KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY

GROUP T LEUVEN CAMPUS

Knowledge graph partitioning for efficient data querying

Eltaj AMIRLI

Promoter: Prof. Dr. Anastasia Dimou

Supervisor: Ioannis Dasoulas

Master's Thesis submitted to obtain the degree of Master of Science in Electronics and ICT programme Engineering Technology

Academic year 2024 - 2025

Acknowledgment

The resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government.

Abstract

Knowledge graphs are powerful tools for representing entities and their relationships; however, as they grow in size and complexity, querying them becomes increasingly challenging, negatively impacting their reliability and usability. Knowledge graph partitioning is an optimization technique that tackles this problem by reducing complexity and query execution time. Although there are graph partitioning techniques derived from established heuristics and approaches, a universal solution does not exist for all knowledge graphs for the best optimization, which makes the partitioning problem dependent on the characteristics and structure of the knowledge graph. We explore the performance of various graph partitioning techniques, developed through an exploratory approach guided by the structural properties and intended query workloads of MLSea-KG, a large-scale real-world knowledge graph. We compare these techniques against one another and identify which methods perform best for specific query types. Our results show that query performance varies significantly depending on the type of query and the applied optimization technique; specific methods consistently yield lower execution times than others. Additionally, the experiments reveal noticeable differences in hardware usage across setups, with some techniques achieving better CPU and memory efficiency.

Keywords: knowledge graph, graph partitioning problem, Virtuoso, SPARQL queries, efficient data querying

Contents

1 Introduction **2**

1.1 Problem Statement . . . 3

2 State of the art **6**

2.1 Terms and Definitions . . . 6

2.2 Related Works . . . 8

2.2.1 Resource Description Framework (RDF) engines . . . 8

2.2.2 General Graph Partitioning Techniques . . . 12

2.2.3 Evaluation Setup and Metrics . . . 18

2.2.4 Analysis of MLSea-KG . . . 20

2.2.5 Virtuoso RDF Engine . . . 20

3 Methodology **22**

3.1 Investigated Methods . . . 22

3.1.1 Named Graphs . . . 22

3.1.2 Indexing Strategies . . . 23

3.1.3 Query Planning with Pragmas . . . 24

3.1.4 Source-Based Data Isolation (Simulated Distribution) . . . 24

3.2 Query Generation . . . 24

3.2.1 Workflow Design . . . 25

3.2.2 Ontology Guided Query Design . . . 25

3.2.3 Grouping and Purpose of the Queries . . . 28

3.3 Experiments . . . 28

3.3.1 Experimental Setup . . . 28

3.3.2 Implementation of Experiments . . . 31

4 Results and Discussion **46**

4.1 Results and Discussion 46
4.1.1 Experiment 1: Baseline Performance on High-Performance Computing (HPC) and Google Compute Engine (GCE) 49
4.1.2 Experiment 2: Simulated Distribution by Source-Based Instance Separation 51
4.1.3 Experiment 3: Source-Based Named Graph Partitioning 51
4.1.4 Experiment 4: Predicate-Based Named Graph Partitioning 52
4.1.5 Experiment 5: Fine-Grained Partitioning of rdf:type Predicate 54
4.1.6 Experiment 6: Semantic Grouping of Predicates into Logical Named Graphs 55
4.1.7 Experiment 7: Custom Bitmap Indexing for Source-Specific Datasets 56
4.1.8 Experiment 8: Enforcing Explicit Join Order in SPARQL Queries . . 58
4.1.9 Remarks 63

5 Conclusion and Future Work 66
5.1 Conclusion 66
5.2 Limitations 66
5.3 Future Work 67

List of Figures

1.1 Example of the bi-partitioning of a graph 5

2.1 Types of Partitioning used in RDF Engines[7]. 9
2.2 Query examples 10
2.3 The predicate co-occurrences table and corresponding weighted graph 10
2.4 Graph representation and partitioning[7]. 11
2.5 Graph partitioning methods, approaches and algorithms 14
2.6 The multilevel approach to GP. The left figure shows a two-level contraction-based scheme. The right figure shows different chains of coarsening-uncoarsening in the multilevel frameworks.[5] 18

4.1 CPU and memory usage measurements of Experiment 1 (HPC) 50
4.2 CPU and memory usage measurements of Experiment 1 (GCE) 50
4.3 CPU and memory usage measurements of Experiment 2 52
4.4 CPU and memory usage measurements of Experiment 3 53
4.5 CPU and memory usage measurements of Experiment 4 54
4.6 CPU and memory usage measurements of Experiment 5 56
4.7 CPU and memory usage measurements of Experiment 6 57
4.8 CPU and memory usage measurements of Experiment 7 58
4.9 CPU and memory usage measurements of Experiment 8 59

List of Tables

3.1 Query set statistics 28
3.2 Predicates by Triple Count 36
3.4 Grouped Predicates 40
3.3 Distribution of `rdf:type` Predicate 42

4.1 Query Set Sorted by Source and Category 47
4.2 Execution Time Statistics for Experiment 1 based on 10 Runs 49
4.3 Execution Time Statistics for Experiment 2 based on 10 Runs 51
4.4 Execution Time Statistics for Experiment 3 based on 10 Runs 52
4.5 Execution Time Statistics for Experiment 4 based on 7 Runs 54
4.6 Execution Time Statistics for Experiment 5 based on 10 Runs 55
4.7 Execution Time Statistics for Experiment 6 based on 10 Runs 56
4.8 Execution Time Statistics for Experiment 7 based on 10 Runs 57
4.9 Execution Time Statistics for Experiment 8 based on 10 Runs 59
4.10 Peak CPU and Memory Utilization Across Experiments 60
4.11 Execution Time Statistics for Experiment 3 based on 9 Runs, excluding the first run 61
4.12 CPU Usage Maxima (Peaks) 62

Acronyms

AQET average query execution time. 49, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62

CQ Complex Queries. 46, 48, 49, 51, 52, 54, 55, 56, 57, 59, 61

GA Genetic Algorithm. 15

GCE Google Compute Engine. vi, vii, 29, 31, 49, 50, 60, 62

GDBS Graph Database Systems. 8

GPP Graph Partitioning Problem. 2, 6, 15

GPS Graph Processing Systems. 8

HPC High-Performance Computing. vi, vii, 28, 29, 31, 32, 49, 50, 60, 62

IQ Intermediate Queries. 46, 47, 48, 49, 51, 52, 54, 55, 56, 57, 59, 61

KG Knowledge Graph. 2, 3, 20

MGP Multi-Level Graph Partitioning. 17, 18

ML Machine Learning. 2, 3, 20

MS Multisource. 46, 47, 48, 49, 52, 54, 55, 56, 59, 61

RDF Resource Description Framework. v, vii, 7, 8, 9, 10, 12, 19, 20, 32

SQ Simple Queries. 46, 47, 48, 49, 51, 52, 53, 54, 55, 56, 57, 59, 61

SSK Single Source Kaggle. 46, 47, 49, 51, 52, 54, 55, 56, 57, 59, 60, 61

SSO Single Source OpenML. 46, 47, 48, 49, 51, 52, 54, 55, 56, 59, 60, 61

SSP Single Source PwC. 46, 47, 48, 49, 51, 52, 54, 55, 56, 57, 59, 60, 61

TT Triple Table. 8

VSC Vlaams Supercomputer Centrum. 28, 29

Chapter 1

Introduction

The term "Knowledge Graph" emerged as early as 1972 [1]. Still, the modern embodiment of the term began in 2012 with Google's announcement of "Google Knowledge Graph"[1], which was then followed by other Knowledge Graphs (KGs) of well-known companies such as Facebook, Microsoft, etc. Moreover, especially since then, the topic of KGs has been the focus of academic study. The term has been subject to varied interpretations across research and industry, often lacking a unified definition. Recognizing this ambiguity, Aidan Hogan and his colleagues proposed a technical definition in their paper aiming to establish a common understanding within the community; they defined it as *"A graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent potentially different relations between these entities"* [2]. KGs have emerged as a compelling abstraction for organizing the world's structured knowledge over the Internet and as a way to integrate information extracted from multiple data sources. They have started playing a central role in Machine Learning (ML) as a method to incorporate world knowledge as a target representation for extracted knowledge and to explain what is learned. The KGs enable contextual understanding and intelligent decision-making across other industries as well; their applications appear in search engines, recommendation systems, cybersecurity, etc.

However, as KGs continue to expand in both size and complexity, efficiently managing and querying these massive graphs becomes increasingly complex. This leads to higher hardware resource demands, longer response times, and increased risks of incorrect or incomplete query results. Such inefficiencies negatively impact the reliability and usability of knowledge-graph-based systems, ultimately reducing user satisfaction and limiting their practical adoption in real-world scenarios. To address these challenges, partitioning techniques and algorithms are employed to reduce computational complexity and improve performance.

The Graph Partitioning Problem (GPP) is an NP-hard combinatorial optimization problem requiring advanced techniques to handle its inherent complexity[3]. While numerous methods and algorithms have been explored to optimize KG performance, a universal solution remains

[1] https://en.wikipedia.org/wiki/Google_Knowledge_Graph

elusive because each KG has distinct characteristics, making optimization inherently dataset-specific. In other words, there is no "one-size-fits-all" strategy for optimizing KG query performance.

This highlights a critical challenge: How can query performance and resource utilization be effectively optimized for a specific, domain-focused KG? This study considers MLSea-KG[4], which is a declaratively constructed KG containing over 1.44 billion RDF triples that catalog ML experiments. Thus, the primary objective of this thesis is to evaluate various partitioning and indexing techniques to optimize query performance on the MLSea-KG dataset. Specifically, the research focuses on reducing query response times, improving hardware utilization, and ensuring the correctness of query results using the Virtuoso RDF store. This thesis aims to answer this research question: **"How do different optimization strategies affect the MLSea-KG query performance and resource efficiency?"**.

This thesis makes practical contributions by thoroughly benchmarking different optimization techniques against a realistic and structured KG(MLSea-KG). The findings provide clear, actionable insights into which optimization strategies are effective in practice, highlighting not only improvements but also limitations and further recommendations for different scenarios. By carefully documenting these results, this research contributes valuable guidelines for KG designers and engineers aiming to improve performance and resource utilization in similar real-world scenarios. This thesis is structured as follows: Chapter 2 reviews related concepts and existing research, covering RDF engines, graph partitioning techniques, and relevant evaluation metrics. Chapter 3 describes the experimental methodology, detailing the partitioning and indexing methods investigated, query design, and experimental setups. Chapter 4 presents and analyzes results obtained from the experiments, discussing their implications and observations. Finally, Chapter 5 concludes by summarizing the findings, identifying limitations, and suggesting directions for future research.

1.1 Problem Statement

To understand the graph partitioning problem, let's define the graph mathematically. A given an undirected graph G can be defined as $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ represent group of vertices and $E = \{e_1, \ldots, e_m\}$ represent group of edges, $E \subseteq V \times V$. Then the balanced k-way GP problem is defined as G is partitioned into k partition sets $V = \{V_1, V_2, \ldots, V_k\}$ such that [3]:

$$\begin{cases} \bigcup_{i=1}^{k} V_i = V, \\ V_i \cap V_j = \emptyset, \quad \forall i \neq j. \end{cases}$$

The k-way partitioning of the graph requires dividing a big graph into k number of distinct subgraphs, while still satisfying two main constraints: **maximizing load balance** and **minimizing cuts**. Maximizing load balance demands that all subgraphs or blocks have about

equal weights. More precisely, it requires that:

$$\forall i \in \{1, \ldots, k\} : |V_i| \leq L_{\max} := (1 + \epsilon) \left\lceil \frac{|V|}{k} \right\rceil .$$

Here $|V|$ is used for denoting the size of V. A block V_i is overloaded if $|Vi| > L_{\max}$. Epsilon (ϵ) is an imbalance parameter where $\epsilon \in \mathbb{R}_{\geq 0}$. In the case of $\epsilon = 0$, we can say the graph is perfectly balanced[5].

The second and perhaps more crucial objective of GP is finding a k-partition that minimizes the cost of all external edges, which in practice translates to communication costs between blocks as well. The *graph cut* between two partition sets, can be represented as:

$$Q = \omega(E_{ij}) = \sum_{i \in V_1, j \in V_2} e_{ij}$$

Here, e_{ij} is the weight of the edges. It is a representative value for cost, length, capacity or other metrics associated with traversing that edge. It is an important aspect of weighted graphs, where the edges have varying values that can influence the properties and behavior of the graph. However, for an unweighted graph, we can consider edge weight to be equal to one. Additionally, depending on the algorithm or method used, the value (Q) can be kept as is or it can be converted to the ratio, normalized, or quotient cut (sometimes referred to as the conductance of a cut in the literature as well). In most cases, quotient or normalized cut is opted. Normalized cut for bi-partitioning (k=2) is formulated as followed [2]:

$$Q = \frac{\omega(E_{ij})}{\text{V}_{\text{ol}}(V_1)} + \frac{\omega(E_{ij})}{\text{V}_{\text{ol}}(V_2)}.$$

Where V_{ol} corresponds to the number of edges that starts from nodes within the cut, for example in Figure 1.1, left most cut that is indicated has volume equal to 13.

The optimization problem of GP is then given by:

$$\begin{cases} max|V_i| \leq (1 + \epsilon) \left\lceil \frac{|V|}{k} \right\rceil , \\ \min Q \end{cases}$$

[2] https://www.youtube.com/watch?v=7DuImjU8eyw&t=17s&ab_channel=LeonidZhukov

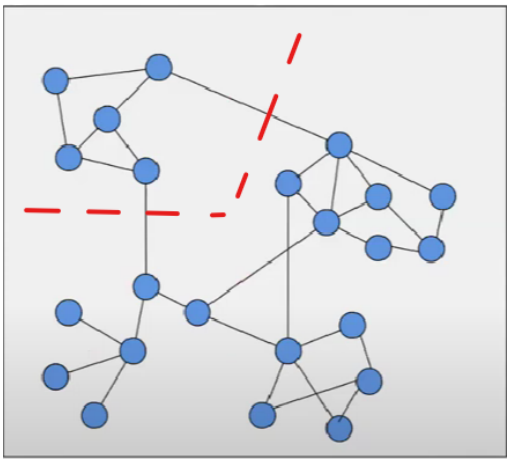


Figure 1.1: Example of the bi-partitioning of a graph

Chapter 2

State of the art

In this chapter, key terms and definitions relevant to GPP are first introduced, followed by a discussion of state-of-the-art methods commonly used in this area of research. Finally, the MLSea-KG and Virtuoso RDF engine will be introduced.

2.1 Terms and Definitions

Starting with terms, this section explains the terminology commonly used for GP.

1. Graph Structures and Properties

   (a) **Directed vs Undirected Graph:** A directed graph, or digraph, is a graph in which each edge has a direction. This means that the edges go from one node to another in a specific order, indicating a one-way relationship. An undirected graph is one in which edges do not have a direction. Each edge connects two nodes without implying any specific order or direction.

   (b) **Sparse Graph:** This is a type of graph in which the number of edges is relatively small compared to the number of nodes (vertices). In other words, a sparse graph has few connections relative to the possible maximum number of edges that could connect all pairs of nodes.

   (c) **Skewed Graph**: This refers to an uneven or imbalanced distribution of nodes (entities or resources) and edges (relationships or properties) across the graph. This means that some nodes have significantly more connections (or relationships) than others, leading to asymmetry in the graph's structure.

   (d) **Connected Graph:** When we say that a graph G = (V, E) is connected, it means that there is a path between any pair of vertices in the graph. In other words, in a connected graph, it is possible to start at any vertex and reach any other vertex by following a series of edges.

   (e) **Edge cut:** An edge that connects nodes in different partitions.

2. Graph Representation

   (a) **Ontologies:** These are formal representations of knowledge within a particular domain. They define concepts, categories, and relationships between them. For instance, an ontology for animals might define the concept of "dog" and its relationship to "mammals", "pets", or "breeds". Ontologies help systems reason and draw conclusions from the data they analyze.

3. Querying and Data Representation

   (a) **RDF:** This is a general framework for representing interconnected data on the web. RDF statements are used to describe and exchange metadata, enabling standardized data exchange based on relationships. RDF is used to integrate data from multiple sources.

   (b) **SPARQL:** It is a query language used to retrieve and manipulate data stored in RDF format.

   (c) **Triplestore:** It is a type of database specifically designed to store, manage, and query data that is represented in the form of RDF triples. In the context of the Semantic Web, triplestores are used to store data as triples, which consist of three parts: subject, predicate, and object.

   (d) **Blank Nodes (also known as bnodes or anonymous nodes):** In RDF, they are used to represent resources that do not have a URI or literal value. They are used when you want to describe a resource, but you either don't know its URI or don't want to assign a URI to it. They are "anonymous" resources, meaning they cannot be directly referenced from outside the RDF graph where they are defined. They are only meaningful locally, in the context of one graph.

   (e) **Cardinality:** In a database context, cardinality refers to the number of unique values in a relational table column relative to the total number of rows in the table. The cardinality of a column is assessed and stored in system tables for optimizer use when the database administrator runs statistics[1].

4. Partitioning Approaches

   (a) **Heterogeneous vs Homogenous Clusters:** When computing nodes are the same in terms of CPU power, memory, and bandwidth, clusters are said to be homogeneous; otherwise, they are heterogeneous.

   (b) **Single machine vs Distributed GP:** As the name suggests, single machine GP uses a single machine to partition. In the distributed approach, the graph is already distributed in a distributed memory application. However, to preserve scalability, not every processor stores the whole graph. As a result, distributed-memory partitioning algorithms frequently rely on their partitioning choices on

[1] https://www.actian.com/what-is-cardinality

partial views of local graph data rather than having an overall view of the entire graph [3].

2.2 Related Works

2.2.1 RDF engines

Graph Computing Systems are developed to deal with graph-based analytics. The design of this system relies on Graph Processing Systems (GPS) and Graph Database Systems (GDBS). GDBS are designed to efficiently store, process, and analyze large-scale graphs by applying core principles of database management systems, such as persistent data storage, data consistency, integrity, and logical or physical data independence.

GDBS utilize various data models to manage graph elements, including vertices, edges, and their associated properties. Unlike traditional relational databases, where connections between data are stored in separate tables, requiring computationally expensive join operations, GDBS treat edges (connections) as a core component of the model, along with vertices. This design enables a more efficient handling of graph relationships [3]. However, in this section, the focus will be placed on RDF engines. An RDF graph can be viewed as a specific type of graph where nodes represent subjects and objects, and edges represent predicates. In this sense, RDF engines can be seen as a specialized subset of GDBS focused on semantic relationships. However, they have common features, such as the fact that both are used for managing data and relationships. Even in modern systems, they often integrate features from both paradigms to support diverse graph use cases.

In this section, different categories of state-of-the-art graph partitioning techniques used in RDF engines will be examined. RDF graph partitioning techniques that are used in RDF engines can be divided into two major categories:

- **Horizontal Partitioning:** In RDF, each row of Triple Table (TT) represents a triple. Horizontal partitioning is the distribution of a complete dataset in a row-wise manner. To explain it better, let `T` be the set of all RDF triples in a dataset and `k` the required number of partitions. The technique assigns the first $\frac{|T|}{k}$ triples in the first partition, the following $\frac{|T|}{k}$ triples in the second partition, and so on[6].
- **Vertical Partitioning:** Contrary to horizontal partitioning, vertical partitioning distributes data column-wise. Instead of storing complete triples, it typically stores only two components, subject and object, in separate two-column tables. Each table corresponds to a unique predicate in the RDF dataset, with the table name reflecting the predicate itself. As a result, the number of tables in vertical partitioning equals the number of distinct predicates present in the dataset.

Figure 2.1 shows categories of the partitioning techniques found in RDF engines [7], note that

horizontal partitioning can be further divided into more categories such as workload-based, hash-based, and graph-based partitioning techniques.

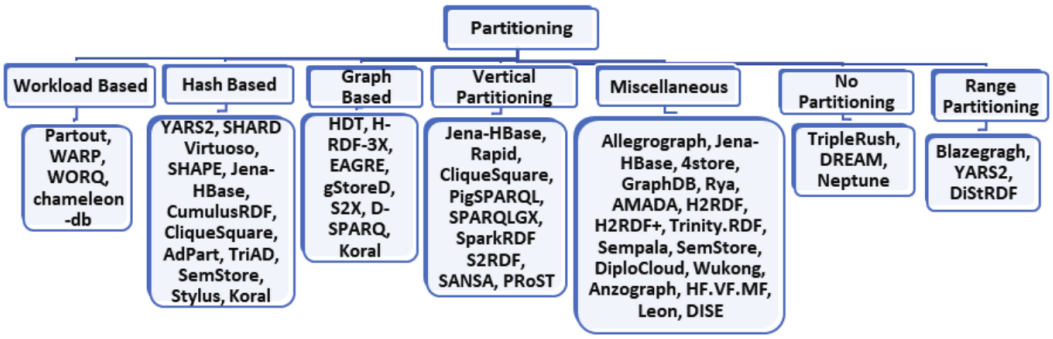


Figure 2.1: Types of Partitioning used in RDF Engines[7].

2.2.1.1 Workload Based Partitioning

This category of partitioning uses historical real-user query workloads to partition the graph. The main idea of this technique is to take data locality into account to minimize the inter-communication between partitions, thus potentially leading to better query runtimes. Ideally, the query workload contains real-world queries. However, real user queries might not be available; in that case, the query workload can be estimated from queries in applications accessing RDF data. Most state-of-the-art partitioning techniques do not use information concerning the likelihood of particular data portions being queried concurrently to answer user queries. Akhter et al. have done interesting work in this area [8]. They propose two RDF graph partitioning techniques:

1. Predicates co-occurrence-based partitioning using a greedy algorithm (PCG),
2. Predicates co-occurrence-based partitioning using extended Markov clustering (PCM).

Both of these techniques comprise three main steps: extract a list of predicate co-occurrences from a querying workload and model them as a weighted graph. Use this weighted graph as an input to generate clusters of predicates, and allocate the obtained clusters to partitions. The partitions are created according to the clusters such that all triples pertaining to predicates in a given cluster are distributed into the same partition. For example, consider a workload of eight queries, as shown in Figure 2.2.

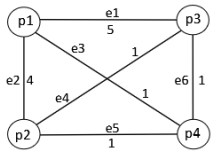
First off, based on the workload (Figure 2.2), the co-occurrence of a set of predicates is counted and stored in a tuple in this way $\langle p_1, p_2, c \rangle$, where p1 and p2 is the first and second predicate, respectively, and c is the count of co-occurrence where both predicates exist inside a query. Then, a weighted graph is constructed based on these tuples. Figure (2.3b) demonstrates an example of this based on the set of tuples shown in Figure(2.3a).

```
SELECT * WHERE          SELECT * WHERE          SELECT * WHERE          SELECT * WHERE          SELECT * WHERE          SELECT * WHERE          SELECT * WHERE          SELECT * WHERE
{                       {                       {                       {                       {                       {                       {                       {
 ?S :P1 ?O1.             ?S :P1 ?O.              ?S :P1 ?O1.             ?S :P1 ?O.              ?S1 :P1 ?O.             ?O :P1 ?S.              ?S1 :P1 ?O.             ?S :P1 ?O.
 ?S :P2 ?O2              ?O :P2 ?O2              ?S :P3 ?O3              ?O :P3 ?O3              ?S3 :P3 ?O              ?S :P3 ?S3              ?S2 :P2 ?O.             ?S :P2 ?O.
}                       }                       }                       }                       }                       }                       ?S :P3 ?O.
                                                                                                                                                            ?S :P4 ?O
                                                                                                                                                           }
```

Figure 2.2: Query examples

P1	P2	Co-occurrences
p1	p2	4
p1	p3	5
p1	p4	1
p2	p3	1
p2	p4	1
p3	p4	1

(a) Predicate Co-occurrences



(b) Weighted graph of the predicate co- occurrences

Figure 2.3: The predicate co-occurrences table and corresponding weighted graph

Then the next step is to apply graph clustering. As mentioned before, they apply **PCM and PSG clustering**. For **PCM**, first, a transition matrix T is built from the graph, with rows and columns representing predicates. Since our weighted graph is shown in Figure 2.3 has four nodes, a 4×4 (one row and column for each predicate vertex) matrix will be created. T is then normalized and undergoes standard iterative expansion and inflation. This process continues until the residual value (change between iterations) falls below a specified threshold, stabilizing the clusters. The last step is to interpret the resulting transition matrix to discover n clusters.

For **PCG**, firstly, the edges of the predicate co-occurrence graph are sorted in ascending order of weights. Then clustering formation starts with the smallest edge, predicates are assigned to the same cluster if the combined size of triples in the cluster stays below a predefined threshold. If the threshold is exceeded, a new cluster is created.

Please note that there may exist many predicates in the RDF dataset that are not used in the query workload. In that case, all unused predicates are assigned to a single partition (in both clustering techniques). Once the clusters are formed, RDF triples from the dataset are assigned to partitions based on their predicates. This ensures that triples with commonly co-occurring predicates are stored together, reducing inter-partition communication and improving query performance.

2.2.1.2 Hash Based Partitioning

Hash Partitioning is random; it is careless about the graph structure. Thus, this results in a simple, fast technique but sometimes lacks load balance. There are three subcategories in this partitioning [6][7]:

1. **Subject-hashed:** The partitioning is based on the subject of the RDF triple. A hash function is applied to the subject of each triple, and the result determines the partition

where the triple is stored. In Figure 2.4, a clear imbalance can be seen.

2. **Predicate-hashed:** Similar to subject-hashed, but in this case, partitioning is based on the predicate. This causes all the triples with the same predicate to be assigned to one partition.

3. **Hierarchy-hashed:** This partitioning technique is based on the assumption that IRIs (Internationalized Resource Identifiers) or URIs (Uniform Resource Identifiers) have path hierarchy and those with the same hierarchy prefix are often queried together. Those with the same hierarchy are assigned to the same partition.

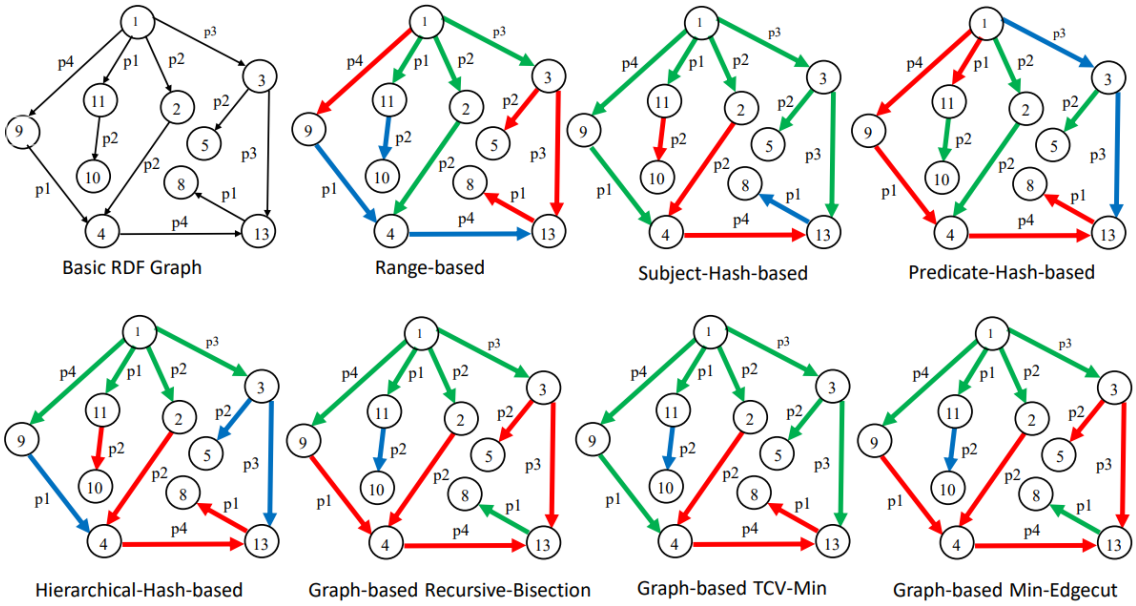


Figure 2.4: Graph representation and partitioning[7].

2.2.1.3 Graph Based Partitioning

Graph-based partitioning applies clustering techniques directly on the graph structure to divide it into pairwise disjoint subgraphs. Three main techniques fall under this category[6][7]:

1. **Recursive-Bisection Partitioning:** This method splits the graph into two parts and recursively applies the same strategy to each subgraph until the desired number of partitions is achieved.

2. **TCV-Min Partitioning:** This technique partitions the graph by minimizing the total communication volume (TCV) between connected nodes, aiming to reduce the data exchange cost during query processing.

3. **Min-Edgecut Partitioning:** While similar to TCV-Min, this method specifically aims to distribute nodes by minimizing the number of edges connected to them.

Listing 2.1 presents an example set of RDF triples, while Figure 2.4 illustrates various ways to partition a graph constructed from those triples.

Listing 2.1: An example of set of RDF tripels

```
@prefix hierarchy1: <http://first/r/> .
@prefix hierarchy2: <http://second/r/> .
@prefix hierarchy3: <http://third/r/> .
@prefix schema: <http://schema/> .

hierarchy1:s1      schema:p1      hierarchy2:s11    .   #Triple 1
hierarchy1:s1      schema:p2      hierarchy2:s2     .   #Triple 2
hierarchy2:s2      schema:p2      hierarchy2:s4     .   #Triple 3
hierarchy1:s1      schema:p3      hierarchy3:s3     .   #Triple 4
hierarchy3:s3      schema:p2      hierarchy1:s5     .   #Triple 5
hierarchy3:s3      schema:p3      hierarchy2:s13    .   #Triple 6
hierarchy2:s13     schema:p1      hierarchy2:s8     .   #Triple 7
hierarchy1:s1      schema:p4      hierarchy3:s9     .   #Triple 8
hierarchy3:s9      schema:p1      hierarchy2:s4     .   #Triple 9
hierarchy2:s4      schema:p4      hierarchy2:s13    .   #Triple 10
hierarchy2:s11     schema:p2      hierarchy1:s10    .   #Triple 11
```

2.2.2 General Graph Partitioning Techniques

In this section, some of the well-known established graph partitioning techniques will be explored, although not explicitly designed for RDF, they can be applied to RDF graphs to improve performance. Since graph partitioning is a complex problem, practical solutions are based on heuristics. At the basic level, they can be categorized into three:

1. **Vertex Partitioning (VP):** Vertex partitioning (also called edge-cut) divides the big graph into many subgraphs by assigning vertices to the different partition sets while minimizing edge cuts concerning the load balance constraint.

2. **Edge Partitioning (EP)**: Edge partitioning (also named vertex-cut)divides a big graph into many subgraphs by assigning edges to the different partition sets while considering a maximum load balance and minimum vertex cut.

3. **Hybrid Partitioning (HP):** Lower and higher-degree vertices are distinguished in this approach. Then, VC or EC methods are employed to achieve more effective partitions. HP combines VC and EC techniques, leveraging the internal structure of the graph for partitioning. Most real-world graphs follow a power-law distribution, where a small proportion of vertices have a high degree, while the majority have a low degree. HP classifies vertices into high-degree and low-degree categories. High-degree vertices have their edges evenly distributed across partitions (via vertex-cut) to balance the computational load. In contrast, all in-edges (or out-edges) of low-degree

vertices are assigned to the same partition (via edge-cut) to minimize inter-partition communication.

These methods can be further divided into four more classification [3] as depicted in Figure 2.5:

1. **Offline Approach:** The offline approach is a traditional graph partitioning method that utilizes the global information of a graph to assign edges or vertices to partitions. Before applying partitioning algorithms, the entire graph is loaded into memory. Numerous algorithms have been developed for this approach, both for single-machine and distributed systems. **Single-Machine Partitioning**: This method uses a single machine to execute the partitioning process. It achieves high partitioning accuracy, but struggles with large-scale graphs due to memory limitations, as the entire graph must fit into memory. The two main challenges of graph partitioning are quality and scalability. Achieving high-quality partitions, evaluated by total cuts and load balance, is challenging because graph partitioning is an NP-hard problem. The increasing size of real-world graphs necessitates scalable solutions, which the single-machine approach cannot fully support. To address scalability, **distributed memory graph partitioning** has been introduced. This method distributes the graph across multiple processors in a distributed memory environment. Since not every processor holds the entire graph, partitioning decisions are based on partial views of local data. Processors communicate with one another to minimize cuts and maintain load balance. However, this comes with a trade-off: the distributed approach enables large-scale partitioning and scalability, but at the cost of reduced partitioning quality compared to the single-machine approach.

2. **Online Approach:** The offline approach involves loading the entire graph into memory before partitioning begins. This allows the partitioner to quickly access the global structure of the graph, enabling it to solve optimization problems with high partitioning quality. However, this method is unsuitable for large-scale graph partitioning due to memory limitations. To address this limitation, the online approach, also referred to as the streaming approach, was developed to enable scalable partitioning. In this method, vertices and their edge sets are processed sequentially in a pipeline fashion. Partitioning decisions are made based on a partial view of the graph, and the system maintains a partitioned state to guide future allocations. This state is essential for assigning incoming edges to the appropriate partitions. Once an edge or vertex is assigned, it cannot be reassigned. Since the entire graph does not need to be retained in memory, the online approach significantly reduces memory overhead, making it possible to use lower-capacity workstations for partitioning large graphs and lowering associated costs. However, in the initial stages, the partitioner has insufficient partition state to make informed decisions, leading to suboptimal early allocations. Over time, as more state is accumulated, the partitioning quality improves, but it generally remains inferior to

the offline approach. Despite this, the online approach is well-suited for partitioning large-scale graphs. Additionally, the performance of the method can be influenced by the order in which graph data arrives, whether it is random, Depth First Search (DFS), or Breadth First Search (BFS). These arrival patterns have a notable impact on partitioning efficiency.

3. **OffStream Approach:** The OffStream partitioning approach combines elements of both offline and streaming methodologies to bridge the gap between pure in-memory and pure streaming algorithms. Its key innovation lies in handling graphs that are too large to fit entirely in memory by processing smaller portions of the graph in-memory while leveraging streaming techniques for the rest. The approach applies effective partitioning methods to both offline and stream components. An example of offStream (edge partitioning) approach is HEP [9]. HEP starts by dividing the edge set into subsets based on vertex degree. The average degree of the graph determines whether a vertex is classified as high-degree or low-degree. Edges connecting two high-degree vertices are partitioned in the stream, and edges with one of the low-degree vertices are partitioned in-memory partitioning.

4. **Dynamic Approach:** Some graphs are inherently dynamic, with their topology constantly evolving as vertices and edges are added or removed over time. These changes can degrade the partitioning quality, leading to imbalanced load distribution across partitions and increased communication overhead. To address this challenge, the dynamic partitioning approach was introduced, enabling the system to adapt to the evolving structure of the graph and maintain partitioning efficiency.

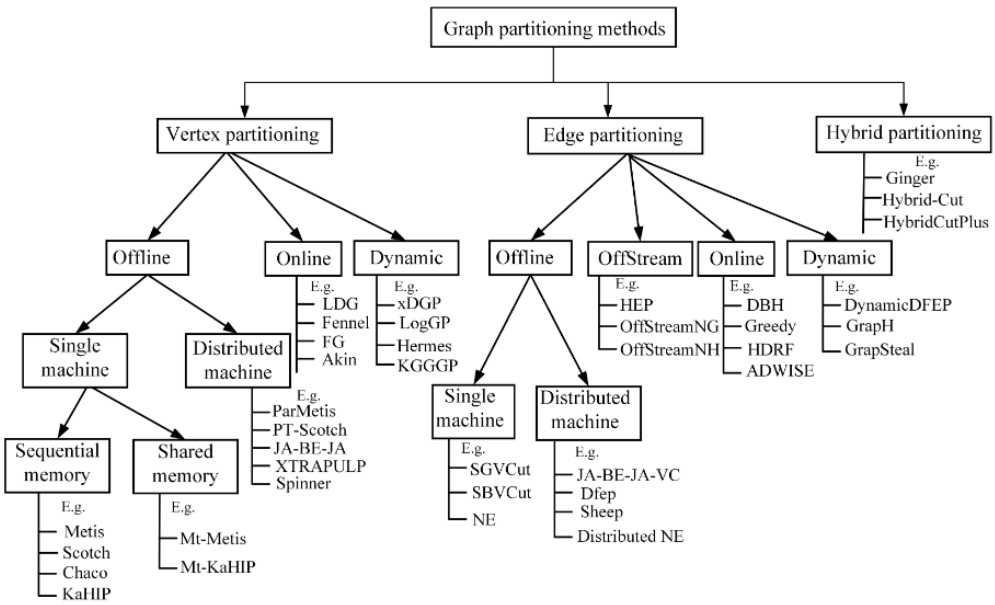


Figure 2.5: Graph partitioning methods, approaches and algorithms

The following sections present an overview of notable research and techniques in the field of graph partitioning (GP).

2.2.2.1 Evolutionary Computation

Evolutionary computation is one of the effective ways to solve the NP-hard GPPs. The usual flow of evolutionary computation starts with encoding. Kim et al. studied different heuristics for vertex ordering and an approach to partition vertex ordering into clusters [10] [11]. In graph partitioning, vertex ordering refers to the process of arranging the vertices (nodes) of a graph in a specific sequence, which can help improve the efficiency and effectiveness of partitioning algorithms by improving data locality (by arranging vertices so that connected nodes are close to each other) and by minimizing edge cuts(vertex ordering can be used to identify an arrangement that places closely connected nodes near each other in the order). You can think of encoding as the gene rearrangement for Genetic Algorithms (GAs). GA is kind of bio-inspired meta-heuristics. GAs are often used as an optimization step after initial partitioning. GA uses population of individuals $\{X_1, X_2, \ldots, X_n\}$ to represent candidate solutions of a problem, and each individual consists of a series of genes[12]. The GAP algorithm encodes an individual solution X_i (where $i = 1, 2, \ldots, n$) as:

$$X_i = x_{i1}, x_{i2}, \ldots, x_{ik}$$

Here, x_{ij} $(j = 1, 2, \ldots, k)$ denotes that the vertex $v_{x_{ij}} \in V^{cut}$ should be assigned to partition P_j. Consequently, each individual represents a specific partitioning outcome.

After population initialization of a random solution, GA applies operators in an iterative manner to generate new solutions. Common GA operators are *mutation, selection, and crossover*. When applied to graph partitioning, these operators help create and refine partitions, allowing the algorithm to explore a broader set of potential solutions and converge on an optimized partitioning.

The Mutation operator is applying small random changes to the randomly chosen solution (partition) to introduce variability. A mutation might involve moving a node from one partition to another or making a slight modification in the way partitions are structured. These small random adjustments help prevent the algorithm from becoming stuck in local optima, as they allow the exploration of new areas in the solution space.

The Crossover operator combines parts of parent solutions to produce offspring. Segments of two partitions can be merged to form a new partition. The idea is to combine the best parts of two good partitions; the offspring may inherit favourable characteristics that lead to a better partitioning solution.

The Selection Operator mimics the natural selection in evolution, it determines which solutions are chosen (with higher fitness than a threshold) for the next phase in the partitioning algorithm.

Building on the principles of GAs, **Memetic Algorithms** represent a more sophisticated and

complex approach to solving GPP. They combine the principles of GA with local search methods to find high-quality solutions. The term "memetic" comes from the concept of a meme (introduced by Richard Dawkins), which represents an idea or behavior that spreads and evolves. Similarly, a memetic algorithm uses *evolutionary principles (like in genetic algorithms)* but also incorporates *local refinement* to improve each solution individually. The memetic algorithm applies local search to each new solution that operators produce to refine the solution. The main goal is to push each solution toward a local optimum, and new, refined solutions are added back into the population. In simple terms, the whole process is replacement and iteration.

We can look at the research conducted by Menghan Li et al., which focuses on GA-based large-scale graph partitioning in heterogeneous clusters [12]. It is an interesting article because most works on GPP do not address the case of heterogeneous clusters. In their experiment, the main concern was to minimize runtime. The runtime consists of the processing time of each computing node and the communication time within the cluster. Once they define their optimization problem mathematically, which in their case was minimizing the processing time of the cluster, they move on to an initial balanced partition to assign a balanced workload to each computing node in a heterogeneous cluster. The next step is to optimize the partition using genetic algorithm. This step is the actual algorithm that will minimize the runtime. After they iteratively apply the operators as I mentioned above, GA achieves the best solution by comparing solutions to each other, updating the best individual in each iteration. For their analysis they compare their partitioning method to Hash partitioning on the performance of:

- Balanced factor of number of vertices (defined as $\max_{i=1,2,\ldots,k}\left\{\text{abs}\left(|V_i| - \frac{|V|}{k}\right)\right\}$) and number of edges (defined as $\max_{i=1,2,\ldots,k}\left\{\text{abs}\left(|E_i| - \frac{|E|}{k}\right)\right\}$).
- Number of cut edges, noted as $|E^{\text{cut}}|$
- Total processing time of SSSP (Dijkstra's Single Source Shortest Path algorithm to find the shortest paths from a single source vertex to all other vertices in a weighted graph).
- Proportion of computational time to total time.

In terms of balanced factors, Hash shows better results, since the hash algorithm assigns vertices evenly across partitions without considering the computational capabilities of the nodes. In contrast, the GAP algorithm prioritizes reducing the total execution time over achieving a balanced distribution of vertices and edges. As a result, GAP may produce unbalanced partitions, often assigning the most vertices to a single node. GAP algorithm can partition the graph according to the computing power of computing nodes: the stronger the computing nodes are, the more vertices and edges are allocated. In particular, the number of vertices of each computing node is proportional to its computing power.

The Hash method tends to produce more cut edges because it does not account for minimizing edge cuts during partitioning. This results in longer communication times for Hash-based SSSP compared to GAP-based SSSP. Additionally, as the number of vertices and edges in a graph increases, the computation time proportionally grows, further influencing the total processing time.

Another study related to this domain is performed by Seo et al. [10]. In their research, they present a new GA based on a new encoding type they created, which is called **edge-set encoding**. Compared to traditional edge-based encoding, where a gene corresponds to an edge, in edge-set encoding based on a spanning tree, only feasible partitions are represented. As their target problem, they chose to go with the MAX CUT problem, which is simple to conceptualize but very complex to solve. The objective of MAX CUT is to partition the set of vertices of a graph into two subsets, such that the sum of the weights of the edges having one endpoint in each subsets is maximum. MAX CUT is an essential combinatorial problem and has applications in many fields, including VLSI circuit design and statistical physics. To demonstrate the effectiveness of their proposed approach, they did experiments on three different edge-set encodings that are derived from different spanning trees. They found out that changing the encoding method can be beneficial for a specific type of graphs. As in their case, edge-set encoding based on a spanning tree is advantageous for sparse graphs, while vertex-based encoding works better for dense graphs. And luckily, there are many examples of sparse graphs used in real-world applications (such as social networks, road networks, and so on), which further underscores the practical significance and relevance of this research.

2.2.2.2 Multi-Level Graph Partitioning (MGP)

The multilevel graph partitioning (MGP) approach is a popular technique for graph partitioning. Its core idea is to reduce the size of the graph during the **coarsening phase (first phase)** by creating a hierarchy of progressively smaller graphs. This reduction continues until the graph is small enough to allow the application of sophisticated but slower algorithms in the **initial partitioning phase (second phase)**. The hierarchy is constructed such that cuts in the coarsened graphs correspond to cuts in the original, finer graph [5]. There are several methods available for generating these graph hierarchies. Subsequently, in the **refinement phase (also known as the uncoarsening phase)(third phase)**, the partition quality is progressively enhanced at each level of the generated hierarchy through a local improvement algorithm.[13]. An example of a coarsening-uncoarsening scheme is outlined (with phases mentioned as well) in Figure 4.9.

In their paper "Recent advances in graph partitioning", Buluç et al. state that "Clearly the most successful heuristic for partitioning large graphs is the multi-level graph partitioning approach"[5]. In fact, most general-purpose methods that obtain good partitions for large real-world graphs use the MGP method. Most common ones are Metis, Mt-Metis, ParMetis, Scotch and so on, also shown in the Figure 2.5.

There are quite logical reasons why MGP works so well. One key advantage is the flexibility it offers, allowing the use of different methods or strategies for each phase of the process. And according to Buluç et al. there are at least three more reasons [5]:

- At the coarse levels, substantial computation can be performed per node with minimal impact on the overall execution time.
- Moving a single node at a coarse level results in a significant change in the final solution, making it easier to identify improvements that may be challenging to find at the finest level.
- Local improvements at the fine level are expected to run efficiently since they begin with a strong initial solution inherited from the coarse level.

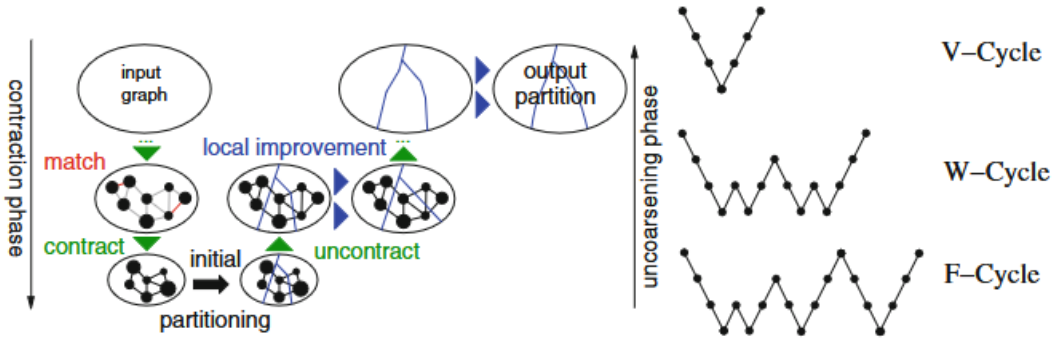


Figure 2.6: The multilevel approach to GP. The left figure shows a two-level contraction-based scheme. The right figure shows different chains of coarsening-uncoarsening in the multilevel frameworks.[5]

As mentioned above, MLP provides the flexibility to adjust each phase. For example, in research conducted by Akhremtsev et al., a high-quality shared-memory MGP algorithm is presented, which parallelizes all three phases mentioned above using C++17 multi-threading [13]. Their approach includes a scalable parallel label propagation algorithm that is able to quickly shrink large, complex networks during the coarsening phase. The label propagation algorithm is basically a simple and efficient method for detecting communities or clusters in large graphs. It is a heuristic algorithm that assigns labels to nodes and propagates these labels throughout the graph, resulting in clusters where each node belongs to a group labelled with the majority of its neighbours. And as an addition to that, they utilize cache-aware hash-tables to limit memory consumption and improve locality.

2.2.3 Evaluation Setup and Metrics

When evaluating GP techniques, several key factors must be considered:

- **Datasets:** The datasets can be categorized into two types: real-world RDF datasets and synthetic datasets. Real-world RDF datasets are valuable because they contain actual, real-world information. In contrast, synthetic datasets are useful for evaluating scalability by varying dataset sizes. These synthetic datasets are generated using specific tools designed to mimic the characteristics and features of real-world datasets[7].

  Most research works regarding GPP are conducted and evaluated using real-world RDF datasets and real-world SPARQL queries submitted by users[6]. The main sources of real-world graph datasets repository are found in SNAP (Stanford Network Analysis Project) [14], Online Network Research Web Portal[15], KONECT (Koblenz Network Collection)[16].

- **Queries:** Saleem et al.[17] propose that a SPARQL querying benchmark should vary queries based on several key characteristics, such as the number of projection variables, triple patterns, result set sizes, query execution time, Basic Graph Patterns, join vertices, mean join vertex degree, mean triple pattern selectivity, join vertex types, and frequently used SPARQL clauses[7].

- **Number of partitions:** To evaluate different GP techniques on equal conditions, they must be partitioned to k (same for each one, if possible, of course) subgraphs.

- **Choice of machine:** CPU power, storage, RAM affects the final result. And must be chosen carefully.

- **Graph Characteristics:** Such as size of the graph($|V|, |E|$) , degree distribution (skewness), and density can significantly influence partitioning results.

We are going to discuss **performance metrics** that are used to evaluate the GP techniques:

- **Query per Second (QpS):** How many queries are executed by a technique in one second [8]

- **Partitioning Imbalance:** a metric that evaluates how evenly the vertices and edges are distributed across different partitions.

- **Number of edge cuts:** A good indicator of communication cost and also one of the two factors that GP techniques tries to optimize by minimizing.

- **Locality:** As mentioned before in the Problem Statement section, normalized cut is defined as:

$$Q = \frac{\omega(E_{ij})}{\mathrm{Vol}(V_1)} + \frac{\omega(E_{ij})}{\mathrm{Vol}(V_2)}.$$

  The lower value of Q indicates that the vertices set are in a good cluster. For edge partitioning, the number of cut vertices are called replicas [3]. It is measured by a replication factor (σ). The σ is calculated as:

$$\sigma = \frac{1}{n} \sum_{i \in k} |P_i(v)|$$

Again, a lower value of σ is preferred because the replication factor also indicates the communication overhead in a way, as communication coexists with vertices.

- **Runtime:** Indicates the time required to partition the graph (also includes loading the input graph into memory and the partitioning time of the graph)[3].
- **Scalability:** Ability to partition increasingly large graphs or handle a growing number of partitions efficiently.
- **Hardware related metrics:** Includes performance measurements such as CPU and memory usage, file size and so on.

2.2.4 Analysis of MLSea-KG

As mentioned, this study's use case will be MLSea-KG. MLSea-KG is a declaratively constructed ML metadata KG containing over 1.44 billion RDF triples that catalogs ML experiments. It integrates metadata from three resources (Kaggle, OpenML, PwC), including datasets, tasks, implementations, hyperparameters, and more.

MLSea-KG consists of three unweighted, undirected RDF graphs, one for each platform (Kaggle, OpenML, PwC). Each RDF graph is stored in Virtuoso[18] triple store. In their study of RDF engines, Saleem et al.[7] analysed that in terms of query runtimes, Virtuoso (version 7.x) is the fastest RDF engine. The performance evaluation was carried on by two very diverse SPARQL benchmarks: FEASIBLE[19] and WatDiv[20]. Three RDF graphs are currently on a single machine. The knowledge graph can be considered as a sparse graph. The dominant share of triples belongs to data obtained from the OpenML platform. Although it consists of 3 RDF graphs, the same partitioning and optimization can be applied to all three of them.

Moreover, there is currently insufficient data in the real user query log. This makes it difficult to effectively use the workload-aware GP technique, as mentioned in Section 2.2.1.1, although it can still be considered an option in the future.

More detailed information regarding MLSea-KG will be presented in the Chapter 3 of this paper.

2.2.5 Virtuoso RDF Engine

In this section, the Virtuoso RDF engine will be introduced; however, more detailed information, such as indexing strategy, tips for better results, tuning, etc., will be explained in the Methodology, Results, and Discussion section later.

Virtuoso is a multi-protocol server that supports access to relational data through both ODBC and JDBC, whether the data is stored internally or in external relational databases. Virtuoso has been used for hosting many of the data sets in the Linking Open Data Project[2], including

[2]https://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

DBpedia[3], Musicbrainz[4], Geonames[5], and others. RDF data in Virtuoso is represented in a single four-column table structure consisting of Subject (S), Predicate (P), Object (O), and Graph (G). Here, S, P, and G are stored as IRIs, while the O column supports the SQL type ANY, allowing it to store a wide range of serializable SQL objects, including scalars, arrays, or user-defined types [18] [7]. SPARQL queries are translated into SQL and executed on the underlying database engine, with query planning guided by cost-based optimization. Data partitioning in Virtuoso is typically done using a subject-hash strategy.

[3]https://www.dbpedia.org/about/
[4]https://musicbrainz.org/doc/About
[5]http://www.geonames.org/about.html

Chapter 3

Methodology

The methods explored in this work are driven by three main objectives: Reducing the amount of data explored during queries by leveraging logical data partitioning (e.g., named graphs). Improving index selectivity and scan efficiency through custom bitmap indexes. Influencing query planning behavior when the default Virtuoso optimizer produces suboptimal execution plans. The following subsections describe the investigated methods in detail, including their intended purpose and the reasoning behind their inclusion in the experimental setup.

3.1 Investigated Methods

This section explains the strategies and mechanisms that were experimented with.

3.1.1 Named Graphs

Named graphs provide a mechanism for logically grouping RDF triples within the same store. By organizing data into separate graphs based on source (e.g., Kaggle, PwC, OpenML), component (e.g., OpenML Tasks, Runs), or semantic role (e.g., types, relationships), it becomes possible to target specific subsets of the data during query execution using the `FROM` clause.

Partitioning strategies investigated include the following:

- Source-based graphs (Experiment 3)
- Predicate-based graphs (Experiment 4)
- Fine-grained `rdf:type` graphs (Experiment 5)
- Semantically grouped predicate graphs (Experiment 6)

Named graph partitioning was used to reduce the search space for SPARQL queries, improve locality, and enable more modular query design. In particular, since many queries access

only a subset of the data, scoping them to relevant graphs can reduce I/O and improve performance.

3.1.2 Indexing Strategies

According to Virtuoso's documentation, the default RDF indexing scheme includes two full indexes on RDF quads and three partial indexes. This configuration is designed to perform well across a wide range of query workloads, regardless of whether queries typically include the graph component. It is generally suitable for both datasets with a large number of small named graphs and those with only a few large graphs. However, default indexes may not be sufficient for specific queries, especially those with known graph and predicate values but unknown subject.

Custom bitmap indexes (e.g., GPOS) were introduced to support cases where the predicate is highly selective, but the query lacks subject constraints. This method was investigated in Experiment 7.

Starting from Virtuoso 7, the indexes are stored in a column-wise format by default, significantly reducing storage requirements, typically using only about one-third of the space of equivalent row-wise structures. (LINK: https://docs.openlinksw.com/virtuoso/rdfperfrdfscheme/) The index scheme consists of the following indices:

- PSOG: primary key
- POGS: bitmap index for lookups on object value
- SP: partial index for cases where only S is specified.
- OP: partial index for cases where only O is specified.
- GS: partial index for cases where only G is specified.

The indexes can be found in the `RDF_QUAD` table, and they can greatly affect the performance of SPARQL queries. To access which indexes Virtuoso is managing, one can use `STATISTICS` command, as shown below in Listing 3.1.

Listing 3.1: Statistics command

```
isql v 1111 dba dba <<EOF
STATISTICS DB.DBA.RDF_QUAD;
EOF
```

The motivation of this approach was to test whether adding custom indexes could reduce lookup time and improve execution efficiency in scenarios where the data distribution or query patterns cause the default optimizer to underperform.

3.1.3 Query Planning with Pragmas

Usually, Virtuoso employs a cost-based query optimizer to determine the most efficient execution plan based on statistics and cardinality estimates (**Give the meaning of cardinality in related works)**. However, this automatic optimization doesn't always yield optimal performance, especially regarding huge graphs and skewed data (such as some predicates or values appearing much more frequently than others).

To address this, pragmas are directive instructions for Virtuoso to execute queries in a certain way. Although pragmatics can improve performance in specific scenarios, degraded performance is also possible.

3.1.4 Source-Based Data Isolation (Simulated Distribution)

Another method investigated in this study was source-based data isolation, where RDF data from each individual source (Kaggle, PwC, and OpenML) was loaded into a separate Virtuoso instance. This approach serves as a conceptual imitation of a distributed environment, where each node is responsible for handling only one subset of the data.

Although this setup does not implement a fully distributed system with inter-partition communication or query federation, it allows us to explore the potential benefits and limitations of separating data along source boundaries. The primary focus of this method is to investigate how such isolation affects query performance, data modularity, and system maintainability when data is queried independently per source. By removing the complexity of cross-partition joins, the approach simplifies the execution environment and highlights how each source behaves in isolation under the same system configuration.

3.2 Query Generation

This section describes the process of generating SPARQL queries used to evaluate the performance of the OpenLink Virtuoso RDF store in the context of the MLSea-KG dataset. A total of 53 SPARQL SELECT queries were manually crafted using the SPARQL Burger Python library[1], supported by ontology mappings derived from the schema and semantics of the dataset. The goal was to design a diverse and realistic query workload that reflects how an actual user or application might interact with the knowledge graph. The primary objective during query design was to ensure balanced coverage across all three data sources, OpenML, Kaggle, and PwC. Queries were written to target different types of entities and relationships from each source, sometimes individually and sometimes jointly. This cross-source focus allowed the experiments to evaluate how different optimization techniques affect query performance on varying graph structures, sizes, and vocabularies. All queries were formulated as SELECT queries to reflect realistic user interactions such as information retrieval, ex-

[1] https://pmitzias.com/SPARQLBurger/

ploration, or analytics tasks. Other query types like ASK, CONSTRUCT, or DESCRIBE were not included, as they typically do not generate complex enough workloads to differentiate performance under different configurations meaningfully. Several queries include filtering mechanisms based on keyword inclusion, such as retrieving entities whose labels contain specific terms (e.g., "classification", "transformer", "notebook"). These filters simulate common exploratory use cases where users search for relevant items without knowing the exact URIs. The queries were designed to reflect meaningful access patterns rather than being randomly generated. Some focus on single-source lookup, others on multi-source joins, and some intentionally repeat predicates like `rdf:type` to observe performance impacts under skewed or high-frequency patterns. This structure allows the experiments to simulate different user behaviors and validate how Virtuoso performs under broad exploratory queries and narrowly targeted lookups across a partitioned knowledge graph.

3.2.1 Workflow Design

The experimental workflow comprised the following similar steps:

1. **Pre-processing N-triples file (.nt files)**: Before bulk loading the N-triples into the Virtuoso database directly, there is an opportunity to manipulate them according to the experiment.
2. **Data Ingestion**: Bulk-loading RDF datasets into Virtuoso.
3. **Query customization**: Modifying the query depending on the query.
4. **Running Queries**: Running the queries in multiple iterations (around 10 rounds for each query).
5. **Performance Monitoring**: Profiling resource usage (CPU, memory) with the VSC's built-in monitoring tools.
6. **Recording the measurements**: After query iterations, time measurements and all the other hardware measurements are collected.

3.2.2 Ontology Guided Query Design

To systematically derive meaningful SPARQL queries, the underlying ontology concepts and relationships defined in the MLSea-KG dataset were carefully studied[2]. This involved analyzing how entities such as datasets, tasks, models, evaluations, etc., are connected through specific predicates. These structural insights were then used as templates to construct valid and semantically relevant triple patterns, forming the basis for the manually written SPARQL queries.

[2]https://github.com/dtai-kg/MLSea-KGC/tree/main/resource_code/Mappings

For example, by analyzing the mapping for PwC scientific papers, as shown in Listing 3.2, it became clear that each paper is modeled as a `mlso:ScientificWork`, identified by a URI derived from its title. The mapping also specifies that the title is stored using `dcterms:title`, the abstract with `fabio:abstract`, and the authors are linked via `dcterms:creator` to agent URIs, which are described with `rdfs:label`. Based on this structure, a SPARQL query was constructed to retrieve the titles, abstracts, and author names of scientific works, as shown in Listing 3.3. The query also includes keyword-based filtering in the title field to simulate a typical user behavior: searching for works related to "neural", "deep" or "learning". The query operates over the PwC graph and reflects the relationships defined in the mapping.

Listing 3.2: PwC paper ontology mapping snippet

```
mappings:
    paper:
      sources:
      access:
        "Mappings/PwC/Data/papers_with_abstracts_sample.json"
      referenceFormulation: jsonpath
      iterator: $.*
     s: http://w3id.org/mlsea/pwc/scientificWork/$(title)
     po:
       - [a, mlso:ScientificWork]
       - [rdfs:label, $(title), rdfs:Literal]
       - [dcterms:title, $(title), rdfs:Literal]
       - [dcterms:source, $(paper_url)~iri]
       - [fabio:hasArXivId, $(arxiv_id), rdfs:Literal]
       - [fabio:abstract, $(abstract), rdfs:Literal]
       - [dcterms:source, $(url_pdf)~iri]
       - [dcterms:issued, $(date), xsd:dateTime]
       - [dcterms:creator,
          http://w3id.org/mlsea/pwc/agent/$(authors)~iri]
       - [mlso:hasRelatedImlpementation,
          http://w3id.org/mlsea/pwc/implementation/$(title)~iri]
       - [dcat:keyword, $(tasks), rdfs:Literal]
       - [dcat:keyword, $(methods.name), rdfs:Literal]
       - p: mlso:hasTaskType
         o:
             mapping: taskTypes
             condition:
               function: equal
               parameters:
                   - [str1, $(tasks), s]
                   - [str2, $(labels), o]
  author:
```

```
      sources:
        access:
          "Mappings/PwC/Data/papers_with_abstracts_sample.json"
        referenceFormulation: jsonpath
        iterator: $.*
      s: http://w3id.org/mlsea/pwc/agent/$(authors)
      po:
        - [a, foaf:Agent]
        - [rdfs:label, $(authors), rdfs:Literal]
        - [foaf:name, $(authors), rdfs:Literal]
```

Listing 3.3: SPARQL Query #17

```
# This query returns scientific work titles along
# with their abstracts and author names that contain
# specific keywords (neural, deep, or learning).
SPARQL
PREFIX mls: <http://www.w3.org/ns/mls#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX mlso: <http://w3id.org/mlso/>
PREFIX fabio: <http://purl.org/spar/fabio/>

SELECT ?workTitle ?abstract ?authorName
FROM <http://example.org/graph/pwc>
WHERE {
  ?work a mlso:ScientificWork;
        dcterms:title ?workTitle.

  OPTIONAL {
    ?work fabio:abstract ?abstract.
  }

  OPTIONAL {
    ?work dcterms:creator ?author.
    ?author rdfs:label ?authorName.
  }

  FILTER (
    CONTAINS(LCASE(?workTitle), "neural") ||
    CONTAINS(LCASE(?workTitle), "deep") ||
    CONTAINS(LCASE(?workTitle), "learning")
  )
}
ORDER BY ?workTitle
```

```
LIMIT 1000;
```

3.2.3 Grouping and Purpose of the Queries

As mentioned at the beginning of the section, the queries were written to target different types of entities and relationships from each source. A more detailed representation of the distribution of which queries originate from which source is provided in Table 3.1.

Table 3.1 Query set statistics

Mappings	Queries
OpenML Dataset	2,5,6,7,9,15,20,21,27,28,29,31,32,36,38,44,48.49,50
OpenML Run	2,5,12,13,14,22,26,35,47,50,51,52,53
OpenML Flow	30,41,45,48,51,52,53
OpenML Task	5,6,18,19,24,40,44
PwC Dataset	2,15,29,44,48,49,50
PwC Paper	1,3,4,8,11,17,27,30,33,37,41,42,43,45,51,52,53
PwC Evaluation	13,14,22,26
PwC Paper Code Links	23,25,34
PwC Model	10,35,39,46,47
Kaggle Dataset	2,7,15,16,29,44,48,49,50

3.3 Experiments

3.3.1 Experimental Setup

This section describes the computational infrastructure and tools used to conduct the experiments.

3.3.1.1 High-Performance Computing Environment

The experiments were performed on the **Vlaams Supercomputer Centrum (VSC)**, a HPC environment designed for large-scale scientific research and data-intensive tasks. The VSC provided the following capabilities, critical to this work:

- **Storage and Memory**: Access to high-speed storage and large memory nodes for handling massive datasets.

- **High Performance Computing**: Access to powerful CPUs (and GPUs) for intensive tasks.
- **Job Scheduling**: Utilization of the SLURM workload manager for the efficient allocation of computational resources.

All experiments were conducted on the same hardware configuration to ensure fair comparison. Specifically, they were run on the VSC using 'Skylake large mem' nodes equipped with Intel Xeon Gold 6240 CPUs, 16 GB of RAM, and over 2 TB of high-speed storage. The operating system used across all experiments was Rocky Linux 8.9 (Green Obsidian). There are three default storage setups in VSC:

- `$VSC_DATA`: 75 GB permanent storage suitable for data, code, software, and results. And in this project, this storage was mainly used for storing results, SLURM files, and bash scripts.
- `$VSC_SCRATCH`: 2 TB temporary storage recommended for intensive, parallel I/O. This storage was used for storing snapshots of MLSea-KG and database files of Virtuoso Instances (virtuoso.db).
- `$VSC_HOME`: 3 GB storage used only for storing SSH keys, configuration files, etc.

3.3.1.2 Cloud Environment (GCE)

To evaluate the influence of infrastructure differences, one version of the baseline experiment was replicated on a virtual machine (VM) hosted on Google Cloud Platform. The VM was configured with 2 virtual CPUs (1 physical core), 16 GB RAM, 500 GB of local SSD storage, Intel Cascade Lake (or later) CPU platform.

3.3.1.3 Initial Software Configuration

Before starting with the experiments, all the N-triples files (or .nt files) were downloaded from the Zenodo repository [3]. And by using data transfer software [4], the files were transferred from the local machine to the remote server in HPC, in $VSC_SCRATCH storage.

At this point, there are 34 N-triples files, which can be left as it is, or there is also the possibility to concatenate them if needed. But, simply concatenating N-triples files can lead to issues with blank nodes (bnodes). BNodes with the same identifier in different files might be incorrectly merged as a single resource. To avoid this, it is advised to use RDF processing tools that handle BNodes appropriately. For example, using the rapper utility from the Redland package[5] can help to handle such cases. But for the specific case of MLSea-KG, this was not a major concern, as .nt files didn't contain BNodes.

[3] https://zenodo.org/records/11264641
[4] https://www.globus.org/
[5] https://librdf.org/

As mentioned before, the software used for query processing and analysis was **OpenLink Virtuoso v7.2**, a database engine. To enable Virtuoso connectivity via ODBC, which is a standard API that allows software applications to communicate with different database systems using a common interface. In the case of this study, Virtuoso uses ODBC to allow SPARQL queries or database tools to access the RDF store as if it were a standard relational database.

Two configuration files `.odbc.ini` and `.odbcinst.ini` were created within my user environment. These files specify the ODBC driver and connection parameters required for Virtuoso to function properly in the HPC setup. `.odbcinst.ini` file defines the ODBC driver (i.e., the software that knows how to talk to Virtuoso). It specifies the path to the driver library and other technical settings. `.odbc.ini` file defines the connection details for your Virtuoso instance, such as the name of the database/DSN, host and port.

Listing 3.4: .odbc.ini file

```
[ODBC Data Sources]
LocalVirtuoso = Virtuoso

[LocalVirtuoso]
Description = Virtuoso Database
Driver      = Virtuoso
Address     = localhost:1111
```

Listing 3.5: .odbcinst.ini file

```
[ODBC Drivers]
Virtuoso = Installed

[Virtuoso]
Description = OpenLink Virtuoso ODBC Driver
Driver      = /apps/leuven/rocky8/icelake/2023b/software/
Virtuoso-opensource/7.2.14-GCC-13.2.0/
lib/virtodbc.so
```

Once the necessary snapshots were transferred to VSC, different virtuoso directories for each the experiment were set up.

A critical component of the experimental setup involved fine-tuning the `virtuoso.ini` configuration file, which governs the behavior of the Virtuoso server. This file specifies key operational parameters such as memory allocation, buffer sizes, and I/O behavior. Proper configuration of these settings is essential for optimizing performance, especially when handling large-scale RDF datasets.

With a dataset comprising approximately 1.44 billion RDF triples, careful adjustment of the `virtuoso.ini` parameters was necessary to ensure efficient data loading and query execution by tailoring settings such as `NumberOfBuffers`, `MaxDirtyBuffers`, and `MaxCheckpointRemap`

to align with the numerous system resources, we aimed to maximize throughput and minimize latency during the experiments.

1. **NumberOfBuffers and MaxDirtyBuffers:** To optimize Virtuoso's performance for loading and querying the 1.44 billion triples dataset, it's essential to adjust the NumberOfBuffers and MaxDirtyBuffers settings in the `virtuoso.ini` configuration file. These parameters control the amount of memory allocated for database caching, significantly impacting data loading and query performance. NumberOfBuffers basically determines the total number of 8KB memory buffers Virtuoso will use and MaxDirtyBuffers specifies the maximum number of these buffers that can contain modified (dirty) data before being flushed to disk. The optimal values for these parameters depend on the available system RAM. A general guideline is to allocate between 2/3 to 3/5 of the system's RAM to Virtuoso's buffers. Values for these two parameters were chosen from Virtuoso's own guidelines for tuning [6]. A general formula for calculating the optimum Virtuoso NumberOfBuffers setting for a given amount of available memory is:

   NumberOfBuffers = (Free Memory * 0.66)/8000 = 16 GB * 0.66 /8KB = 1360000
   MaxDirtyBuffers = approximately 75% of NumberOfBuffers = 1000000

2. **MaxCheckPointRemap:** Specifies how many pages Virtuoso is allowed to remap. Setting `MaxCheckpointRemap` to 1/4th of the database size is recommended. This is in pages, 8K per page. Which then becomes: MaxCheckPointRemap = 100 GB / 8K = 3276800

To optimize Virtuoso for the experiment involving GCE (**see Section 3.3.2.1**), `NumberOfBuffers` and `MaxDirtyBuffers` were adjusted to 1020000 and 750000, respectively. These values were selected to make better use of available memory and minimize I/O overhead during intensive query execution. Additionally, Virtuoso version 8.3 was used instead of version 7.2 to take advantage of performance improvements and updated features.

3.3.2 Implementation of Experiments

In this section, implementations of experiments will be discussed and in the following chapter, the results will be discussed. All measurements and customized queries for all experiments can be found in the repository as well[7].

3.3.2.1 Default Settings

Experiment 1: Baseline Performance on HPC and GCP This experiment aimed to establish baseline performance metrics for Virtuoso by evaluating its default configuration,

[6] https://vos.openlinksw.com/owiki/wiki/VOS/VirtRDFPerformanceTuning
[7] https://github.com/eltajj18/Knowledge-Graph-Partitioning/tree/main/Experiments

without applying any form of optimization. The goal was to provide a reference point against which the effectiveness of the various optimization strategies explored in later experiments could be compared.

The complete RDF dataset, containing approximately 1.44 billion triples was ingested using Virtuoso's built-in bulk loader. Before loading, a `global.graph` file was created to specify a single target named graph.

In this case, the file simply contained the URI: `http://example.org/mygraph`.

Once the data was loaded, the set of 53 queries was executed via the ISQL command-line interface. Query execution times were measured individually to serve as baseline metrics for comparison with future experiments.

To explore the impact of the execution environment on performance, the same procedure was replicated across two different infrastructures:

- A high-performance computing (HPC) node provided by VSC.
- A virtual machine (VM) hosted on Google Cloud Platform, configured with a different CPU platform (as mentioned in Section 3.3.1.3).

By executing the same dataset and queries in both environments under identical Virtuoso settings, this experiment also allowed us to assess how hardware characteristics and system context affect query performance. In particular, it provided a basis for evaluating whether the shared software model of HPC environments has any measurable influence compared to isolated VM deployments.

Experiment 2: Simulated Distribution by Source-Based Instance Separation

This experiment applied the source-based isolation strategy described in Section 3.1.4. The RDF data from Kaggle, PwC, and OpenML was loaded into three separate Virtuoso instances, each handling only one dataset. Each instance received a full N-triples file for its respective source, resulting in three distinct database files.

Since each instance contained data from only one source, not all queries could be executed meaningfully across all setups. Therefore, for this experiment, the analysis focuses only on queries that involve information from a single data source and could be executed within the corresponding isolated instance.

Listing 3.6: Bulk Loading Script

```
# Register RDF files and run bulk loader
DATA_DIR="$VSC_SCRATCH/mlseaKG_RDF_SNAPSHOTS"
isqlv 1111 dba dba <<EOF
ld_dir_all('$DATA_DIR', '*.nt', NULL);
rdf_loader_run();
checkpoint;
EOF
```

3.3.2.2 Named Graphs

Experiment 3: Source-Based Named Graph Partitioning

In the third experiment, a simple form of logical partitioning by loading the RDF data into multiple named graphs based on the source and type of the data was explored. Specifically, six named graphs were created: one for the Kaggle dataset, one for the PwC dataset, and four separate graphs for different OpenML components, namely tasks, runs, flows, and datasets. The N-triples files were organized accordingly, and for each file or group of files, a corresponding .graph file was created specifying the appropriate graph URI. This allowed Virtuoso's bulk loader to assign the triples to the correct named graph during the import process.

The same set of SPARQL queries used in the first experiment was reused, but with minor modifications to better align with the named graph-based partitioning strategy applied in this experiment. In particular, a `FROM` clause was added to each query to explicitly target the relevant named graph. This ensured that each query accessed only the subset of data corresponding to its intended source or component. Execution times were then measured to evaluate the impact of this logical partitioning approach on query performance.

Listing 3.7: SPARQL Query #17 adapted for Experiment 3

```
# This query returns scientific work titles along with their
# authors and publication dates that contain specific
# keywords (transformer, attention, or bert).
SPARQL
PREFIX mls: <http://www.w3.org/ns/mls#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX mlso: <http://w3id.org/mlso/>
PREFIX fabio: <http://purl.org/spar/fabio/>


SELECT ?workTitle ?abstract ?authorName
FROM <http://example.org/graph/pwc>
WHERE {
  ?work a mlso:ScientificWork;
        dcterms:title ?workTitle.

  OPTIONAL {
    ?work fabio:abstract ?abstract.
  }

  OPTIONAL {
    ?work dcterms:creator ?author.
    ?author rdfs:label ?authorName.
```

```
  }

  FILTER (
    CONTAINS(LCASE(?workTitle), "neural") ||
    CONTAINS(LCASE(?workTitle), "deep") ||
    CONTAINS(LCASE(?workTitle), "learning")
  )
}
ORDER BY ?workTitle
LIMIT 1000;
```

Experiment 4: Predicate-Based Named Graph Partitioning

The sixth experiment investigated the effect of fine-grained predicate-based partitioning on query performance.

To achieve this, the RDF data was split at a more granular level, by individual predicate, using a custom shell script shown in Listing 3.8. This script takes a single .nt file as input and produces multiple .nt files, each containing only triples that share the same predicate. This approach effectively partitions the data based on the predicate, and it was inspired by the study conducted on workload-aware graph partitioning done by Akhter et al.[8], which was discussed in the related works, but the difference lies in grouping predicates of the graph individually, instead of using query logs.

Listing 3.8: Bash script to split N-triples files based on predicate

```
#!/bin/bash
INPUT_FILE="mlseaKG_RDF_SNAPSHOTS_openml/openml_runs_merged.nt"
OUTPUT_DIR="splitted_openml_runs_files/"
awk '
{
    predicate = $2
    gsub(/[^a-zA-Z0-9_]/, "_", predicate)
    filename = "splitted_openml_runs_files/" predicate ".nt"
    print $0 >> filename
}
' "$INPUT_FILE"
```

The script uses `awk` to extract the predicate from each triple, sanitizes the predicate string to be file-system safe, and writes each triple to a corresponding .nt file named after its predicate. Table 3.2 presents the distribution of RDF predicates in the MLSea-KG dataset. A small number of predicates, such as `rdf:type`, `mls:specifiedBy`, and `mls:hasValue`, occur extremely frequently and dominate the dataset. This skewed distribution highlights the need to handle high-frequency predicates carefully, as they can significantly impact query performance.

After splitting the data for all sources (Kaggle, PwC, and OpenML), all the resulting .nt files

were bulk loaded into Virtuoso. Each file was loaded into a separate named graph associated with its predicate. The SPARQL queries used in this experiment were modified accordingly by explicitly specifying the relevant named graph using the `FROM` clause to target only the graph corresponding to the predicate in the query pattern, as it can be seen in Listing 3.9.

Listing 3.9: SPARQL Query #17 adapted for Experiment 4

```
# This query returns scientific work titles along with their
# authors and publication dates that contain specific
# keywords (transformer, attention, or bert).
SPARQL
PREFIX mls: <http://www.w3.org/ns/mls#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX mlso: <http://w3id.org/mlso/>
PREFIX fabio: <http://purl.org/spar/fabio/>

SELECT ?workTitle ?abstract ?authorName
FROM <http://example.org/_http___www_w3_org_1999_02_22_rdf
_syntax_ns_type_>
FROM <http://example.org/_http___www_w3_org_2000_01_rdf_schema
_label_>
FROM <http://example.org/_http___purl_org_dc_terms_title_>
FROM <http://example.org/_http___purl_org_spar_fabio_abstract_>
FROM <http://example.org/_http___purl_org_dc_terms_creator_>
WHERE {
  ?work a mlso:ScientificWork;
        dcterms:title ?workTitle.

  OPTIONAL {
    ?work fabio:abstract ?abstract.
  }

  OPTIONAL {
    ?work dcterms:creator ?author.
    ?author rdfs:label ?authorName.
  }

  FILTER (
    CONTAINS(LCASE(?workTitle), "neural") ||
    CONTAINS(LCASE(?workTitle), "deep") ||
    CONTAINS(LCASE(?workTitle), "learning")
  )
}
ORDER BY ?workTitle
LIMIT 1000;
```

This experiment aimed to explore whether predicate-level partitioning and precise graph targeting in queries can reduce query overhead and improve execution efficiency by minimizing the amount of scanned data.

Table 3.2 Predicates by Triple Count

Predicate	**Number of Triples**
http://www.w3.org/1999/02/22-rdf-syntax-ns#type	307,685,291
http://www.w3.org/ns/mls#specifiedBy	281,427,066
http://www.w3.org/ns/mls#hasValue	280,918,491
http://www.w3.org/2000/01/rdf-schema#label	198,597,230
http://www.w3.org/ns/mls#hasOutput	174,107,745
http://www.w3.org/ns/mls#hasInput	116,088,365
http://purl.org/dc/terms/creator	11,944,785
http://www.w3.org/ns/mls#executes	10,068,558
http://www.w3.org/ns/mls#achieves	10,037,962
http://www.w3.org/ns/prov#atLocation	8,726,962
http://www.w3.org/mlso#hasPredictionsLocation	8,678,635
http://www.w3.org/mlso#hasDataCharacteristicType	7,203,202
http://www.w3.org/ns/mls#hasQuality	7,203,202
http://purl.org/dc/terms/title	5,796,290
http://www.w3.org/ns/mls#hasPart	3,631,294
http://www.w3.org/ns/dcat#keyword	1,714,990
http://schema.org/codeRepository	1,476,273
http://purl.org/dc/terms/modified	1,436,089
http://purl.org/dc/terms/created	1,431,233
https://w3id.org/okn/o/sd#programmingLanguage	1,087,840
https://w3id.org/okn/o/sd#hasSourceCode	1,085,129

Continued on next page

Table 3.2 (Continued)

Predicate	**Number of Triples**
http://purl.org/dc/terms/source	930,627
http://w3id.org/mlso/hasRelatedImplementation	918,875
http://xmlns.com/foaf/0.1/name	886,629
http://www.w3.org/mlso#hasRelatedSoftware	848,828
http://www.w3.org/mlso#hasTaskType	569,403
http://www.w3.org/ns/mls#implements	508,885
http://purl.org/dc/terms/issued	481,740
http://purl.org/spar/fabio/abstract	442,229
http://purl.org/dc/terms/identifier	415,883
http://purl.org/spar/fabio/hasArXivId	389,006
http://www.w3.org/ns/dcat#landingPage	352,255
http://purl.org/dc/terms/license	351,320
https://w3id.org/okn/o/sd#softwareRequirements	225,014
http://purl/org/dc/terms/description	151,751
http://www.w3.org/ns/mls#hasHyperParameter	114,079
http://www.w3.org/ns/mls#definedOn	47,273
http://www.w3.org/ns/mls#defines	47,273
http://purl.org/dc/terms/language	24,806
http://purl.org/dc/terms/hasVersion	22,332
http://www.w3.org/mlso#hasNumberOfReferences	15,018
http://www.w3.org/mlso#isVariantOf	12,955
http://www.w3.org/mlso#hasVariant	12,955
http://www.w3.org/mlso#hasModality	9,151
http://www.w3.org/mlso#hasScientificReference	7,789
http://www.w3.org/mlso#hasFormat	5,481

Continued on next page

Table 3.2 (Continued)

Predicate	**Number of Triples**
http://www.w3.org/ns/dcat/distribution	5,481
http://www.w3.org/mlso# hasDataLoaderLocation	5,261
http://purl.org/dc/terms/accessRights	4,980
http://open.vocab.org/terms/hasMD5	4,980
http://www.w3.org/mlso#hasCacheFormat	4,980
http://www.w3.org/ns/dcat/downloadURL	4,572
http://www.w3.org/mlso# hasDefaultTargetFeature	3,855
http://www.w3.org/mlso# hasEvaluationProcedureType	3,163
http://www.w3.org/mlso#hasAlgorithmType	2,107
http://www.w3.org/mlso#hasIdFeature	1,168
http://purl.org/dc/terms/contributor	648
http://www.w3.org/mlso# scientificReferenceOf	308
http://www.w3.org/mlso# hasEvaluationMeasureType	29

Experiment 5: Fine-Grained Partitioning of rdf:type Predicate

This experiment extends the approach taken in Experiment 4 by applying a more fine-grained partitioning strategy specifically targeting the `rdf:type` predicate. During analysis, it was observed that `rdf:type` appears in nearly all queries and represents a significant portion of the dataset, resulting in a heavily skewed distribution. To address this, the RDF triples associated with `rdf:type` were further subdivided based on their specific object types using a shell script shown in Listing 3.10.

Listing 3.10: Bash script to split N-triples files further based on type

```
#!/bin/bash

INPUT_FILE="/mlseaKG_RDF_SNAPSHOTS_splitted/splitted_openml_runs_files
/_http___www_w3_org_1999_02_22_rdf_syntax_ns_type_.nt"
OUTPUT_DIR="/splitted_types_openml_runs"

mkdir -p "$OUTPUT_DIR"

echo "Splitting $INPUT_FILE into files grouped by rdf:type object..."

# Extract unique object types
grep -oP '\s<[^>]+>\s\.' "$INPUT_FILE" | sort | uniq | while read -r line; do
```

```
    # Extract object IRI
    object=$(echo "$line" | awk '{print $1}')

    # Sanitize for filename (replace special characters)
    safe_name=$(echo "$object" | sed 's|[<>:/#]|_|g')

    # Create output file
    out_file="$OUTPUT_DIR/${safe_name}.nt"

    # Extract all lines with this object
    grep "$object" "$INPUT_FILE" >> "$out_file"

    echo "Written: $out_file"
done
echo "Done. Output in: $OUTPUT_DIR/"
```

Each group of `rdf:type` triples was assigned to a dedicated named graph, enabling more precise query targeting. In the corresponding SPARQL queries, the `FROM` clause was adjusted accordingly to reference the appropriate named graph depending on the specific type being queried as shown in Listing 3.11. This setup aimed to improve selectivity and allow Virtuoso to operate over smaller, more focused subsets of data during query execution.

Listing 3.11: SPARQL Query #17 adapted for Experiment 5

```
# This query returns scientific work titles along with their
# authors and publication dates that contain specific
# keywords (transformer, attention, or bert).
SPARQL
PREFIX mls: <http://www.w3.org/ns/mls#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX mlso: <http://w3id.org/mlso/>
PREFIX fabio: <http://purl.org/spar/fabio/>

SELECT ?workTitle ?abstract ?authorName
FROM <http://example.org/_http___w3id.org_mlso_ScientificWork_>
FROM <http://example.org/_http___www_w3_org_2000_01_rdf_schema_
label_>
FROM <http://example.org/_http___purl_org_dc_terms_title_>
FROM <http://example.org/_http___purl_org_spar_fabio_abstract_>
FROM <http://example.org/_http___purl_org_dc_terms_creator_>
WHERE {
  ?work a mlso:ScientificWork;
        dcterms:title ?workTitle.

  OPTIONAL {
    ?work fabio:abstract ?abstract.
  }

  OPTIONAL {
    ?work dcterms:creator ?author.
    ?author rdfs:label ?authorName.
```

```
  }

  FILTER (
    CONTAINS(LCASE(?workTitle), "neural") ||
    CONTAINS(LCASE(?workTitle), "deep") ||
    CONTAINS(LCASE(?workTitle), "learning")
  )
}
ORDER BY ?workTitle
LIMIT 1000;
```

Experiment 6: Semantic Grouping of Predicates into Logical Named Graphs

Building on the partitioning strategy of Experiment 5, this experiment involved grouping RDF predicates based on their semantic relationships and expected co-occurrence in queries. Predicates that commonly appear together in data mappings or are likely to be queried together in future use cases were identified and assigned to the same named graph. These groups were labeled as logical graphs (e.g., Graph A, Graph B, Graph C), each representing a semantically coherent subset of the data.

The .nt files corresponding to these grouped predicates were loaded into Virtuoso using named graphs that reflect their grouping as shown in the Table 3.4, as it can be seen from the table that only a portion of the predicates were chosen and grouped together for the reason mentioned earlier. In SPARQL queries, the `FROM` clause was modified to reference the appropriate graph based on the semantics of the query. This experiment aimed to create a more modular and query-friendly structure, facilitating more efficient and maintainable access to RDF data.

Table 3.4 Grouped Predicates

Predicate	**Assigned URL of their new named graph**
http://www.w3.org/ns/mls#specifiedBy	http://example.org/graph_A
http://www.w3.org/ns/mls#hasValue	http://example.org/graph_A
http://www.w3.org/ns/mls#hasOutput	http://example.org/graph_B
http://www.w3.org/ns/mls#hasInput	http://example.org/graph_B
http://www.w3.org/ns/mls#executes	http://example.org/graph_B
http://www.w3.org/ns/mls#achieves	http://example.org/graph_C
http://www.w3.org/ns/prov#atLocation	http://example.org/graph_E
http://www.w3.org/mlso# hasPredictionsLocation	http://example.org/graph_C

Continued on next page

Table 3.4 (Continued)

Predicate	**Assigned URL of their new named graph**
http://www.w3.org/mlso#hasDataCharacteristicType	http://example.org/graph_A
http://www.w3.org/ns/dcat#keyword	http://example.org/graph_G
http://schema.org/codeRepository	http://example.org/graph_D
https://w3id.org/okn/o/sd#hasSourceCode	http://example.org/graph_D
http://www.w3.org/mlso#hasTaskType	http://example.org/graph_G
http://purl.org/spar/fabio/abstract	http://example.org/graph_H
http://purl.org/spar/fabio/hasArXivId	http://example.org/graph_H
https://w3id.org/okn/o/sd#softwareRequirements	http://example.org/graph_D
http://www.w3.org/mlso#hasNumberOfReferences	http://example.org/graph_F
http://www.w3.org/mlso#hasVariant	http://example.org/graph_F
http://www.w3.org/mlso#scientificReferenceOf	http://example.org/graph_E

3.3.2.3 Indexes

Experiment 7: Custom Bitmap Indexing for Source-Specific Datasets

The seventh experiment focused on evaluating the impact of custom bitmap indexes on query performance. Specifically, new bitmap indexes were created for the Kaggle and PwC datasets, which were each loaded into separate Virtuoso instances. This setup followed Virtuoso's indexing guidelines, which suggest that the default index (OGPS) is typically sufficient when the graph is always specified using `FROM` or `FROM NAMED` clauses, and there are no query patterns where only the graph and the predicate are known. However, in cases where the predicate is selective but the subject is not provided, creating a GPOS style index can improve performance[8].

To explore this, a `RDF_QUAD_GPOS` bitmap index was created using the following command:

Listing 3.12: GPOS Bitmap Index Creation

```
CREATE BITMAP INDEX RDF_QUAD_GPOS
ON DB.DBA.RDF_QUAD (G, P, O, S)
PARTITION (O VARCHAR (-1, 0hexffff));
```

[8]https://docs.openlinksw.com/virtuoso/rdfindexschemesel/

Table 3.3 Distribution of `rdf:type` Predicate

Type	**Number of Instances**
http://www.w3.org/ns/mls#ModelEvaluation	174077149
http://www.w3.org/ns/mls#HyperParameterSetting	107349917
http://www.w3.org/ns/mls#Run	10068558
http://www.w3.org/ns/mls#FeatureCharacteristic	7039794
http://www.w3.org/ns/mls#Feature	3519897
https://w3id.org/okn/o/sd#Software	1267261
https://w3id.org/okn/o/sd#SourceCode	1088656
http://xmlns.com/foaf/0.1/Agent	885457
http://www.w3.org/ns/mls#Implementation	467588
http://w3id.org/mlso/ScientificWork	451045
http://www.w3.org/ns/mls#Dataset	364414
http://www.w3.org/ns/dcat#Catalog	346278
http://www.w3.org/ns/mls#Software	165431
http://w3id.org/mlso/HyperParameterCharacteristic	114079
http://www.w3.org/ns/mls#HyperParameter	114079
http://www.w3.org/ns/mls#ImplementationCharacteristic	114079
http://www.w3.org/ns/mls#DatasetCharacteristic	49329
http://www.w3.org/ns/mls#EvaluationProcedure	47273
http://www.w3.org/ns/mls#Task	47273
http://www.w3.org/ns/mls#EvaluationSpecification	47273
http://www.w3.org/ns/mls#Model	30596
http://www.w3.org/ns/dcat#Dataset	18136
http://www.w3.org/ns/mls#EvaluationMeasure	3997
http://www.w3.org/ns/mls#algorithm	2212
http://w3id.org/mlso/DataModality	39

The **PARTITION** clause at the end of a CREATE INDEX statement instructs Virtuoso to first partition and then populate the index. This is necessary because without it, new indexes cannot be added to tables that already contain data.

Virtuoso supports two types of hashing for partitioning: one for integer-based types (such as INTEGER, BIGINT, and IRI IDs), and one for strings (VARCHAR and similar types).

For integer partitioning, a bitmask is applied to the numeric value before computing the hash. For example, using a mask like 0xFFFF00 means that the hash will be derived from the second and third least significant bytes, so the values of 0-255 hash to the same value, 256-511 to the same value, etc. If the mask is set to 0x1000000, values like 0 and 0x1000000 would still hash identically. This clustering of nearby integers into the same partition is beneficial for key compression and storage efficiency. When no mask is explicitly provided, the default is 0xFFFF, which distributes values based on their lowest 16 bits, ensuring distinct hashes for consecutive integers.

For VARCHAR partitioning, the default behavior is to compute a hash using all bytes in the string. However, to improve key compression and locality, it's often helpful to group strings with common prefixes into the same partition. Partitioning can be controlled using two parameters: length and mask. If length is positive, only the first length characters of the string are used for hashing. If it's negative, Virtuoso uses the entire string except for the last abs(length) characters[9].

This indexing strategy was successfully applied to the Kaggle and PwC datasets, and performance measurements were recorded using the same query set as in previous experiments.

However, applying this index to the OpenML dataset failed due to a uniqueness constraint violation:

Listing 3.13: Error that is encountered on OpenML Graph

```
SR175: Uniqueness violation : Violating unique index RDF_QUAD_GPOS
on table DB.DBA.RDF_QUAD. Transaction killed.
```

This error indicates that the RDF data contained duplicate entries for the (G, P, O, S) combination, which violates the uniqueness requirement of the index. As a result, this part of the experiment could not be completed for OpenML.

3.3.2.4 Pragmas

Experiment 8: Enforcing Explicit Join Order in SPARQL Queries

In this experiment, the impact of explicitly controlling the join order in SPARQL queries executed by Virtuoso was explored. Typically, Virtuoso employs a cost-based query optimizer to determine the most efficient execution plan based on statistics and cardinality estimates. However, this automatic optimization doesn't always yield optimal performance, especially

[9] https://docs.openlinksw.com/virtuoso/clusteroperationpart/

when it comes to very large graphs and skewed data (such as some predicates or values appearing much more frequently than others).

To address this issue, the `sql:select-option "order"` pragma was added at the beginning of the SPARQL queries used in Experiment 3. This directive instructs Virtuoso to execute joins in the exact order in which they appear in the query, thereby bypassing the default join reordering performed by the optimizer. This approach was intended to mitigate potential incorrect estimations in cardinality and improve overall query performance[10].

While enforcing a specific join order can lead to performance improvements in certain scenarios, it requires a thorough understanding of the data and query structure. An inappropriate join sequence might degrade performance.

Listing 3.14: SPARQL Query #17 adapted for Experiment 8

```
# This query returns scientific work titles along
# with their abstracts and author names that contain
# specific keywords (neural, deep, or learning).
SPARQL
PREFIX mls: <http://www.w3.org/ns/mls#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX mlso: <http://w3id.org/mlso/>
PREFIX fabio: <http://purl.org/spar/fabio/>

SELECT ?workTitle ?abstract ?authorName
FROM <http://example.org/graph/pwc>
WHERE {
  ?work a mlso:ScientificWork;
        dcterms:title ?workTitle.

  OPTIONAL {
    ?work fabio:abstract ?abstract.
  }

  OPTIONAL {
    ?work dcterms:creator ?author.
    ?author rdfs:label ?authorName.
  }

  FILTER (
    CONTAINS(LCASE(?workTitle), "neural") ||
    CONTAINS(LCASE(?workTitle), "deep") ||
    CONTAINS(LCASE(?workTitle), "learning")
  )
```

[10] https://docs.openlinksw.com/virtuoso/rdfsparqlimplementatioptragmas/#rdfsparqlimplementatioptragmasccg

```
}
ORDER BY ?workTitle
LIMIT 1000;
```

Chapter 4

Results and Discussion

In this chapter, the results of each experiment will be demonstrated and discussed, and a few final notes will be provided that can prove helpful.

4.1 Results and Discussion

Given that 53 queries were generated and each was executed 10 times in multiple experiments, the volume of data quickly became extensive. Queries were categorized according to their source (SSP, SSK, SSO, MS) and complexity (SQ, IQ, CQ) to present the results more clearly. In total, 7 categories were defined:

1. **Single Source PwC (SSP)**: Queries that can return results from the PwC dataset and are not dependent on another source.
2. **Single Source Kaggle (SSK)**: Queries that can return results from the Kaggle dataset and are not dependent on another source.
3. **Single Source OpenML (SSO)**: Queries that can return results from the OpenML dataset and are not dependent on another source.
4. **Multisource (MS)**: Queries that return results from multiple sources and are dependent on more than one source
5. **Simple Queries (SQ)**: Single triple pattern, no `FILTER/OPTIONAL`, single dataset, no aggregation, basic keyword searches.
6. **Intermediate Queries (IQ)**: Multiple triple patterns, `FILTER` with logical operators, `OPTIONAL`, 1-2 datasets, basic joins.
7. **Complex Queries (CQ)**: Aggregation (`GROUP BY, COUNT, MAX`), nested queries, multi-dataset joins, advanced logic.

It should be noted that some queries can be argued to be categorized in a different complexity category.

Table 4.1 Query Set Sorted by Source and Category

Query ID	Sources Touched	Category
1	PwC Paper	SSP,SQ
2	OpenML Dataset, PwC Dataset, Kaggle Dataset, OpenML Run	MS,SQ
3	PwC Paper	SSP,SQ
4	PwC Paper	SSP,IQ
5	OpenML Dataset, OpenML Run, OpenML Task	MS,IQ
6	OpenML Dataset, OpenML Task	MS,SQ
7	OpenML Dataset, Kaggle Dataset	MS,IQ
8	PwC Paper	SSP, IQ
9	OpenML Dataset	SSO,IQ
10	PwC Model	SSP,SQ
11	PwC Paper	SSP,IQ
12	OpenML Run	SSO,SQ
13	PwC Evaluation, OpenML Run	MS,IQ
14	OpenML Run, PwC Evaluation	MS, IQ
15	OpenML Dataset, PwC Dataset, Kaggle Dataset	MS,IQ
16	Kaggle Dataset	SSK,SQ
17	PwC Paper	SSP,IQ
18	OpenML Task	SSO, IQ
19	OpenML Task	SSO, IQ
20	OpenML Dataset	SSO, IQ
21	OpenML Dataset	SSO, IQ
22	OpenML Run, PwC Evaluation	MS,IQ
23	PwC Paper Code Links	SSP,IQ
24	OpenML Task	SSO,IQ

Continued on next page

Table 4.1 (Continued)

Query ID	Sources Touched	Category
25	PwC Paper Code Links	SSP,IQ
26	OpenML Run, PwC Evaluation	MS,IQ
27	OpenML Dataset, PwC Paper	MS,IQ
28	OpenML Dataset	SSO,IQ
29	OpenML Dataset, PwC Dataset, Kaggle Dataset	MS,IQ
30	OpenML Flow, PwC Paper	MS,SQ
31	OpenML Dataset	SSO,CQ
32	OpenML Dataset	SSO,CQ
33	PwC Paper	SSP,SQ
34	PwC Paper Code Links	SSP,CQ
35	OpenML Run, PwC Model	MS,IQ
36	OpenML Dataset	SSO,IQ
37	PwC Paper	SSP,IQ
38	OpenML Dataset	SSO ,IQ
39	PwC Model	SSP,IQ
40	OpenML Task	SSO,IQ
41	OpenML Flow, PwC Paper	MS,SQ
42	PwC Paper	SSP,IQ
43	PwC Paper	SSP,IQ
44	OpenML Dataset, OpenML Task, PwC Dataset, Kaggle Dataset	MS,CQ
45	OpenML Flow, PwC Paper	MS,CQ
46	PwC Model	SSP,CQ
47	OpenML Run, PwC Model	MS,CQ
48	OpenML Dataset, OpenML Flow, PwC Dataset, Kaggle Dataset	MS,CQ
49	OpenML Dataset, PwC Dataset, Kaggle Dataset	MS,CQ
50	OpenML Dataset, OpenML Run, PwC Dataset, Kaggle Dataset	MS,CQ

Continued on next page

Table 4.1 (Continued)

Query ID	Sources Touched	Category
51	OpenML Run, OpenML Flow, PwC Paper	MS,CQ
52	OpenML Run, OpenML Flow, PwC Paper	MS,CQ
53	OpenML Run, OpenML Flow, PwC Paper	MS,CQ

4.1.1 Experiment 1: Baseline Performance on HPC and GCE

As mentioned in the methodology, the first experiment establishes a baseline for evaluating SPARQL query performance in a default Virtuoso setup without any optimizations. These results serve as a reference point for all subsequent experiments.

In Table 4.2, the average query execution time (AQET) is shown, which is the inverse of Query per Second (QpS) that was explained in Section 2.2.3. And the average standard deviation is calculated by getting the square root of the mean variance of the averaged query execution times of individual queries.

Comparing the two environments, while the HPC environment consistently achieved lower AQET values, the GCE exhibited more consistent performance across runs, as indicated by its lower average standard deviation. This consistency may be attributed to the greater isolation of GCE, whereas HPC environments may introduce fluctuations due to factors such as shared hardware resources, job scheduling overhead, and etc. If multiple applications try to access the same resources (like memory, CPU, or network) simultaneously, it can lead to bottlenecks and delays.

Table 4.2 Execution Time Statistics for Experiment 1 based on 10 Runs

Category	Queries	AQET (ms)(HPC)	Average Std Dev(ms) (HPC)	AQET(ms)(GCE)	Average Std Dev(ms)(GCE)
SSP	16	2113.34	1246.30	2014.5	480.04
SSK	1	1501.4	0	1981.8	0
SSO	13	42117.3	41734.64	44787.70	8927.03
MS	23	82062	35256.41	129729.24	20910.04
SQ	10	46661.85	47584.09	81820.65	11759.25
IQ	30	62440.89	30761.43	87142.95	18588.39
CQ	13	10032.38	4185.85	12902.42	1425.94

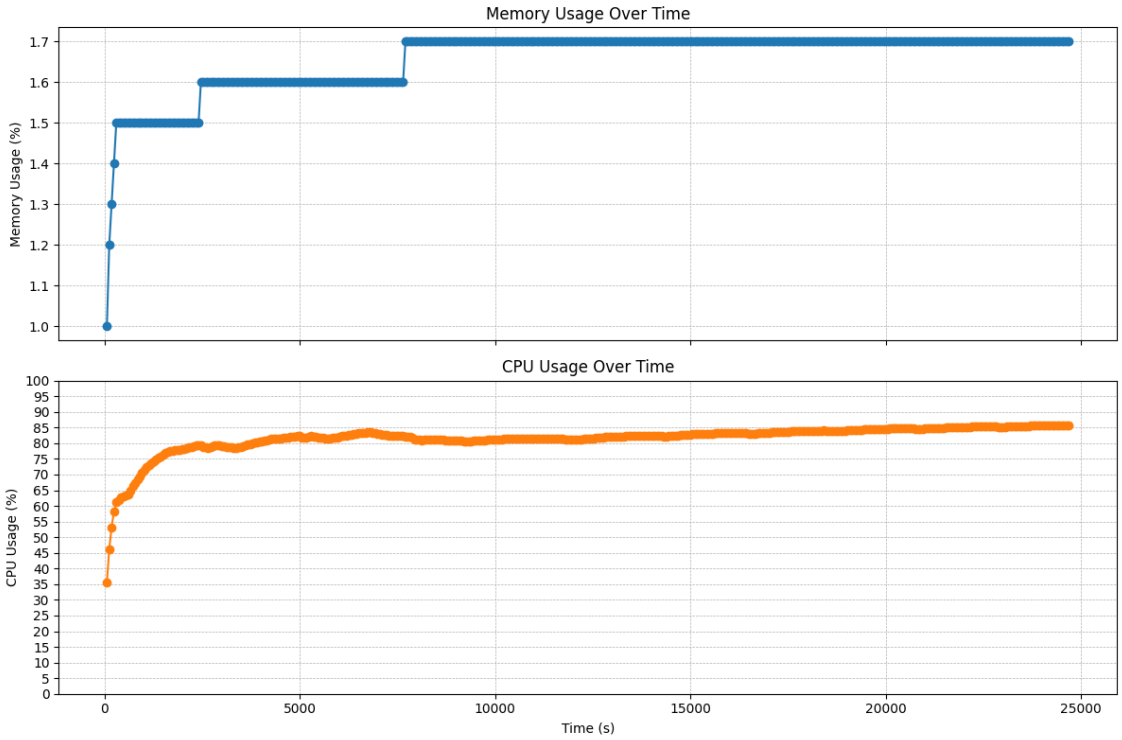


Figure 4.1: CPU and memory usage measurements of Experiment 1 (HPC)

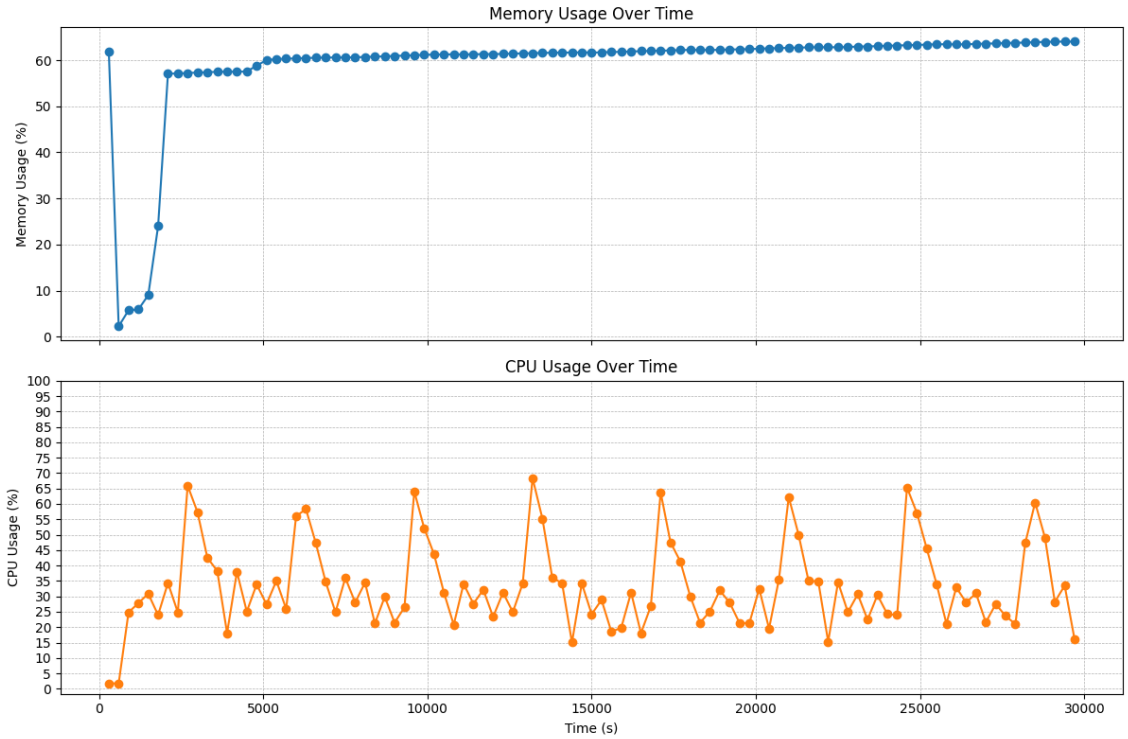


Figure 4.2: CPU and memory usage measurements of Experiment 1 (GCE)

4.1.2 Experiment 2: Simulated Distribution by Source-Based Instance Separation

As indicated by the experiment's title, these results are simulated and do not account for actual network communication delays between physically separated machines. Typically, clustered databases utilize partitioning schemes where data rows are assigned to servers based on key column values. Virtuoso supports flexible partitioning, allowing each index to be partitioned differently. However, according to Virtuoso's documentation[18], partitioning based on the graph or predicate components of a quad is not advisable because it often results in highly uneven data distributions, negatively affecting performance. Instead, partitioning by subject or object is preferred. Moreover, carefully splitting N-Triples files into multiple partitions introduces added complexity in simulated environments. Ensuring query completeness becomes difficult, as the relevant triples needed to answer a query might be scattered across partitions. This can result in incomplete query results or outright failures during execution.

For these reasons, implementing a simulated distributed environment based on predicate partitioning would have been impractical. Therefore, in this experiment, a graph-based approach was adopted instead, which allowed for a more manageable simulation setup. Ironically, a real distributed setup would likely have been simpler to execute correctly (partitioning by subject or object) than the simulation.

Table 4.3 Execution Time Statistics for Experiment 2 based on 10 Runs

Category	Queries	AQET (ms)	Average Std Dev (ms)
SSP	26	888.73	221.24
SSK	3	1029.3	155.35
SSO	23	75596.31	47551.28
SQ	11	741.3	337.82
IQ	37	46385.78	38003.08
CQ	3	13246.9	2114.27

4.1.3 Experiment 3: Source-Based Named Graph Partitioning

The results of experiment 3 show rather good results, as actual improvement can be seen compared to the baseline performance measurements shown in Section 4.1.1. Specifically, a huge reduction in AQET values is observed in SSO,SQ categories, along with more consistent results too.

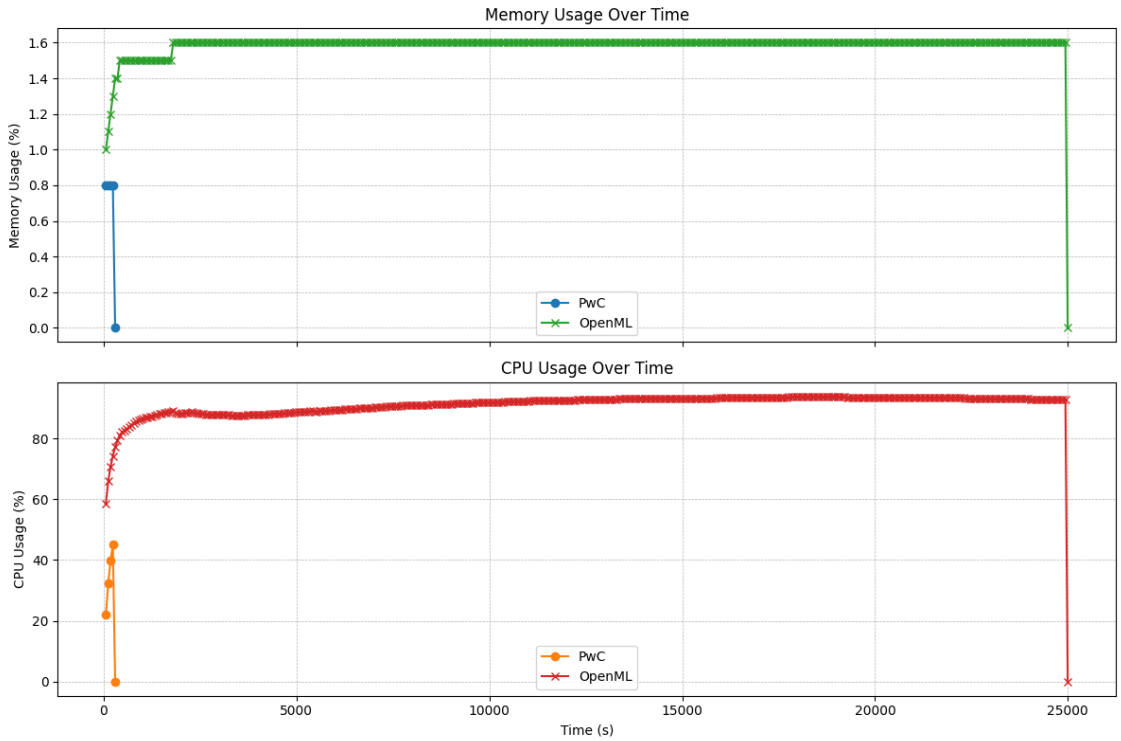


Figure 4.3: CPU and memory usage measurements of Experiment 2

Table 4.4 Execution Time Statistics for Experiment 3 based on 10 Runs

Category	Queries	AQET (ms)	Average Std Dev (ms)
SSP	16	1816.04	1282.04
SSK	1	1166.5	0
SSO	13	7378.32	1101.25
MS	23	111729.37	83708.54
SQ	10	1068.09	966.77
IQ	30	84670.96	73226.3
CQ	13	11162.07	5066.94

4.1.4 Experiment 4: Predicate-Based Named Graph Partitioning

Based on the results provided in Table 4.5, Experiment 3 demonstrated comparatively better results in categories such as Simple Queries (SQ), Multisource (MS), and Complex Queries (CQ). One possible explanation for the increased AQET in the SQ category is the introduction

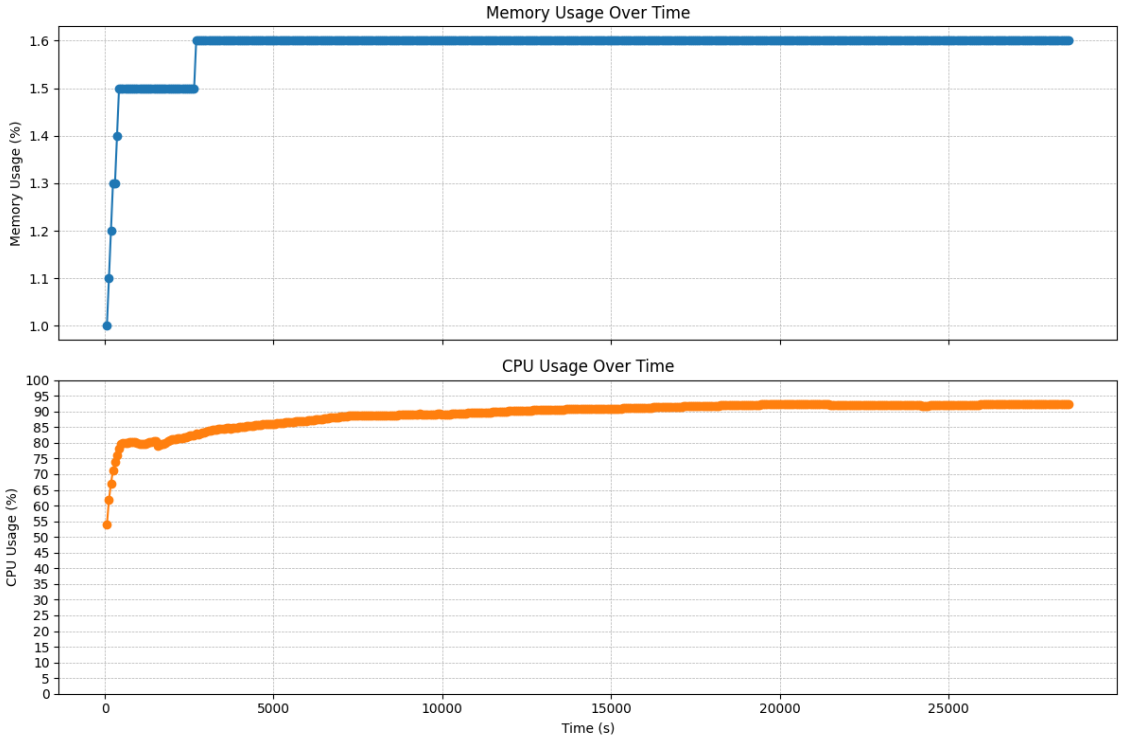


Figure 4.4: CPU and memory usage measurements of Experiment 3

of multiple `FROM` clauses. Since SQ queries were originally designed to operate over a single dataset, splitting the dataset into multiple smaller graphs may have negatively impacted performance.

On the other hand, a notable improvement was observed in the Intermediate Queries (IQ) category, where the AQET decreased significantly, from 84,670.96 ms to 64,139.19 ms. This suggests that the method used in this experiment may be more effective for certain types of queries, particularly those involving moderate complexity and joins across multiple patterns.

Table 4.5 Execution Time Statistics for Experiment 4 based on 7 Runs

Category	Queries	AQET (ms)	Average Std Dev (ms)
SSP	16	1896.64	727.92
SSK	1	1450.29	0
SSO	13	52686.68	23184.44
MS	23	207169.97	9771.03
SQ	10	57623.39	26425
IQ	30	64139.19	8121.56
CQ	13	229324.88	4210.97

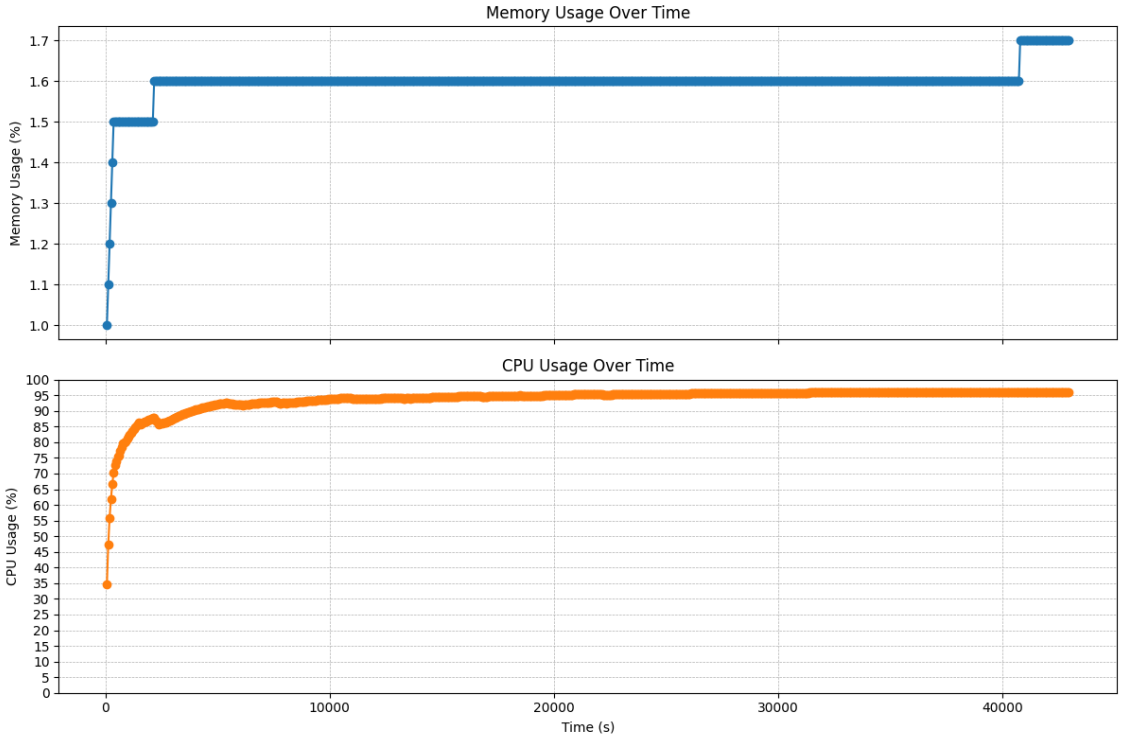


Figure 4.5: CPU and memory usage measurements of Experiment 4

4.1.5 Experiment 5: Fine-Grained Partitioning of rdf:type Predicate

A comparison between the results of this experiment and the previous one, where the `rdf:type` predicate was not split, reveals a significant reduction in AQET across several query categories. Notably, in the MS category, the AQET dropped from 207,169.97 ms

in Experiment 4 to nearly half that value in Experiment 5. The improvement is even more pronounced in the CQ category, where a ninefold decrease in AQET was observed. While a few categories in Experiment 4 performed marginally better by only a few thousand milliseconds, these minor gains do not outweigh the substantial performance improvements seen in Experiment 5. So, further dividing the graph into subgraphs based on the occurrence of the predicates proves to improve performance. However, comparing the results in this experiment to those that were obtained in Experiment 3, it can be seen that it is somewhat similar, except a drastic improvement in consistency and a slight decrease in AQET values are observed in MS category. Moreover, although AQET values worsened in CQ category, the results show more consistency now.

Table 4.6 Execution Time Statistics for Experiment 5 based on 10 Runs

Category	**Queries**	**AQET (ms)**	**Average Std Dev (ms)**
SSP	16	1942.03	894.06
SSK	1	1183.1	0
SSO	13	35168.33	35451.72
MS	23	103647.48	30646.93
SQ	10	36566.15	40411.76
IQ	30	70983.62	26744.6
CQ	13	29089.68	3558.19

4.1.6 Experiment 6: Semantic Grouping of Predicates into Logical Named Graphs

As outlined in the methodology section, Experiment 6 was conducted by logically regrouping a subset of predicates that had been split in the setup of Experiment 5. Only a limited number of predicates were regrouped, leading to modifications in a small set of queries. Specifically, the affected queries were those with IDs: *3, 5, 9, 17, 20, 23, 24, 25, 28, 34, 35, 38, 44, 47, 48, 50, 51, 52, and 53*, most of which coincidentally fall under the Complex Query (CQ) category.

When comparing the results of Experiment 6 with those of Experiment 5, the overall performance remains largely similar. However, a significant improvement is observed in the CQ category, where the AQET decreased from 29,089.68 ms to 18,284.25 ms, representing an improvement of approximately 37%.

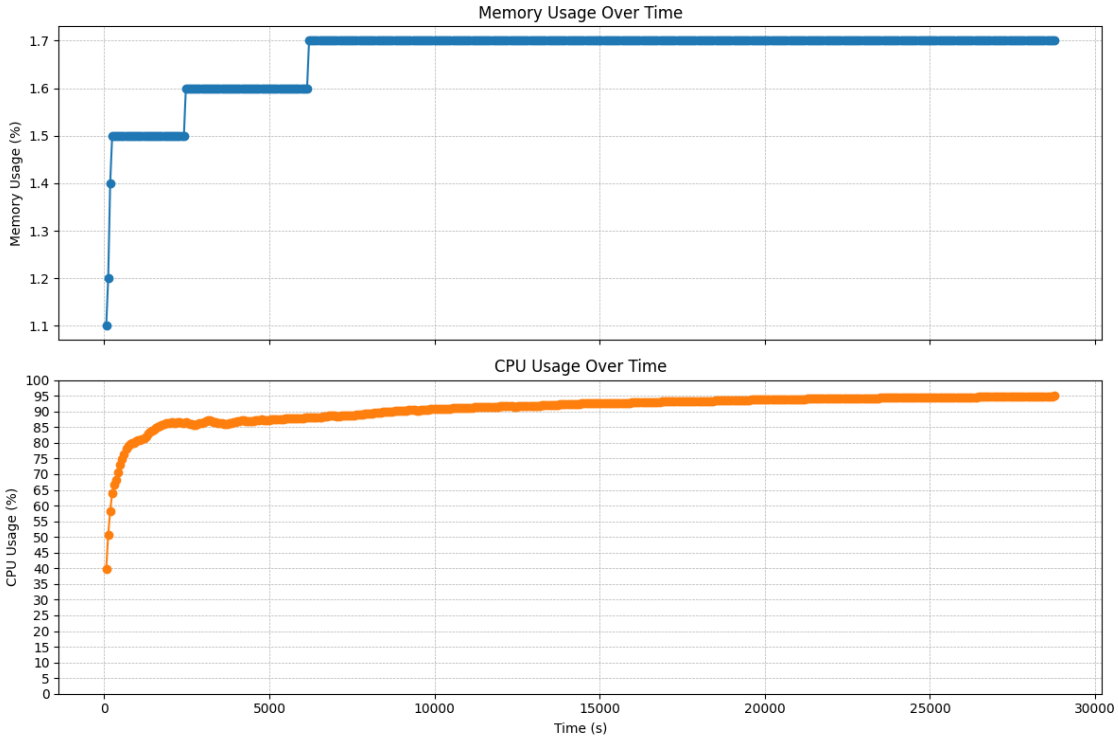


Figure 4.6: CPU and memory usage measurements of Experiment 5

Table 4.7 Execution Time Statistics for Experiment 6 based on 10 Runs

Category	Queries	AQET (ms)	Average Std Dev (ms)
SSP	16	1805.525	667.55
SSK	1	1261.50	0
SSO	13	42576.84	35180.06
MS	23	95484.07	33765.86
SQ	10	45554.03	40100.24
IQ	30	69551.58	29539.21
CQ	13	18284.25	2188.07

4.1.7 Experiment 7: Custom Bitmap Indexing for Source-Specific Datasets

This experiment was conducted using a similar setup to Experiment 2, and therefore its results are compared against those of that experiment. Contrary to expectations, the results were unsatisfactory, with an increase in (AQET) values were observed across all query categories.

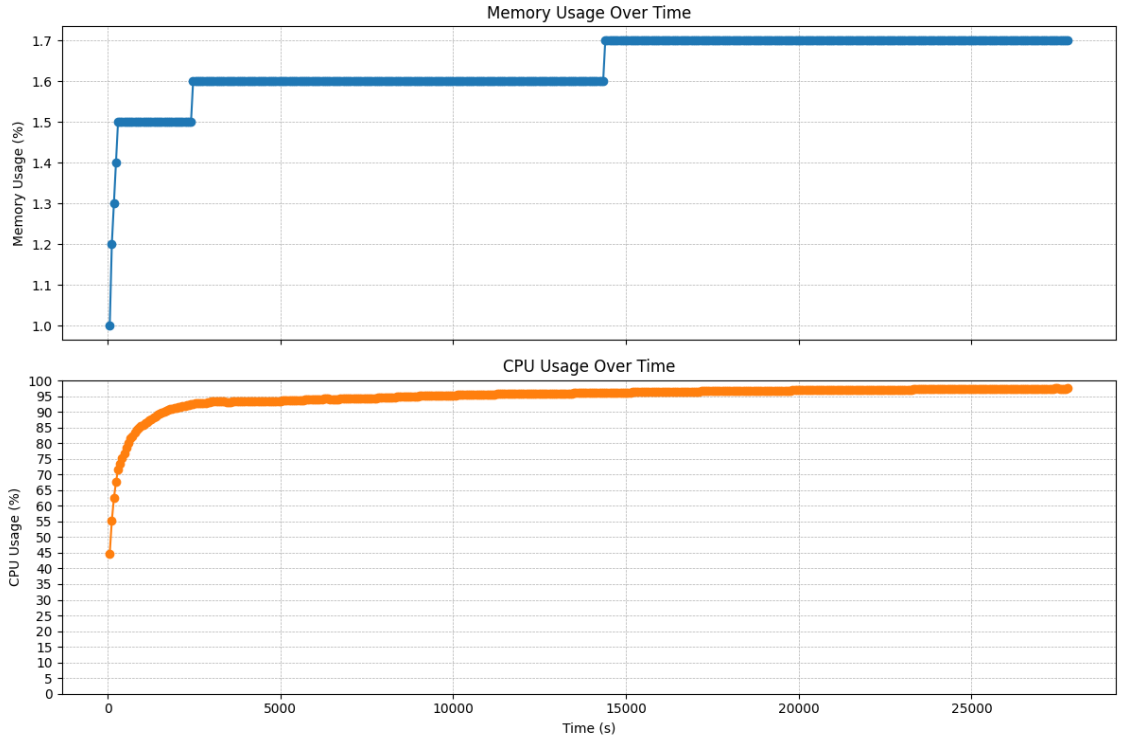


Figure 4.7: CPU and memory usage measurements of Experiment 6

This performance degradation may be attributed to several factors, such as the increased index size resulting from the newly added bitmap index, overall growth in storage size, which the impact of this will be explained later, or potential negative impacts on Virtuoso's query optimizer, specifically, inaccurate cardinality estimates caused by the new index structure. The only improvement that is seen in this experiment is more consistent results in the IQ category.

Table 4.8 Execution Time Statistics for Experiment 7 based on 10 Runs

Category	**Queries**	**AQET (ms)**	**Average Std Dev (ms)**
SSP	26	44846.23	4463.4157
SSK	3	1154.27	500.07
SQ	8	995.58	833.19
IQ	20	58050.86	27888.19
CQ	1	483.1	0

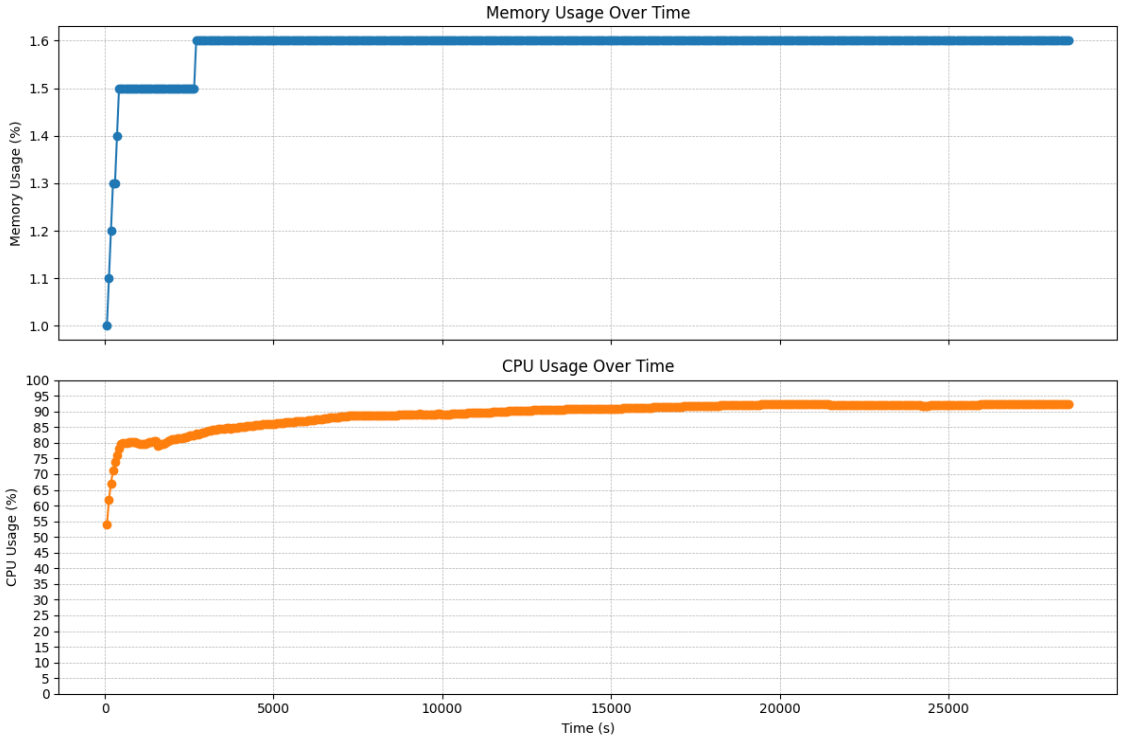


Figure 4.8: CPU and memory usage measurements of Experiment 7

4.1.8 Experiment 8: Enforcing Explicit Join Order in SPARQL Queries

As described in the methodology section, this experiment was conducted similarly to Experiment 3, with the only difference being the inclusion of the `sql:select-option "order"` pragma at the beginning of each query. As previously discussed, Virtuoso's query optimizer does not always produce the most efficient execution plan, either in terms of response time or from the perspective of the knowledge graph designer.

When comparing the results in Table 4.9 with those in Table 4.4, it becomes evident that the use of the pragma did not lead to an overall improvement in AQET in this specific case.

However, since pragmas influence query execution rather than the graph structure itself, it would be premature to dismiss this method entirely. With thorough query analysis and a deeper understanding of specific query execution plans, pragmas can be a valuable tool for performance tuning in targeted scenarios.

Table 4.9 Execution Time Statistics for Experiment 8 based on 10 Runs

Category	Queries	AQET (ms)	Average Std Dev (ms)
SSP	16	9703.2	16872.93
SSK	1	1832.1	0
SSO	13	10580.74	6193.90
MS	23	141797.47	113046.41
SQ	10	1575.65	1145.86
IQ	30	110202.3	98479.12
CQ	13	18011.47	24847.19

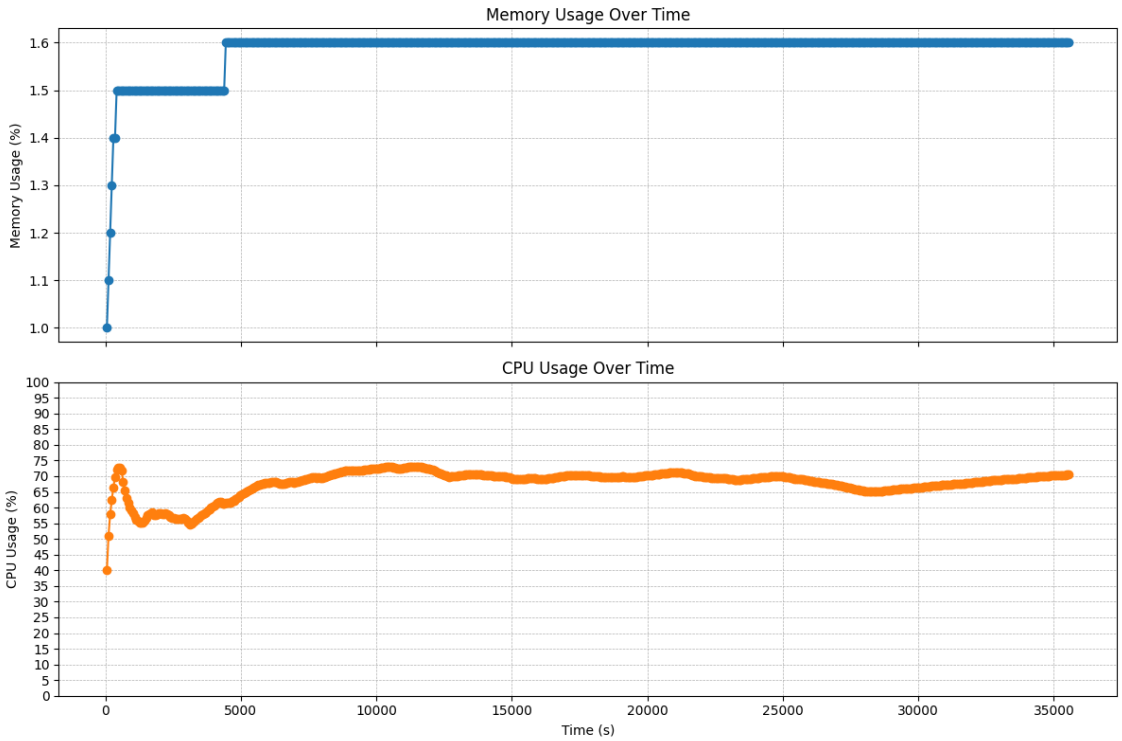


Figure 4.9: CPU and memory usage measurements of Experiment 8

A closer look at the results reveals several noteworthy insights. Across all query categories, most experiments showed performance improvements compared to the baseline. However, the degree of improvement varied, with distinct patterns emerging based on the query type and the specific optimization strategy employed.

For Multisource (MS) queries, the fastest query response was surprisingly still achieved in Ex-

Table 4.10 Peak CPU and Memory Utilization Across Experiments

Experiment ID	CPU Peak (%)	Memory Peak (%)
1 (HPC)	85.8	1.7
1 (GCE)	68	64.2
2 (OpenML)	93.8	1.6
2 (PwC)	45	0.8
3	92.4	1.6
4	96.1	1.7
5	95.0	1.7
6	97.5	1.7
7 (PwC)	97.6	1.3
8	73.1	1.6

periment 1, followed by Experiment 6, which employed semantic grouping of predicates. As for the consistency of the query results, average standard deviation values show that Experiment 1 performed poorly compared to Experiment 6, which also outperformed Experiments 4 and 5. Interestingly, Experiment 6 also yielded the lowest AQET and average standard deviation values for SSP queries, making it a suitable choice for this category.

In contrast, Experiments 3, which used source-based named graph partitioning, and 5 produced the best results for SSK queries.

It is worth noting that the results for SSP and SSK queries in Experiments 2 and 7 are only compared relative to each other, rather than to the other experiments. This is because the size and structure of the knowledge graph in those experiments differ significantly, making direct comparison invalid. Nevertheless, the values obtained in these measurements are valuable information demonstrating how dataset isolation impacts the query performance.

In the case of SSO queries, both Experiment 3 and Experiment 8 showed an 8-fold reduction in AQET compared to Experiment 1. Similarly, for Simple Queries (SQ), the best results were also obtained in Experiment 3 and 8.

One particularly interesting observation is that Intermediate Queries (IQ) almost consistently showed higher AQET values than Complex Queries (CQ). At first glance, this seems counterintuitive. However, it highlights the fact that the categorization of queries (Simple, Intermediate, and Complex) is based on structural and logical features such as the number of triple patterns, the presence of `FILTER` and `OPTIONAL` clauses. These structural classifications do not necessarily correlate with execution time, which is more strongly influenced by factors such as data distribution, indexing strategies, and Virtuoso's query optimization

behavior. The only exception to what is discussed above is Experiment 4 showing a lot higher values in CQ category, but rather promising results in variance.

Experiments 1 and 4 yielded the best results for Intermediate Queries (IQ), with Experiment 4 showing a lot more consistency, making it a suitable approach for this category. For Complex Queries (CQ), performance was relatively consistent across all experiments, with the best results observed in Experiments 1 and 3. Yet, in consistency ranking, Experiment 6 outperforms both.

The last crucial observation that is made in the execution time statistics of the experiments is that there is a pattern in individual query execution times in the first run of the experiment. The values are relatively higher than the subsequent runs, the reason behind this is explained in the next section. Table 4.11 is created based on the average of 9 runs, while totally ignoring the first run. And comparing this table to the Table 4.4, major improvements can be seen across all categories, especially in average standard deviation values, which further proves that the first run measurements are usually outliers. At first glance, this might be viewed as a huge disadvantage, but fortunately, there are ways to counter this pattern, which is explained in Section 4.1.9.

Table 4.11 Execution Time Statistics for Experiment 3 based on 9 Runs, excluding the first run

Category	**Queries**	**AQET (ms)**	**Average Std Dev (ms)**
SSP	16	1638.48	707.44
SSK	1	1104.44	0
SSO	13	7207.93	557.47
MS	23	98408.25	62749.64
SQ	10	892.13	182.54
IQ	30	74521.82	54930.5
CQ	13	10756.7	2035.53

Overall, the analysis of query execution times reveals that certain query categories experienced clear improvements in both speed and consistency. However, for others, the experiments that yielded the fastest results did not necessarily offer the most consistent performance. These findings can serve as a practical reference point, depending on whether speed or stability is prioritized in a given use case.

In terms of hardware usage within the HPC environment, most experiments exhibited a typical exponential saturation pattern in CPU usage over time. The notable exception was Experiment 8, which showed an initial sharp drop in CPU usage before stabilizing, an anomaly not seen in the other experiments.

Table 4.10 summarizes the peak CPU and memory usage for each experiment. As shown, Experiment 8 had the lowest CPU and memory usage, despite processing the entire dataset of 1.44 billion triples. Interestingly, this same experiment also achieved the best AQET values in simple query (SQ) category. The second most efficient in terms of CPU usage was Experiment 1. Overall, hardware metrics across all experiments were relatively similar, likely because the same set of queries, or slightly modified versions, was executed in each case.

By only looking at CPU usage measurements at HPC, it is impossible to identify the queries that required the lowest or highest resources, as mentioned before, it becomes somewhat stabilized with no significant fluctuations or recurring peaks. But if we look at Figure 4.2, the plot that was constructed from GCE Experiment, it can be observed that there is an almost periodic pattern throughout 10 runs. Using this, we can deduct which query (or queries) required the most resource. Table 4.12 shows the local maxima occurrences by time and CPU usage in percentage. With this table and also accounting for the execution order of the queries for Experiment 1 in GCE environment as shown in Listing 4.1, the occurrences of local maxima happens at every run when it is executing query #30 and the local minima at queries #12 and #13.

Listing 4.1: The order of the queries

```
query_order = [
    1, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2, 20, 21, 22, 23,
    24, 25, 26, 27, 28, 29, 3, 30, 31, 32, 33, 34, 35, 36, 37, 38,
    39, 4, 40, 41, 42, 43,44, 45, 46, 47, 48, 49, 5, 50, 51, 52,
    53, 6, 7, 8, 9
]
```

Table 4.12 CPU Usage Maxima (Peaks)

Time (s)	**CPU Usage (%)**
2700	65.89
6300	58.35
9600	64.06
13200	68.38
17100	63.74
21000	62.21
24600	65.33
28500	60.45

And for the memory usage, the graph follows the shape of a step function, similar in every experiment, and it seems to increase until one full run is completed then only increase in very

small increments (this is also supported by `monitor.log`[1] files, where file sizes continue to grow incrementally but without dramatic changes).

Lastly, it should be noted that due to the Kaggle query set being smaller and completing quickly, it was excluded from being displayed in Figures 4.3 and 4.8.

Finally, it is worth noting that the results presented in this discussion were based on grouped query categories, as mentioned before, as presenting individual results for each query would have been too overwhelming. However, the complete set of query response times, averages, standard deviations, and coefficients of variation (CV) can be found in the GitHub repository[2], organized under their respective experiment directories. Additionally, more detailed hardware usage statistics are also available there. While this thesis focused on analyzing results at the category level, further insights could be uncovered by examining individual query performance differences, which will reveal more nuanced behavior and optimization effects.

4.1.9 Remarks

This section outlines several noteworthy observations and actionable insights that may serve as useful reference points for future research or implementation. It is not intended as a "future work" section, but rather as a collection of notes, which is common in most graph databases. This might also become more relevant depending on the direction MLSea-KG evolves over time.

Cache Warming:

First of all, as mentioned before, the query execution time measurements at the first run of the experiment are generally the slowest. The reason behind this is that after restarting a database or launching the database, the cache is almost empty. Thus, all queries have to read from the disk, which is one of the slowest functions of all DBMSs. But as more data is queried and active memory "fills up", it shifts from a "cold state" to a "warm state". Depending on the query set that is used to warm up the cache, it can sometimes take even longer than 30 minutes to prepare the DBMS to be in the working state. This is where cache warming becomes critical. **Cache warming is a technique used to preload a cache with frequently accessed data or resources before they are actually requested by users**[3]**.** And historically, the usual way to make that shift from cold state to warm state is to simulate user activity with pre-prepared queries, also referred to as manual preloading. So, cache warming plays a critical role in improving the response time and, most importantly, the consistency of the results. The latest versions of Virtuoso have SQL functionalities to record and restore a database's working instance:

- `ws_save()` saves the current set of database pages held in RAM to a file located in the server's working directory.

[1] https://github.com/eltajj18/Knowledge-Graph-Partitioning/blob/main/Experiments/Experiment%203/monitor.log
[2] https://github.com/eltajj18/Knowledge-Graph-Partitioning/tree/main/Experiments
[3] https://www.geeksforgeeks.org/what-is-cache-warming/

- `ws_restore()` restores the previously saved working set[4].

Using these functions to load identified critical data into the cache can be beneficial.

Data Striping:

Enabling data striping in Virtuoso distributes the database across multiple segments, each containing stripes. Striping enhances performance by spreading data across different file system locations, while segmentation facilitates expansion and overcomes file size limits. **To allow for database growth when a file system is near capacity or near the OS file size limitation, new segments can be added on the same or other file systems[5].**

Crucially, striping only improves performance across multiple physical devices; striping across partitions on a single device often harms performance by increasing disk seek times. As the MLSea-KG accumulates more triples, data striping may become necessary or advantageous.

Storage Space:

If you are using an `ext2fs` or `ext3fs` file system (traditional file systems for Linux), it is recommended to keep at least 20% of the disk space free. When the disk gets close to full, it may allocate space for database files inefficiently, which can noticeably slow down disk access. This isn't specific to Virtuoso, but rather a general best practice for any database-style application that performs random access on large files. Free space can be used to leave a portion of the table space or index empty and available to store newly added data. The specification of free space in a table space or index can reduce the frequency of reorganization, reduce contention, and increase the efficiency of access and insertion[21].

Managing Public Web Service Endpoints:

Public web service endpoints have proven to be sources of especially bad queries. Unlike local application developers, who can consult database administrators and use ISQL to fine-tune execution plans, external users typically lack knowledge of the database configuration or the expertise to write efficient queries. A common issue is that public endpoints frequently receive queries that do not specify the target graph. This happens because many of these queries were originally designed for generic triple stores without a specific graph context.

According to the Virtuoso guidelines, if a web service grants access to only one graph (or a small number of known graphs), it is advised to configure the endpoint by inserting an entry into `DB.DBA.SYS_SPARQL_HOST`. The rationale is that if a client explicitly defines a default graph or uses named graphs and group graph patterns, they are likely more knowledgeable and will submit better formed queries[6].

Listing 4.2: SYS_SPARQL_HOST Table

```
create table DB.DBA.SYS_SPARQL_HOST (
  SH_HOST        varchar not null primary key, -- host mask
  SH_GRAPH_URI   varchar, -- default graph uri
```

[4] https://community.openlinksw.com/t/controlling-the-database-working-set-in-openlink-virtuoso/1044
[5] https://docs.openlinksw.com/virtuoso/ch-server/#ini_striping
[6] https://docs.openlinksw.com/virtuoso/rdfperfindexes/

```
  SH_USER_URI    varchar, -- reserved for any use in applications
  SH_BASE_URI varchar, -- for future use to set BASE in sparql
  -- queries. Should be NULL for now.
  SH_DEFINES long varchar,  - additional defines for requests
  PRIMARY KEY (SH_HOST)
)
```

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis explored how different optimization strategies affect the performance and hardware efficiency of MLSea-KG, a large-scale knowledge graph. This study aimed to answer the main research question: How do different optimization strategies affect the MLSea-KG query performance and resource efficiency? Different methods and experimental setups were explored using manually designed SPARQL queries that target different types of entities and relationships from each source to answer this question. The findings demonstrated interesting insights into hardware usage and average query execution times. The results showed that no single method works best for all queries; performance depends on the structure of the query and the graph.

So, a critical contribution of this paper was to present tailored optimizations and partitionings that were applied thoughtfully by the characteristics of the specific knowledge graph in a particular environment (in our case, Virtuoso) and the results they produce in a clear fashion.

Second, I outlined practical optimization tips that can serve as helpful reference points, which are common in most RDF engines. This study can help developers and researchers working with large graphs understand what works and what does not, as well as how to make better performance decisions based on the characteristics of their data.

5.2 Limitations

The empirical results reported in this study should be considered in the light of some limitations. **The first is the effect of the storage growth**. During the execution of Experiments 1 through 8, previously loaded data in the Virtuoso triple store was not deleted between runs due to the reproducibility of experiments later, if necessary. As a result, the size of the `virtuoso.db` file continuously increased with each experiment. While each experiment worked with distinct datasets, the cumulative data may have affected query performance

due to larger disk size, potential increases in indexing overhead, and reduced buffer pool efficiency. This growth in the database may have contributed to progressively slower response times, particularly in the later experiments (The experiment numbers are not arranged in chronological order; the order can be found in the GitHub repository, in the naming of measurement_*.md files under each experiment directory[1]). Listing 5.1 shows the final storage size and how much it is used. **The second limitation concerns the order in which queries were executed.** Depending on the order of queries that are executed, one can get different results, which Virtuoso's caching mechanism can explain. However, this can also be considered realistic, as the user who sends queries to the MLSea-KG knowledge graph is not concerned with how Virtuoso caches data.

Listing 5.1: Final Storage Usage

```
~ $ myquota

file system $VSC_HOME
    Blocks: 31096K of 3072M
    Files: 1368 of 100k
file system $VSC_DATA
    Blocks: 4262M of 76800M
    Files: 50713 of 10000k
file system $VSC_SCRATCH
    Blocks: 1.216T of 2T
```

5.3 Future Work

A further study can be conducted using different RDF Engines, such as GraphDB or Koral (more examples can be found in Figure 2.1) with the same type of hardware configuration. Such a study would contribute to analyzing the performance difference with different types of partitioning. Another future study can focus on partitioning with Virtuoso's Enterprise Edition, which allows for cluster operations, RDF views, higher flexibility in CPU affinity, and so on.

[1] https://github.com/eltajj18/Knowledge-Graph-Partitioning/tree/main/Experiments

Bibliography

[1] E. W. Schneider, "Course modularization applied: The interface system and its implications for sequence control and data analysis.", 1973.

[2] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. D. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, *et al.*, "Knowledge graphs", *ACM Computing Surveys (Csur)*, vol. 54, no. 4, pp. 1–37, 2021.

[3] T. A. Ayall, H. Liu, C. Zhou, A. M. Seid, F. B. Gereme, H. N. Abishu, and Y. H. Yacob, "Graph computing systems and partitioning techniques: A survey", *IEEE Access*, vol. 10, pp. 118 523–118 550, 2022.

[4] I. Dasoulas, D. Yang, and A. Dimou, "MLSea: A Semantic Layer for Discoverable Machine Learning", in *European Semantic Web Conference*, Springer, 2024, pp. 178–198.

[5] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent advances in graph partitioning*. Springer, 2016.

[6] A. Akhter, A.-C. Ngomo Ngonga, and M. Saleem, "An empirical evaluation of rdf graph partitioning techniques", in *Knowledge Engineering and Knowledge Management: 21st International Conference, EKAW 2018, Nancy, France, November 12-16, 2018, Proceedings 21*, Springer, 2018, pp. 3–18.

[7] M. Saleem *et al.*, "Storage, indexing, query processing, and benchmarking in centralized and distributed rdf engines: A survey", *Authorea Preprints*, 2023.

[8] A. Akhter, M. Saleem, A. Bigerl, and A.-C. N. Ngomo, "Efficient rdf knowledge graph partitioning using querying workload", in *Proceedings of the 11th Knowledge Capture Conference*, 2021, pp. 169–176.

[9] R. Mayer and H.-A. Jacobsen, "Hybrid edge partitioner: Partitioning large power-law graphs under memory constraints", in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1289–1302.

[10] K. Seo, S. Hyun, and Y.-H. Kim, "An edge-set representation based on a spanning tree for searching cut space", *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 4, pp. 465–473, 2014.

[11] Y. Yoon and Y.-H. Kim, "Vertex ordering, clustering, and their application to graph partitioning", *Applied Mathematics & Information Sciences*, vol. 8, no. 1, p. 135, 2014.

[12] M. Li, H. Cui, C. Zhou, and S. Xu, "Gap: Genetic algorithm based large-scale graph partition in heterogeneous cluster", *IEEE Access*, vol. 8, pp. 144 197–144 204, 2020.

[13] Y. Akhremtsev, P. Sanders, and C. Schulz, "High-quality shared-memory graph partitioning", *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2710–2722, 2020.

[14] J. Leskovec and A. Krevl, *Snap datasets: Stanford large network dataset collection. snap*, 2014.

[15] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks", in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, 2007, pp. 29–42.

[16] J. Kunegis, "Konect: The koblenz network collection", in *Proceedings of the 22nd international conference on world wide web*, 2013, pp. 1343–1350.

[17] M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood, and A.-C. Ngonga Ngomo, "How representative is a sparql benchmark? an analysis of rdf triplestore benchmarks", in *The World Wide Web Conference*, 2019, pp. 1623–1633.

[18] O. Erling and I. Mikhailov, "Virtuoso: Rdf support in a native rdbms", in *Semantic web information management: a model-based perspective*, Springer, 2009, pp. 501–519.

[19] M. Saleem, Q. Mehmood, and A.-C. Ngonga Ngomo, "Feasible: A feature-based sparql benchmark generation framework", in *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I 14*, Springer, 2015, pp. 52–69.

[20] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of rdf data management systems", in *The Semantic Web–ISWC 2014: 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I 13*, Springer, 2014, pp. 197–212.

[21] C. Mullins, *Database administration: the complete guide to practices and procedures*. Addison-Wesley Professional, 2002.