$ 
12.    $P\_C$  =  $RM\_AR.CPU / (100 / CPU\_C)$ 
13.   If Query  $q$  is Frequent:
14.      $WM\_value = (P\_WMq + P\_DWq) * (C\_RC / P\_RC)$ ;
15.     If  $WM\_value > RM\_AR$ :
16.        $WM\_value = RM\_AR * 0.9$ ;
17.   Else
18.      $WM\_value = (((RM\_AR.RAM / P\_C) * (TR / CPU\_C)) * (J\_C / 4.0))$ ;
19.   Return  $WM\_value$ ;
```

best QET time. Whenever required work memory exceeds the available memory, 90% of available memory is allocated to achieve the near best QET.

For a 16GB main memory system with quad free CPU cores, the equation can allocate a maximum of 4GB RAM to each query in the first part. J_C part increases the RAM allocation to 1.25x for complex queries having more than 4 JOINS. While it also assures that simple queries get 0.25x memory which is 1GB or less, to avoid over allocation of RAM resources. The work memory value is returned to MUAR, and *work_mem* is set using an SQL query in real time for the given query execution session. A unique query ID is assigned to every new query. The time and resources used by each query are recorded to identify frequent queries, improve future query runs, and calculate actual improvements in QET. The amount of data written to disks due to insufficient work memory during the first run is used to accurately estimate work memory value for future query runs.

MUAR: Complexity The complexity of the proposed MUAR algorithm is $O(x*(w_l))$. It can be seen from pseudo code that most steps of MUAR algorithm execute only once for each workload query. Steps 2, 5 to 8, and 10-14 runs in $O((w_l))$. However, the wait logic written in step 3 may execute once for each workload query or unknown times (x) depending on resource availability. For systems with more number of resources, this x wait time will be lower. These steps add a waiting time of 0.1sec each time resources are unavailable, and the loop continues until enough resources are available. In best case scenarios algorithm runs in $O((w_l))$. Due to wait steps, the complexity of MUAR increases to $O(x*(w_l))$. However, the complexity of MUAR stays linear, which means MUAR runs in polynomial time.

CHAPTER 8

Experimental Setup

This chapter explains the experimental setup used to conduct experiments. Figure 8.1 shows the block diagram of the experimental setup and connection between different modules of RAW-HF. Optimizing resource utilization module provides the modified workload queries and dataset schema to the raw data query processing (RQP) module. RQP module should load the complex query partitions in DBMS. At the same time, the simple query partitions and unused partitions must be kept in raw format to reduce the DLT time. The raw partition files must be linked to the in-situ engine for executing queries without file paths.

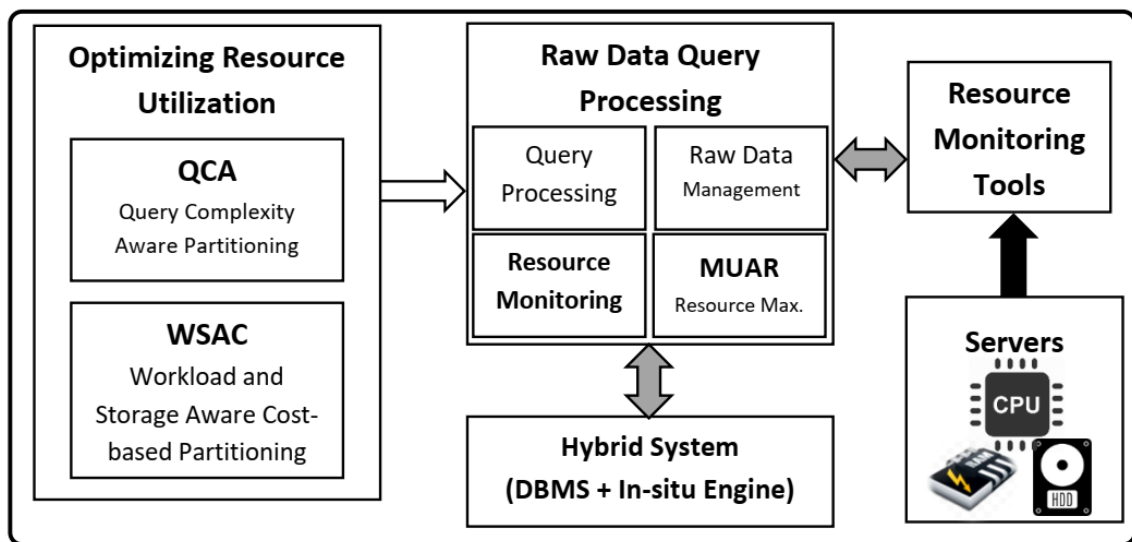


Figure 8.1: Experimental Setup: Block Diagram

The resource monitoring module receives real-time resource utilization data from resource monitoring (RM) tools. The RM tools should be able to monitor the total CPU, RAM, and IO hardware resource utilization of entire systems and indi-

vidual processes. RM module needs to filter the received data and provide the resource availability information to the resource maximization module in real-time. MUAR should calculate how many resources can be allocated to each query task and schedule them for execution using a hybrid system. The hybrid system must contain an in-situ engine and DBMS to execute queries on raw and loaded data. Some experiments may require a hybrid system that can join data stored in multiple formats, i.e., raw and database. The raw data query processing framework needs to record the DLT, QET, and RM data for each workload task for offline analysis.

The experiments have been conducted for all four identified phases. Section 8.1 discusses the implementation setup, which lists all the software tools used to implement RAW-HF. The datasets used in different phases of experiments have been discussed in Section 8.2. Two real-world datasets have been used to address extreme cases of data processing by workload queries. Both datasets have been compared based on dataset size and query types in Section 8.2.3. Section 8.3 briefs the evaluation parameters used to compare the results. The last Section 8.4 explain the experimental flow of each phase.

8.1 Hardware & Software Setup

The hardware configuration of the machine included a quad-core Intel i5-6500 CPU clocked at 3.20GHz. It had 16GB of RAM and Intel HD Graphics 530 (Skylake GT2) in-built, running 64-bit Ubuntu 18.04 LTS operating system. A SATA hard disk drive had 500GB of space and 7200RPM rotation speed is used as a permanent storage medium. The software required to run the WSAC algorithm is Jupyter running Python code. The raw data query processing framework is used to handle the raw data. The raw data query processing framework and its task modules are developed using Java code on Eclipse. The database management system PostgreSQL (PgSQL) is used to load raw data into a database. At the same time, the framework uses NoDB (PostgresRAW) extension to execute queries directly on raw files. Raw data storage, data loading, and query execution tasks of

the raw data query processing module are also coded using Java. The *top* and *iostat* tools provided the real-time resource utilization data to the resource monitoring module in response to the terminal commands.

8.1.1 Implementation Block Diagram

This section provides implementation setup detail of each RAW-HF component. Figure 8.2 shows the external tools and language names used to implement the entire framework. It can be seen that QCA and WSAC are implemented using Python. Python is easy to code and provides a rich library of functions that reduces lines of code to develop complex algorithms. The framework is implemented using Java code because Java is generally faster and more efficient than Python. Java is a compiled language. It reduces algorithm execution time (AET) for real-time algorithms compared to Python. Therefore, time bound real-time algorithms like resource monitoring and MUAR have been implemented using Java.

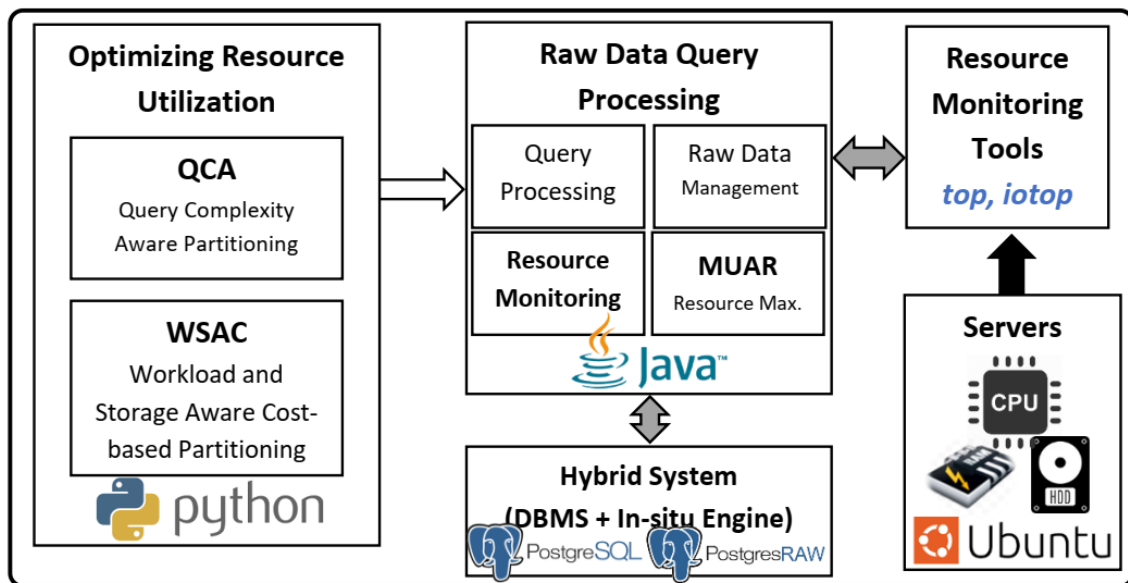


Figure 8.2: Implementation Setup Diagram

The RAW-HF uses state-of-the-art DBMS PostgreSQL to handle database tasks. PostgreSQL is also open source. The in-situ engine used here is the NoDB [33]. NoDB is one of the first few in-situ engines which allows querying raw data using SQL and indexes processed data in main memory. NoDB was developed for Human

Brain Project and is available on GitHub [11]. The *top* and *iotop* resource monitoring tools have been used to monitor the resources utilized by Ubuntu system and other processes.

8.2 Dataset & Query Set

Experiments of thesis phases have been conducted using two different real-world datasets called Sloan Digital Sky Survey (SDSS) and Linked Observation Data (LOD).

8.2.1 SDSS

The Sloan Digital Sky Survey (SDSS) dataset is a real-world astronomical dataset of stars, galaxies, and other sky objects. The size of the recent data release named DR-16 is 273TB [12, 30]. SDSS DR-16 has 134 tables and 59 views. Analysis of the query workload of 0.4M showed that 55% of queries of the workload belong to the *PhotoPrimary* view. Therefore, we extracted 18GB of data from *PhotoPrimary* view for the experiment. The number of records in the extracted table is 4M.

The SDSS keeps track of queries executed on the dataset and logs them in *SQLlogAll* table. We had extracted the top 1000 unique queries executed on *PhotoPrimary* view. These queries represented 51% of DR-16 workload. The similar queries have been grouped based on the similarity of the attributes and query type, forming 12 query groups. One representative query has been chosen from each group for experiments.

8.2.2 LOD

The Linked Observation Data (LOD) dataset containing descriptions of blizzard and hurricane observations is a benchmark RDF dataset [96]. The Linked Sensor Data part of Linked Observation Data containing sensor data of temperature, rain, and humidity has been used for all the resource maximization experiments. The dataset was extracted from .ttl files and converted to triple format for exper-

iments with relational DBMS. LOD dataset is embedded with URI links that can identify each record uniquely by computers. The 2.9GB dataset size in triple format included 10M records. Each record has only three values: subject, object, and predicate.

The triple format requires multiple self-joins to get details of one single record. Therefore, most queries require multiple joins. 16 standard RDF queries having different numbers of joins are used for experiments. Nine queries out of 16 queries of workload have 5 joins. Queries with multiple self-joins have been considered because these queries need to process complex join operations, which require significant resources. The effect of different resource configuration settings can be identified easily due to their high resource requirements.

8.2.3 Comparison of Datasets

Both LOD and SDSS datasets have different characteristics, as shown in Table 8.1. The SDSS dataset is a broad dataset having 509 attributes in a single table. In contrast, LOD had only 3 attributes as it uses a triple format. The dataset size of 1M records for SDSS is 4.6GB, while LOD takes only 253MB. The LOD data is originally in URI format, so it is in string format. While the SDSS dataset is generated by converting image data to CSV format. Most columns in SDSS are in integer or float format.

Table 8.1: Dataset Details

Parameters	Dataset Domain	Source	No. of Columns	1M Dataset size	Data Types	Format
SDSS	Astronomy	Telescope	509	4.6 GB	Integer, Float	CSV
LOD	Weather	Sensors, Satellite	3	253 MB	String	CSV

The workload queries from both datasets can represent the majority of query types. The classification of both workload queries is shown in Table 8.2 based on the number of joins required by the query. LOD query workload consists of nine analytical queries having 5 joins, which require more resources to process

the data for the queries. The SDSS queries represent simple 0 join queries to 2 join analytical queries. Most LOD queries need to access all columns of the dataset due to its triple format, while SDSS queries access only few columns of the dataset.

Table 8.2: Query Classification

# No. of Joins	0	1	2	3	4	5	6
SDSS	5	6	1	-	-	-	-
LOD	1	1	4	1	0	9	0

8.2.4 Datasets used in Experiments

Different experiments had different workload and dataset requirements. The experiment section tried to cover all experiments for both datasets to identify which techniques performed best with what kind of dataset and query workload. Table 8.3 lists all phases and the datasets used in those phases with the reason behind choosing that dataset. Phase-II & III used only SDSS datasets to identify which queries work best with which tool and the resource requirements for different types of queries. Phase-IV experiments use both LOD and SDSS datasets, because LOD dataset workload consists of more complex queries. LOD workload can provide evident results for RAM maximization experiments.

Table 8.3: Dataset used in Phases

# Phases	Phase-I	Phase-II	Phase-III	Phase-IV
Dataset Used	SDSS, LOD	SDSS	SDSS	SDSS, LOD
Reason	Identify tools best for which type of queries.	Identify Resource requirements.	Broad Table Dataset required.	Complex Queries required for RAM Maximization Exp.s.

The goal of Phase-III experiments is to identify frequently accessed part of the dataset and process only that part to answer frequent queries. WSAC technique required a broad dataset to find frequently accessed attributes for vertical partitioning of the dataset. LOD dataset is a narrow three column dataset. Most

LOD queries need to access all three columns to provide results. Therefore, SDSS dataset has been used for Phase-III experiments. Phase-III and Phase-IV combination experiments are performed using SDSS dataset to find overall improvement.

8.3 Evaluation Parameters

This section discusses all the evaluation parameters used in Section 9 to compare experimental results. The parameters have been grouped into two categories: 1) Input Parameters, and 2) Output Parameters.

8.3.1 Input Parameters

The following list of parameters has been provided as input to different algorithms of RAW-HF.

8.3.1.1 Workload Parameters

The following list of workload parameters like schema files, and list of workload queries helps in developing workload-aware techniques like QCA, WSAC, and MUAR of the proposed RAW-HF framework.

Schema Files: It is the DDL schema of tables used to create the workload database.

Workload List: It is the list of queries with a unique query ID used to generate the experiment workload.

File Paths: It is the URL or Path to files on disk. These file paths help in locating inputs and output files used to retrieve or store result data to disk as files.

Flags: Boolean flags are used to set instructions for experiments. For example: decide which datasets to use, whether to start monitoring threads, or which operations to perform on the dataset.

8.3.1.2 Resource Utilization Parameters

These parameters represent the percentage of CPU, RAM, or IO resources a query utilizes during execution. Different algorithms of RAW-HF have used the historical or real-time values of these parameters.

CPU (%): The percentage of CPU free for task processing. MUAR uses this parameter in real-time to schedule tasks based on *Min_RR* threshold values.

RAM (%): The percentage of RAM free for caching or processing data. MUAR uses this parameter in real-time to allocate RAM resources to complex queries.

IO_wait (%): The percentage of time CPU spent in waiting for IO resources. CPU stayed idle because it was waiting to receive the data for processing.

Work_mem (MB): It is the size of RAM allocated to each workload query to improve QET.

DB_cost (MB): It is the size of each attribute accessed by workload queries in the database format.

RAW_cost (MB): It is the size of each attribute accessed by workload queries in the raw format from disk.

Total Read/Write (MB): It is the total size of data read or written to disk during workload execution.

8.3.2 Output Parameters

The following list of output parameters helps us identify the reductions or improvements achieved by applying proposed techniques compared to existing tools & techniques.

8.3.2.1 Query Performance Parameters

These parameters display time taken by in-situ engines or DBMS to complete given workload tasks.

QET (sec): Query Execution Time is the time taken by a query to execute.

DLT (sec): Data Loading Time is the time data loading tasks take to load a given partition or complete dataset into DBMS.

WET (sec): Workload Execution Time is the total time system takes to execute a given workload. For single-thread execution, WET can be simply calculated by summing DLT and QET. However, in multi-thread execution of workload, the total time is calculated by subtracting the experiment end time from the start time.

Fraction of Attributes Accessed (FAA) (%): This parameter shows the number of attributes accessed by the workload queries. **Add equation**

8.3.3 Resource Utilization Parameters

These parameters represent the percentage of CPU, RAM, or IO resources utilized to execute a given workload.

CPU (sec): The overall CPU time utilized to complete execution of all workload tasks.

RAM (MB): It is the size of RAM used by DBMS or In-situ engine to complete workload tasks.

DB_APS (MB): It is the size of database partitions that workload queries accessed or cached from disk.

RAW_APS (MB): It is the size of raw files that workload queries accessed or cached from disk.

Total Read/Write (MB): It is the total size of data read or written to disk during workload execution.

8.4 Experiment Flow

This section discusses how experiments for all phases have been conducted. The setup modification and flow for each experiment have been discussed.

8.4.1 Resource Monitoring Framework for Raw Data Query Processing

The raw data query processing framework implementation details and modules have been briefly explained. Most of the raw datasets are available in comma-

separated values (CSV) or JSON formats. The CSV is a storage efficient raw format as it separates the data using a single special character comma or some other delimiter. In comparison, JSON and XML formats required key-value pairs or begin-end tags to identify data, increasing the file size for the stored raw data. CSV format stores a single header line to identify the records, to reduce the file size. It is also easy to extract data from CSV files, while data extraction from JSON and XML requires additional support libraries at the implementation level. The proposed framework stores streaming data in CSV files and processes it efficiently using hybrid system to reduce data to query time. The framework consists of two main modules 1) Raw Data Query Processing and 2) Resource Monitoring modules.

8.4.1.1 Raw data Query Processing (RQP) module

The raw data query processing (RQP) module consists of functions like raw data storage, data loading, query execution, and recording QET time. The RQP module is written in Java using Eclipse. Raw data storage module stores the streaming raw data into CSV files linked to the NoDB. The connection to NoDB and PostgreSQL is made using JDBC. The connection allows execution of queries immediately on arrived raw data using SQL language. The data loading function can load the stored data into PostgreSQL to improve the execution time of future queries. Basic results confirmed that the COPY method used by A. Dziedzic et al. is the fastest bulk data loading technique [56]. Therefore, all the data loading operations are performed using COPY method for all the remaining experiments.

The query execution function handles the tasks of executing workload queries stored in a CSV file on the hybrid system. The NoDB considers linked CSV files a relational table facilitating the execution of join queries between raw files and database tables. It eliminates the need to transform queries from one query language to another or merge results. Additionally, the raw files can be accessed and modified using external applications like Python to apply QCA and WSAC techniques. The data loading and query processing functions connect to the PostgreSQL DBMS and NoDB extension using a single connection link. It enables the frame-

work to redirect workload queries to raw files or DBMS easily. A function saves the time required by tools to answer queries and saves the time as DLT or QET. The total time required to complete all workload tasks is also recorded as WET in the output result files for later analysis.

8.4.1.2 Resource Monitoring (RM) Module

Most traditional DBMS do not consider real-time resource utilization due to statistics collection and analysis overhead. These databases are configured to work for general workloads and datasets. The specialized tuning of resource allocation is done by database admins based on their knowledge of the application domain, workload requirements, and DBMS software for specific applications. This section discusses how resource monitoring tools work and their setup with minimal overhead.

Resource Monitoring Tools The operating system interfaces with the CPU, RAM, and IO hardware to process, retrieve, and store data. The system monitor or task manager is an inbuilt resource monitoring application that provides the user an interface to view resource utilization. These inbuilt system monitoring tools may not have a customizable output based on our requirements. Therefore, the installation of additional tools is required. We had checked *top*, *iostat*, *SAR*, *Sysstat*, *iostat*, *iosnoop*, and a few other resource monitoring tools that provide customizable resource utilization outputs [56]. We tried to find a single tool that can provide all required information and combine *top* and *iostat* output in a single command. But we could not find any such tool with the required details, nor a combined command worked during framework implementation using java.

For experiment purposes, *top* and *iostat* tools are used for their detailed output that matches our needs. The *top* tool provides total utilization and individual process utilization of CPU, RAM, and SWAP. The *iostat* provided IO utilization in % with total and individual process read/write in K/s. Sample resource monitoring outputs of both tools are shown in Figures 8.3 & 8.4. Figure 8.3 shows the *top* tool output with CPU, RAM, and important processes marked with red rectangles. Figure 8.4 displays IO utilization by processes and total read/writes. Some

```

top - 23:43:41 up 1:42, 1 user, load average: 1.20, 0.67, 0.46
Tasks: 268 total, 2 running, 214 sleeping, 0 stopped, 1 zombie
%Cpu(s): 21.8 us, 2.9 sy, 0.0 ni, 59.6 id, 13.7 wa, 0.0 hi, 2.0 si, 0.0 st
KiB Mem : 16257856 total, 3572580 free, 2052448 used, 10632828 buff/cache
KiB Swap: 2097148 total, 2097148 free, 0 used. 13510608 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 3553 om        20   0 174716  45092 43396 R  82.4   0.3   0:33.39 postgres
 2438 om        20   0 2291336 215472 45972 S   8.3   1.3  14:12.69 anydesk
 1068 om        20   0 595520 110000 94380 S   2.7   0.7   3:44.15 Xorg
 3544 root       20   0 52952  14676  6716 S   2.3   0.1   0:00.88 iotop
 3542 om        20   0 44672   4544  3316 S   2.0   0.0   0:00.88 top

```

Figure 8.3: *top* Tool Output: CPU & RAM Resource Monitoring

```

File Edit View Search Terminal Help
Total DISK READ : 40501.48 K/s | Total DISK WRITE : 22058.46 K/s
Actual DISK READ: 20452.48 K/s | Actual DISK WRITE: 9988.81 K/s

  TID  PRIO  USER      DISK READ  DISK WRITE  SWAPIN     IO>   COMMAND
 20134 be/4  root      391528.00 K  308.00 K   0.00 %   1.07 % mount.ntfs /dev
  238 be/3  root         0.00 K   4260.00 K   0.00 %   0.88 % [jbd2/sda3-8]
 16244 be/4  om         0.00 K 165280.00 K   0.00 %   0.77 % postgres: wal w
 20396 be/4  om      328704.00 K 203120.00 K   0.00 %  16.04 % postgres: om om
  1261 be/4  om         432.00 K    0.00 K   0.00 %   0.12 % Xorg vt1 -displ
 27706 be/4  om         4848.00 K   1644.00 K   0.00 %   0.10 % chrome

```

Figure 8.4: *iotop* Tool Output: IO (Disk) Resource Monitoring

of the key values are marked with red rectangles provided by *iotop* tool.

Resource Monitoring Tools setup The in-situ extension and DBMS do not provide resource utilization details for each workload task. Therefore, a data passing interface between resource monitoring tools and framework is required. The RM modules link workload tasks with resource utilization outputs. The modules use *top* and *iotop* resource monitoring tools to monitor real-time resource utilization. These tools are installed on the operating system Ubuntu. RM threads pass terminal commands (TC) to the external RM tools and receive output. The output stream contains a lot of information. Storing all the information can quickly increase the result size. The basic experiments created result output file sizes reaching more than 2GB in size with only 3 to 4 hours of resource monitoring observations. Therefore, the RM modules filter the required CPU, RAM, and IO utilization values of specific processes before saving. This output is then combined with current workload task information, i.e., query id, before saving to the result file. The task ID with resource utilization helps correlate the resources required by each workload task.

8.4.1.3 Experiment Flow

For experimental purposes, we have used 4.6GB *PhotoPrimary* table data and 700MB of LOD triples data in CSV file format, which contained 1M records with 509 attributes in each row. In-situ engines use this file to execute queries using in-situ engine NoDB. While performing experiments for PostgreSQL, this CSV file is loaded into the database using COPY command. Each query is executed four times to get the average query execution time. Figure 8.5 shows the experiment flow for the phase-I & II experiments. First, we set the raw files paths and other parameters, then based on the selected tool PostgreSQL or NoDB, the remaining steps are performed. If the PostgreSQL is selected, then data is loaded into the database before executing queries. While for NoDB, queries can be executed as soon as the streaming data is stored in the linked CSV files. Here, we read the existing CSV file to get the data for loading or writing to the CSV file linked to the in-situ extension to mimic data streaming. Once all the workload queries are executed, the DLT and QET times are saved in the output result file for later analysis.

8.4.2 RAW-HF: Optimizing Required Resources

This section explains the experimental flow for QCA and WSAC techniques used by RAW-HF to optimize required resources during workload execution. QCA partitions the original raw dataset file based on query complexity for hybrid system. QCA groups simple query attributes and complex query attributes to generate final partitions. WSAC tries to find best partition for loading in database using a cost function that considers frequency of attributes, cost of loading attributes, size on disk and attribute access costs from raw and database formats.

8.4.2.1 QCA: Experiment Flow

Multiple experiments have been performed for QCA proposed partitions to evaluate the performance of QCA in typical single-core execution and resource optimization modes. Initial experiments have been done with the original dataset to find actual WET using state-of-the-art DBMS (PostgreSQL) [22] and In-situ engine

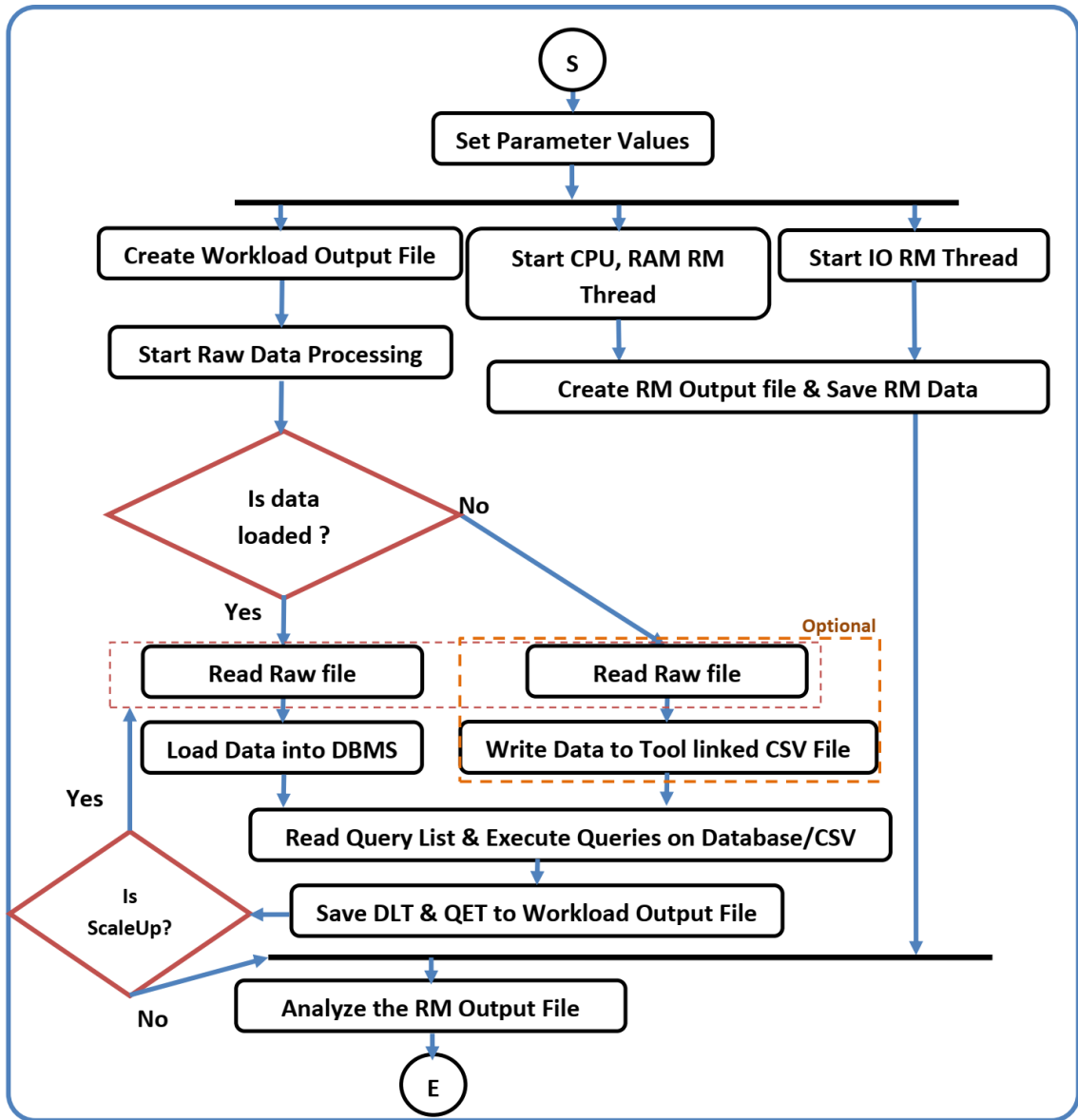


Figure 8.5: Experiment flow: Phase-I & II Resource Monitoring for Raw Data Query Processing

(NoDB) [34]. Section 9.1 shows the results achieved by executing workload on the original dataset.

Figure 8.6 shows the experimental flow of applying QCA technique to the SDSS dataset. The QCA technique takes dataset schema, workload list, and actual raw files of datasets in CSV format as input. The Extract from Schema function uses DDL files of schema to find all table and their attributes of a dataset and stores them in s_d dictionary. The Extract from Workload function creates a dictionary of workload queries (que_d) containing table names and attributes used in a statement based on s_d . The QCA algorithm first identifies the types of queries by checking the number of tables coming in a query. Once the query complexity identification is complete, the attributes of similar types of queries are grouped to form simple query partition (QT_P0) to be stored as raw and complex query partition (QT_P1) that needs to be loaded into the database. The intersection of both the list provides CAP list. The first iteration of QCA ended with all queries partially covered because all the workload queries had 2 or more attributes in CAP. Out of 54 workload attributes, database partition had 25 attributes, 21 in raw partition and 10 attributes in CAP with primary key attributes replicated in all partitions to allow joins. The different sets of partitions have been created to cover three out of five cases discussed in Section 6.2.1.3.

The Section 9 analyzes results obtained for data distribution Case-I, II, and V. The Case-III & IV have not included because all queries would require join, which would be slower than other cases where 41-58% of workload queries did not require any additional joins. The different QCA case results have been compared with the original dataset execution on NoDB [34], Workload Aware (WA) partitioning techniques like Partial Loading [125] & WSAC. The best result of the WA techniques is considered, where all the required data is loaded in DBMS due to availability of enough storage budget.

8.4.2.2 WSAC: Experiment Flow

The algorithm flow has been discussed in detail in Section 6.2.2. Therefore, this section discusses the parameters and files that are given as input to the sub-

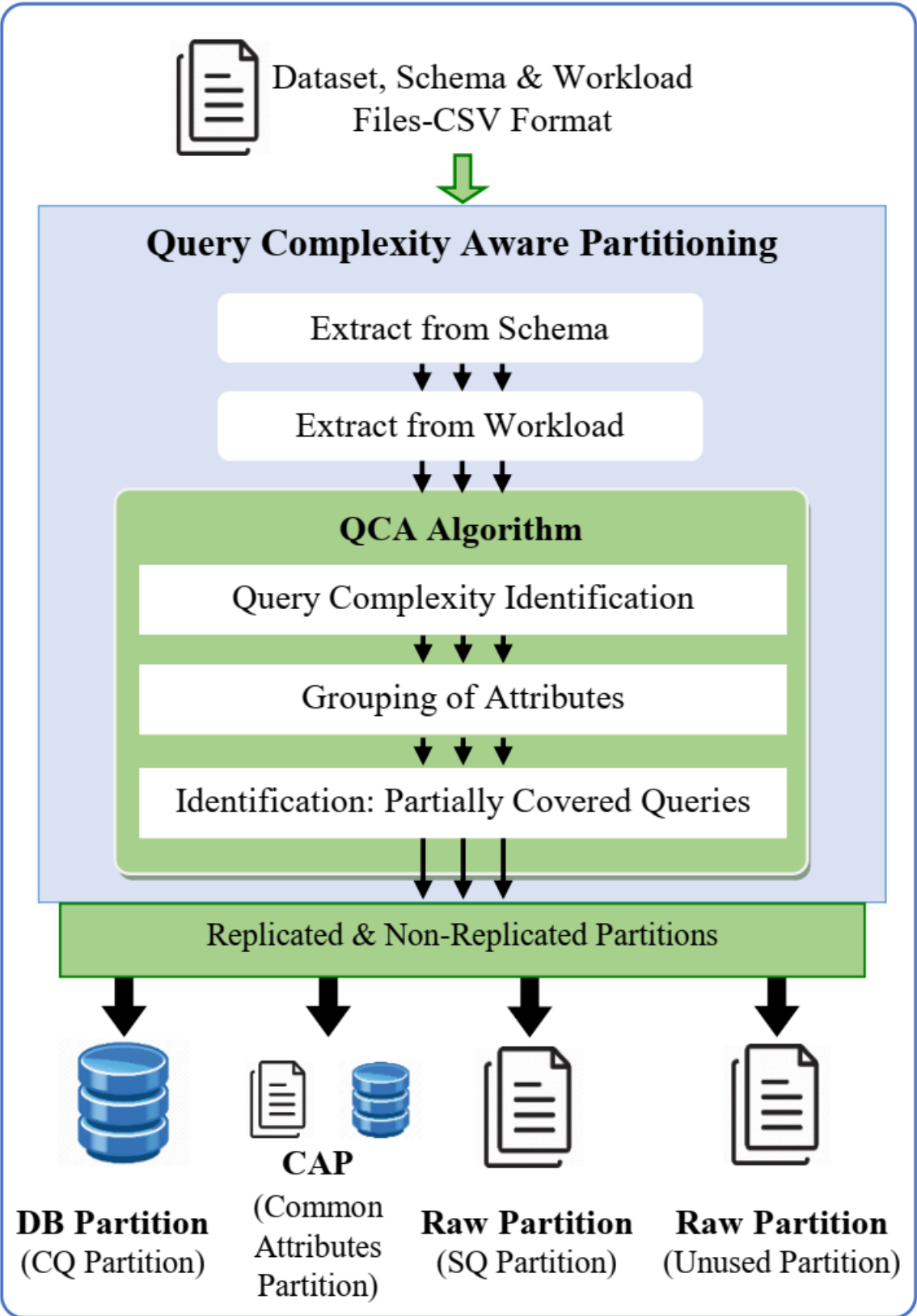


Figure 8.6: Experiment Flow: QCA Technique

algorithms. As shown in Figure 8.7, First the database schema – the DDL files of *PhotoPrimary* table having 509 attributes, and the frequent query set files are provided for the initial attribute and entity extraction phase. The input for the cost calculation phase is 1M records of the *PhotoPrimary* table in CSV format. The cost calculation function calculated the read-extraction, load cost, and size for attributes used by the workload. Once the attribute is loaded in a single column table, the function checks the actual table size S_i required to store the attribute i . QC & AUF sub-algorithms of WSAC then use the output costs to find covered attributes and queries.

The partition consisting of covered attributes is loaded in PostgreSQL. The raw partition files are linked to tables using the NoDB extension configuration files. Now, the java code executes the covered queries on PostgreSQL and Partially covered queries on the combination of PostgreSQL and NoDB. The database and raw partitioned are joined using the ObjID Primary key PK_A. The 12 queries from the frequent query set were executed four times, and the average is considered for accurately counting the QET time. If any new queries come that don't have any covered attributes, they can be answered using the 3rd raw partition.

8.4.3 RAW-HF: Maximizing Utilization of Existing Resources

This section discusses the experimental flow for the individual, multiple resource maximization experiments, and MUAR.

8.4.3.1 Single Resource Maximization

This section discusses individual resource maximization experimental flow. *CPU Maximization*: The CPU maximization technique adds the same number of query threads as CPU cores. These basic experiments did not check the CPU utilization and assigned a new thread to every incoming workload task. The data loading tasks can be performed in parallel. However, DLT time may not improve for disk-based storage devices [56]. To avoid parallel disk access, query processing tasks have been executed in parallel by assigning one thread to each query utilizing data cached by DBMS.

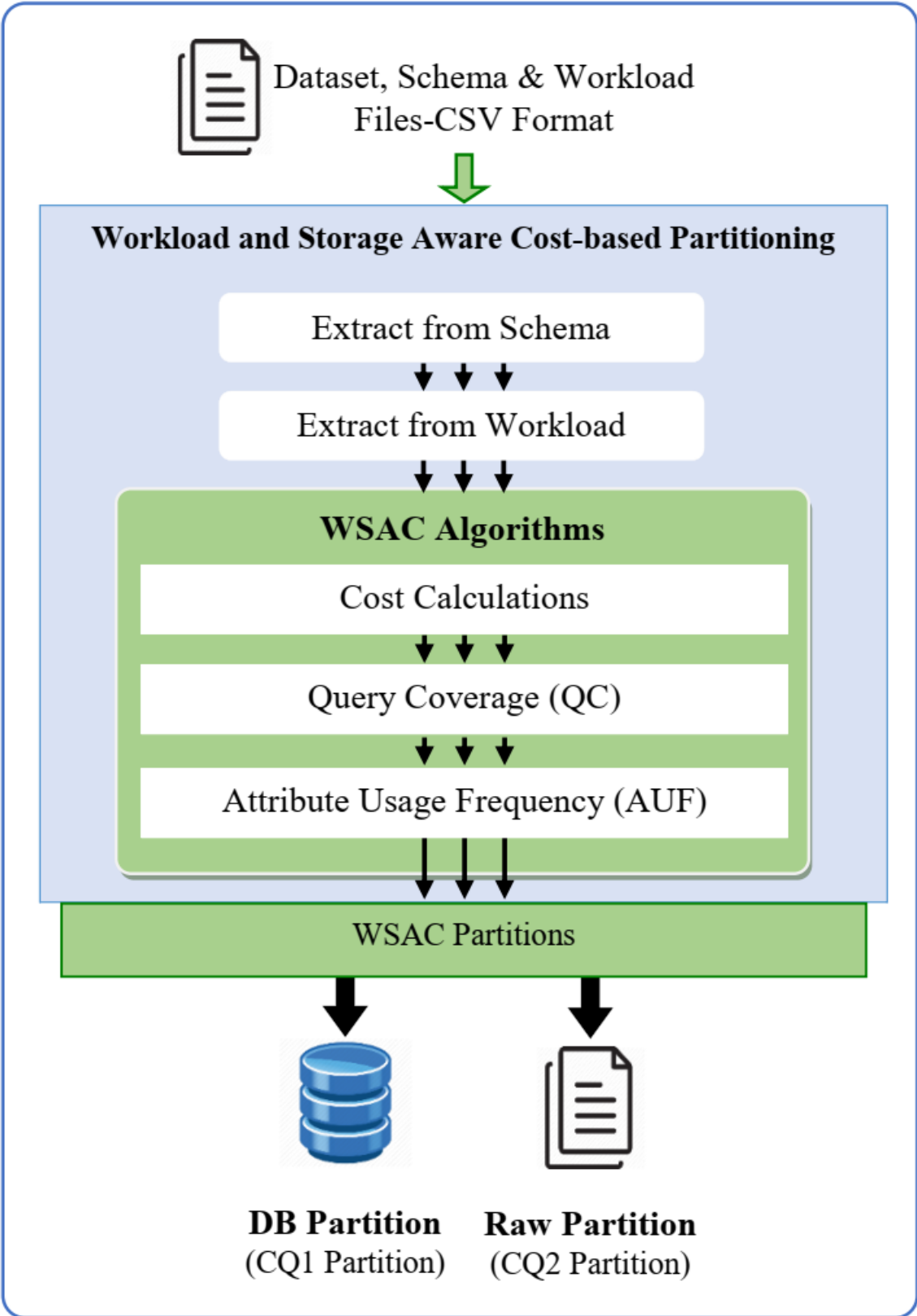


Figure 8.7: Experiment Flow: WSAC Technique

RAM Maximization: There are several options to configure RAM utilization values like *shared_buffers*, *temp_buffers*, *work_mem*, and others. Basic experiments showed that *work_mem* change impacts QET significantly because it allows storing more intermediate results data in RAM. By default, the *work_mem* is set to 4MB. The value has been set to 500MB to store 5 times more results than the dataset containing 1M records.

IO Maximization: According to researchers, changing the IO device from HDD to RAM is the only way to improve DLT [56]. Therefore, IO maximization experiments used the RAM file system RAMFS to utilize 12GB of RAM as a database storage device. Table space is moved to the RAMFS folder to get faster read-write speeds.

8.4.3.2 Multiple Resource Maximization

The combination of individual resource maximization techniques has been performed to observe their effect on DLT and QET. There are only four unique combinations possible CPU+RAM, CPU+IO, RAM+IO, and CPU+RAM+IO. Here, the IO configuration is static and cannot be changed during runtime. The CPU maximization technique uses all four CPU cores, and work memory is pre-set to 500MB. For DLT, single resource maximization experiments showed that only IO change could improve DLT. RAM can allow parallel access to read and write data faster. Therefore, only the CPU+IO combination has been considered for experiments to improve DLT. In contrast, QET time improved with all single resource maximization techniques. Therefore all possible combinations have been experimented with to improve QET.

8.4.3.3 MUAR Technique

The MUAR algorithm tries to maximize CPU and RAM resource utilization automatically during workload execution. Figure 8.8 shows the flow of MUAR experiment. The figure shows that basic parameters are set first. The resource monitoring threads are started and they run in parallel to query execution flow. All queries are read one by one from the workload query list for execution.

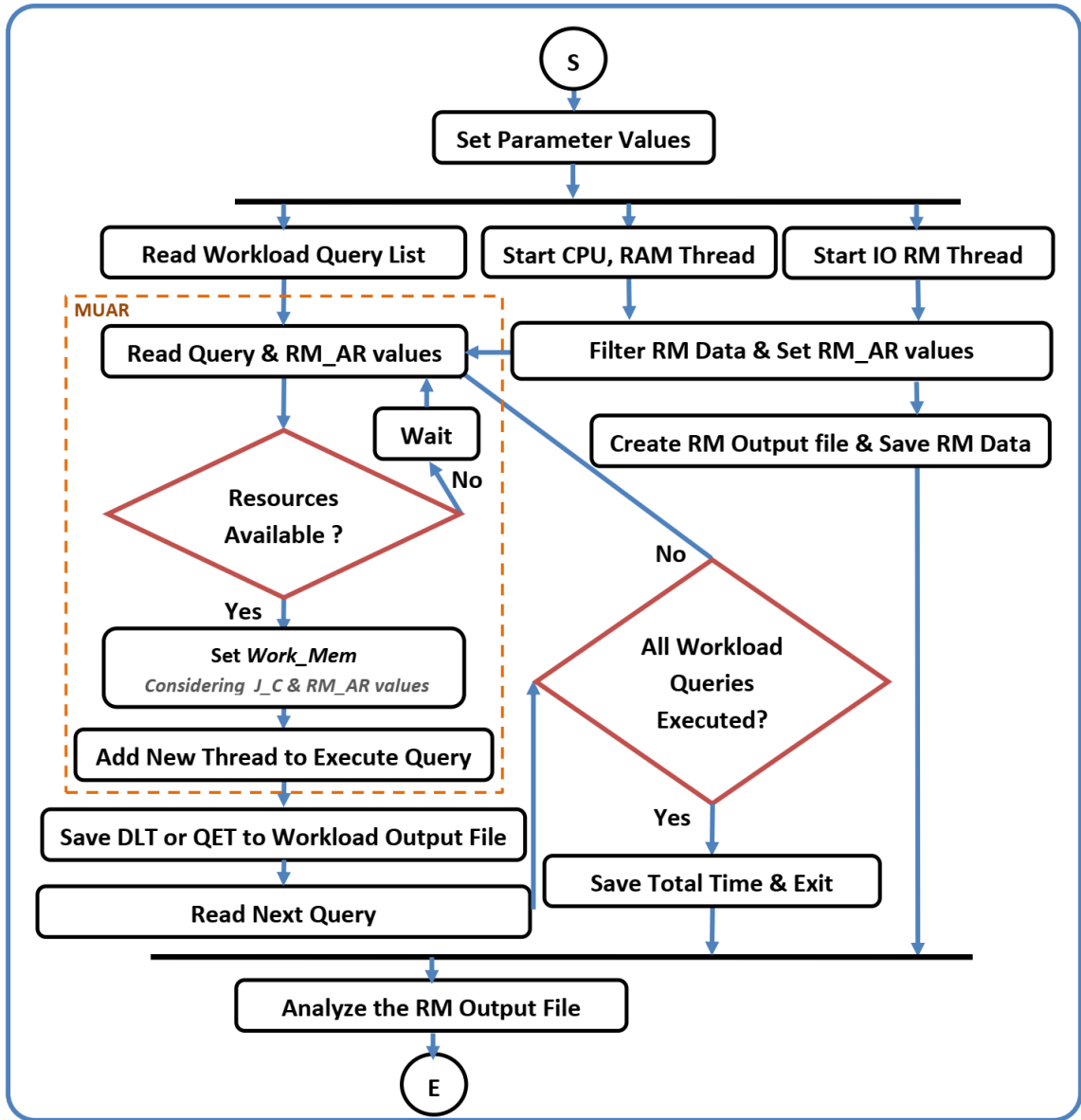


Figure 8.8: Experiment Flow: MUAR Technique

The MUAR algorithm receives the real-time resource monitoring data of CPU, RAM, and IO_wait in percentage from the resource monitoring module. The resource monitoring frequency is set to 1sec to keep the resource monitoring overhead on CPU resources less than 2%. The minimum resource requirement to start a thread *Min_AR* is set considering the minimum resources required to execute any query on the experimental machine. If available *CPU* is $\geq 20\%$, available *RAM* $\geq 10\%$ and *IO_wait* $\leq 50\%$ then thread is added to one of the four thread pools. The experimental machine had four CPU cores, thus four thread pools have been created. The RAM requirement of each query is different. Therefore, MUAR uses a lightweight *WM_Query* function to set appropriate memory to the given query considering Join Count (*J_C*) of query and available CPU, RAM, and IO resources (*RM_AR*) at runtime. This function can find the complexity of new queries by analyzing their SQL statements without using offline learning stages or other historical records. Once work memory value is calculated, a new thread and database connection is created to set work memory and execute the given query. The basic CPU maximization suffered from over-allocation of resources, making query tasks starve for resources and fail. In contrast, MUAR keeps upcoming tasks in a queue and assigns a new thread and *work_mem* only when enough resources are available. The total WET and QET for each query is recorded to result files for later analysis of results.

CHAPTER 9

Results & Discussion

This chapter discusses general tool configuration results and experiments performed using SDSS and LOD datasets. The results of both datasets have been explained separately to avoid confusion. The results of each phase have been written in their respective sections for better understanding. However, a summary containing improvements in WET achieved by optimization and maximization phases is discussed in the end. The QET, DLT, and WET parameters are presented in seconds. The dataset size is presented as million records (M). Algorithm execution time (AET) optimization has been conducted only for the WSAC technique because the cost function was dependent on the dataset size. In contrast, QCA and MUAR techniques do not require analysis of original raw dataset files reducing AET. The X and Y axis in all the results graphs show the comparison parameter, technique, tool, or resource.

9.1 Raw Data Query Processing and Resource Monitoring framework

This section presents basic results of Raw Data Query Processing and Resource Monitoring framework developed in Phase-I and II of the thesis. It compares the time and resources required by PostgreSQL (PgSQL) and NoDB (PostgresRAW) tools to complete the execution of a given SDSS workload on the dataset of 1M to 4M records. It contains comparison of both tools for QET, WET, and resource utilization parameters. The results for scaled data are also included.

9.1.1 Workload Execution on Raw

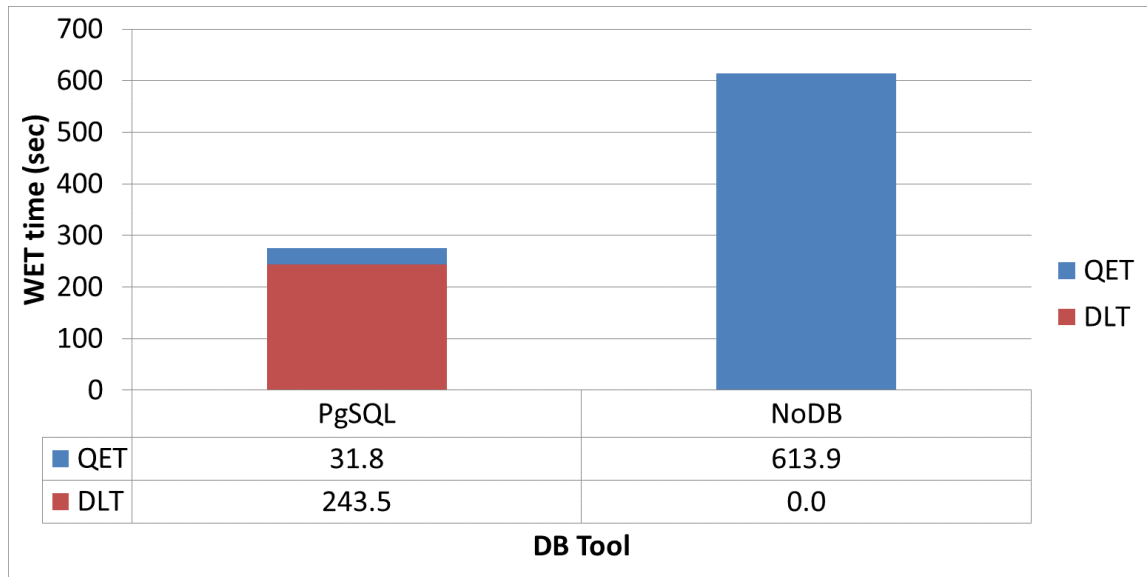


Figure 9.1: Workload Execution Time

The raw data query processing framework uses open source PostgresRAW, which is a NoDB implementation [34]. Figure 9.1 shows the comparison of WET time in PgSQL with NoDB for the SDSS dataset having 1M records. The 1M records dataset required 4.6GB of disk space in raw format. It can be observed that raw data query processing engines like NoDB have zero data loading time as they can start executing queries on the raw file immediately. At the same time, PgSQL required a considerable amount of time to load data existing in a CSV file with the fastest data loading technique COPY. However, PgSQL required only 4% time to execute 12 SDSS queries on loaded data compared to NoDB. Therefore, loading data into a database system is crucial to reduce QET of future queries.

9.1.2 Query Processing for Scaled Data

The data scaling experimental results have been presented in Figure 9.2. It can be seen that the workload execution time (WET) of PgSQL increases linearly with the dataset size. On the other hand, WET of NoDB increases exponentially after 1.5M. The 2M-4M experiments with NoDB took more than 2hrs without any results for several workload queries. One query Q5 took 10hr at 1M without re-

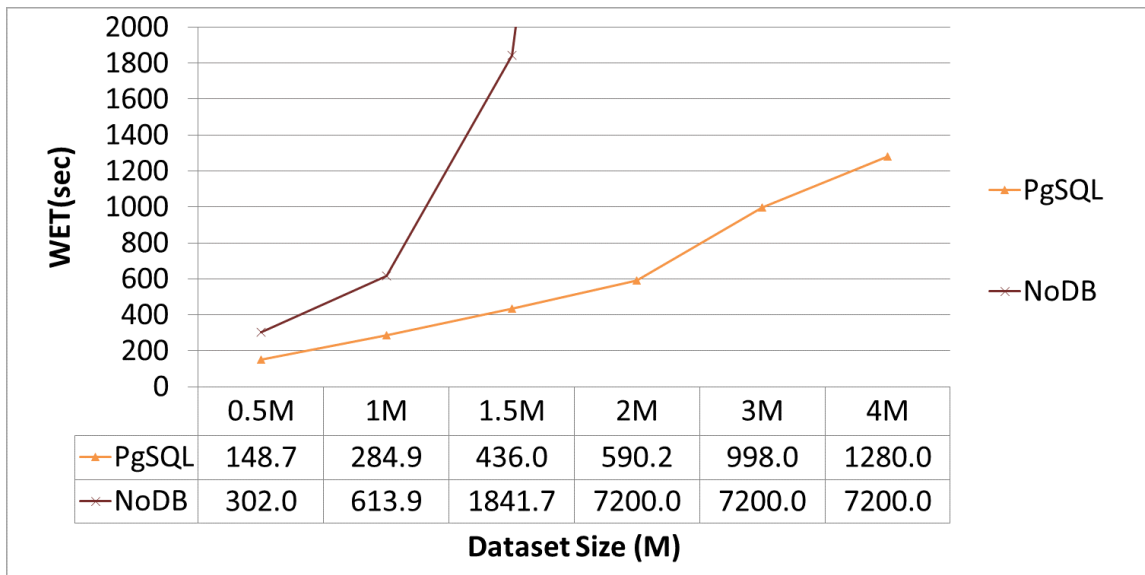


Figure 9.2: Scaled Data: Workload Execution Time

sults. Therefore, the experiments were stopped for 2M-4M records when NoDB could not provide results after 2hrs. The resource monitoring phase showed that almost all the remaining RAM(>14GB out of 16GB) is utilized by NoDB to cache and index data while processing dataset sizes of 1.5M and higher. NoDB requires 2-5x more time to execute workload queries compared to PostgreSQL for dataset sizes lower than 2M.

9.2 Resource Monitoring Tools Configuration

This section discusses the resource monitoring tools configuration settings. Objective of the experiments is to find out the resource efficient way of monitoring resources. Therefore, a resource monitoring tool needs to be tuned to utilize minimal resources. The *top* and *iostat* tools have multiple configuration settings, including filtering output details, sorting, and delays before refreshing resource utilization. The delay `-d` setting controls resource monitoring frequency for both tools. The freq. for both tools needs to be equal to correlate the outputs for analysis. Figure 9.3 shows the graph of resource monitoring frequency and CPU utilization. It can be seen that with the increase in monitoring frequency, CPU utilization increases for both tools. The CPU utilization does not increase more than 25% be-

cause the machine has four cores. The 25% utilization shows that the entire one core out of four is used to monitor resources from 1000 and above observations per second. The higher observation rate also increases result output file size. The *top* and *iostat* RM threads need to filter those monitored records before storing, which may utilize additional two CPU cores and increase IO utilization.

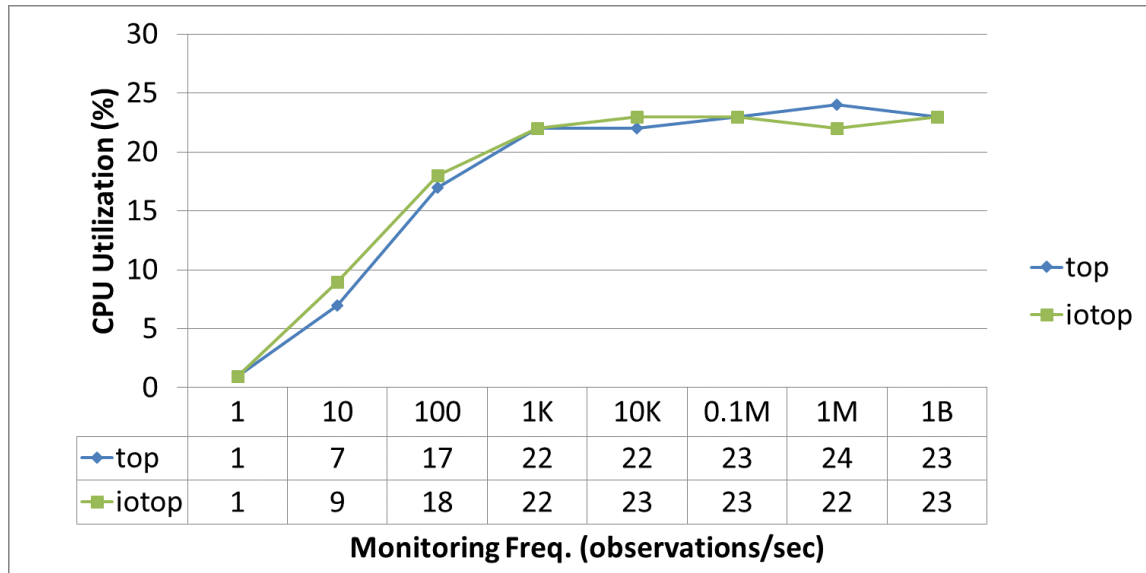


Figure 9.3: RM Tools: Overhead

CPU is the fastest resource of all resources. The CPU speed of 3.20GHz indicates that it can process 3.2×10^9 instructions every second. The machine used for experiments is a 64bit, which means the CPU can process $3.2 \times 10^9 \times 64\text{bit} = 23.84\text{GB}$ of data every second. However, CPU is dependent on RAM to fetch data from the disk. Now, DDR3 RAM speed is less than a few GBs per second, while magnetic disks can only provide 300MB of data per second. While accessing smaller files, disks can provide only 30MB of data per second due to the seek-delays of 2-4ms for each non-continuous data block. This means the change in resource utilization of RAM and IO resources is slow, while the CPU is dependent on them, which indicates the overall utilization change is not rapid. IO's 300MB per second data read speed represents less than 2% change in RAM resource utilization. Additionally, RM tools record resource utilization cumulatively, avoiding missed reading issues. Therefore, the monitoring frequency is set to 1 observation per second for most experiments to minimize resource monitoring overhead and keep resources

available for actual workload processing.

9.3 Resource Monitoring

The resource utilization results recorded by the resource monitoring (RM) module are presented here. This section also compares the resource utilization results observed by the RM module for PostgreSQL and NoDB. The changes in resource utilization patterns with increased dataset size observed during data scaling experiments are presented in Section 9.3.3.

9.3.1 Resource Utilization

PostgreSQL and NoDB follow different data processing techniques. This section provides insights into the actual resource requirements of these DB tools to process 1M records of SDSS dataset. PostgreSQL needs to load the entire dataset before query execution can start, while NoDB starts the execution of queries immediately on raw data. Figures 9.4 & 9.5 show resource utilization graphs of NoDB and DBMS. The RM tools provided CPU utilization, IO utilization, and CPU IO_wait in percentage. Read/write bandwidth was calculated using "total" read/write observations considering max read/write speeds of 300MB/sec and 200MB/sec, respectively. The "actual" disk read and write observations were not used because read/write speed can be higher than actual data read/write. It would not have provided any valuable insights. The tools provided IO utilization for each process, so the plotted IO utilization is the SUM of all *Postgres* and *Java* processes representing the raw data query processing framework.

Figure 9.4 shows resource utilization for PostgreSQL. It can be seen that CPU utilization during data loading tasks is 20%. This is because data loading is an IO-dependent process. The CPU had to wait for IO for data to be read from the disk. The high IO utilization and high IO_wait spikes confirm that observation. It can be seen that once data loading task is completed, read and write to the disk are almost zero. It is because the entire dataset gets cached in the main memory to reduce QET of future queries. However, the IO and IO_wait readings reflect

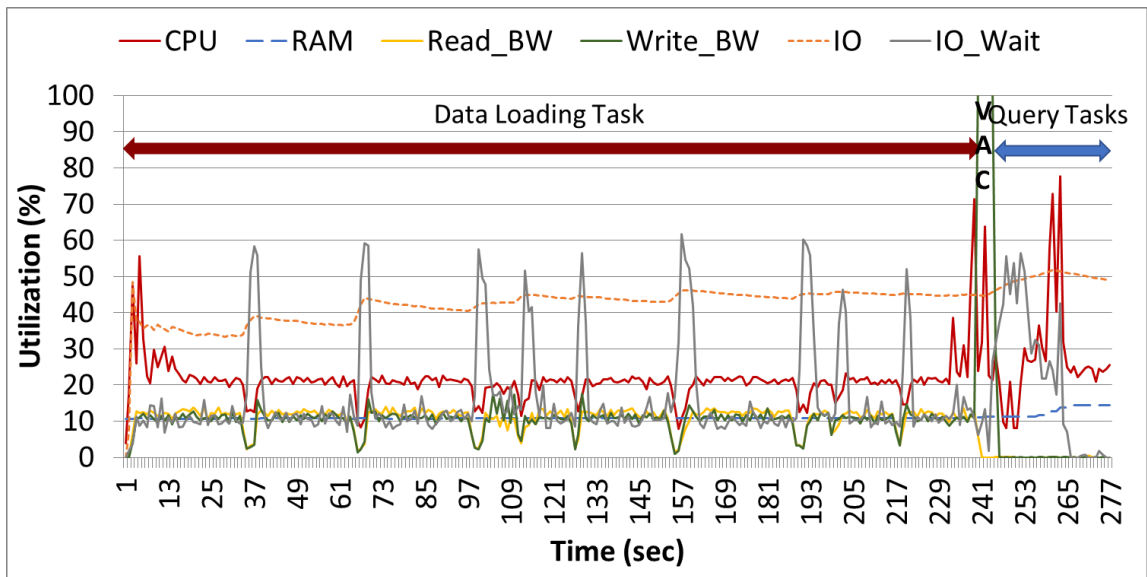


Figure 9.4: DBMS (PgSQL): Resource Utilization:

those changes a little later due to slower reflection of storage hardware utilization readings and VACUUM process reclaiming space delaying access to disk for most processes. The VACUUM process is behind a high write spike after data loading was completed. Here, VACUUM removes old tuples, facilitating faster disk access to the new data. The query processing tasks increased RAM utilization by 3.1% as the query processing tasks process the cached data in main memory to provide results. The query processing tasks utilized a single CPU core, which is 25% for a quad-core CPU. The *Postgres* processes like wal, checkpointer process, parse, bind, and other processes, including Java processes, could utilize up to 77% CPU resources but only for one second.

Figure 9.5 shows resource utilization for the NoDB tool. The tool accesses the data residing on the disk after the arrival of the query. For the first two queries up to 262 sec, NoDB accessed the required data from disk, which increased IO utilization by more than 20%. The raw engine caches and indexes the data into the disk. Therefore, there are zero disk reads during the remaining time of workload execution. The raw engine does not write processed data back to disk, utilizing write bandwidth almost 0%. However, the caching and indexing of raw data into main memory increased RAM utilization up to 76%. The CPU wait is almost zero once all data is cached in RAM. Therefore, query processing tasks utilized

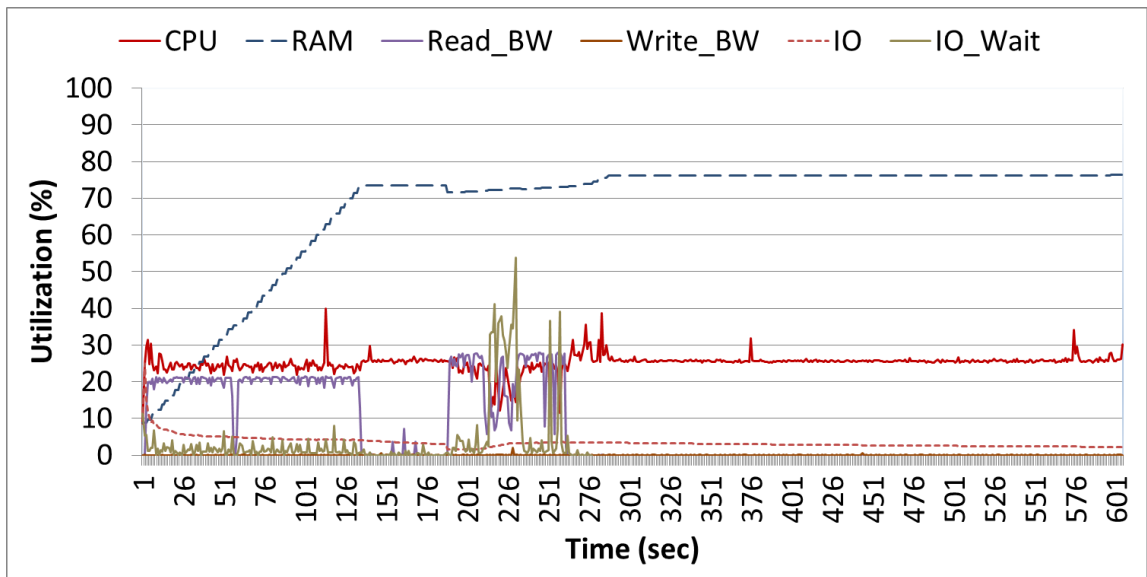


Figure 9.5: NoDB: Resource Utilization

allocated CPU core completely, unlike PostgreSQL.

9.3.2 Average Resource Utilization

The average resource utilization of both tools is plotted in Figure 9.6. It can be observed that NoDB utilized 3% more CPU than PostgreSQL due to 9x lower IO_wait. The RAM utilization of NoDB is 6x more than PostgreSQL because of the caching and indexing of data in main memory.

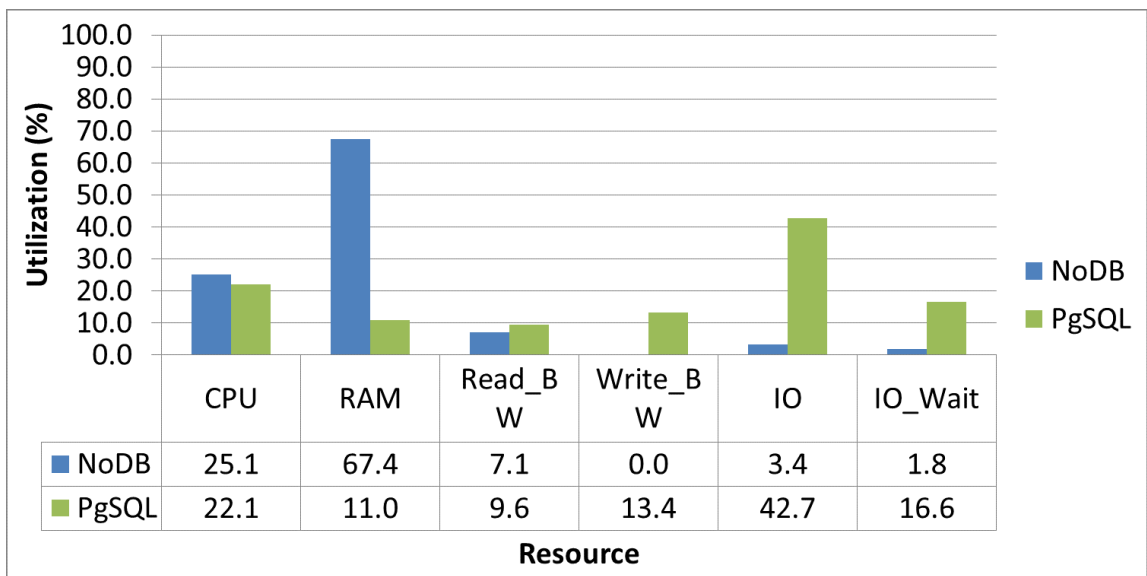


Figure 9.6: Average Resource Utilization

However, write IO utilization of NoDB is zero due to no data loading, and read IO utilization is 2.5% less due to in-memory processing of queries. It can be noted that both tools require different resources. NoDB utilized RAM most, while IO resource is critical for PostgreSQL data processing. These results conclude that both tools require different resources to perform raw data query processing tasks. Therefore, once raw engine caches the data for query execution, the IO resource is available for data loading.

9.3.3 Resource Utilization for Scaled Data

This section discusses the data scaling experiment results for both tools. Most researchers experiment with datasets smaller than the main memory size [117]. However, the experiments performed in this section used 18.8GB dataset size, which is 2.8GB larger than the RAM size. The data scaling experiments were performed by partitioning 18.8GB dataset file into eight smaller partitions, each containing 0.5M records. The 0.5M partitions were saved as CSV files, and these CSV files were loaded using the COPY command, followed by query execution tasks.

9.3.3.1 DBMS

This section discusses data scaling results performed using PostgreSQL DBMS using SDSS dataset.

Figure 9.7 shows average resource utilization readings for the PostgreSQL processing SDSS dataset. It can be seen that CPU utilization is below 21% and decreases by 1-2% with the increase in dataset size. This transpires because IO_wait increased from 15.4% to more than 21% for dataset sizes larger than 1M. The RAM utilization by *Postgres* process increased linearly from 0.4% to 1.6% for dataset sizes 2.4GB (0.5M) to 18.8GB (4M). The overall RAM utilization calculates the database files cached for processing queries. Therefore, the RAM utilization is plotted after including dataset size and *Postgres* process RAM utilization. It can be seen that RAM utilization increased from 8.5% to 66.6%, while IO_wait increased by 9.7% with dataset size. The CPU, Read, and Write bandwidth utilization stayed

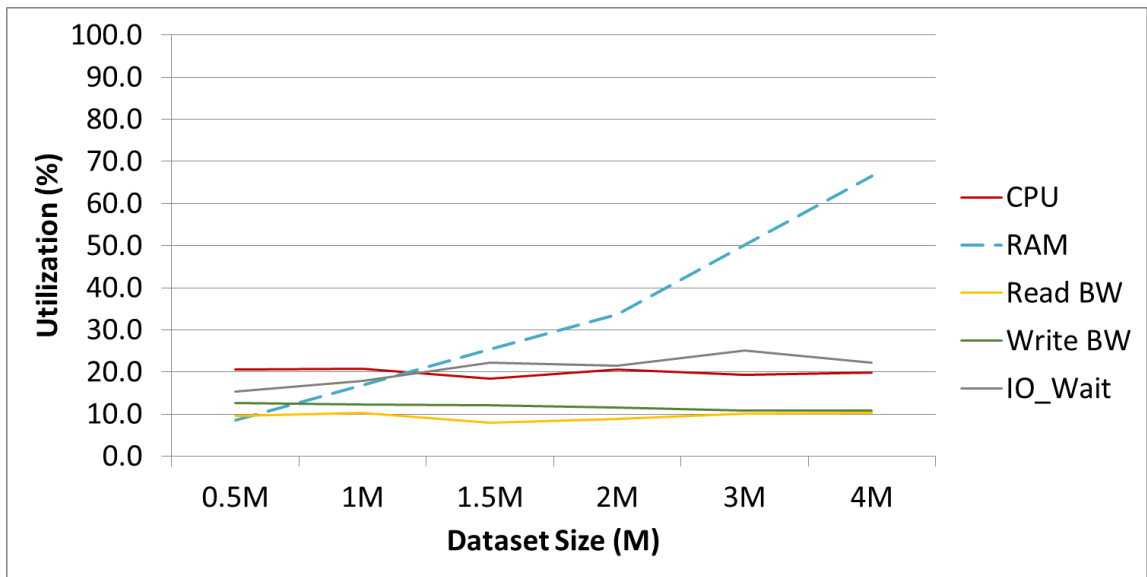


Figure 9.7: DBMS (PgSQL): Resource Utilization

below 21%. The *Postgres* process used only one core to load and query, following similar patterns shown in Figure 9.4.

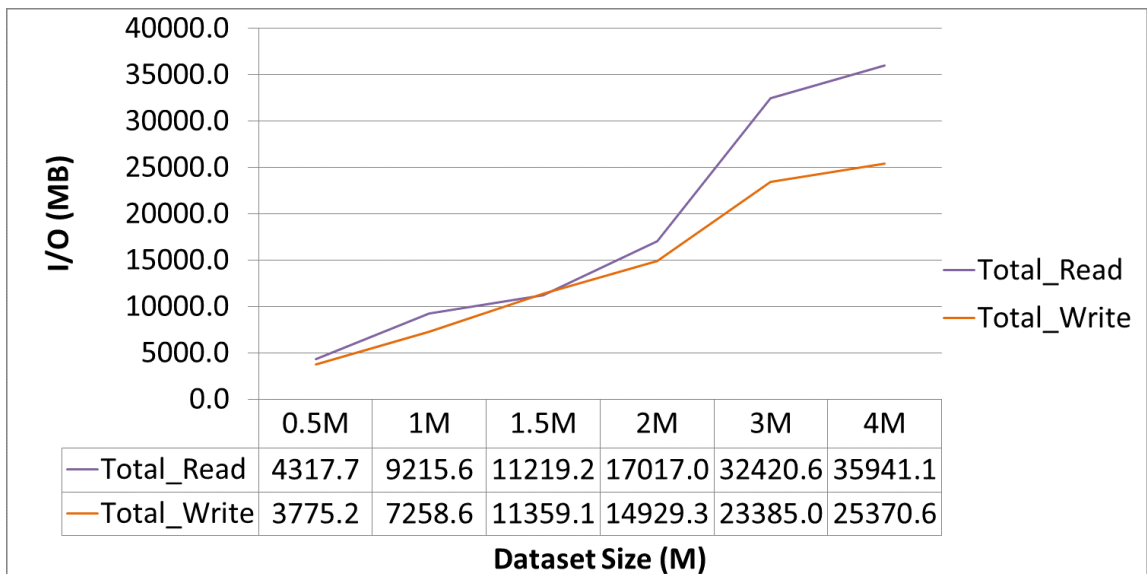


Figure 9.8: DBMS (PgSQL): Total IO Utilization

Figure 9.8 shows total IO utilization for PgSQL DBMS. While, Figure 9.9 shows the comparison of dataset sizes in database and CSV raw file size. It can be seen that disk space utilization of database is 45% smaller than the raw file size. The *PhotoPrimary* table of SDSS consists of large number of numerical attributes. Numerical values take less space in binary format compared to their string format.

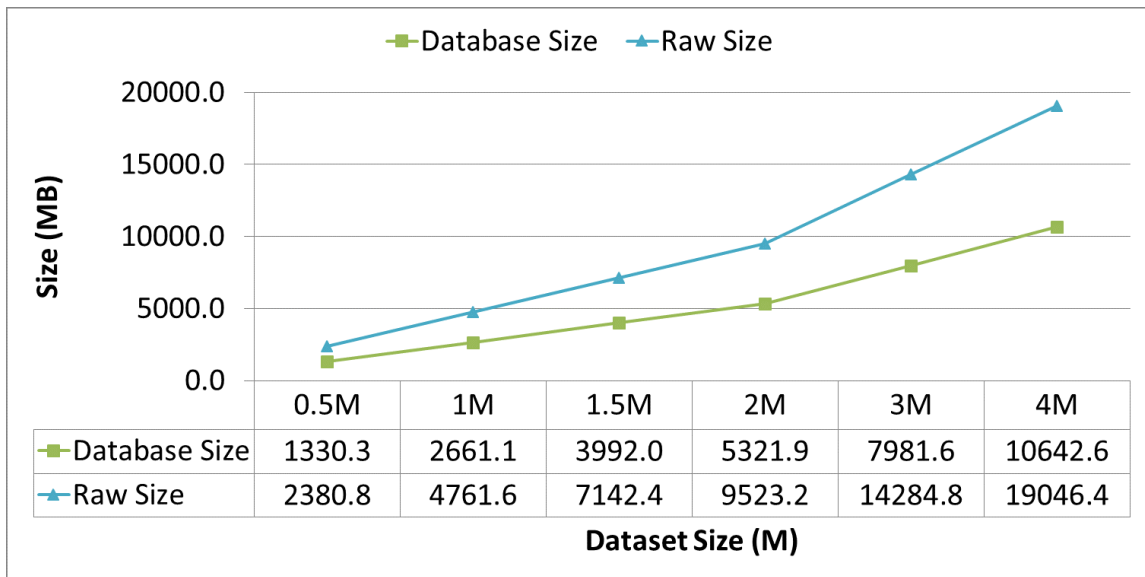


Figure 9.9: Storage Space Utilization: Database (PgSQL) vs. Raw (NoDB)

The IO utilization shows that total write is 1.5x, and the total read data size is 2.2x compared to the raw file size for PgSQL. The COPY command reads actual raw files having 4.7GB size and writes the compressed DB file of 2.6GB. The WAL writes logs of all inserted records in the non-compressed format before loading data, utilizing the write bandwidth by 1.5x. The *Postgres* process only reads the compressed database for query processing if not cached in RAM. The caching of database files in the main memory increases RAM utilization and Total Read size. The VACUUM, stat collection and check pointer processes use IO resources to arrange data on disk and collect statistical data.

9.3.3.2 NoDB

The experiments performed in this section use a single large raw data file to answer queries using in-situ processing NoDB. Figure 9.10 shows the resource utilization for data scaling experiments from 0.5M to 1.5M. The average RAM utilization increased up to 67% for 7.1GB (1.5M) file size. This means for 7.1GB of raw data additional 4GB of main memory is utilized by NoDB. The NoDB utilized more RAM to cache and index entire dataset. However, maximum RAM utilization had already reached 96.3% for 7.1GB (1.5M), and SWAP memory use had begun. The *Postgres* process had used 5-50% SWAP space in two different

runs. The *Postgres* process went into sleeping mode, and other processes started crashing due to nearly 100% RAM utilization and high IO_wait. Therefore, data scaling experiments had to be limited to 7.1GB. The high IO_wait can be seen at 7.1GB point, which reduced CPU utilization by more than 12% compared to 4.7GB.

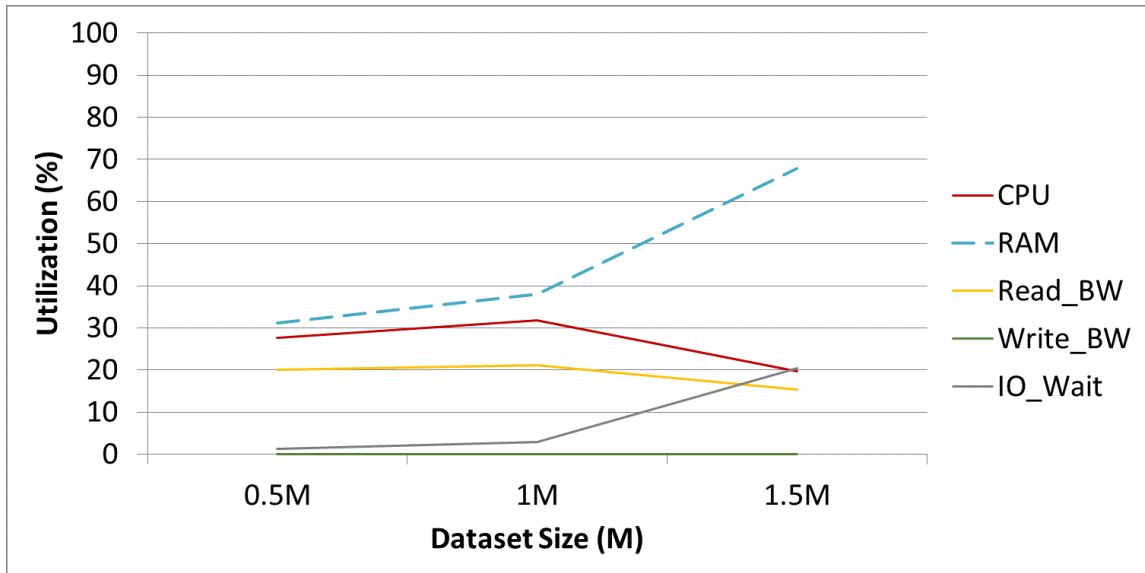


Figure 9.10: NoDB: Resource Utilization

Figure 9.11 shows total IO utilization by NoDB tool to process SDSS dataset for 2.4GB (0.5M), 4.7GB (1M), and 7.1GB (1.5M) scaling experiments. It can be seen that the total read size is around 1.7x as compared to raw data file sizes. The detailed analysis of the *Postgres* process showed that the total data read was exactly same as the raw file size. The additional data read might be due to OS and other processes caching the data from disk. The total data written to disk is less than 10MB which was written by RM threads of raw data query processing framework and stat collector process of *Postgres*.

The results analysis has shown that NoDB required 64.4% around 10.3GB of RAM to run queries on 4.6GB of raw data, limiting the data scaling as it would have triggered swapping and increased WET. The RAM utilization of *PgSQL* is around 3x less, while NoDB has almost zero write bandwidth requirements. The *PgSQL* is highly dependent on IO speed, while NoDB is on RAM size. The CPU utilization of both tools is less than 31%. The results showed that *PgSQL* and

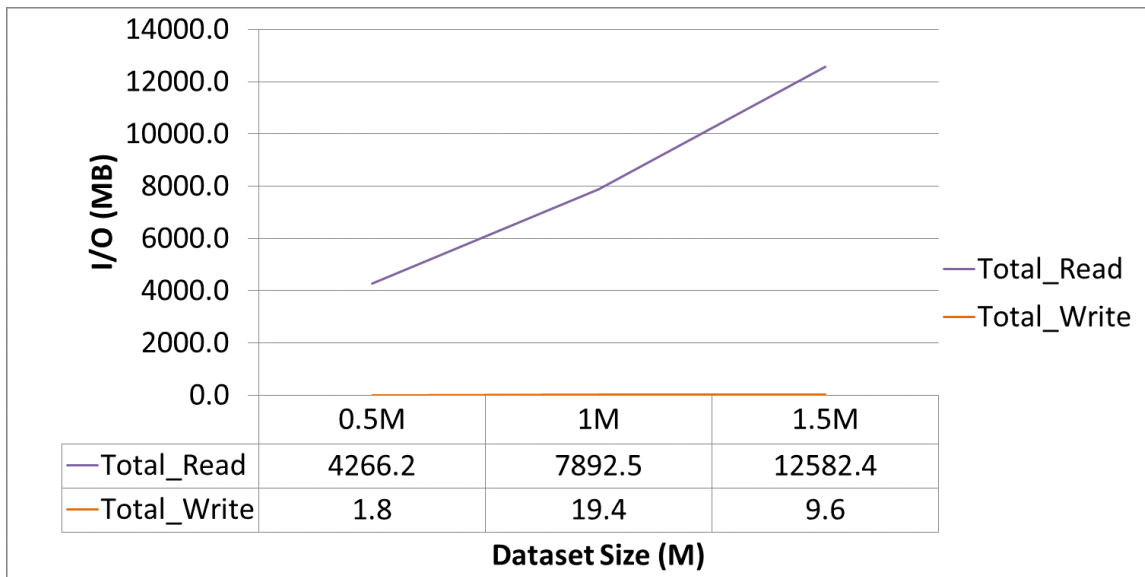


Figure 9.11: NoDB: Total IO Utilization

NoDB tools used only one CPU core for data processing tasks. It is because most traditional systems are designed with OLTP queries in mind, requiring sequential query task processing. Parallelizing data processing tasks for each query is difficult because it increases the overhead of identifying if a task is dividable without affecting the final result. The parallelization of a task needs division into smaller subtasks, distribution of subtasks, and combining intermediate results of those subtasks to produce the final result. The Cloudera and Hadoop based systems are highly parallelized systems. However, they required more resources to handle these additional parallelization tasks, increasing workload processing time.

9.4 Query Classification

This section considers QET time and resources required by each query to classify and group similar queries. This classification helps identify what types of queries can be executed using which data processing tool.

9.4.1 Query Execution Time

The experimental results presented here used SDSS dataset having 1M records size and 12 workload queries. Figure 9.12 shows the QET of workload queries ex-

executed in sequence using one process thread. It can be observed that few queries are executing faster in NoDB compared to PostgreSQL. NoDB caches the entire dataset into main memory, which improved QET. The figure showed that there 4 queries that had better QET. This means some earlier queries might have cached the data required by these four queries, which improved their QET. The SQL statement analysis showed that all 4 queries were single join queries. Therefore, the next Figure 9.13 presents classification based on number of joins in a query. It shows average query execution time of queries group based on Join Count. The analysis showed that out of 5 simple queries 4 outperformed PostgreSQL. In contrast, 1 or high join queries had high QET.

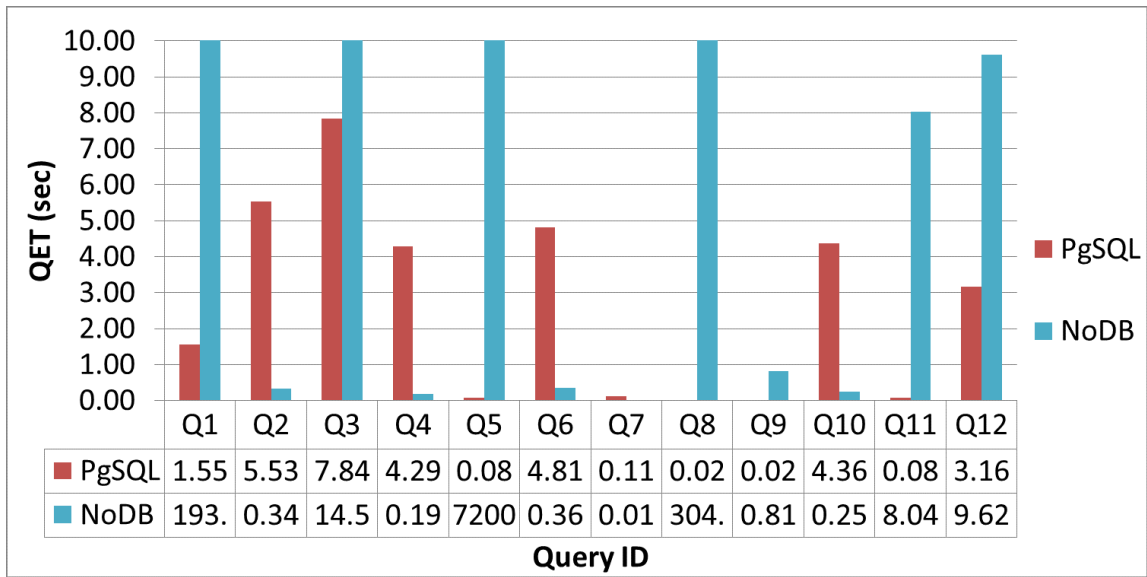


Figure 9.12: SDSS: QET comparison

To verify the finding, the experiments were repeated with LOD dataset and query set with higher join counts. The results presented in Figure 9.14 showed that 2 and 5 join count queries performed worst in NoDB. Therefore, they must be executed using PostgreSQL. However, 2 queries having -1-Join and 3-join performed better in NoDB than PostgreSQL. 1-join query might have better QET due to smaller dataset and all required data are pre-cached. However, 3-join query is an exception as it falls in the complex query category.

3 join query analysis: The 3-join query improvement needs to be analyzed as it does not fit with our earlier explanation. Further analysis of results and dataset

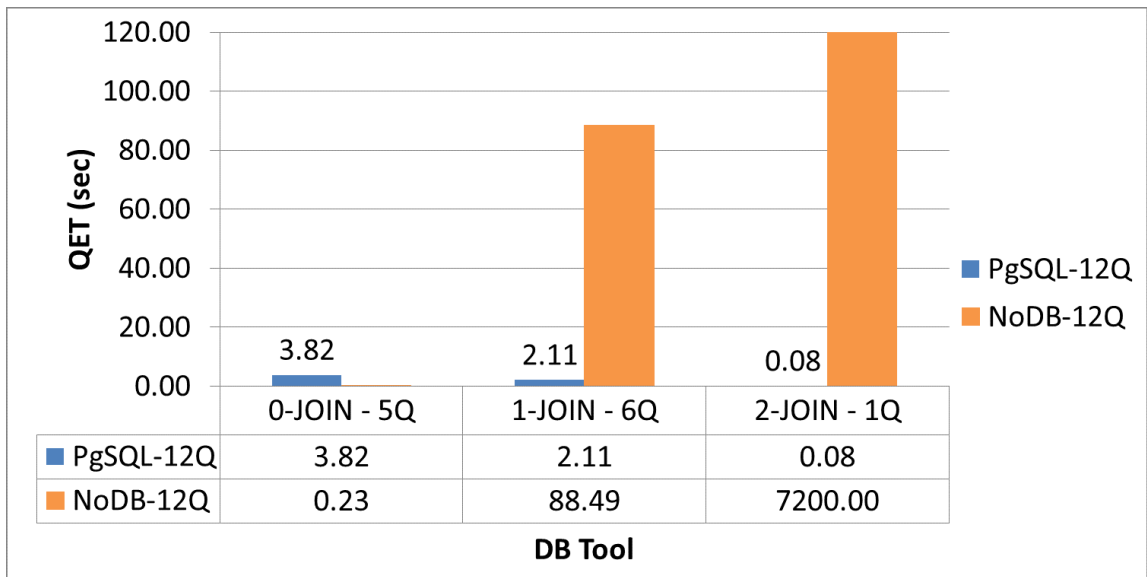


Figure 9.13: SDSS: Query Classification based on Join Count

data showed that this query had only one record in the results. This means query had high selectivity (number of distinct values/total number of rows). Therefore, less records will be there in join process reducing QET time. Therefore, it can be said that NoDB can handle simple 0-join queries or queries with high selectivity. On the other hand, queries having 1 or more joins with low selectivity should be executed using PostgreSQL. Because results have shown that PostgreSQL is well optimized to execute such multi-join (complex) queries.

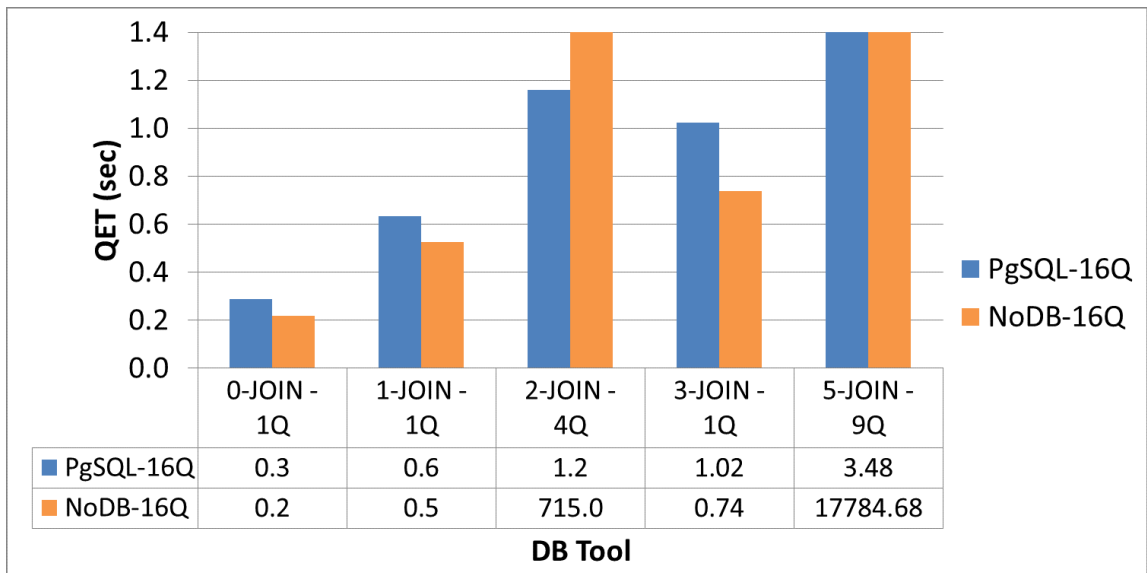


Figure 9.14: LOD: Query Classification based on Join Count

From the query classification results shown in Figures 9.13 and 9.14, it can be determined that 0 JOIN simple queries can be executed using NoDB while execution of 1 or more JOIN complex queries (CQ) should be done using PostgreSQL after the dataset is loaded into a DBMS. The resource monitoring experiments done in the next section tries to find how much resources are utilized by both tools during workload execution. The analysis would provide insights into the possibilities of parallel execution of queries using NoDB while data is loaded into PostgreSQL.

9.4.2 Resource Utilization

The analysis of resource requirements of both tools established that NoDB does not need IO resources after the data is cached and indexed in the main memory. While PostgreSQL needs IO resources longer to load data into DBMS. Both tools used a single CPU core. Therefore, CPU resources are available to handle both processes in parallel. However, high RAM utilization causes processes to stop in NoDB, and high IO_wait increases data loading time in PostgreSQL. The experiments performed in this section analyze individual query resource requirements in cold and hot runs. This experiment will observe how much resources each query needs. Section 9.4 suggested that complex queries should be executed using DBMS, and the resource utilization result Figures 9.2, 9.6, and 9.9 already established that DBMS (PostgreSQL) required 6x less RAM, half storage size and even fewer CPU time(QET) compared to NoDB. It can be derived that simple queries in PostgreSQL also use less RAM than NoDB because total RAM utilization is 6x lower. However, NoDB executed simple queries faster than DBMS. The next section tries to find how much resources NoDB utilizes to execute simple queries.

9.4.2.1 Data Caching

This section tries to find out RAM and IO resources required by NoDB to execute simple queries. NoDB is not optimized to execute complex multi-join queries, as results have shown high QET time. Therefore, complex queries are not considered, as they can be executed using PostgreSQL using 6x less RAM, CPU, and IO

resources.

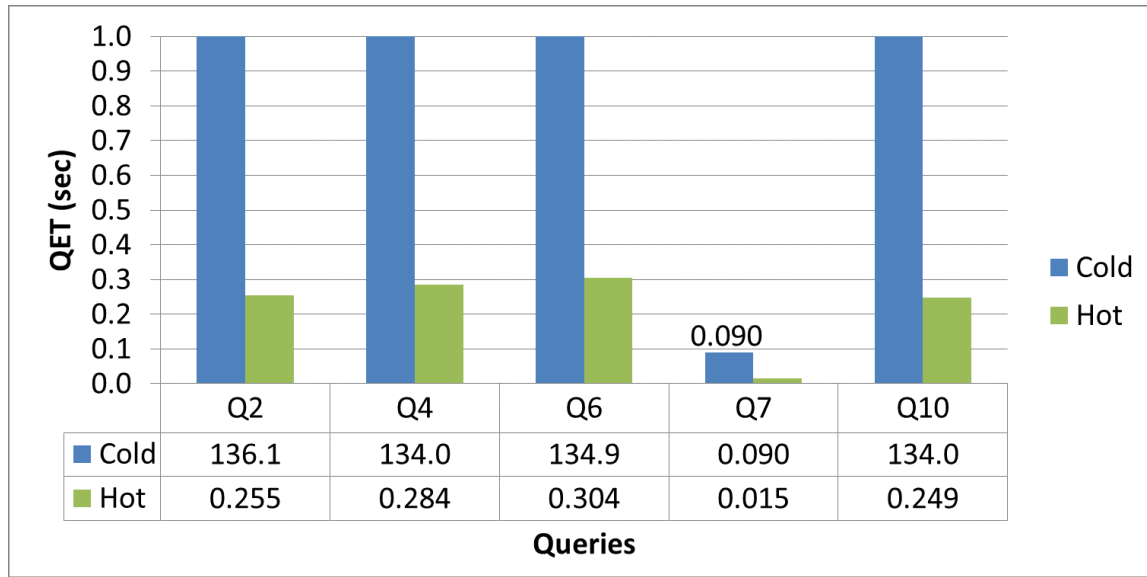


Figure 9.15: NoDB: Simple Queries using Caching

The caching experiments executed each simple query 4 times using NoDB to find individual query resource requirements. The time required to complete the first run is cold run QET, while the average QET of the remaining 3 runs is considered hot run time. The cache is cleared before executing each simple query to make sure no query uses preprocessed data cached by NoDB. Figure 9.15 shows the difference between Cold and Hot runs of simple queries executed using NoDB of 1M records dataset. It can be seen that Q2, Q4, Q6, and Q10 require around 134seconds in processing raw data when processed data is not in main memory, similar to the data loading task of PostgreSQL. However, PostgreSQL required 109sec more to load data into DBMS before queries could be executed. The cold runs of Q7 provided the query results within 0.09sec, which is 2700x faster than PostgreSQL. It means 2700 queries like Q7 can be executed using NoDB before PostgreSQL can finish data loading tasks. PostgreSQL caches the entire dataset into the main memory alongside data loading tasks. A comparison of Cold and Hot runs for PostgreSQL had shown only 14sec difference. The difference between Cold and Hot runs of NoDB is higher than PostgreSQL because PostgreSQL only needs to cache the created database from the disk. On the other hand, when the NoDB cache is cleared, it needs to process and index the entire raw dataset every time.

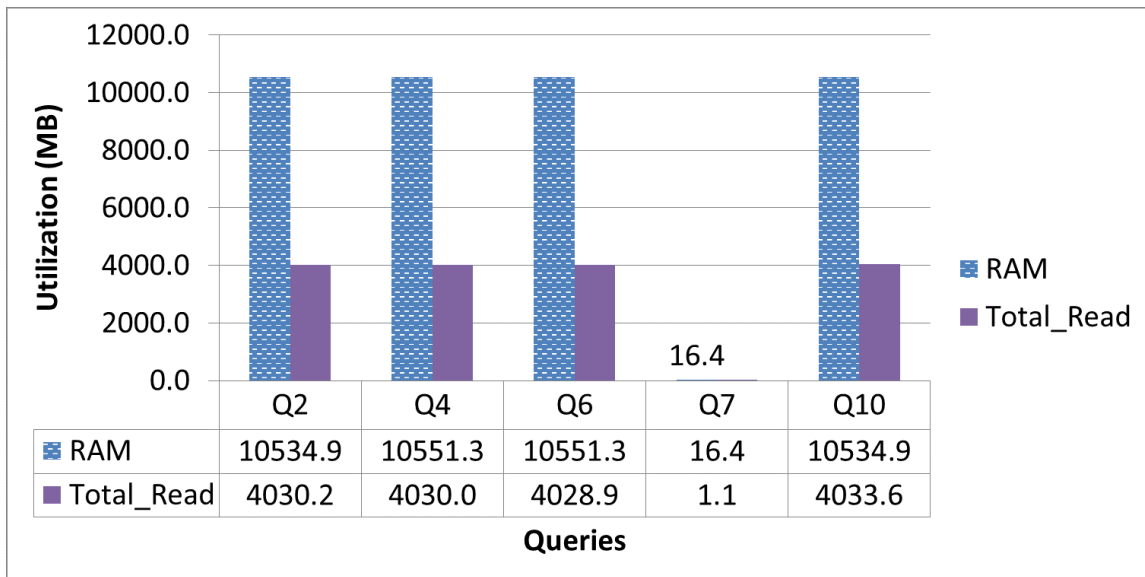


Figure 9.16: Resource Utilization of Simple Queries

Figure 9.16 displays RAM and IO resource utilization of all simple queries executed using NoDB. It can be observed that Q7 requires less than 2MB of data to be read from disk, and RAM utilization was also less than 0.1%. The query Q7 is a sampling type query. Q7 needs to find top 10 records satisfying some conditions. Therefore, the query execution stops as soon as the Q7 result touches 10 records. At the same time, other simple queries required almost entire dataset to provide accurate results. Therefore, Q2, Q4, Q6, and Q10 utilized 10.3GB of RAM after accessing 4GB data from disk.

9.5 Summary of Raw Data Query Processing and Resource Monitoring framework Results

This section summarizes key important findings of Phase-I & II results. The findings listed below have been used as facts or rule of thumb to solve the partitioning, task scheduling, and resource allocation problems and develop QCA, WSAC, and MUAR techniques.

- Phase-I & II identified three types of queries, 1) Simple queries (0-JOIN), 2) Complex queries (1 or more JOIN), and 3) Sampling queries (0-JOIN).

- The query classification results determined that 0 JOIN simple and sampling queries can be executed using NoDB. While the execution of 1 or more JOIN complex queries (CQ) should be done using DBMS (PgSQL) after the dataset is loaded.
- Resources are underutilized as both tools can not utilize all resources efficiently. NoDB utilized 6x more RAM compared to PgSQL. However, IO utilization is almost zero once data is cached in the main memory. Both tools utilize less than 26% CPU, which represents a single CPU core.
- The SDSS dataset size reduces by 45% in database format compared to raw format.

9.6 RAW-HF: Optimizing Required Resources

This section presents QCA and WSAC results proposed in Phase III of the thesis to optimize resource utilization. Both techniques have been compared with state-of-the-art in-situ engine NoDB [33] and workload aware Partial Loading technique [125]. The WET time taken by QCA is compared with Partial Loading technique, while the number of attributes covered by WSAC for a given storage budget is compared in Section 9.6.2.1.

9.6.1 Query Complexity Aware (QCA) Partial Loading Technique

This section discusses the results of partial loading experiments based on partitions provided by the QCA technique.

9.6.1.1 QCA: Partial Loading

Figure 9.17 compares Query Complexity Aware (QCA) partitioned dataset's workload execution time (WET) for Case-I, II, & V with the original dataset WET and Workload Aware (WA) partitioning techniques like Partial Loading [125] & WSAC having enough storage budget B to load all attributes. Case-III & IV are not presented because all workload queries require joining with CAP partition, increas-

ing QET time compared to other cases. The NoDB is the original dataset WET. In a sequential run, NoDB takes 87.8% of the time executing complex queries, while simple query execution time is only 12.2%. One of the simple queries took 74.1sec due to raw data access, while the remaining four were executed in just 0.8sec. The WA technique load only required 54 workload attributes, reducing WET by 94.6%. The QCA Case-I load all 34 attributes required by complex queries in the database, which improves CQ QET by 98.7%. The CQ QET improvement of Case-I is 2.5% compare to WA due to the smaller partition size, as shown in Figure 9.17.

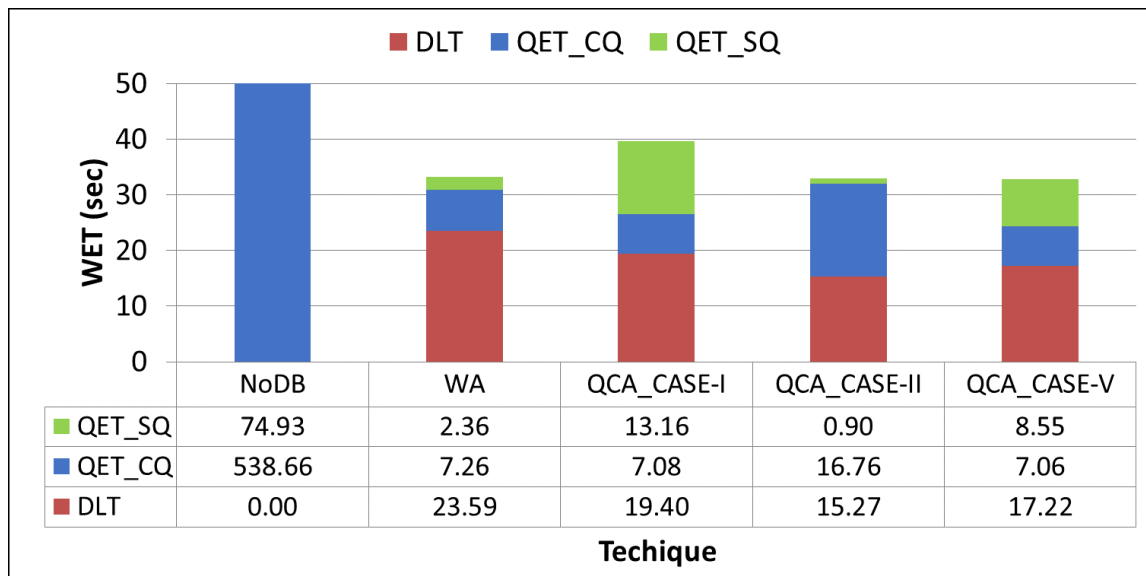


Figure 9.17: QCA: Data Distribution CASEs

The SQ QET is higher than WA in Case-I because attributes required by simple queries reside on the storage device until the first SQ accesses the raw partition. Case-II has the lowest DLT due to the fewer attributes getting loaded into the database. The access to CAP partition residing as raw format increases QET time of complex queries. However, Case-II achieved the lowest QET time for simple queries because complex queries had already cached and indexed the raw data before the execution of SQ started. The Case-V represents the replication of CAP on both formats. All the data required by each workload query is kept in a single partition, which means no additional joins are needed. The CQ achieved the lowest possible QET while SQ had to access the data from raw, which increased

the SQ time by almost 9.5x times compared to Case-II. The Case-I, II & V reduced overall WET by 93.53%, 94.63%, and 94.64% compared to the original dataset WET of NoDB [33].

9.6.1.2 QCA: Resource Utilization

The results discussed in this section provides insights to reduction of resource utilization in terms of RAM, database accessed partition size (DB_APS), raw accessed partition size (Raw_APS), and Total Read/Write required by hybrid system to execute given workload.

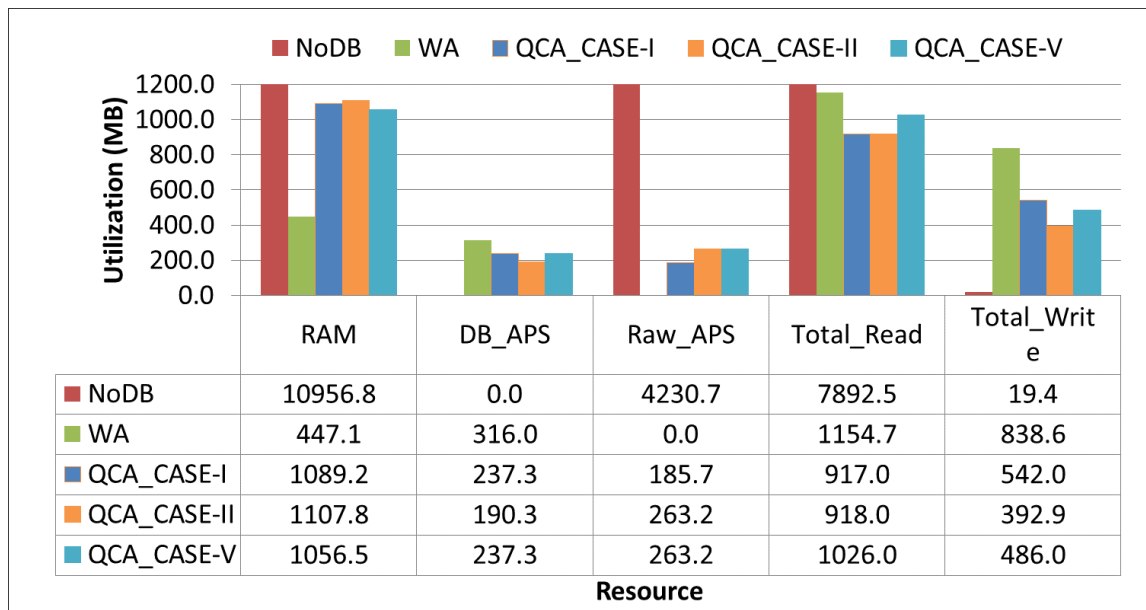


Figure 9.18: QCA: Resource Utilization

Figure 9.18 compares resources required by the NoDB, WA techniques, and QCA cases. The RAM utilization is reduced by 89.9 – 95% by QCA and WA techniques. The QCA technique required 57-59% more RAM than WA because WA required a single DB tool. In comparison, QCA uses DB and Raw engine tools, which increases its RAM utilization. The accessed partition size APS of DB and Raw is displayed in the graph. All the APSs of WA and QCA cases are less than 7.5% in size compared to the original dataset size of 4.1GB. Smaller partition size reduces disk access, and required partitions occupy less RAM than the original non-partitioned dataset. The in-memory caching and indexing also become more manageable. The overall WET for WA, Case-II, and Case-V remained similar in

single-core sequential workload execution. Therefore, WA and QCA techniques have been applied to a multi-node and multi-core setup to identify their benefits and limitations.

9.6.2 Workload and Storage Aware Cost-based (WSAC) Partial Loading Technique

This section discusses the results of partial loading experiments based on partitions provided by the WSAC technique. The WSAC technique is motivated from the fact that loaded data require 45% less disk space. WSAC proposes to consider actual storage size of attribute in DBMS to efficiently utilize given storage budget.

9.6.2.1 WSAC: Storage Resource Utilization

Figure 9.19 compares attributes covered by the algorithm choosing a static number of attributes as storage budget [125] with WSAC. The red bar shows fixed attributes as a storage budget named without WSAC. The green bar represents WSAC without AUF, which used only Query Coverage (QC) sub-algorithm, and the blue bar shows attributes covered when WSAC used QC and AUF sub-algorithms for different storage budget options. We assumed that without WSAC, the covered attributes might have been calculated based on the average size of attributes that can fit in a given storage budget B . The average size of attributes for *Photo-Primary* table columns is 38.41MB. That means without WSAC average number of attributes covered for 200, 800, 1400, and 2000 MB storage budget B would be 5, 21, 37, and 53. The number of queries covered by WSAC algorithms is 1, 5, 9, 12 for a corresponding storage budget of 200, 800, 1400, and 2000MB. It is crucial to cover all query attributes to get results using only the loaded part of the dataset to improve the QET. There is no guarantee that most frequent attributes would cover queries of the frequent query set in Without WSAC cases where QC is not a sub-algorithm. WSAC without AUF covered 2, 18, 28, 54 attributes, while WSAC covered 5, 22, 39, 54 attributes for the given storage budget.

The results show that the WSAC without AUF algorithm alone could not uti-

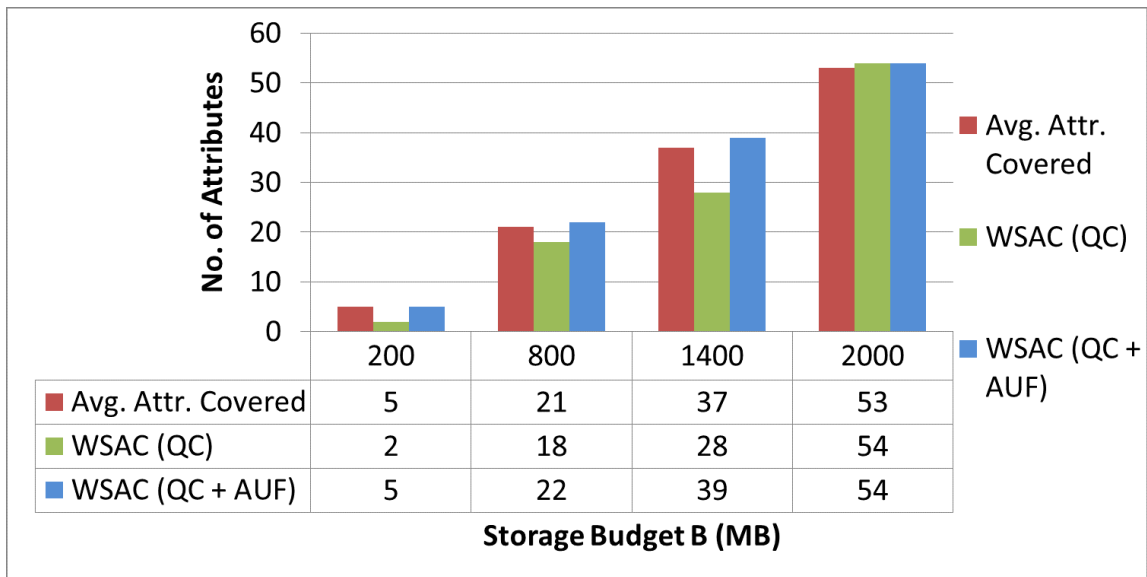


Figure 9.19: WSAC: Storage Utilization

lize all the available storage budget. The WSAC’s AUF sub-algorithm tries to utilize the remaining space with attributes having lower storage budget requirements in the end to improve storage budget utilization. The WSAC improved the storage utilization by 20-60% compared to WSAC without AUF. For the *PhotoPrimary* table, WSAC technique covered 5% more attributes than the algorithms choosing average values to decide storage budget represented as without WSAC. When stored in a database, most columns in the *PhotoPrimary* table had long int or real datatypes having similar column sizes due to the 4-byte storage space requirement on disk. The WSAC can cover more attributes if the dataset consists of boolean or char(1) datatypes that require only 1-byte storage space to store one tuple.

9.6.2.2 WSAC: Partial Loading

This section compares the WSAC performance compared to NoDB and PostgreSQL. The experiments performed here show the partitions provided by WSAC. The entire workload of 12 queries was provided as list of complex queries to WSAC algorithms.

The comparison of PostgreSQL with NoDB is plotted in Figure 9.1 of RQP to get the actual WET required by the original non-partitioned dataset having 509 attributes

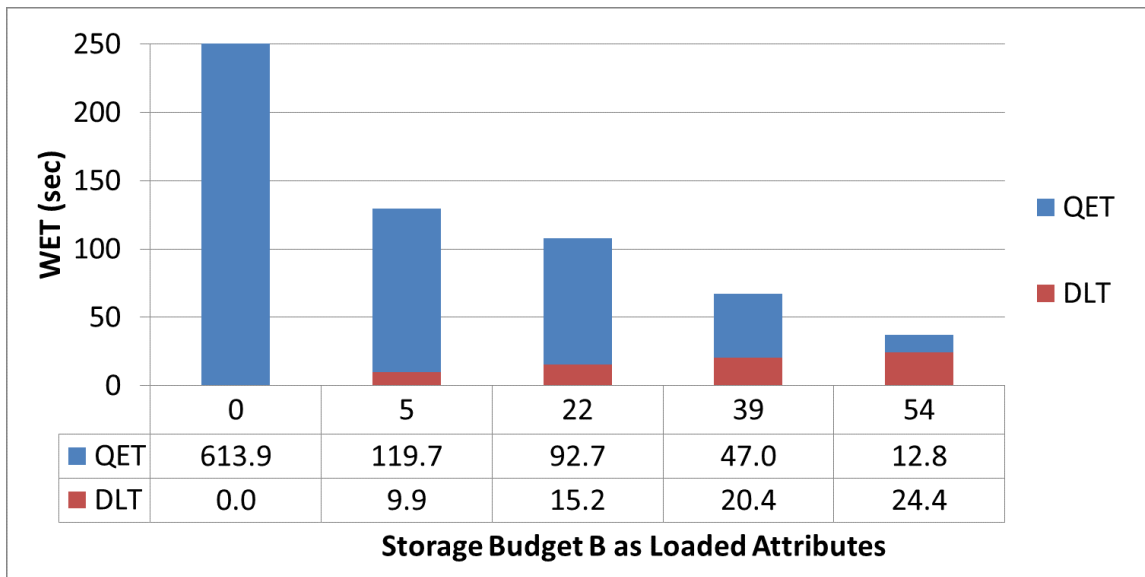


Figure 9.20: WSAC: Partial Loading WET

for both systems independently. It can be observed that raw data query processing engines like NoDB have zero data loading time. At the same time, PostgreSQL required 188 sec to load the data existing in a CSV file using the fastest loading method COPY. However, PostgreSQL required 4.05% time to complete the query execution of 12 queries on loaded data compared to NoDB. One thing to note is that NoDB is an open-source tool developed by research group from EPFL. Therefore it is not entirely created with industry standards nor optimized to execute JOIN queries as effectively as PostgreSQL. The NoDB required 11GB of RAM to run queries on 4.6GB of raw data, limiting the data scaling as it would have triggered swapping and increased WET.

Figure 9.20 compares total WET (Equation 6.2) before and after the proposed WSAC technique partitioned the *PhotoPrimary* table into three partitions from which workload queries use only two partitions. The first bar in the result shows the QET time when zero attributes are loaded into the database, which means all the queries are executed using NoDB. The second bar displays the WET when only five attributes are loaded into the database. It reduced the WET by 83.88%, with just one query covered by PostgreSQL. This happens because the main primary key column objID is loaded in the database, which helps in reducing the time required to join records. The raw files are only accessed to get the data to be pro-

jected. The 22 and 39 attributes are loaded into the database based on the storage budget, which reduced the overall WET by 86.57% and 91.61% as compared to zero loaded attribute workload. The last bar shows the WET time when all attributes are loaded into the database. The 54 loaded attributes reduced the WET by 95.37% compared to zero loaded attributes. For the 54 attribute partition, the DLT and QET time gets reduced by 87.06% and 60.66% compared to the original database loaded in PostgreSQL. The created partitions can be cached into the main memory to reduce QET further.

9.7 RAW-HF: Maximizing Utilization of Existing Resources

This section discusses Phase-IV results obtained using CPU and RAM resource maximization techniques to maximize utilization of existing resources. This phase also performed experiments that combined multiple resources maximization techniques to improve WET for SDSS and LOD datasets.

9.7.1 MUAR for SDSS Dataset

Figure 9.21 compares CPU, RAM, and CPU + RAM maximization techniques to the total time required to execute 12 SDSS queries. The experiments conducted to generate results for this section used only PostgreSQL. NoDB could not execute queries in parallel as each new connection cached the entire raw file causing the memory to run out of space. Because NoDB uses more than 10GB RAM just to query 1M records in each thread. The machine had only 16GB RAM, which caused errors with just 2 parallel query threads. Additionally, NoDB uses maximum available main memory with default settings, rendering RAM maximization experiments useless.

For PostgreSQL, RAM maximization is achieved by executing queries on pre-cached data. It can be observed that RAM maximization reduced QET time by 33%, while the combination of CPU and RAM maximization techniques achieved 77.1% re-

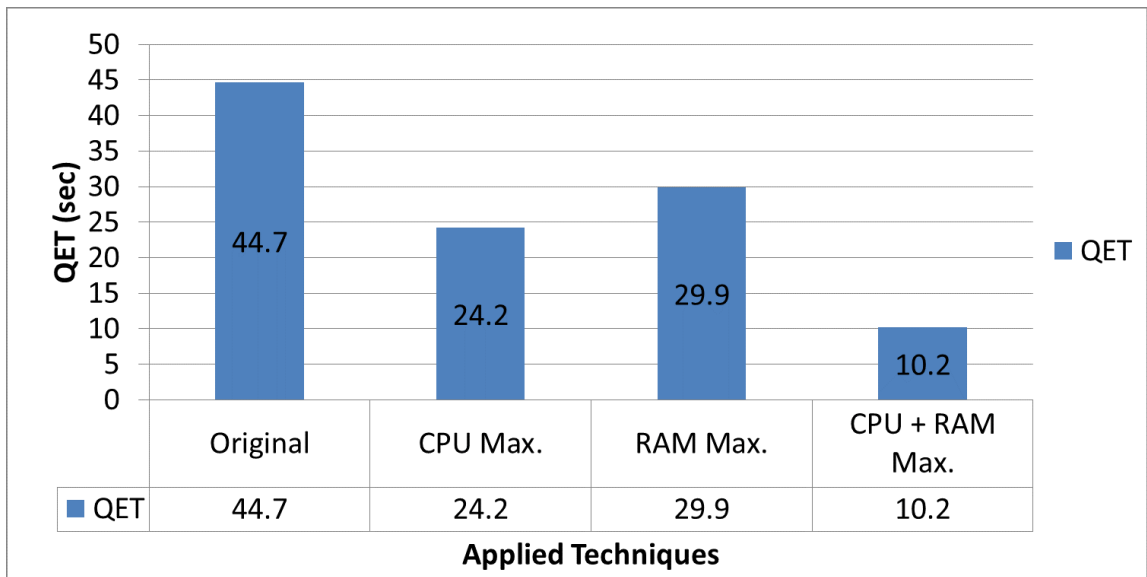


Figure 9.21: RAW-HF: Maximizing Utilization of Existing Resources

duction in total query execution time. PostgreSQL provides multiple RAM tuning settings. One such configuration is work memory. The default value of *work_mem* is set to 4MB. The RAM maximization experiments with 500MB *work_mem* setting have been performed using SDSS. Although, it did not show any improvement in WET for SDSS. The LOD workload queries could achieve reductions of 51% and 84.8% in WET using static memory allocation and dynamic memory allocation using MUAR due to in memory storage of large hash join tables. Therefore, more extensive experiments have been performed with the LOD dataset to analyze the benefits of applying MUAR technique. Section 9.7.2 presents the analysis of the LOD dataset results. DLT time is not improved using CPU resource maximization techniques. COPY method is already optimized to load data using only one data loading thread because sequential disk access provides the best IO speeds for magnetic disk storage devices [56]. Therefore, the combination of Phase-III techniques QCA and WSAC with Phase-IV resource maximization techniques is experimented with to find the maximum reduction in WET.

9.7.2 MUAR for LOD Dataset

This section discusses the static single and multiple resource maximization experiments performed using the LOD dataset to find the best combination of tech-

niques for systems having limited resources. An analysis summary of all results has been discussed to compare the impact of static, existing state-of-art, and the proposed dynamic resource allocation technique MUAR on DLT and QET. The experiments performed in this section are compared with PostgreSQL DBMS using default resource utilization settings. The time taken to execute 16 workload queries with default settings is considered the original WET. It is represented as PgSQL-default in results.

9.7.2.1 Single Resource Maximization without MUAR

The static single resource maximization techniques applied to PostgreSQL are represented only by the resource name they try to maximize. For example, PostgreSQL with CPU maximization technique (multi-threading) is represented using CPU only in result graphs. Similarly, multiple resource maximization techniques have been presented with a plus sign.

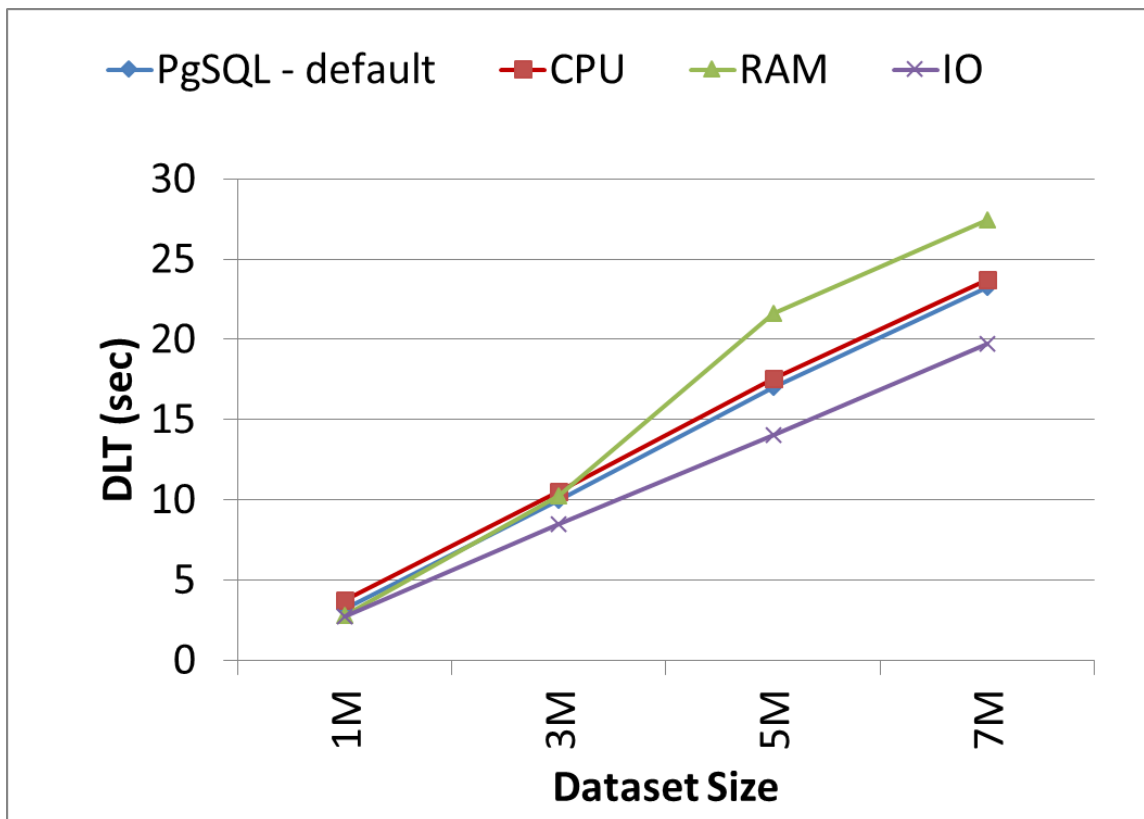


Figure 9.22: Single Resource Maximization: DLT

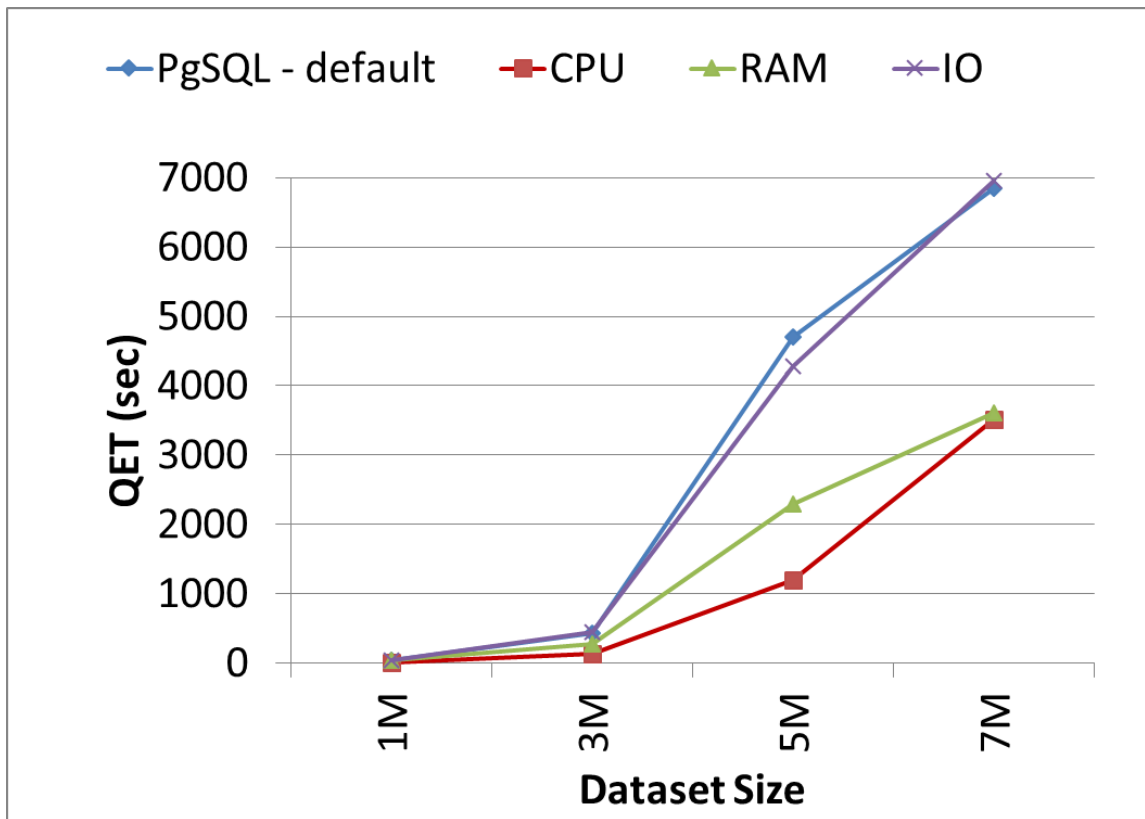


Figure 9.23: Single Resource Maximization: QET

Figures 9.22 and 9.23 present the impact of CPU, RAM, and IO resource maximization techniques on DLT & QET. It can be observed that DLT is reduced by 15-17.5% after changing DBMS storage location from HDD to RAM using RAMFS. In comparison, RAM and CPU maximization techniques do not improve DLT. At the same time, QET improves by 48.7-74.5% by executing queries in parallel utilizing maximum CPU. RAM maximization techniques with work memory set to 500MB also improved QET by 35.6-51.2% for the 3M-7M dataset sizes. However, QET improvement at 1M is less than 0.6%. RAM maximization improves the QET of queries executing using large dataset sizes because of reduced disk read-write to store intermediate join results. IO maximization technique could only improve QET by 8.9% for the 5M dataset size. However, IO maximization performed poorly at 3M & 7M, increasing QET time. In most other cases, IO maximization can not improve QET because DBMSs cache the required data to RAM, making IO maximization impractical.

9.7.2.2 Multiple Resources Maximization without MUAR

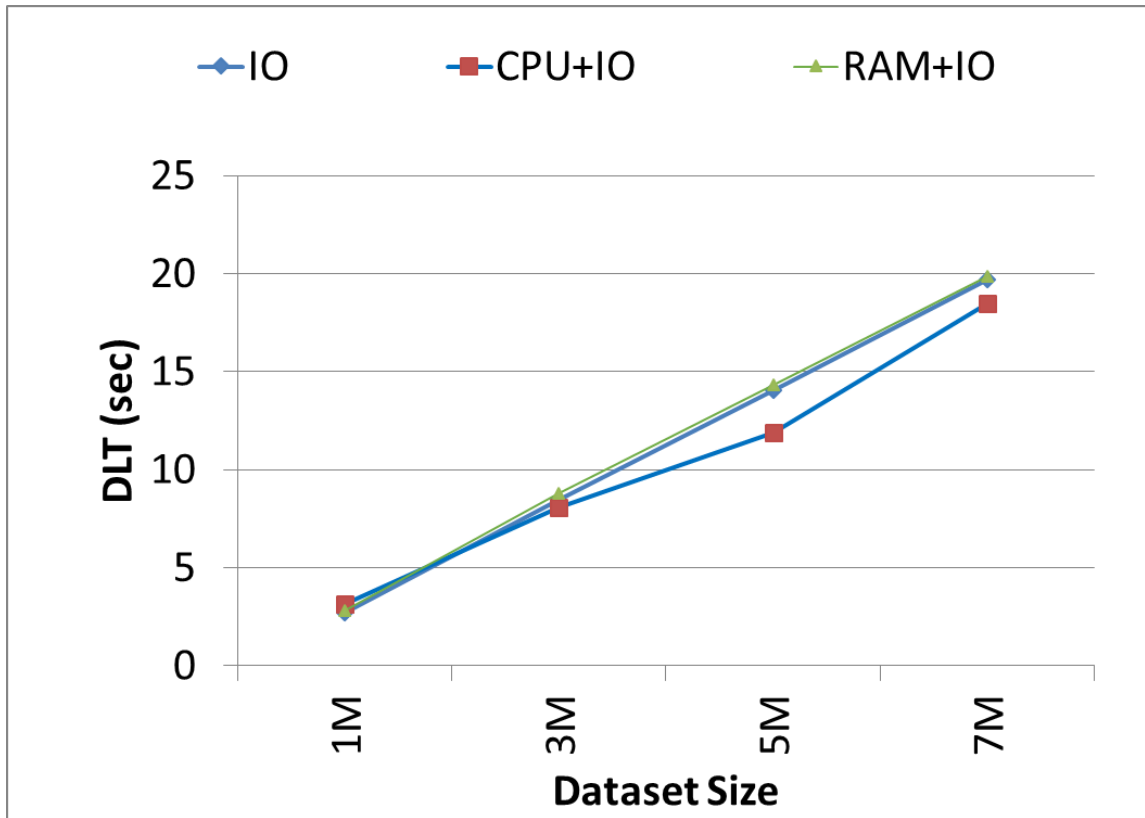


Figure 9.24: Multiple Resource Maximization: DLT

Figures 9.24 & 9.25 present the impact of multiple resource maximization techniques on DLT & QET. The experiments on DLT were limited to combining CPU and RAM resource maximization techniques with IO because individual resource maximization techniques with CPU and RAM could not improve DLT. The combination of RAM & IO resource maximization techniques required 0.8-4.8% more time to complete data loading operations than only IO maximization. At the same time, CPU+IO could improve DLT time by up to 30% compared to the original and 4.4-15.4% compared to IO maximization. CPU+IO could improve DLT because RAM storage increases IO speeds and allows faster parallel access to data compared to HDD [56].

Figure 9.25 shows that combining CPU & RAM resource maximization techniques can improve QET by 67.4-81.1% compared to the original QET utilizing maximum RAM and CPU resources. CPU+IO and CPU+RAM+IO combinations could not execute queries on dataset sizes greater than 3M because of experimen-

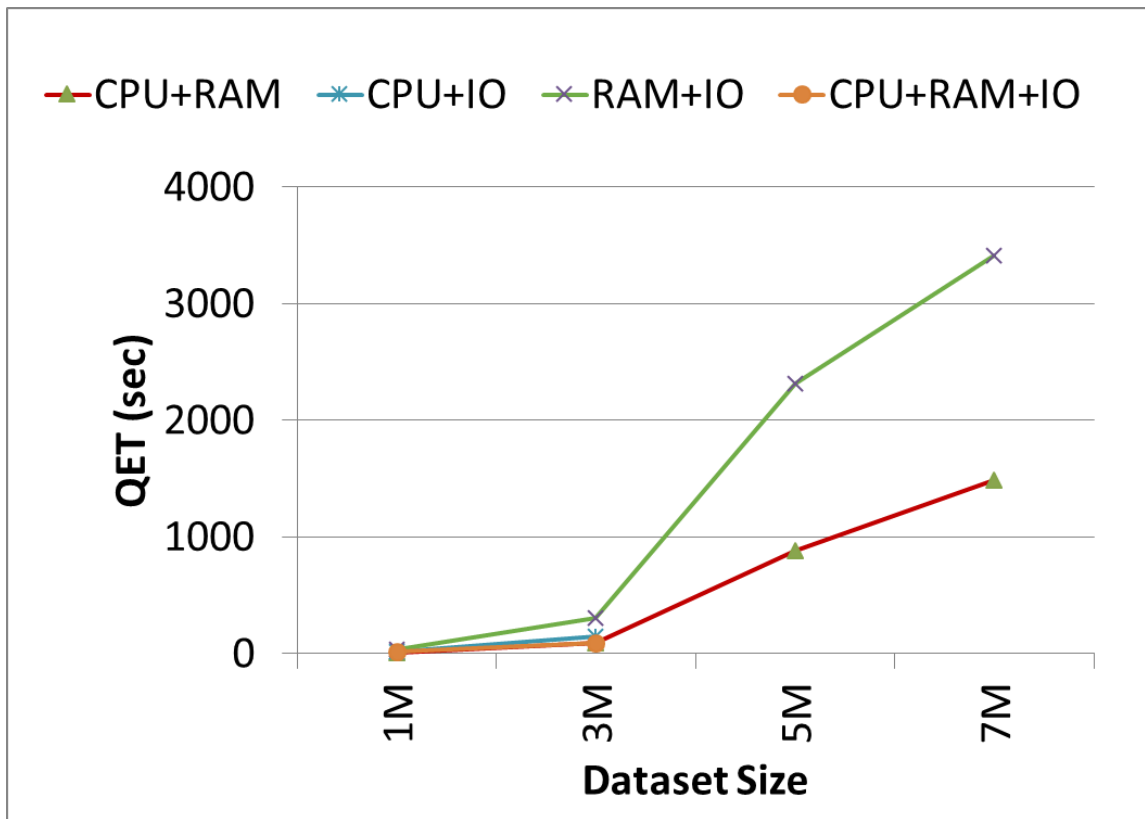


Figure 9.25: Multiple Resource Maximization: QET

tal machine hardware limitations. Therefore, CPU+IO and CPU+RAM+IO results are not plotted for comparison. However, CPU+RAM+IO results are almost equal to CPU+RAM results up to 3M because IO maximization does not help improve QET as the entire dataset is already cached in RAM, and queries have enough RAM due to the application of RAM maximization. However, for the dataset having 5M records, multiple queries being executed in parallel stored more than 10GB of intermediate join result data on RAMFS space. This caused the *OutOfMemory* errors and limited data scaling experiments.

On the other hand, CPU+RAM maximization uses a fixed amount of RAM set in the *work_mem* parameter. The additional intermediate join results are stored on disk. This allows the execution of complex multi-join queries on datasets with 5M and 7M records without causing *OutOfMemory* errors. However, more disk usage slows down the query execution. Therefore, the following section discusses MUAR, which tries to allocate maximum available RAM to queries based on real-time availability.

9.7.2.3 MUAR: WET

Multiple resource maximization technique results showed that the IO maximization technique could only improve DLT time. Datasets having more records or sizes cannot be processed efficiently when IO resource maximization is considered. Additionally, IO maximization is static and cannot be configured or changed dynamically at run time. Therefore, the proposed technique MUAR has not considered IO resource maximization and its combinations to develop a real-time dynamic algorithm. This section compares the WET and resource utilization results of the most effective combination of multiple resource maximization techniques, i.e., CPU+RAM results with MUAR.

Figure 9.26 presents that MUAR improved QET by 66.7-82.2% compared to the original by maximizing CPU and RAM resources dynamically at runtime. While QET improvement achieved is 13.8-37.7% better than the static CPU+RAM resource maximization. MUAR achieves significant reduction in QET because it allocated 2-5GB of work memory to complex queries based on join count J_C , ensuring maximum utilization of available RAM. Independent resource maximization of CPU resource is 0.05-4.3% slower than static CPU resource maximization techniques because of the wait imposed by the algorithm to avoid over-allocation.

9.7.2.4 MUAR: Resource Utilization

This section discusses the impact of resource maximization techniques on actual resource utilization. The CPU, RAM, and IO wait results are presented in percentages to identify the overall utilization of resources out of 100. At the same time, disk IO utilization is shown in MB to present the amount of data written by queries on disk due to limited work memory. Figure 9.27 presents the average resource utilization observed during the original workload execution (when no maximization technique is applied) with the best combination of multiple resource maximization techniques without MUAR and with MUAR. It can be seen that the Original CPU and RAM resource utilization is below 36%. Basic CPU & RAM maximization techniques without MUAR and with MUAR reached peak CPU utilization of 100%. However, average CPU utilization stayed around 63-

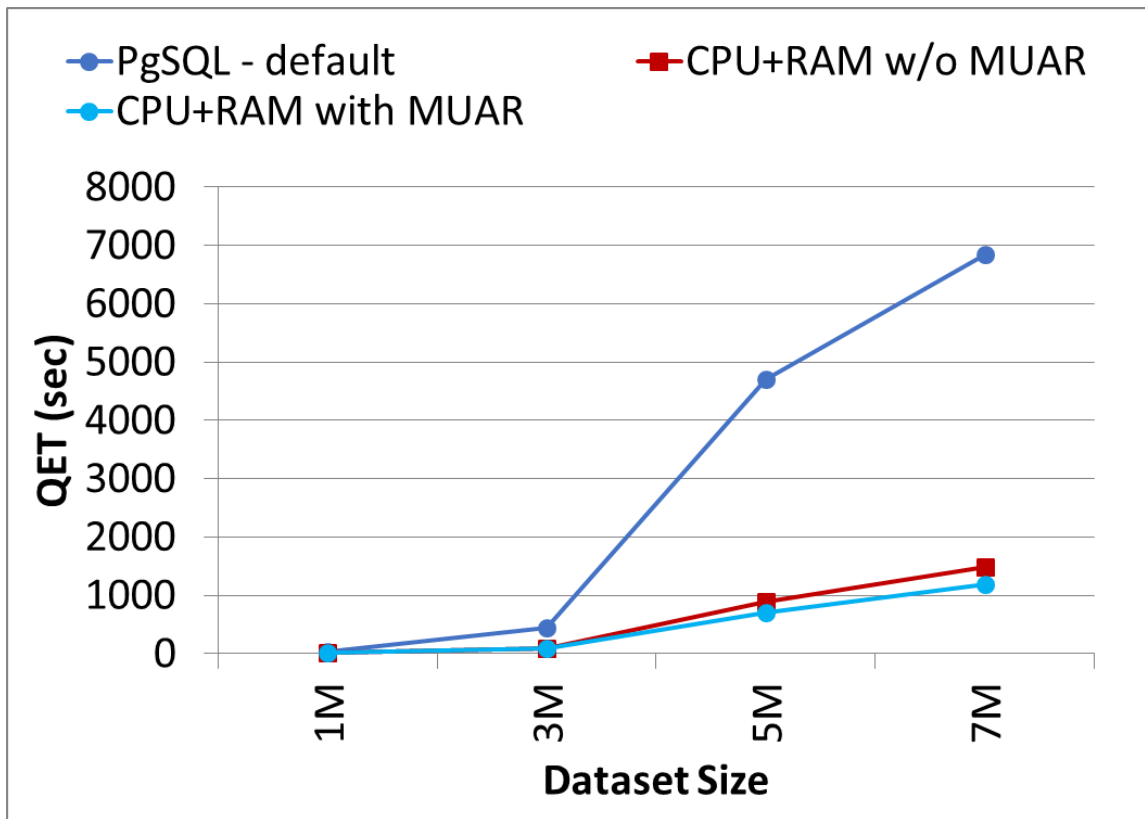


Figure 9.26: MUAR: QET

67% because of the smaller workload the last query ran alone, utilizing only 25% CPU.

The average RAM utilization stays below 45% for default and static main memory allocation. Static RAM resource maximization increased RAM utilization by 6.2% compared to the original. However, MUAR increased RAM resource utilization by 18.5% due to 2-5GB of work memory allocated to complex queries in real-time. *IO_Wait* shows the percentage of CPU time wasted in waiting for IO resources. MUAR allocates more RAM space and waits for resources to be available before starting any task, reducing the overall IO wait to 3%. Static CPU & RAM maximization techniques without MUAR had 4 times more IO wait on average than dynamic resource allocation using MUAR.

Fig. 10. MUAR: Total IO

Figure 9.28 shows the total data written to disk by 16 workload queries for default, static CPU+RAM, and MUAR resource allocation techniques. It can be seen that with a default memory allocation of 4MB, the total data written to disk is

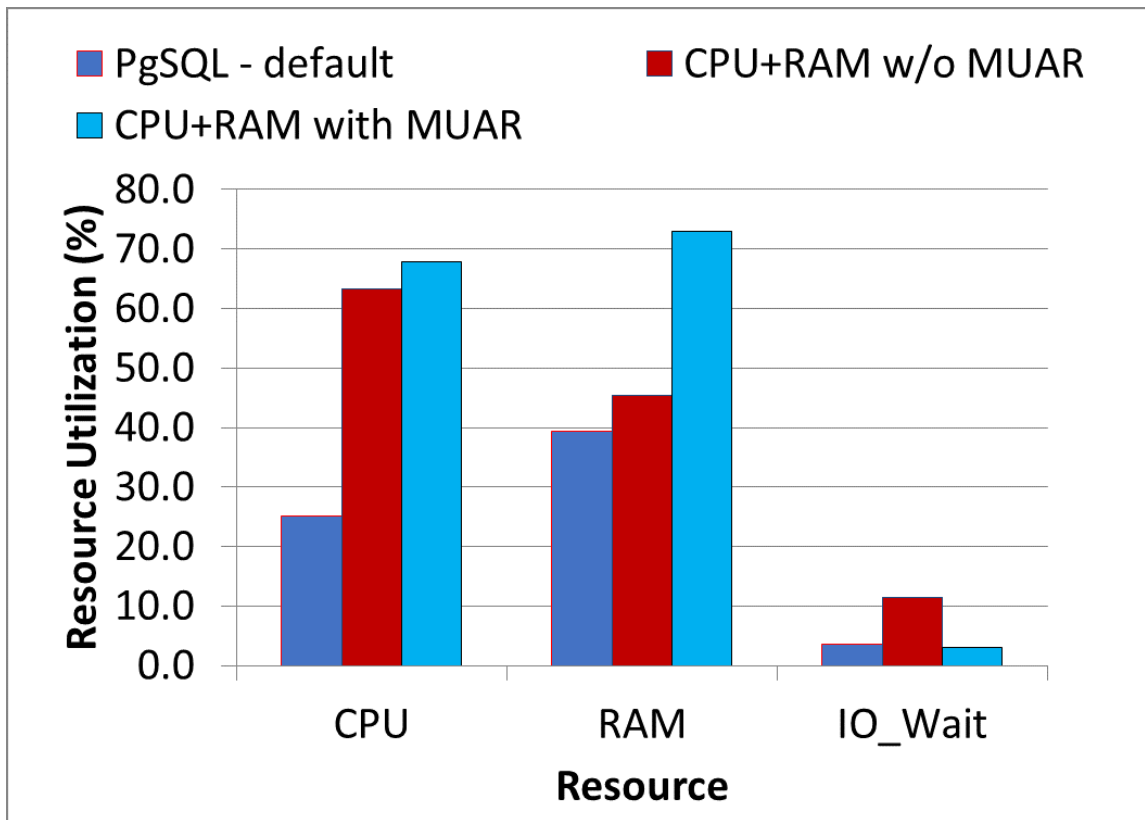


Figure 9.27: MUAR: Average Resource Utilization

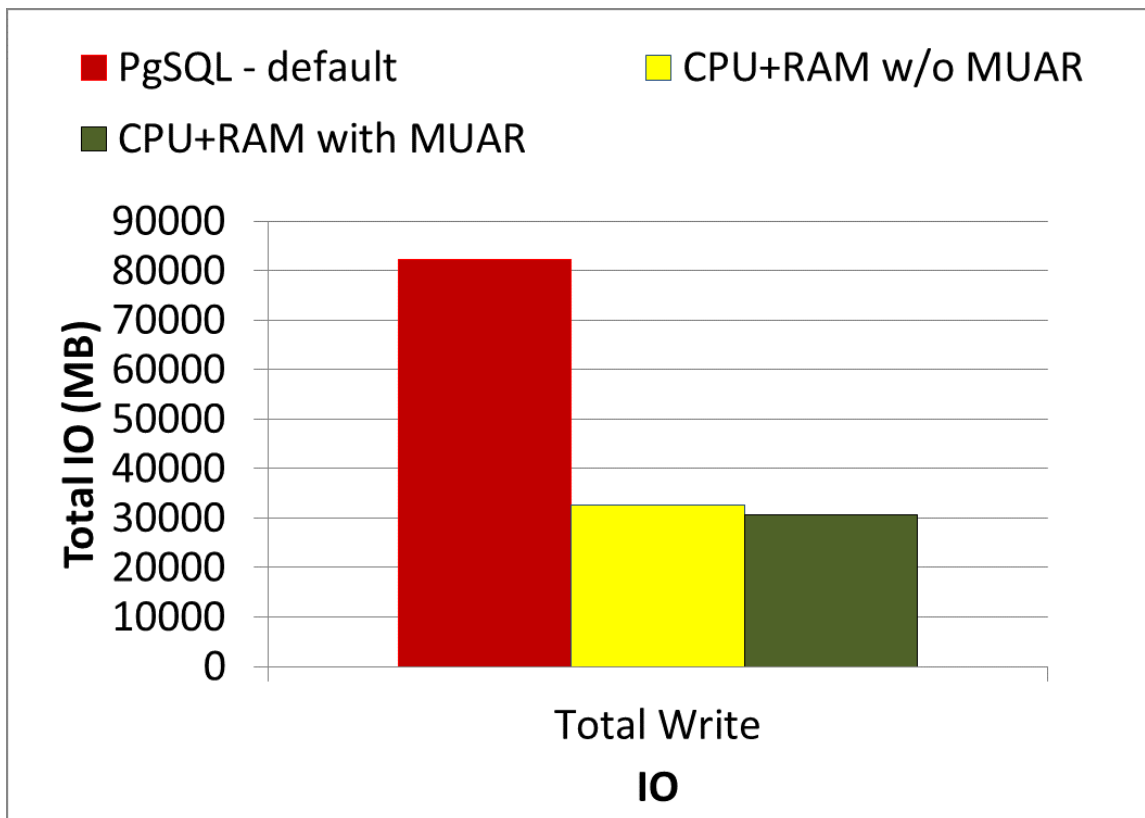


Figure 9.28: MUAR: Disk Writes

82GB. The smaller work memory cannot store large intermediate hash join tables or sort records. Therefore, a large amount of data is written to disk, increasing the QET time of complex queries. In comparison, static memory allocation of 500MB reduced disk writes by 60%, because most temporary sorting data or hash join tables are written to RAM. The high reduction in disk writes. While MUAR was able to reduce disk writes by 62.5% compared to the default work memory configuration. This helps MUAR achieve lower IO waits and 85% faster workload execution.

9.7.2.5 Comparison of MUAR with State-of-the-art techniques

This section compares the proposed MUAR technique with the existing static and dynamic state-of-the-art techniques, which try to increase the utilization of existing system resources or allocate resources efficiently to improve WET. Table 9.1 compares MUAR performance with static techniques, while Figure 9.29 & Table 9.2 presents the comparison of MUAR with state-of-the-art dynamic or ML techniques based on parameters like QET, real-time resource utilization monitoring, whether the technique divides single query tasks for parallel processing, uses lightweight algorithms, and its ability to manage ad-hoc queries.

Table 9.1: WET: Comparison with static resource allocation techniques

Technique	Maximized Utilization of Resources	DLT(%)	QET(%)	WET(%)
Single Resource Maximization w/o MUAR	CPU	-02.94	74.54	74.26
	RAM	-26.92	51.17	50.89
	IO	17.47	08.95	08.98
Multiple Resource Maximization w/o MUAR	CPU+RAM	-02.94	81.15	80.85
	CPU+IO	30.19	NA	NA
	RAM+IO	15.73	50.79	50.66
	CPU+RAM+IO	30.19	NA	NA
MUAR	CPU+RAM	00.00	85.12	84.81

Figure 9.29 compares 1st and 2nd run QET of Q14 achieved by MUAR with PostgreSQL configured to allocate default resources, Elastic[106], PCC[103], and Best (AutoToken[109]). We had performed experiments with multiple complex queries like Q10 & Q14, which wrote 8-10GB of intermediate join results to disk. For

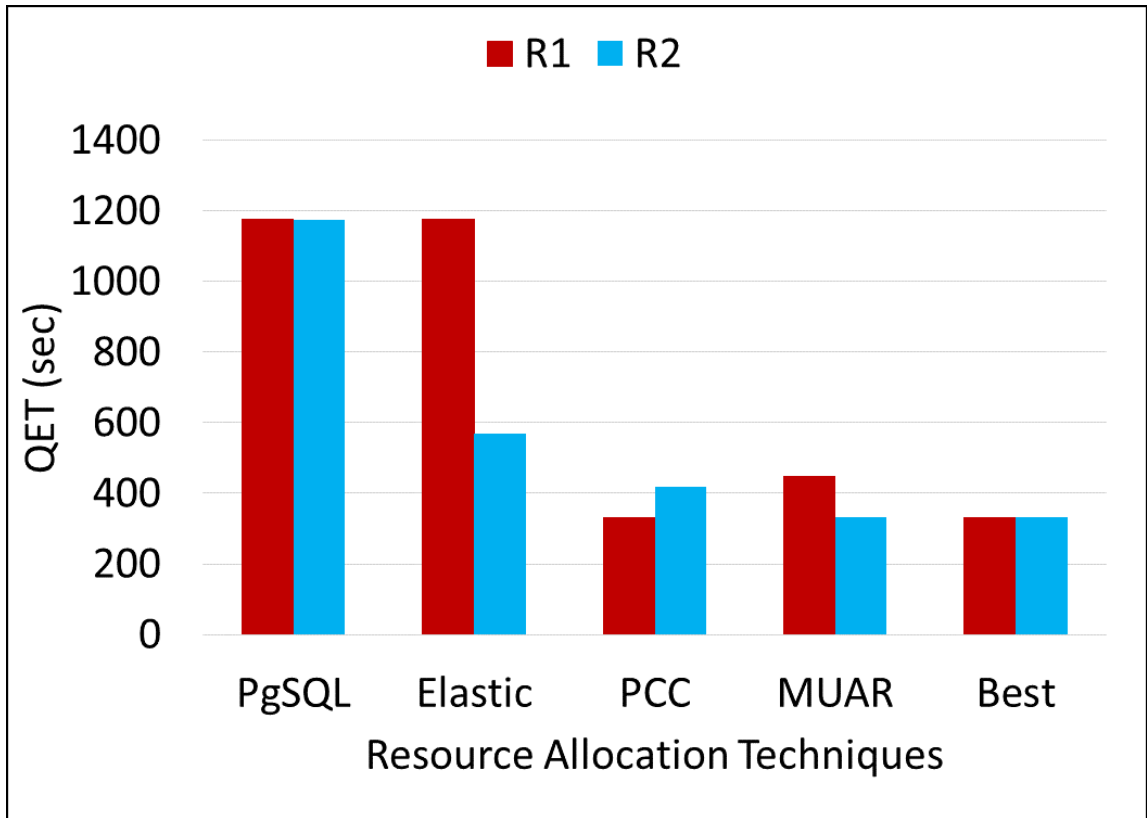


Figure 9.29: MUAR: Comparison with state of the art

Table 9.2: Comparison with State-of-the-art Resource Allocation Techniques

#	Tools/ Techniques	Resources Maximized			R2 QET (%) w.r.t. PgSQL	Real- time RM	Divide Query Tasks	Light- weight	Ad-hoc Query Supp.
		CPU	RAM	IO					
1	Elastic [106]	Yes	Yes (5.6%)	No	51.7%	No	No	Yes	No
2	PCC [103]	Yes	Yes (21.2%)	No	64.4%	No	No	No	No
3	MUAR	Yes	Yes (66.3%)	No	71.8%	Yes	No	Yes	Yes
4	AutoToken (Best)[109]	Yes	Yes (62.8%)	No	71.8%	No	No	No	Yes

the first run, PgSQL & Elastic allocates default resources, i.e., 4MB *work_mem*. While MUAR allocates 1.8GB of work memory by analyzing query complexity and available RAM to improve 1st query run performance by 62%. At the same time, PCC and AutoToken may allocate 8GB-10GB of RAM resources to achieve best performance during 1st query run as they suggest resources are over-allocated in the serverless cloud. During 2nd run, Elastic resource allocation QET results are achieved by allocating only 500MB of work memory enough to reduce OLAP (complex query) QET by 50%. PCC and AutoToken use past data to train ML models with multiple features to allocate optimal (3.4GB) or maximum resources (10GB) during the 2nd run with 12-20% estimation error. During 2nd query run executed on the same 7M records dataset, MUAR allocated 10.6GB of work memory by adding previous work memory of 1.8GB & 8.8GB of disk writes recorded during 1st run. MUAR uses a simple linear equation that considers fewer parameters like join count, dataset size, and past disk writes to achieve the best performance with 15-20% estimation error. This makes MUAR lightweight and faster compared to ML techniques. The main memory utilization of MUAR & AutoToken is 3x to 20x higher than PCC and Elastic. In summary, MUAR is capable of finding best resource allocation value for work memory parameter with single query run data as PCC[103].

9.7.3 MUAR for Different Datasets

Database management softwares support runtime tuning of resource utilization with the help of configuration parameters like *work_mem*. Work memory parameter helps in allocating query specific RAM resources for faster execution of a query. However, the effects of changing any parameter value on WET depend on several parameters, like dataset size, query complexity, or number of attributes accessed. This section analyzes the experimental results after applying MUAR on different datasets like SDSS and LOD.

Figure 9.30 shows the impact of RAM and CPU maximization techniques used by MUAR on LOD and SDSS datasets. It can be seen that individual and combination of resource maximization techniques used by MUAR are more effective on

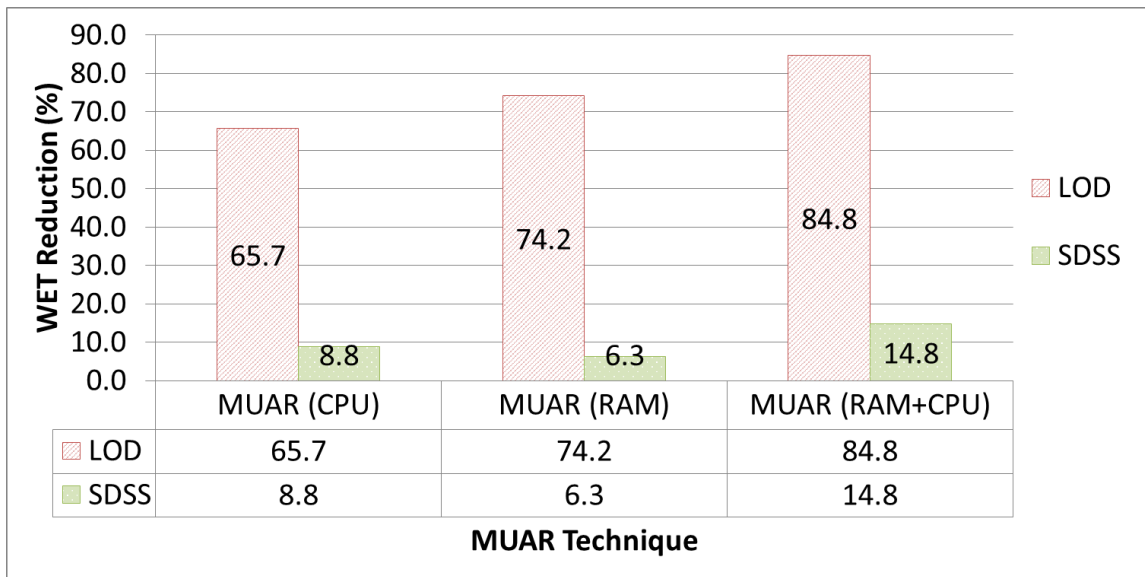


Figure 9.30: Impact of MUAR on WET for LOD & SDSS datasets

the LOD dataset. It is visible that the WET reduced for LOD dataset is 5.7 times more compared to the SDSS dataset. The large difference in WET improvement is due to the different characteristics of both datasets. The PhotoPrimary table of SDSS contains 509 attributes, while the LOD dataset is a narrow table dataset with only three attributes. For SDSS, loading all 509 attributes in the dataset required 188.63sec while QET time is only 44.7sec. For SDSS, 80.8% of WET is spent loading data into DBMS due to the 4.7GB size per 1M records. While for LOD dataset only spent 0.4% of the WET time loading data. MUAR is not using the IO maximization technique. Therefore, DLT time cannot be improved using CPU & RAM maximization techniques. This means the effect of CPU and RAM maximization techniques on WET depends on QET time reduction only. For the LOD dataset, 99.6% of the time is spent on query execution. In comparison, SDSS QET time is less than 20% of WET. Therefore, executing queries in parallel helps reduce overall WET by only 8.8% for SDSS. On the other hand, 99.6% of the workload can be executed in parallel for the LOD dataset. Therefore, executing queries in parallel achieved 65.7% reduction in WET for the LOD dataset workload.

The RAM maximization technique used by MUAR allocates more work memory to complex queries to reduce QET. The allocation of more work memory helps complex multi-join queries execute faster as disk access reduces significantly. For

the LOD dataset, 56% of the query workload had five join queries, while 87% had two or more joins. On the other hand, query workload of the SDSS dataset had less than 1% of queries with two joins. Therefore, MUAR RAM maximization techniques also achieved better results in reducing QET for the LOD dataset than SDSS. It can be seen in Figure 9.30 that MUAR RAM maximization achieved 11 times more reduction in WET than the LOD dataset. For SDSS, allocating more RAM did not help reduce QET due to a simpler query workload with fewer joins, as disk writes were already less or non-existing. However, caching the entire dataset into main memory helped reduce overall WET by 6.3% for the SDSS workload. Therefore, MUAR(CPU+RAM) is more effective for datasets having complex query workloads like LOD.

9.8 RAW-HF: Optimizing Required Resources & Maximizing Utilization of Existing Resources

This section presents the results of RAW-HF after combining all techniques proposed in Phase-III & IV. The results also compare RAW-HF performance with state-of-the-art tools and techniques based on the WET and resource utilization parameters.

9.8.1 RAW-HF: Workload Execution Time

Figure 9.31 shows the comparison of all the techniques used during thesis work. First column shows the raw data query processing time using NoDB, which has zero data loading time. The 2nd column shows WET time required by traditional DBMS PostgreSQL. The 3rd and 4th columns show the WET results for individual optimization and maximization phases of RAW-HF. The combination of Phase-III technique WSAC and Phase-IV resource maximization techniques shows that entire workload can be executed within 30.6 sec using workload aware partitioning techniques like Partial Loading [125] compared to 613.6 sec by NoDB [33]. At the same time, RAW-HF takes only 22.6 sec to complete the execution of given work-

load tasks. It can be observed that the combination of all techniques achieved total reduction of 96.32% using only RAW-HF compared to NoDB while 26.14% compared to the workload aware partial loading technique. RAW-HF benefits from low DLT time achieved by only loading attributes used by complex queries. Additionally, simple queries complete execution in parallel to data loading tasks utilizing available resources efficiently.

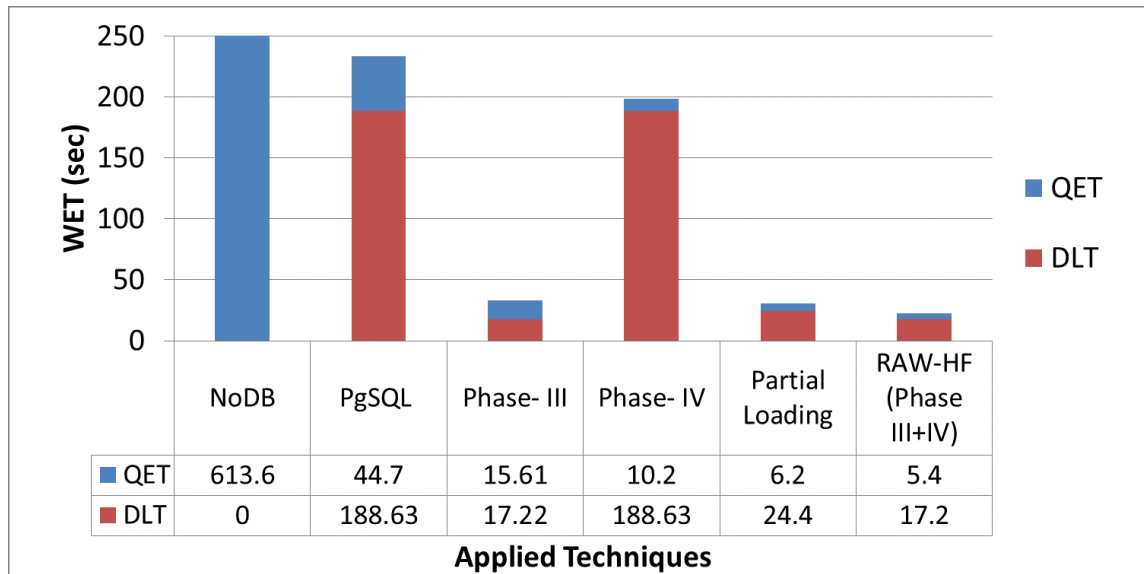


Figure 9.31: RAW-HF: WET Comparison

9.8.2 RAW-HF: Resource Utilization

Figure 9.32 shows the comparison of resources utilized by Phase-III & IV of RAW-HF with NoDB [33], PgSQL DBMS [22], and Partial loading technique [125]. Phase-I proposed a raw data query processing framework to process raw data in its place without loading. The NoDB has a high QET, which utilizes CPU for a longer time. RAM utilization is more than double the size of actual raw data. Here, the 1M records dataset used in experiment utilized 4.7GB of space on IO device. PgSQL is the better choice for data processing as it reduced the CPU, RAM, and IO utilization by 72.5%, 74.8%, and 43.5%, respectively.

Phase-IV results showed that CPU utilization time is reduced by only 6.34% because most of the time is spent in data loading process compared to PgSQL. Phase-IV can only utilize other CPU cores to execute read queries in parallel. Fig-

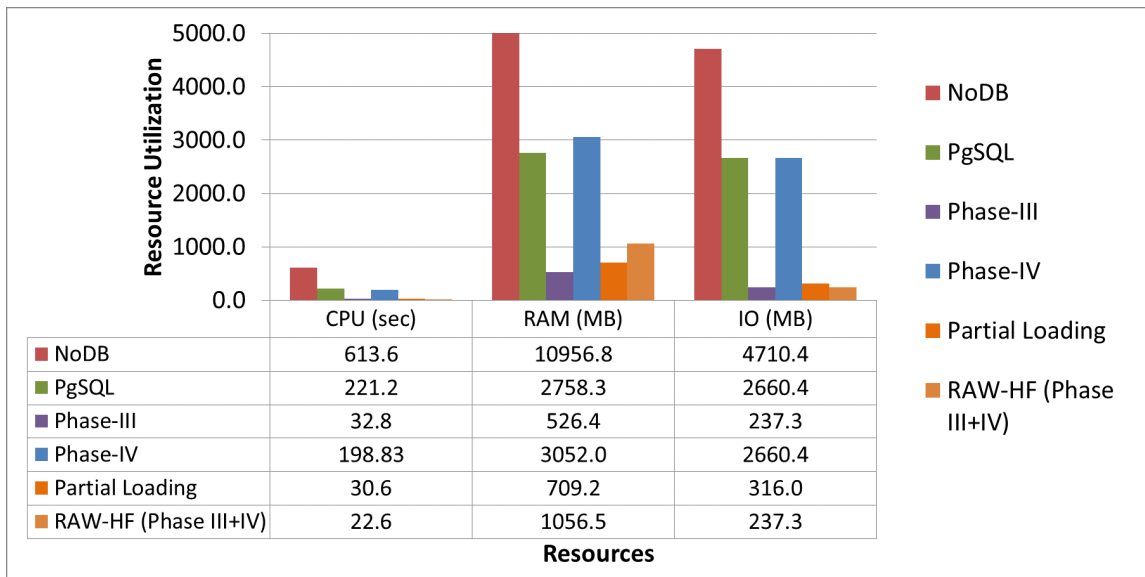


Figure 9.32: RAW-HF: Resource Utilization

Figure 9.32 confirms that CPU utilization is reduced by 77% only during query processing tasks due to parallel processing. Similarly, IO utilization stays same as PostgreSQL in Phase-IV as the experiments used original 1M SDSS dataset having 509 attributes. The QCA with WSAC technique in Phase-III reduced the required DB partition size(IO) by 91.08% reducing WET, CPU, and RAM utilization by 85.9%, 85.1%, and 80.9%.

The Partial Loading technique [125] loaded only 10.6% of original data into DBMS, which reduced the WET time by 88.1% compared to NoDB. It also reduced the CPU and RAM utilization by 81.3% and 86.4%. The RAW-HF experiments combined Phase-III and Phase-IV techniques, which showed a 32.8% increase in RAM utilization compared to the Partial Loading technique due to parallel processing of queries. However, RAW-HF improved DLT, QET, WET, CPU, and DB Size(IO) requirements by 29.5%, 12.9%, 26.14%, 26.14%, 24.92% compared to Partial Loading technique [125] executing all read queries in parallel after data loading is complete. The maximum CPU utilization reached 94% for RAW-HF while executing the given query workload. However, due to the 12 query workload, the average CPU utilization stayed at 31% as most of the CPU time goes into data loading operations, which utilized a single CPU core. A more detailed comparison of RAW-HF with state-of-the-art techniques and tools is presented in Section

9.11.

9.8.3 RAW-HF: Processing Capacity Estimation

This section estimates the data processing capacity of a given machine for cloud or other distributed setups by analyzing the resource utilization of workload task processes. The RAM resource utilization recorded by the proposed framework for different non-partitioned and partitioned dataset cases has been considered for estimation. The resource monitoring and data scaling results have shown that both PostgreSQL and PostgresRAW tools require different resources for different amounts of time. The data processing capacity estimation is a complex problem and greatly depends on the application requirements. A given machine can process terabytes of data if given enough CPU time. However, faster response time requirements may need results within a suitable time limit, preferably within a few seconds for real-time systems. The QET or CPU utilization time reduces significantly if the processed data is already cached in RAM for both tools. Therefore, the best way to estimate the capacity of a machine would be to determine the amount of data that can be cached in the main memory. The machine can process datasets larger than the available main memory (RAM), but it would have to access the disk or swap the data, increasing query execution time.

Query complexity is another parameter that needs to be considered to estimate the data processing capacity of a machine. More complex queries with multiple joins would require more memory for data processing and caching intermediate join results [100]. Processing the RDF dataset stored in triple format requires a large amount of main memory due to multiple self-join requirements [96]. The SDSS workload had only one query with two joins. Therefore, the RAM resource requirements of SDSS queries do not change extensively with dataset size.

The estimation of original and partitioned SDSS dataset sizes that a machine having 16GB of RAM can handle has been shown in Figure 9.33. The PgSQL and PgRAW data scaling experiments have shown that the main memory capacity of the used machine is 70%. OS and other applications use the remaining 30% RAM. Now, the estimation is calculated based on the size of the required database par-

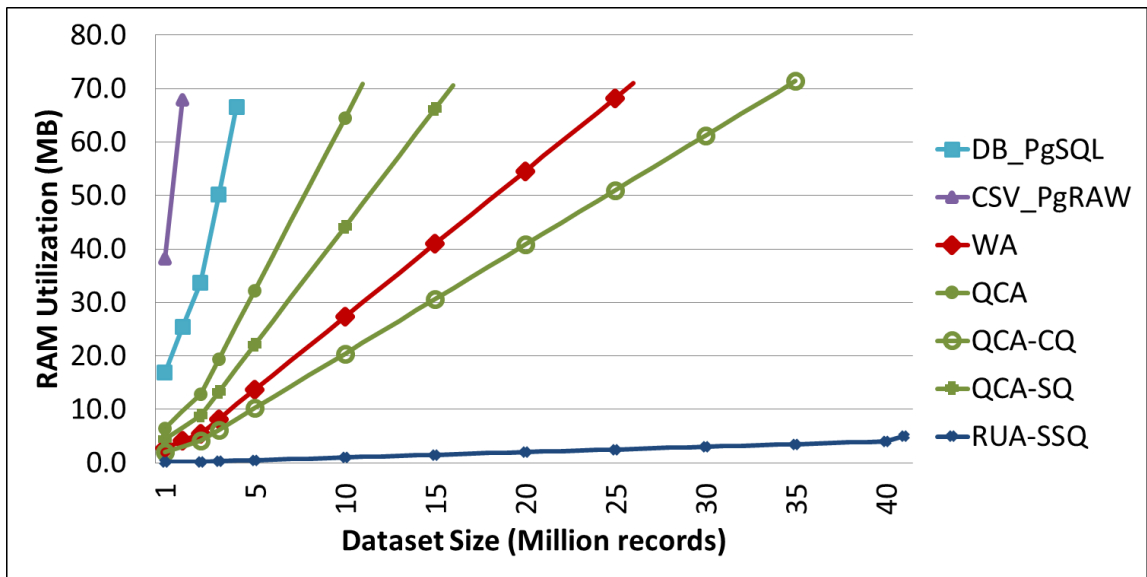


Figure 9.33: Raw Data Processing Capacity Estimation

tion and the most complex query's RAM requirement. Figure 9.33 compares PostgreSQL, and PgRAW using the entire dataset with workload-aware (WA) partitioning techniques like Vertical Partitioning[125], WSAC, QCA & RUA accessing partitions required by workload queries. The database size in PostgreSQL is 40-50% smaller than the raw format size on disk. The compressed size significantly increases the size of the raw dataset that can be cached in the main memory, increasing machine's capacity to process larger datasets efficiently. SDSS has most attributes in numeric format, which reduces the loaded dataset size. However, database size may not differ significantly for other datasets having more string attributes.

The PgRAW and QCA-SQ techniques cache and index the raw partitions in the main memory, utilizing 1.6x to 1.8x more RAM than their raw format size. The estimation plotted in Figure 9.33 is based on a simple linear equation derived from the original dataset's resource requirements monitored for PostgreSQL and PgRAW tools in Section 9.3.3 and Figure 9.32. The calculations show that the given machine can process around 26M records (122GB) datasets using WA techniques, while 11M records (51.7GB) dataset can be processed using the QCA technique. QCA-CQ and CQA-SQ partitions can be used to distribute the complex query and simple query workload on multiple nodes. Here, the QCA-SQ node does

not require loading any data because all queries are executed using PgRAW, and the required SQ partitions are cached and indexed in the main memory to reduce reparsing. The partitions created using the RUA technique may allow execution of simple sampling type queries (SSQ) on 700M or larger datasets efficiently using minimal resources because sampling queries running on PgRAW does not process the entire dataset.

Partitioning techniques presented in Figure 9.33 partition the original dataset, and cache workload-specific partitions required by queries in the main memory. This increases the size of the original dataset that can be processed. In reality, the capacity of a machine stays the same, and the cached partition size also stays 70%(11.2 GB) of main memory. The estimation is based on partitions required by queries to provide exact results as if the queries have been executed using the original dataset. The complex query CQ partition of QCA independently might be able to process 35M records (164GB) dataset because the CQ partition size is 25.7% smaller than WA technique partitions.

9.9 RAW-HF for Different Datasets

This section discusses the impact of RAW-HF on WET for different types of datasets like LOD & SDSS. Table 9.3 compares LOD and SDSS datasets based on the Fraction of Attributes Accessed (FAA) by workload queries and the Fraction of Attributes Loaded (FAL) by RAW-HF. It can be seen that SDSS is a broad table dataset. All the SDSS workload queries access only 10.6% of attributes. On the other hand, LOD dataset is a narrow table dataset containing only three attributes. Due to fewer attributes, almost all queries use two or more attributes. The impact of broad and narrow tables and queries accessing only small part of the dataset can be seen in the ORR results for SDSS in Figure 9.34.

The ORR phase of RAW-HF uses vertical partitioning methods to reduce DLT and improve QET by accessing only required fractions by creating database and raw file partitions. As discussed in QCA results presented in Figure 9.17, RAW-HF only loads attributes required by complex queries to reduce DLT time. This helps

Table 9.3: ORR: Fraction of Attributes Accessed (FAA) and Loaded (FAL)

	Total Attributes	Accessed Attributes	FAA (%)	Loaded Attributes	FAL (%)	DLT (%)	QET (%)	WET (%)
SDSS	509	54	10.6	34	6.7	90.9	87.9	85.9
LOD	3	3	100.0	3	100.0	0	0	0

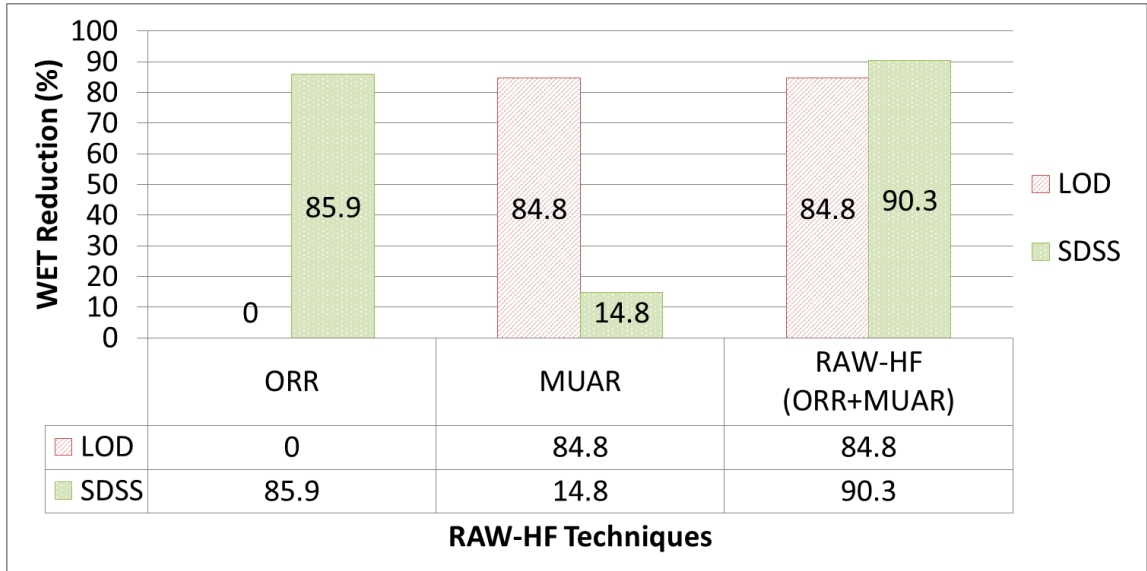


Figure 9.34: Impact of RAW-HF on WET for LOD & SDSS datasets

datasets like SDSS, which requires only a small fraction of the dataset (10.6%) to answer queries by loading only 6.7% of attributes. The remaining 93.3% of attributes are not loaded into DBMS, reducing WET by 85.9% for the SDSS dataset. On the other hand, vertical partitioning cannot help datasets like LOD that access 100% of attributes. Therefore, the WET reduction achieved by applying ORR phase is 0% for LOD. However, the MUAR phase achieves 84.8% of reduction in WET by efficiently utilizing existing CPU and RAM resources for complex queries. A detailed analysis of MAUR results has already been discussed in Section 9.7.3. The RAW-HF can be applied to different types of datasets like LOD and SDSS to achieve 84-90% reduction in WET. In summary, RAW-HF can be applied to different types of real-world datasets to achieved significant reduction in WET by combining ORR and MUER phases.

9.10 Characterization of RAW-HF

This section discusses time required to execute RAW-HF algorithms (AET) and steps taken to optimize AET. The Section 9.10.3 also discusses AET taken by RAW-HF algorithms for different datasets like SDSS and LOD.

9.10.1 AET

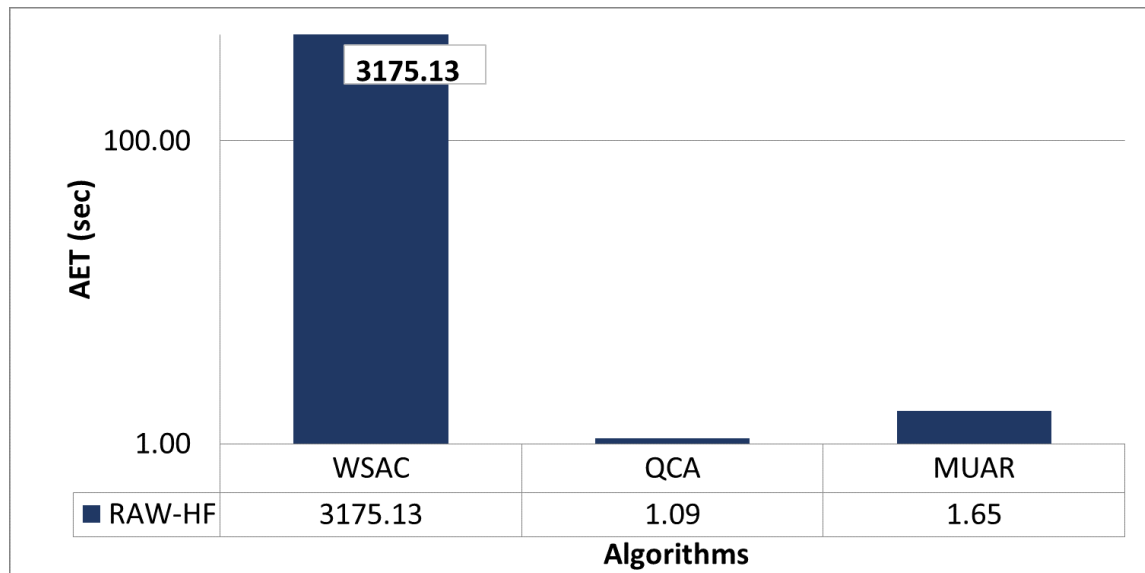


Figure 9.35: AET of RAW-HF Algorithms for SDSS

The RAW-HF uses combination of WSAC, QCA and MUAR algorithms. Figure 9.35 presents time taken by WSAC, QCA and MUAR algorithms used in ORR and MUER phase of RAW-HF. It can be seen that WSAC required more time to complete the algorithm, because it consists of cost calculation. Section 9.10.2 discusses the steps taken to optimize different sub-algorithms or functions used by WSAC. QCA uses the pre-optimized functions, therefore it takes only 1.09sec. Additionally, QCA AET does not depend on the size of dataset, as QCA does not use cost calculations functions. MUAR continuously runs in parallel throughout the workload execution on multiple threads of CPU so the finding AET of difficult. However, CPU utilized to execute all the steps of MUAR required less than 2% of CPU as resource monitoring was optimized. MUAR uses 0.13sec to calculate and

set work memory for each workload query. Therefore, MUAR AET is achieved by multiplying 0.13sec to the number of queries in workload list.

9.10.2 WSAC: Algorithm Optimization

For any algorithm to work efficiently in real-time, the time required by the algorithm to execute the functions needs to be as low as possible. However, cost function in WSAC depends on the actual dataset size and workload list. The larger these files, the more time the function takes to process them. This makes the WSAC dependent on the number of records n in the dataset. These optimization experiments have been performed only for WSAC algorithms to eliminate the dependency on dataset size. Different input file sizes have been considered to reduce the algorithm execution time (AET) of WSAC.

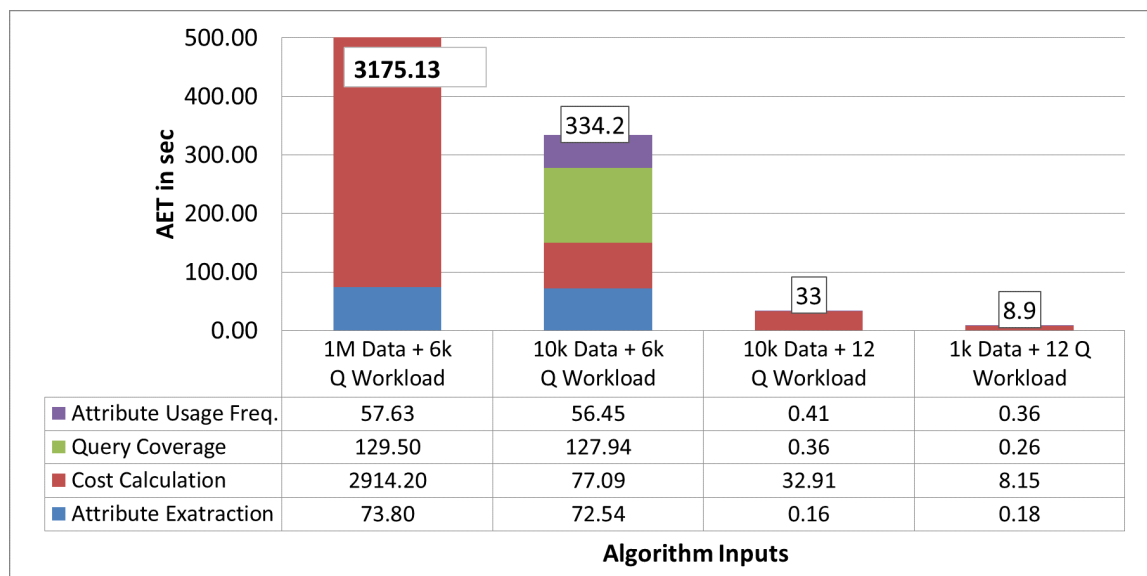


Figure 9.36: WSAC: Algorithm Optimization

Figure 9.36 shows the AET for different inputs. It can be observed that when the algorithm is provided with the entire 1M records dataset for cost-calculation, the time required by cost-function is more than 48mins. Therefore, we tried to reduce the cost calculation time by providing a smaller 10k sample of the dataset, which reduced the AET by 89.48%. The 3rd bar shows the time of AET when only the most frequent query set of 12 queries was provided as workload. The final data point showed when the sample dataset size was only 1000, which reduced

AET to an acceptable time of 8.9sec. However, the storage size received from the smaller set was slightly larger than the original dataset sample. The direct multiplication by 1000 to project the original column size led to a reduction in the attributes covered by the algorithm. We can optimize the proposed technique as per the accuracy requirements of the application.

9.10.3 RAW-HF AET for Different Datasets

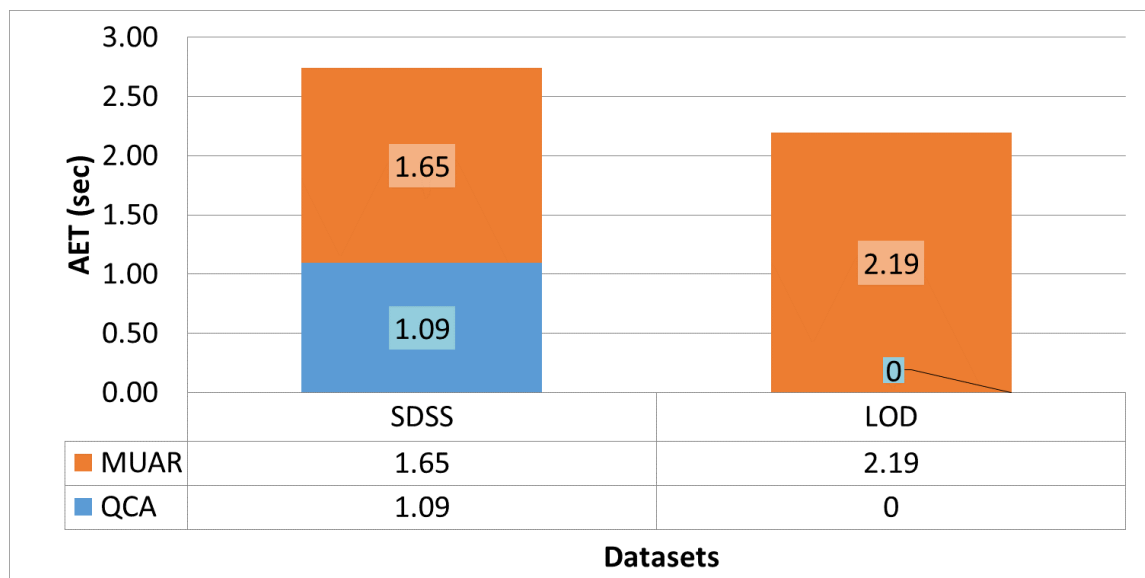


Figure 9.37: RAW-HF AET for SDSS and LOD

RAW-HF uses best combination out of WSAC, QCA and MUAR to optimize resource utilization. As discussed in Section 9.10.2, WSAC still required 8.15sec in calculating cost of attributes used by workload queries. RAW-HF uses combination of QCA and MUAR technique to keep it lightweight. Figure 9.37 displays AET of RAW-HF for SDSS and LOD dataset. It can be seen that ORR (QCA) phase is not applied to LOD dataset, therefore AET of QCA is 0sec. On the other hand, MUAR AET achieved by multiplying 0.1sec AET of MUAR with number of queries in workload list produced 1.65sec for SDSS and 2.19sec for LOD dataset.

Figure 9.38 displays the AET of RAW-HF when dataset size is increased from 1M to 7M for SDSS and LOD datasets. This figure shows that AET of RAW-HF does not depend on the dataset size, because QCA does not use any cost calculation functions. While MUAR only analyzes the query statement to allocate

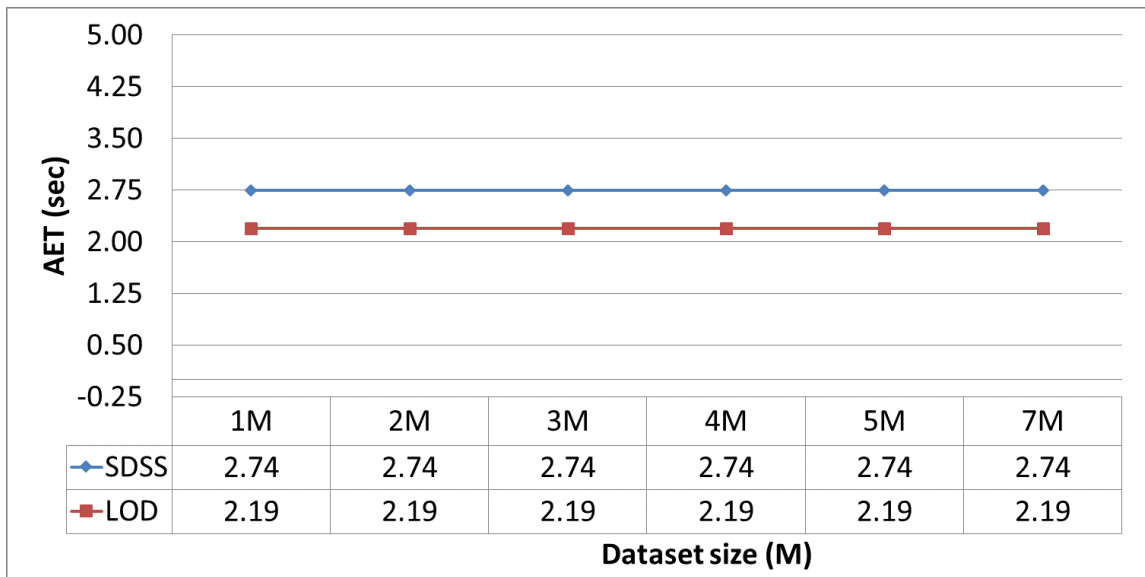


Figure 9.38: Scaled Data: AET for SDSS and LOD

resources in real-time. The complexity of QCA and MUAR has already been discussed in Sections 6.2.1.2 and 7.2.4. It is $O(j^*(w_l))$ for QCA and $O(x^*(w_l))$ for MUAR. Therefore, AET of both depends on the number of unique queries in the workload.

9.11 Comparison with State-of-the-art

This section presents a comparison of the RAW-HF technique with other state-of-the-art techniques. The RAW-HF techniques compared with NoDB [33], Slalom [94], DBMS [22], Partial Loading [125], PDC [117], and PCC [103]. NoDB is an open source in-situ processing engine with main memory caching and indexing features [33]. Although Slalom is an improvement over NoDB, it is not available as an open source tool [94]. PostgreSQL (PgSQL) is a widely used open source DBMS [22]. The Partial Loading technique proposes distributing dataset partitions among DBMS, and raw format considering storage resource limitations [125]. PDC proposes to cache the dataset partition summaries and distribute query tasks to relevant nodes [117]. PCC proposes using a performance characteristic curve to allocate appropriate resources to frequent queries [103].

Table 9.4 presents a comparison of state-of-the-art raw data query processing

Table 9.4: RAW-HF Technique Comparison with State-of-the-art

#	Technique / Tool	Partitioning	DBMS Data %	Workload Aware	Ad-hoc queries	RUA	Multi-format Join	Remarks
1	NoDB (PostgresRaw) [33]	-	0	No	NA	No	No	Required More Memory. High QET.
2	Slalom [94]	Logical HP	0	Yes	Yes	No	No	Requires less Memory. Adapts to workload changes.
3	DBMS (PgSQL) [22]	-	100	No	-	No	No	High DLT. Low QET & Resource Utilization.
4	Partial Loading [125]	VP	10.6	Yes	-	Yes	No	Technique Not Lightweight
5	PDC [117] (ODBMS)	HP	0	Yes	No	No	No	Resources Underutilized, Distributed System
6	PCC (Cloud/Serverless) [103]	-	100	Yes	No	Yes	No	Resource Intensive, Distributed System
7	RAW-HF (Hybrid)	VP	6.7	Yes	Yes	Yes	Yes	Lightweight Technique, Can be extended to a distributed setup

techniques with RAW-HF. DBMS Data% shows the percentage of original dataset loaded into DBMS by the technique or data processing tool. Resource Utilization Aware (RUA) shows whether the technique considered resource utilization information or not. The Multi-Format Join column represents whether the tool can execute join queries on data residing in raw and database formats. NoDB eliminates DLT by querying raw files. However, QET time and main memory utilization are very high. Slalom is an improvement over NoDB. It logically partitions raw files and adapts to workload changes using less main memory than NoDB. The PgSQL reduces the QET time at the cost of high DLT. NoDB, PgSQL, and PCC do not use partitioning techniques and require the entire dataset in a single format. Partial loading and RAW-HF use hybrid systems, so both can query multi-format data. The SCANRAW tool used to implement Partial Loading techniques can not join data existing in database and raw partition. Therefore, the multi-format (MF) join feature is not present. Whereas RAW-HF uses NoDB as an extension to PgSQL (PostgreSQL), allowing execution of join queries on the database and raw format. Therefore, the multi-format join feature is present in RAW-HF.

Partial loading, PDC, PCC, and RAW-HF are workload aware. PCC uses workload information to identify appropriate resources, while others use workload information to partition the dataset. Only RAW-HF supports allocating appropriate resources to ad-hoc queries based on query complexity. While, NoDB, Slalom, PgSQL, and Partial loading techniques allocate static resources to all the queries, including the ad-hoc ones. PDC uses a static partition of main memory (50%) to keep lookup tables. PCC needs historical data for allocating appropriate resources to each query. However, it does not work well for ad-hoc queries. In comparison, although RAW-HF performs workload aware partitioning, resource allocation is done based on query complexity. Therefore, RAW-HF is capable of allocating appropriate resources to ad-hoc queries.

The Partial Loading technique [125] and RAW-HF partition the raw dataset into a database and raw partitions considering memory or storage budget. The storage budget limits the amount of data that needs to be loaded into DBMS. The Partial Loading technique tries to load attributes that cover maximum num-

Table 9.5: RAW-HF Performance Parameters Comparison

#	Technique/ Tool	Query Performance %			Resource Utilization %		
		DLT (sec)	QET (sec)	WET (sec)	CPU (sec)	RAM (MB)	IO (MB)
1	NoDB [33]	0	613.59 (99.12%)	613.59 (96.32%)	613.59 (96.32%)	10956.8 (90.36%)	4710.4 (94.96%)
2	DBMS [22]	188.63 (90.88%)	44.7 (87.92%)	233.33 (90.31%)	233.33 (89.78%)	2758.3 (61.70%)	2660.4 (91.08%)
3	Partial Loading [125]	24.4 (29.51%)	6.2 (12.90%)	30.6 (26.14%)	30.6 (26.14%)	709.2 (+32.87%)	316.0 (24.92%)
4	RAW-HF	17.2	5.4	22.6	22.6	1056.5	237.3

ber of workload queries. RAW-HF uses lightweight QCA, WSAC, and MUAR techniques to partition, schedule tasks, and allocate resources to reduce WET. The Partial Loading technique is not lightweight. It requires attribute access time from database & raw formats, data loading time, workload analysis, and other values to find the optimal partitions for hybrid systems, increasing AET. SCANRAW loads data into DBMS whenever resources are available, which makes this technique Resource Utilization Aware. RAW-HF considers available resources to schedule tasks. The PCC uses historical resource utilization of queries. RAW-HF proposes to load only attributes required by complex queries to reduce DLT time and required to load only 6.7% of data. The multi-format join feature present in RAW-HF helps in achieving 0% replication. In comparison, Partial Loading may load 10.6% of the dataset accessed by workload queries when enough storage budget is available. PCC and PDC are implemented for cloud based systems making them distributed systems. Although RAW-HF is implanted on a single machine, it can be used for cloud based systems or extended to distributed setup.

Figure 9.31 & 9.32 presented comparison of these techniques with RAW-HF for SDSS dataset. Table 9.5 shows the RAW-HF performance parameters comparison with state-of-the-art techniques. The table shows the time and resources required by techniques to execute the given workload on the SDSS dataset of 1M records. NoDB accessed the actual raw dataset file of size 4.7GB. PostgreSQL loaded the entire dataset into DBMS, reducing disk space by 43.5%. Partial Loading technique [125]

loaded only the data required by workload queries. The improvement achieved by RAW-HF in each performance parameter is written below the actual data in parentheses in blue color. While a decrease in performance is shown in red. It can be seen that RAW-HF improved WET by 26.14 to 96.32% compared to others. The resource utilization RAW-HF reduced CPU and Disk space utilization for loaded data by 26.14% and 24.92%. But, RAM utilization is increased by 32.87% compared to the Partial Loading technique.

CHAPTER 10

Conclusions and Future work

This thesis develops a Resource Availability and Workload aware Hybrid Framework (RAW-HF) to process raw datasets efficiently. RAW-HF optimized required resources using Query Complexity Aware (QCA) and Workload and Storage aware Cost-based (WSAC) algorithms. This work also addresses the issue of under-utilized resources during query processing. It maximizes utilization of existing resources using the MUAR (Maximizing Utilization of Available Resources) algorithm, which considers the availability of hardware resources in real time with the help of the Resource Monitoring (RM) module. RM module communicates with external resource monitoring tools to send hardware utilization data to MUAR. RAW-HF is demonstrated using a scientific experiment datasets like SDSS and LOD. RAW-HF being resource efficient allows processing queries efficiently on large datasets.

A comparison of the RAW-HF technique and performance with state-of-the-art techniques is presented. RAW-HF allows the execution of join queries on data stored in DBMS and raw format. At the same time, the Partial loading technique does not support the execution of join queries on data residing in multiple formats. RAW-HF never loads partitions used by simple queries, thereby reducing data loading requirements in ORR phase. Unlike PCC, MUER phase of RAW-HF allocates appropriate resources to ad-hoc queries. Result analysis has concluded that MUAR algorithm proposed in MUER phase reduces total workload execution time WET by up to 85% compared to the original WET. MUAR improved WET by 20-55% compared to static CPU & RAM resource maximization techniques, PCC, and Elastic resource allocation techniques. MUAR allocates appro-

priate resources to new queries considering available resources by avoiding time-consuming offline analysis. It can also estimate the near-best resource allocation value of work memory with 15-20% error for frequent queries to achieve the best QET using fewer parameters compared to PCC and AutoToken. The RAW-HF including ORR & MUER reduced the total workload execution time by 26% and 96% compared to the state-of-the-art Partial Loading and NoDB techniques. The overall CPU, RAM, and IO resource utilization has been improved by 61-91% over DBMS. Partial loading technique requires 33% lesser RAM than RAW-HF, but it needs 24% more IO to achieve its best performance. Results analysis has shown that ORR techniques works better for broad table datasets like SDSS, while MUAR is capable of improving WET for workload with complex multi-join queries like LOD. RAW-HF reduced WET by 84.8% for SDSS and 90.3% for LOD datasets compared to traditional DBMS system PostgreSQL.

RAW-HF can not process data streams in real time as the hybrid setup lacks real time stream processing engine. However, it can store data in CSV files to execute queries using the in-situ extension (NoDB). In future, RAW-HF can be implemented and compared using different distributed or cloud setups. Additionally, Machine Learning techniques can be used to accurately allocate query-specific resources considering past resource consumption patterns. RAW-HF can help in building resource efficient cloud based systems.

References

- [1] Citypulse smart city datasets - datasets. <http://iot.ee.surrey.ac.uk:8080/datasets.html>.
- [2] Nasa earth science data | nasa's earth observing system. <https://eospsso.gsfc.nasa.gov/content/nasa-earth-science-data>.
- [3] System monitor - gnome library. <https://help.gnome.org/users/gnome-system-monitor/>.
- [4] Ubuntu manpage: htop - interactive process viewer. <http://manpages.ubuntu.com/manpages/bionic/man1/htop.1.html>.
- [5] Ubuntu manpage: iotop - simple top-like i/o monitor. <http://manpages.ubuntu.com/manpages/focal/en/man8/iotop.8.html>.
- [6] Ubuntu manpage: sar - collect, report, or save system activity information. <https://manpages.ubuntu.com/manpages/xenial/man1/sar.sysstat.1.html>.
- [7] Ubuntu manpage: top - display linux processes. <https://manpages.ubuntu.com/manpages/xenial/man1/top.1.html>.
- [8] Ubuntu manpage: vmstat - report virtual memory statistics. <http://manpages.ubuntu.com/manpages/bionic/man8/vmstat.8.html>.
- [9] 1.2 billion vehicles on world's roads now, 2 billion by 2035: Report. https://www.greencarreports.com/news/1093560_

1-2-billion-vehicles-on-worlds-roads-now-2-billion-by-2035-report
2014.

- [10] Data growth, IoT will lead to unlimited energy consumption if not controlled, scientists warn. https://circleid.com/posts/20160811_internet_data_growth_iot_leading_to_unlimited_energy_consumption/, 2016.
- [11] Github - hbpmedical/postgresraw. <https://github.com/HBPMedical/PostgresRAW>, 2018.
- [12] Dr16-data volume table | sdss. <https://www.sdss.org/dr16>, 2019.
- [13] A new schedule for the LHC and its successor | CERN. <https://home.cern/news/news/accelerators/new-schedule-lhc-and-its-successor>, 2019.
- [14] CERN open data portal. <http://opendata.cern.ch/record/15010>, 2020.
- [15] Github - rawvis: In-situ visual analytics system. <https://github.com/VisualFacts/RawVis>, 2020.
- [16] The linked open data cloud. <https://lod-cloud.net/>, 2020.
- [17] Data volume | sdss. <https://www.sdss.org/dr17>, 2021.
- [18] Amazon athena - serverless interactive query service - aws. <https://aws.amazon.com/athena/>, 2022.
- [19] Bigquery: Cloud data warehouse | google cloud. <https://cloud.google.com/bigquery>, 2022.
- [20] Cloud data warehouse – amazon redshift – aws. <https://aws.amazon.com/redshift/>, 2022.
- [21] Earthdata cloud evolution | earthdata. <https://earthdata.nasa.gov/eosdis/cloud-evolution>, 2022.

- [22] PostgreSQL: The world's most advanced open source database. <https://www.postgresql.org/>, 2022.
- [23] Storage | cern. <https://home.cern/science/computing/storage>, 2022.
- [24] Task manager (windows). [https://en.wikipedia.org/wiki/Task_Manager_\(Windows\)](https://en.wikipedia.org/wiki/Task_Manager_(Windows)), 2022.
- [25] Tpc-homepage. <http://tpc.org/>, 2022.
- [26] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *VLDB Endowment*, 2(2):1664–1665, 2009.
- [27] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs row-stores: How different are they really? categories and subject descriptors. *Sigmod*, June 9-12:967–980, 2008.
- [28] Abdurro'uf, K. Accetta, C. Aerts, V. Silva Aguirre, and et al. The seventeenth data release of the sloan digital sky surveys: Complete release of manga, mastar, and apogee-2 data. *The Astrophysical Journal Supplement Series*, 259(2):35, Apr 2022.
- [29] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: Access-driven data transfer from raw files into database systems. In *16th International Conference on Extending Database Technology - EDBT '13*, page 1–10. ACM Press, 2013.
- [30] R. Ahumada, C. A. Prieto, A. Almeida, F. Anders, S. F. Anderson, B. H. Andrews, B. Anguiano, R. Arcodia, E. Armengaud, M. Aubert, S. Avila, V. Avila-Reese, C. Badenes, C. Balland, K. Barger, J. K. Barrera-Ballesteros, S. Basu, J. Bautista, R. L. Beaton, T. C. Beers, B. I. T. Benavides, C. F. Bender, M. Bernardi, M. Bershad, F. Beutler, C. M. Bidin, J. Bird, D. Bizyaev, G. A. Blanc, M. R. Blanton, M. Boquien, J. Borissova, J. Bovy, W. N. Brandt, J. Brinkmann, J. R. Brownstein, K. Bundy, M. Bureau, A. Burgasser,

E. Burtin, M. Cano-Díaz, R. Capasso, M. Cappellari, R. Carrera, S. Chabanier, W. Chaplin, M. Chapman, B. Cherinka, C. Chiappini, P. Doohyun Choi, S. D. Chojnowski, H. Chung, N. Clerc, D. Coffey, J. M. Comerford, J. Comparat, L. da Costa, M.-C. Cousinou, K. Covey, J. D. Crane, K. Cunha, G. d. S. Ilha, Y. S. Dai, S. B. Damsted, J. Darling, J. W. Davidson, R. Davies, K. Dawson, N. De, A. de la Macorra, N. De Lee, A. B. d. A. Queiroz, A. Deconto Machado, S. de la Torre, F. Dell'Agli, H. du Mas des Bourboux, A. M. Diamond-Stanic, S. Dillon, J. Donor, N. Drory, C. Duckworth, T. Dwelly, G. Ebelke, S. Eftekhazadeh, A. Davis Eigenbrot, Y. P. Elsworth, M. Eracleous, G. Erfanianfar, S. Escoffier, X. Fan, E. Farr, J. G. Fernández-Trincado, D. Feuillet, A. Finoguenov, P. Fofie, A. Fraser-McKelvie, P. M. Frinchaboy, S. Fromenteau, H. Fu, L. Galbany, R. A. Garcia, D. A. García-Hernández, L. A. G. Oehmichen, J. Ge, M. A. G. Maia, D. Geisler, J. Gelfand, J. Goddy, V. Gonzalez-Perez, K. Grabowski, P. Green, C. J. Grier, H. Guo, J. Guy, P. Harding, S. Hasselquist, A. J. Hawken, C. R. Hayes, F. Hearty, S. Hekker, D. W. Hogg, J. A. Holtzman, D. Horta, J. Hou, B.-C. Hsieh, D. Huber, J. A. S. Hunt, J. I. Chitham, J. Imig, M. Jaber, C. E. J. Angel, J. A. Johnson, A. M. Jones, H. Jönsson, E. Jullo, Y. Kim, K. Kinemuchi, C. C. Kirkpatrick IV, G. W. Kite, M. Klaene, J.-P. Kneib, J. A. Kollmeier, H. Kong, M. Kounkel, D. Krishnarao, I. Lacerna, T.-W. Lan, R. R. Lane, D. R. Law, J.-M. Le Goff, H. W. Leung, H. Lewis, C. Li, J. Lian, L. Lin, D. Long, P. Longa-Peña, B. Lundgren, B. W. Lyke, J. Ted Mackereth, C. L. MacLeod, S. R. Majewski, A. Manchado, C. Maraston, P. Martini, T. Masseron, K. L. Masters, S. Mathur, R. M. McDermid, A. Merloni, M. Merrifield, S. Mészáros, A. Miglio, D. Minniti, R. Minsley, T. Miyaji, F. G. Mohammad, B. Mosser, E.-M. Mueller, D. Muna, A. Muñoz-Gutiérrez, A. D. Myers, S. Nadathur, P. Nair, K. Nandra, J. C. do Nascimento, R. J. Nevin, J. A. Newman, D. L. Nidever, C. Nitschelm, P. Noterdaeme, J. E. O'Connell, M. D. Olmstead, D. Oravetz, A. Oravetz, Y. Osorio, Z. J. Pace, N. Padilla, N. Palanque-Delabrouille, P. A. Palicio, H.-A. Pan, K. Pan, J. Parker, R. Paviot, S. Peirani, K. P. Ramírez, S. Penny, W. J. Percival, I. Perez-Fournon, I. Pérez-Ràfols, P. Petitjean, M. M. Pieri,

M. Pinsonneault, V. J. Poovelil, J. T. Povick, A. Prakash, A. M. Price-Whelan, M. J. Raddick, A. Raichoor, A. Ray, S. B. Rembold, M. Rezaie, R. A. Riffel, R. Riffel, H.-W. Rix, A. C. Robin, A. Roman-Lopes, C. Román-Zúñiga, B. Rose, A. J. Ross, G. Rossi, K. Rowlands, K. H. R. Rubin, M. Salvato, A. G. Sánchez, L. Sánchez-Menguiano, J. R. Sánchez-Gallego, C. Sayres, A. Schaefer, R. P. Schiavon, J. S. Schimoia, E. Schlafly, D. Schlegel, D. P. Schneider, M. Schultheis, A. Schwobe, H.-J. Seo, A. Serenelli, A. Shafieloo, S. J. Shamsi, Z. Shao, S. Shen, M. Shetrone, R. Shirley, V. S. Aguirre, J. D. Simon, M. F. Skrutskie, A. Slosar, R. Smethurst, J. Sobek, B. C. Sodi, D. Souto, D. V. Stark, K. G. Stassun, M. Steinmetz, D. Stello, J. Stermer, T. Storchi-Bergmann, A. Streblyanska, G. S. Stringfellow, A. Stutz, G. Suárez, J. Sun, M. Taghizadeh-Popp, M. S. Talbot, J. Tayar, A. R. Thakar, R. Theriault, D. Thomas, Z. C. Thomas, J. Tinker, R. Tojeiro, H. H. Toledo, C. A. Tremonti, N. W. Troup, S. Tuttle, E. Unda-Sanzana, M. Valentini, J. Vargas-González, M. Vargas-Magaña, J. A. Vázquez-Mata, M. Vivek, D. Wake, Y. Wang, B. A. Weaver, A.-M. Weijmans, V. Wild, J. C. Wilson, R. F. Wilson, N. Wolthuis, W. M. Wood-Vasey, R. Yan, M. Yang, C. Yèche, O. Zamora, P. Zarrouk, G. Zasowski, K. Zhang, C. Zhao, G. Zhao, Z. Zheng, Z. Zheng, G. Zhu, and H. Zou. The 16th Data Release of the Sloan Digital Sky Surveys: First Release from the APOGEE-2 Southern Survey and Full Release of eBOSS Spectra. *The Astrophysical Journal Supplement Series*, 249(1):3, jul 2020.

- [31] A. Ailamaki. Managing scientific data. In *International conference on Management of data - SIGMOD '11*, page 1045. ACM Press, 2011.
- [32] A. Ailamaki. Databases and hardware: The beginning and sequel of a beautiful friendship. *VLDB Endowment*, 8(12):2058–2061, 2015.
- [33] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: Efficient query execution on raw data files. In *International conference on Management of Data - SIGMOD*, volume 58, page 241. ACM Press, 2012.
- [34] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb in action. *VLDB Endowment*, 5(12):1942–1945, Aug 2012.

- [35] A. Alvarez-Ayllon, M. Palomo-Duarte, and J. M. Dodero. Interactive data exploration of distributed raw files: A systematic mapping study. *IEEE Access*, 7:10691–10717, 2018.
- [36] R. R. Amossen. Vertical partitioning of relational oltp databases using integer programming. In *26th International Conference on Data Engineering Workshops (ICDEW)*, page 93–98. IEEE, 2010.
- [37] A. Anagnostou, M. Olma, and A. Ailamaki. Alpine: Efficient in-situ data exploration in the presence of updates. *ACM International Conference on Management of Data - SIGMOD*, page 1651–1654, 2017.
- [38] A. S. G. Andrae. Total consumer power consumption forecast. *Nordic Digital Business Summit*, 10(October):1, 2017.
- [39] T. Azim, M. Karpathiotakis, and A. Ailamaki. Recache: Reactive caching for fast analytics over heterogeneous data. *VLDB Endowment*, 11(3):324–337, 2017.
- [40] T. Azim, A. Nadeem, A. Ailamaki, and F. Polytechnique. Adaptive cache mode selection for queries over raw data. In *In Ninth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, number September, 2018.
- [41] W. Bachman Charles. Integrated data store-the information processing machine that we need. In *General Electric Computer Users Symposium. Kiamesha Lake. New York May*, pages 17–18, 1962.
- [42] M. Balazinska, A. Deshpande, M. J. Franklin, P. B. Gibbons, J. Gray, M. Hansen, M. Liebhold, S. Nath, A. Szalay, and V. Tao. Data management in the worldwide sensor web. *IEEE Pervasive Computing*, 6(2):30–40, Apr 2007.
- [43] S. Baunsgaard, M. Boehm, A. Chaudhary, B. Derakhshan, S. Geißelsoder, P. M. Grulich, M. Hildebrand, K. Innerebner, V. Markl, C. Neubauer, S. Osterburg, O. Ovcharenko, S. Redyuk, T. Rieger, A. Rezaei Mahdiraji, S. B.

- Wrede, and S. Zeuch. Exdra: Exploratory data science on federated raw data. In *International Conference on Management of Data*, page 2450–2463. ACM, Jun 2021.
- [44] N. Bikakis, S. Maroulis, G. Papastefanatos, and P. Vassiliadis. Rawvis: visual exploration over raw data. In *European Conference on Advances in Databases and Information Systems*, pages 50–65. Springer, 2018.
- [45] M. Boissier, T. Dürken, R. Schlosser, and M. Faust. A cost-aware and workload-based index advisor for columnar in-memory databases. *International Conference on Information and Software Technologies, Springer International Publishing Switzerland*, 639:285–299, 2016.
- [46] M. Boissier, C. Meyer, M. Uflacker, and C. Tinnefeld. And all of a sudden: Main memory is less expensive than disk. In T. Rabl, K. Sachs, M. Poess, C. Baru, and H.-A. Jacobson, editors, *Workshop on Big Data Benchmarks*, volume 8991 of *Lecture Notes in Computer Science*, page 132–144. Springer, 2015.
- [47] M. Boissier, A. Spivak, and C. Meyer. Improving tuple reconstruction for tiered column stores: A workload-aware ansatz based on table reordering. In *Australasian Computer Science Week Multiconference*, pages 1–10, 2017.
- [48] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser. Smooth scan: robust access path selection without cardinality estimation. *VLDB Journal*, 27(4):521–545, 2018.
- [49] R. Borovica-Gajić, R. Appuswamy, and A. Ailamaki. Cheap data analytics using cold storage devices. *VLDB Endowment*, 9(12):1029–1040, 2016.
- [50] M. Castillo. From hard drives to flash drives to dna drives. *American Journal of Neuroradiology*, 35(1):1–2, 2014.
- [51] R. Chaiken, B. Jenkins, P.- Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *VLDB Endowment*, 1(2):1265–1276, Aug 2008.

- [52] Y. Cheng. In-situ data processing over raw file. University of California, Merced, 2016.
- [53] Y. Cheng and F. Rusu. SCANRAW: A database meta-operator for parallel in-situ processing and loading. *ACM Transactions on Database Systems*, 40(3):1–45, Oct 2015.
- [54] Y. Cheng, W. Zhao, and F. Rusu. OLA-RAW: Scalable exploration over raw data. *arXiv preprint arXiv:1702.00358*, page 1–23, Feb 2017.
- [55] C. Ding, D. Tang, X. Liang, A. J. Elmore, and S. Krishnan. Ciao: An optimization framework for client-assisted data loading. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, page 1979–1984. IEEE, Apr 2021.
- [56] A. Dziedzic, M. Karpathiotakis, I. Alagiannis, R. Appuswamy, and A. Ailamaki. Dbms data loading: An analysis on modern hardware. In *Data Management on New Hardware*, volume 10195 LNCS, page 95–117. Springer, 2017.
- [57] G. M. D’Silva, A. Khan, Gaurav, and S. Bari. Real-time processing of IoT events with historic data using apache kafka and apache spark with dashing framework. *RTEICT 2017 - 2nd IEEE International Conference on Recent Trends in Electronics, Information and Communication Technology, Proceedings*, page 1804–1809, 2018.
- [58] C. Ge, Y. Li, E. Eilebrecht, B. Chandramouli, and D. Kossmann. Speculative distributed csv data parsing for big data analytics. page 883–899, 2019.
- [59] C. Gorenflo, L. Golab, and S. Keshav. Managing sensor data streams. In *29th International Conference on Scientific and Statistical Database Management*, page 1–11. ACM, Jun 2017.
- [60] M. Grund, J. Kr, A. Zeier, P. Cudre-mauroux, and S. Madden. HYRISE — A Main Memory Hybrid Storage Engine. *VLDB Endowment*, 4(2):105–116, 2011.

- [61] H. Guo, L. Wang, and D. Liang. Big earth data from space: a new engine for earth science. *Science Bulletin*, 61(7):505–513, Apr 2016.
- [62] A. D. Gvishiani, M. N. Dobrovolsky, B. V. Dzeranov, and B. A. Dzeboev. Big data in geophysics and other earth sciences. *Izvestiya, Physics of the Solid Earth* 2022 58:1, 58(1):1–29, Feb 2022.
- [63] P. Helman. A family of NP-complete data aggregation problems. *Acta Informatica*, 26(5):485–499, Mar 1989.
- [64] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? *5th Biennial Conference on Innovative Data Systems Research, Conference Proceedings - CIDR*, page 57–68, 2011.
- [65] S. Idreos and T. Kraska. From auto-tuning one size fits all to self-designed and learned data-intensive systems. *ACM SIGMOD International Conference on Management of Data*, 19:2054–2059, Jun 2019.
- [66] M. Ivanova, M. Kersten, and S. Manegold. Data vaults: A symbiosis between database technology and scientific file repositories. In *International Conference on Scientific and Statistical Database Management*, volume 7338 LNCS, page 485–494. Springer, 2012.
- [67] A. Jain, T. Padiya, and M. Bhise. Log based method for faster IoT queries. In *IEEE Region 10 Symposium (TENSymp)*, page 1–4, 2017.
- [68] L. Jiang and Z. Zhao. JSONski: Streaming Semi-structured Data with Bit-Parallel Fast-Forwarding. page 12, 2022.
- [69] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast queries over heterogeneous data through engine customization. *VLDB Endowment*, 9(12):972–983, 2016.
- [70] M. Karpathiotakis, I. Alagiannis, T. Heinis, B. Miguel, and A. Ailamaki. Just-in-time data virtualization: Lightweight data management with ViDa. In *7th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

- [71] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive query processing on raw data. *VLDB Endowment*, 7(12):1119–1130, 2014.
- [72] M. S. Kester, M. Athanassoulis, and S. Idreos. Access path selection in main-memory optimized data systems: Should i scan or should i probe? *ACM SIGMOD International Conference on Management of Data*, Part F1277:715–730, 2017.
- [73] S. Kim, J. Lee, T. Kim, and B. Moon. Scalable parallel data loading in scidb. *International Conference on Big Data, Big Data*, page 3443–3446, 2017.
- [74] A. Kipf, V. Pandey, J. Bottcher, L. Braun, T. Neumann, and A. Kemper. Scalable analytics on fast data. *ACM Transactions on Database Systems*, 44(1), 2019.
- [75] L. L. Knud. State of the IoT 2018: Number of IoT devices now at 7b – market accelerating. <https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/>, 2018.
- [76] J. J. Levandoski, P.-A. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *29th International Conference on Data Engineering (ICDE)*, pages 26–37. IEEE, apr 2013.
- [77] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: A fast json parser for data analytics. *VLDB Endowment*, 10(10):1118–1129, Jun 2017.
- [78] Y. Li, Y. Wen, and X. Yuan. Online Aggregation: A Review. In *Lecture Notes in Computer Science*, volume 11242 LNCS, pages 103–114. Springer, Cham, sep 2018.
- [79] Y. Li, Y. Wen, and X. Yuan. Online aggregation: A review. 11242 LNCS:103–114, 2018.

- [80] T. Lindemann, J. Kauke, and J. Teubner. Efficient stream processing of scientific data. *Proceedings - IEEE 34th International Conference on Data Engineering Workshops, ICDEW 2018*, (1020):140–145, 2018.
- [81] G. Liu, L. Chen, and S. Chen. Zen+: a robust NUMA-aware OLTP engine optimized for non-volatile main memory. *The VLDB Journal*, apr 2022.
- [82] X. Lu, C. Cheng, J. Gong, and L. Guan. Review of data storage and management technologies for massive remote sensing data. *Sci China Technol Sci*, 54(12):3220–3232, Dec 2011.
- [83] Q. Ma, A. M. Shanghooshabad, M. Almasi, M. Kurmanji, and P. Triantafyllou. Learned approximate query processing: Make it light, accurate and fast. In *11th Annual Conference on Innovative Data Systems Research (CIDR)*, page 11. ACM, 2021.
- [84] A. Magalhaes, J. M. Monteiro, and A. Brayner. Jose Maria Monteiro, and Angelo Brayner. 2021. Main Memory Database Recovery: A Survey. *ACM Comput. Surv*, 54, 2021.
- [85] R. Mancini, S. Karthik, B. Chandra, V. Mageirakos, and A. Ailamaki. Efficient massively parallel join optimization for large queries. pages 122–135, 2022.
- [86] S. Maroulis, N. Bikakis, G. Papastefanatos, P. Vassiliadis, and Y. Vassiliou. Adaptive indexing for in-situ visual exploration and analytics. *23rd International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*, 2021.
- [87] S. Maroulis, N. Bikakis, G. Papastefanatos, P. Vassiliadis, and Y. Vassiliou. Resource-aware adaptive indexing for in situ visual exploration and analytics. *The VLDB Journal*, pages 1–29, 2022.
- [88] K. Mershad and A. Hamieh. Sdms: smart database management system for accessing heterogeneous databases. *International Journal of Intelligent Information and Database Systems*, 14(2):115, 2021.

- [89] T. Milo. Getting rid of data. *Journal of Data and Information Quality*, 12(1):1–7, Jan 2020.
- [90] H. Mohammad. Number of connected IoT devices growing 18% to 14.4 billion globally. <https://iot-analytics.com/number-connected-iot-devices/>, May 2022.
- [91] O. Moll, A. Zalewski, S. Pillai, S. Madden, M. Stonebraker, and V. Gadeppally. Exploring big volume sensor data with vroom. *VLDB Endowment*, 10(12):1973–1976, 2017.
- [92] T. Muhlbauer, W. Rodiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *VLDB Endowment*, 6(14):1702–1713, 2014.
- [93] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting through raw data via adaptive partitioning and indexing. *VLDB Endowment*, 10(10):1106–1117, 2017.
- [94] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Adaptive partitioning and indexing for in situ query processing. *The VLDB Journal*, 29(1):569–591, Jan 2020.
- [95] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer New York, 2011.
- [96] T. Padiya and M. Bhise. DWAHP: Workload Aware Hybrid Partitioning and Distribution of RDF Data. In *21st International Database Engineering & Applications Symposium (IDEAS)*, page 235–241. ACM, 2017.
- [97] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia. Filter before you parse. *VLDB Endowment*, 11(11):1576–1589, Jul 2018.
- [98] A. Pandat, N. Gupta, and M. Bhise. Load balanced semantic aware distributed rdf graph. In *25th International Database Engineering Applications Symposium (IDEAS)*, page 127–133. ACM, Jul 2021.

- [99] H. S. Patel, Q. Shi, A. Jindal, M. K. Bag, R. Sen, and C. A. Curino. Resource optimization for serverless query processing, Apr. 2021. US Patent App. 16/697,960.
- [100] J. Patel. Scalable big data modelling. *International Journal of Big Data Intelligence*, 7(4):194, 2020.
- [101] E. Petraki, S. Idreos, and S. Manegold. Holistic indexing in main-memory column-stores. In *ACM SIGMOD International Conference on Management of Data*, pages 1153–1166, 2015.
- [102] E. Petraki and S. Manegold. Holistic indexing in main-memory column. *SIGMOD’15*, 2015.
- [103] A. Pimpley, S. Li, R. Sen, S. Srinivasan, and A. Jindal. Towards optimal resource allocation for big data analytics. In *International Conference on Extending Database Technology*, 2022.
- [104] D. A. N. Puiu, P. Barnaghi, S. Member, R. Tonjes, D. Kumper, M. I. Ali, A. Mileo, J. X. Parreira, M. Fischer, S. Kolozali, N. Farajidavar, F. Gao, T. Iggena, T.-l. Pham, C.-s. Nechifor, D. Puschmann, and J. Fernandes. City-pulse: Large scale data analytics framework for smart cities. *IEEE Access*, 4, 2016.
- [105] Y. Qin, Q. Z. Sheng, N. J. G. Falkner, S. Dustdar, H. Wang, and A. V. Vasilakos. When things matter: A survey on data-centric internet of things. *Journal of Network and Computer Applications*, 64:137–153, 2016.
- [106] A. Raza, P. Chrysogelos, A. C. Anadiotis, and A. Ailamaki. Adaptive htap through elastic resource scheduling. In *ACM SIGMOD International Conference on Management of Data*, page 2043–2054. ACM, Jun 2020.
- [107] V. U. Reus. Query processing on raw files. *Seminar “Recent Trends in Database Research*, page 13, 2013.

- [108] D. Saxena and A. K. Singh. A proactive autoscaling and energy-efficient vm allocation framework using online multi-resource neural network for cloud data center. *Neurocomputing*, 426:248–264, Feb 2021.
- [109] R. Sen, A. Jindal, H. Patel, and S. Qiao. AutoToken: Predicting Peak Parallelism for Big Data Analytics at Microsoft. *VLDB Endowment*, 13(12), aug 2020.
- [110] A. Shaikhha, M. Schleich, A. Ghita, and D. Olteanu. Multi-layer optimizations for end-to-end data analytics. *18th ACM/IEEE International Symposium on Code Generation and Optimization, CGO*, 13:145–157, Feb 2020.
- [111] Z. Shang, X. Liang, D. Tang, C. Ding, A. J. Elmore, S. Krishnan, and M. J. Franklin. Crocodiledb: Efficient database execution through intelligent deferment. *10th Annual Conference on Innovative Data Systems Research (CIDR)*, 2020.
- [112] J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. P. L. F. d. Carvalho, and J. Gama. Data stream clustering: A survey. *ACM Computing Surveys*, 46(1):1–31, Oct 2013.
- [113] V. Silva, J. Leite, J. J. Camata, D. de Oliveira, A. L. Coutinho, P. Valduriez, and M. Mattoso. Raw data queries during data-intensive parallel workflow execution. *Future Generation Computer Systems*, 75:402–422, Oct 2017.
- [114] H. Singh, A. Bhasin, and P. R. Kaveri. Qras: efficient resource allocation for task scheduling in cloud computing. *SN Applied Sciences*, 3(4):1–7, Apr 2021.
- [115] U. Sirin, S. Dwarkadas, and A. Ailamaki. Workload interference analysis for htap*. *11th Annual Conference on Innovative Data Systems Research (CIDR)*, 2021.
- [116] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, P. O’Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *Making Databases Work*:

the Pragmatic Wisdom of Michael Stonebraker, volume 10, page 491–518. Association for Computing Machinery, Dec 2018.

- [117] H. Tang, S. Byna, B. Dong, and Q. Koziol. Parallel query service for object-centric data management systems. *34th International Parallel and Distributed Processing Symposium Workshops, IPDPSW*, page 406–415, 2020.
- [118] L. Viswanathan, A. Jindal, and K. Karanasos. Query and resource optimization: Bridging the gap. In *34th International Conference on Data Engineering (ICDE)*, page 1384–1387. IEEE, Apr 2018.
- [119] Y. Wang, Q. Zhang, X. Huang, M. R. Ghorri, A. S. Sadiq, and A. Ghani. Smart traffic management system for traffic control using automated mechanical and electronic devices. *IOP Conference Series: Materials Science and Engineering*, 377(1):012201, Jun 2018.
- [120] A. Watson, S. K. Das, and S. Ray. Daskdb: Scalable data science with unified data analytics and in situ query processing. In *8th International Conference on Data Science and Advanced Analytics (DSAA)*, page 1–10. IEEE, Oct 2021.
- [121] G. Weikum. Database researchers: plumbers or thinkers? *14th Int. Conf. on Extending Database Technology*, 31(3):9–10, 2011.
- [122] A. Witkowski, M. Colgan, A. Brumm, T. Cruanes, and H. Baer. Performant and scalable data loading with oracle database 11g. *Oracle*, (March), 2011.
- [123] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a learning optimizer for shared clouds. *Towards a Learning Optimizer for Shared Clouds. PVLDB*, 12(3):210–222, 2018.
- [124] W. E. Zhang, Q. Z. Sheng, K. Taylor, and Y. Qin. Identifying and Caching Hot Triples for Efficient RDF Query Processing. In *International Conference on Database Systems for Advanced Applications*, volume 1, pages 259–274. Springer, 2015.

- [125] W. Zhao, Y. Cheng, and F. Rusu. Vertical partitioning for query processing over raw data. In *27th International Conference on Scientific and Statistical Database Management*, volume 29-June-20, page 1–12. ACM, Jun 2015.
- [126] W. Zhao, F. Rusu, B. Dong, K. Wu, A. Y. Q. Ho, and P. Nugent. Distributed Caching for Complex Querying of Raw Arrays. *Radiology*, 266(1):326–336, mar 2018.

CHAPTER A

Appendix A

List of SDSS Queries

- Q1 SELECT p.objid, p.run, p.rerun, p.camcol, p.field, p.obj, p.type, p.ra, p.dec, p.u,p.g,p.r,p.i,p.z, p.Err_u, p.Err_g, p.Err_r,p.Err_i,p.Err_z FROM (select objID from PhotoPrimary where objID<1237658364002994835 and objID>1237658198131757067) n, PhotoPrimary p WHERE n.objID=p.objID limit 10;
- Q2 select p.objid,p.ra,p.dec,p.u,p.g,p.r,p.i,p.z FROM PhotoPrimary AS p WHERE type=6 AND ((cast(flags_r as bigint) & 268435456) != 0) AND ((cast(flags_r as bigint) & 141837000769700) = 0) AND (((cast(flags_r as bigint) & 70368744177664) = 0) or (psfmagerr_g <= 0.03)) AND (((cast(flags_r as bigint) & 17592186044416) = 0) or (cast(flags_r as bigint) & 4096) = 0) AND p.ra BETWEEN (352.427925-0.183351) AND (352.427925+0.183351) AND p.dec BETWEEN (0.141831-0.183350) AND (0.141831+6.013350) ORDER BY p.objid;
- Q3 SELECT p.objid, p.run, p.rerun, p.camcol, p.field, p.obj, p.type, p.ra, p.dec, p.u,p.g,p.r,p.i,p.z, p.Err_u, p.Err_g, p.Err_r,p.Err_i,p.Err_z FROM (select objID from PhotoPrimary where objID<1237665364002994835 and objID>1237610198131757067) n, PhotoPrimary p WHERE n.objID=p.objID limit 500000;
- Q4 SELECT objID, ra ,dec FROM PhotoPrimary WHERE ra >185 and ra <185.1 AND dec >56.2 and dec <56.3 limit 100;

- Q5 SELECT P.ObjID FROM PhotoPrimary AS P JOIN PhotoPrimary AS N ON P.ObjID = N.ObjID JOIN PhotoPrimary AS L ON L.ObjID = N.ObjID+1 WHERE P.ObjID <L. ObjID and $\text{abs}((P.u-P.g)-(L.u-L.g)) < 0.05$ and $\text{abs}((P.g-P.r)-(L.g-L.r)) < 0.05$ and $\text{abs}((P.r-P.i)-(L.r-L.i)) < 0.05$ and $\text{abs}((P.i-P.z)-(L.i-L.z)) < 0.05$ limit 10;
- Q6 SELECT count(*) as total, sum(case when (Type=3) then 1 else 0 end) as Galaxies, sum(case when (Type=6) then 1 else 0 end) as Stars, sum(case when (Type not in (3,6)) then 1 else 0 end) as Other FROM PhotoPrimary WHERE ((u - g >2.0) or (u >22.3)) and (i between 0 and 19) and (g - r >1.0) and ((r - i <0.08 + 0.42 * (g - r - 0.96)) or (g - r >2.26)) and (i - z <0.25);
- Q7 SELECT * FROM PhotoPrimary WHERE(dered_r-dered_i) <2 AND cmodelmag_i-extinction_i BETWEEN 17.5 AND 19.9 AND (dered_r-dered_i) - (dered_g-dered_r)/8. >0.55 AND fiber2mag_i <21.7 AND de-
 vrad_i <20.0 AND dered_i <19.86 + 1.60*((dered_r-dered_i) - (dered_g-dered_r)/8. - 0.80) limit 10;
- Q8 SELECT '' || cast(p.objId as varchar(20)) || '' as objID, p.run, p.rerun, p.camcol, p.field, p.obj, p.type, p.ra, p.dec, p.u,p.g,p.r,p.i,p.z, p.Err_u, p.Err_g, p.Err_r,p.Err_i,p.Err_z FROM (select objID from PhotoPrimary where objID<1237658364002994835 and objID>1237658198131757067) n, PhotoPrimary p WHERE n.objID=p.objID limit 10;
- Q9 SELECT '' || cast(p.objId as varchar(20)) || '' as objID, p.run, p.rerun, p.camcol, p.field, p.obj, p.type, p.ra, p.dec, p.u,p.g,p.r,p.i,p.z, p.Err_u, p.Err_g, p.Err_r,p.Err_i,p.Err_z FROM (select objID from PhotoPrimary where objID<1237668364002994835 and objID>1237648198131757067) n, PhotoPrimary p WHERE n.objID=p.objID AND p.u between 0 AND 17 AND p.g between 0 AND 15 limit 10;

Q10 select specObjID, objID, ra, dec, mode, type, clean, cModelMag_u, cModelMag_g, cModelMag_r, cModelMag_i, cModelMag_z, cModelMagErr_u, cModelMagErr_g, cModelMagErr_r, cModelMagErr_i, cModelMagErr_z from PhotoPrimary where dec>22.6 and dec<=22.69;

Q11 select s.ObjID, s.ra,s.dec,s.deVRad_r,s.deVAB_r, s.raErr,s.decErr, s.type,s.modelMag_u,s.modelMagErr_u,s.modelMag_g,s.modelMagErr_g, s.modelMag_r,s.modelMagErr_r,s.modelMag_i,s.modelMagErr_i, s.modelMag_z,s.modelMagErr_z from PhotoPrimary s, (select objID from PhotoPrimary where objID<1237658364002994835 and objID>1237658198131757067) n where s.ObjID = n.ObjID AND s.mode = 1;

Q12 SELECT '' || cast(p.objId as varchar(20)) || '' as objID, p.run, p.rerun, p.camcol, p.field, p.obj, p.type, p.ra, p.dec, p.u,p.g,p.r,p.i,p.z, p.Err_u, p.Err_g, p.Err_r,p.Err_i,p.Err_z FROM (select objID from PhotoPrimary where objID<1237658364002994835 and objID>1237610198131757067) n, PhotoPrimary p WHERE n.objID=p.objID limit 500000;

CHAPTER B

Appendix B

List of LOD Queries

```
Q1 select      L1.sub,L1.obj,L2.sub,L2.obj,L3.sub,L3.obj      from
LODTriples  L1,  LODTriples  L2,  LODTriples  L3  where
L1.pred     like      '<http://knoesis.wright.edu/ssw/ont/sensor-
observation.owl#hasLocation>'      and      L1.sub      like
'<http://knoesis.wright.edu/ssw/LocatedNearRel4UT01>'      and
L2.pred     like      '<http://knoesis.wright.edu/ssw/ont/sensor-
observation.owl#floatValue>'      and      L2.obj      like
'"6.0"^^<http://www.w3.org/2001/      XMLSchema#float>'      and
L3.pred     like      '<http://knoesis.wright.edu/ssw/ont/sensor-
observation.owl#uom>'      and      L3.obj      like
'<http://knoesis.wright.edu/ssw/ont/weather.owl#fahrenheit>'      and
L2.sub=L3.sub;
```


- Q2 select L1.sub,L2.obj,L2.sub,L2.obj from LODTriples L1, LODTriples L2, LODTriples L3 where L1.pred like '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L1.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl/RelativeHumidityObservation>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.obj like '"25.0"^^<http://www.w3.org/2001/XMLSchema#float>' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L2.sub = L3.obj and L1.sub=L3.sub;
- Q3 select sub, obj from LODTriples where pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocation>' and sub like '<http://knoesis.wright.edu/ssw/LocatedNearRelAAMA1>';
- Q4 select L1.sub,L1.obj, L2.sub,L2.obj, L3.sub,L3.obj from LODTriples L1, LODTriples L2, LODTriples L3 where L1.pred like '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L1.sub like '<http://knoesis.wright.edu/ssw/Observation_WindSpeed%>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.obj like '"9.0"^^<http://www.w3.org/2001/XMLSchema#float>' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L1.sub=L3.sub and L2.sub=L3.obj;

Q5 select L1.sub,L3.sub,L3.obj,L4.obj from LODTriples L1, LODTriples L2, LODTriples L3, LODTriples L4, LODTriples L5, LODTriples L6 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocation>' and L1.sub like '<http://knoesis.wright.edu/ssw/LocatedNearRelAP268>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.sub like '<http://knoesis.wright.edu/ssw/MeasureData_Precipitation%/' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L3.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#centimeters>' and L4.pred like '<http://www.w3.org/2006/time#inXSDDateTime>' and L4.obj like '"2004-08-09T10:40:00-06:00^^http://www.w3.org/2001/XMLSchema#dateTime"' and L5.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and L5.sub like '<http://knoesis.wright.edu/ssw/Observation_Precipitation%/' and L6.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L6.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#_Precipitation>' and L2.sub=L3.sub and L4.sub=L5.obj and L5.sub=L6.sub;

Q6 select L1.sub,L3.sub,L3.obj,L4.obj from LODTriples L1, LODTriples L2, LODTriples L3, LODTriples L4, LODTriples L5, LODTriples L6 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocation>' and L1.sub like '<http://knoesis.wright.edu/ssw/LocatedNearRelA21>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.sub like '<http://knoesis.wright.edu/ssw/MeasureData_WindSpeed%' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L3.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#milesPerHour>' and L4.pred like '<http://www.w3.org/2006/time#inXSDDateTime>' and L4.obj like '"2004-08-09T10:40:00-06:00^^http://www.w3.org/2001/XMLSchema#dateTime"' and L5.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and L5.sub like '<http://knoesis.wright.edu/ssw/Observation_WindSpeed%' and L6.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L6.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#_WindSpeed>' and L2.sub=L3.sub and L4.sub=L5.obj and L5.sub=L6.sub;

Q7 select L1.sub,L1.obj,L2.obj from LODTriples L1, LODTriples L2 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#ID>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>' and L1.sub = L2.sub;

- Q8 select L1.sub,L1.obj,L2.obj,L3.obj from LODTriples L1, LODTriples L2, LODTriples L3 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>' and L2.pred like '<http://www.w3.org/2003/01/geo/wgs84_pos#alt>' and L3.pred like '<http://www.w3.org/2003/01/geo/wgs84_pos#lat>' and L1.obj=L2.sub and L1.obj=L3.sub;
- Q9 select L1.sub,L3.sub,L3.obj,L4.obj from LODTriples L1, LODTriples L2, LODTriples L3, LODTriples L4, LODTriples L5, LODTriples L6 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocation>' and L1.sub like '<http://knoesis.wright.edu/ssw/LocatedNearRelA25>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.sub like '<http://knoesis.wright.edu/ssw/MeasureData_Precipitation%>' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L3.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#centimeters>' and L4.pred like '<http://www.w3.org/2006/time#inXSDDateTime>' and L4.obj like '"2004-08-09T10:40:00-06:00^^http://www.w3.org/2001/XMLSchema#dateTime"' and L5.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and L5.sub like '<http://knoesis.wright.edu/ssw/Observation_Precipitation%>' and L6.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L6.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#_Precipitation>' and L2.sub=L3.sub and L4.sub=L5.obj and L5.sub=L6.sub;

Q10 select L1.sub,L3.sub,L3.obj,L4.obj from LODTriples L1, LODTriples L2, LODTriples L3, LODTriples L4, LODTriples L5, LODTriples L6 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocation>' and L1.sub like '<http://knoesis.wright.edu/ssw/LocatedNearRelA25>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.sub like '<http://knoesis.wright.edu/ssw/MeasureData_AirTemperature%' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L3.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#fahrenheit>' and L4.pred like '<http://www.w3.org/2006/time#inXSDDateTime>' And L4.obj like '"2004-08-09T10:40:00-06:00^^http://www.w3.org/2001/XMLSchema#dateTime"' and L5.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and L5.sub like '<http://knoesis.wright.edu/ssw/Observation_AirTemperature%' and L6.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L6.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#_AirTemperature>' and L2.sub=L3.sub and L4.sub=L5.obj and L5.sub=L6.sub;

Q11 select L1.sub,L3.sub,L3.obj,L4.obj from LODTriples L1, LODTriples L2, LODTriples L3, LODTriples L4, LODTriples L5, LODTriples L6 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocation>' and L1.sub like '<http://knoesis.wright.edu/ssw/LocatedNearRelA25>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.sub like '<http://knoesis.wright.edu/ssw/MeasureData_WindSpeed%' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L3.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#milesPerHour>' and L4.pred like '<http://www.w3.org/2006/time#inXSDDateTime>' and L4.obj like '"2004-08-09T10:40:00-06:00^^http://www.w3.org/2001/XMLSchema#dateTime"' and L5.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and L5.sub like '<http://knoesis.wright.edu/ssw/Observation_WindSpeed%' and L6.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L6.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#_WindSpeed>' and L2.sub=L3.sub and L4.sub=L5.obj and L5.sub=L6.sub;

Q12 select L1.sub,L3.sub,L3.obj,L4.obj from LODTriples L1, LODTriples L2, LODTriples L3, LODTriples L4, LODTriples L5, LODTriples L6 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocation>' and L1.sub like '<http://knoesis.wright.edu/ssw/LocatedNearRelA25>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.sub like '<http://knoesis.wright.edu/ssw/MeasureData_WindGust%>' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L3.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#milesPerHour>' and L4.pred like '<http://www.w3.org/2006/time#inXSDDateTime>' and L4.obj like '"2004-08-09T10:40:00-06:00^^http://www.w3.org/2001/XMLSchema#dateTime"' and L5.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and L5.sub like '<http://knoesis.wright.edu/ssw/Observation_WindGust%>' and L6.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L6.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#_WindGust>' and L2.sub=L3.sub and L4.sub=L5.obj and L5.sub=L6.sub;

Q13 Select L1.sub,L1.obj,L2.obj,L3.obj from LODTriples L1, LODTriples L2, LODTriples L3, LODTriples L4 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>' and L2.pred like '<http://www.w3.org/2003/01/geo/wgs84_pos#alt>' and L3.pred like '<http://www.w3.org/2003/01/geo/wgs84_pos#lat>' and L4.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#ID>' and L4.obj like '"A07"' and L1.obj=L2.sub and L1.obj=L3.sub and L1.sub=L4.sub;

Q14 select L1.sub,L3.sub,L3.obj,L4.obj from LODTriples L1, LODTriples L2, LODTriples L3, LODTriples L4, LODTriples L5, LODTriples L6 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocation>' and L1.sub like '<http://knoesis.wright.edu/ssw/LocatedNearRelAKFLO>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.sub like '<http://knoesis.wright.edu/ssw/MeasureData_Dew%>' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L3.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#fahrenheit>' and L4.pred like '<http://www.w3.org/2006/time#inXSDDateTime>' and L4.obj like '"2004-08-09T10:40:00-06:00^^http://www.w3.org/2001/XMLSchema#dateTime"' and L5.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and L5.sub like '<http://knoesis.wright.edu/ssw/Observation_Dew%>' and L6.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L6.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#_DewPoint>' and L2.sub=L3.sub and L4.sub=L5.obj and L5.sub=L6.sub;

Q15 select L1.sub,L3.sub,L3.obj,L4.obj from LODTriples L1, LODTriples L2, LODTriples L3, LODTriples L4, LODTriples L5, LODTriples L6 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocation>' and L1.sub like '<http://knoesis.wright.edu/ssw/LocatedNearRelAKFLO>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.sub like '<http://knoesis.wright.edu/ssw/MeasureData_WindDirection%>' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L3.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#degrees>' and L4.pred like '<http://www.w3.org/2006/time#inXSDDateTime>' and L4.obj like '"2004-08-09T10:40:00-06:00^^http://www.w3.org/2001/XMLSchema#dateTime"' and L5.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and L5.sub like '<http://knoesis.wright.edu/ssw/Observation_WindDirection%>' and L6.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L6.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#_WindDirection>' and L2.sub=L3.sub and L4.sub=L5.obj and L5.sub=L6.sub;

Q16 select L1.sub,L3.sub,L3.obj,L4.obj from LODTriples L1, LODTriples L2, LODTriples L3, LODTriples L4, LODTriples L5, LODTriples L6 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocation>' and L1.sub like '<http://knoesis.wright.edu/ssw/LocatedNearRelAKFLO>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.sub like '<http://knoesis.wright.edu/ssw/MeasureData_Pressure%>' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L3.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#inches>' and L4.pred like '<http://www.w3.org/2006/time#inXSDDateTime>' and L4.obj like '"2004-08-09T10:40:00-06:00^^http://www.w3.org/2001/XMLSchema#dateTime"' and L5.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and L5.sub like '<http://knoesis.wright.edu/ssw/Observation_Pressure%>' and L6.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L6.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#_Pressure>' and L2.sub=L3.sub and L4.sub=L5.obj and L5.sub=L6.sub;

CHAPTER C

Appendix C

List of Publications

1. Book Chapters

- a. M. Patel, N. Yadav, and M. Bhise, "Workload aware Cost-based Partial loading of Raw data for Limited Storage Resources," *Futuristic Trends in Networks and Computing Technologies, Lecture Notes in Electrical Engineering* 936, (Chapter 74), Springer Nature, 2022, doi: 10.1007/978-981-19-5037-7_74.

2. Journal Papers

- a. M. Patel and M. Bhise, "RAW-HF: Resource Availability & Workload aware Hybrid Framework for Raw Data Query Processing", 2023. [In preparation]
- b. M. Patel and M. Bhise, "Resource Monitoring Framework for Big Raw Data Processing," *International Journal of Big Data Intelligence, Inderscience*, 2023, doi: 10.1504/IJBDI.2023.10053408.

3. Peer-Reviewed Conference Proceedings

- a. M. Patel and M. Bhise, "MUAR: Maximizing Utilization of Available Resources for Query Processing", in *3rd International Workshop on Advanced Data Systems Management (MegaData) at the 23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2023.

- b. M. Patel and M. Bhise, "Query Complexity Based Optimal Processing of Raw Data," 10th Region 10 Humanitarian Technology Conference (R10-HTC), IEEE, 2022, doi: 10.1109/R10-HTC54060.2022.9929945.
- c. M. Patel and M. Bhise, "Raw Data Processing Framework for IoT," 11th International Conference on Communication Systems & Networks (COMSNETS), IEEE, 2019, doi: 10.1109/COMSNETS.2019.8711408.
- d. U. Vyas, P. Panchal, M. Patel, and M. Bhise, "STSDB: Spatio-Temporal Sensor Database for Smart City Query Processing," In Proceedings of the 20th International Conference on Distributed Computing and Networking (ICDCN), ACM, New York, NY, USA, 2019, doi 10.1145/3288599.3296015.