Master’s Thesis

Evaluation of Physical Structures for Graph Database Usage

Author:
Sameh Manaa

November 12, 2018

Advisors:
M.Sc. Gabriel Campero Durand
Databases and Software Engineering Working Group

Prof. Dr. rer. nat. habil. Gunter Saake
Databases and Software Engineering Working Group
Manaa, Sameh:
*Evaluation of Physical Structures for Graph Database Usage*
Abstract

Graph database management systems have emerged as specialized databases that can store and process relationships among the data entities in an efficient way. Graph databases are equipped with many components that work together in order to handle even the very complex relationships that can exist in a network of connected data. One of the main components of a graph database is its storage system. The storage system of a graph database is utilizing a set of graph data structures in order to efficiently manage the storage of graph data.

For the construction of a graph storage system; many graph data structures are available as alternatives to choose from. Each graph data structure has its unique way in representing graph data. The different representations of graph data implies a difference in the complexity of the data storage and retrieval operations performed on each of the graph data structures. Choosing the right graph data structure for the storage of graph data is challenging without a comprehensive benchmarking of the many available graph data structures.

In this thesis, we study the way each graph data structure is representing the graph data stored from logical and physical perspective. We evaluate the performance of the graph data structures according to a set of well-defined dimensions. Our evaluation of the graph data structures is based on an in-memory implementation of the data structures.

By the end of this thesis, we will answer evaluation questions concerning the scalability, loading, and query performance of a set of graph data structures. We believe the answers to those questions will aid graph database designers in their choices for the graph data structure to use.
Acknowledgements

It is a pleasure to finally get this work done. I would like to thank my advisors Prof. Gunter Saake, and M.Sc. Gabriel Campero Durand for their support, guidance, and supervision throughout my journey in writing this thesis.

Also, I would like to thank my family and friends for the continuous support, and encouragement they have shown me during my entire master’s studies, and especially during my thesis writing.
Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used.

Magdeburg, November 12th 2018

__________________________
Sameh Manaal
## Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>xiv</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>xv</td>
</tr>
<tr>
<td>List of Code Listings</td>
<td>xvii</td>
</tr>
</tbody>
</table>

### 1 Introduction

#### 2 Background

2.1 Graphs .................................................. 5
   2.1.1 What is a Graph? .................................. 6
   2.1.2 Graph Types ........................................ 6
2.2 Graph Data Models .......................... 8
   2.2.1 Property Graph Model .......................... 9
   2.2.2 Resource Description Framework Model (RDF) .. 9
2.3 Graph Data Structures .................. 10
   2.3.1 Graph Topology .................................. 10
      2.3.1.1 Adjacency Matrix .......................... 11
      2.3.1.2 Compressed Sparse Row (CSR) .............. 12
      2.3.1.3 Adjacency List ............................ 12
   2.3.2 Graph Properties ............................ 14
      2.3.2.1 Universal Table .......................... 14
      2.3.2.2 Emerging Schema ......................... 14
      2.3.2.3 Nested Key-Value Store .................. 15
2.4 Data Structures in C++ ............... 17
   2.4.1 std::vector ...................................... 17
   2.4.2 std::map ......................................... 17
   2.4.3 std::unordered_map ............................ 18
   2.4.4 std::pair ....................................... 18
2.5 Summary ............................................. 19

### 3 Physical Design of Graph Data Structures

3.1 Evaluation Questions ................................. 22
3.2 Graph Topology Structures ................ 24
   3.2.1 Adjacency Matrix ............................... 24
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.2 Compressed Sparse Row (CSR)</td>
<td>25</td>
</tr>
<tr>
<td>3.2.3 Adjacency List</td>
<td>25</td>
</tr>
<tr>
<td>3.3 Graph Properties Structures</td>
<td>26</td>
</tr>
<tr>
<td>3.3.1 Universal Table</td>
<td>27</td>
</tr>
<tr>
<td>3.3.2 Emerging Schema</td>
<td>28</td>
</tr>
<tr>
<td>3.3.3 Nested Key-Value Store</td>
<td>30</td>
</tr>
<tr>
<td>3.4 Parallel Graph Structures</td>
<td>31</td>
</tr>
<tr>
<td>3.4.1 Parallel Adjacency List</td>
<td>31</td>
</tr>
<tr>
<td>3.4.2 Parallel Nested Key-Value Store</td>
<td>32</td>
</tr>
<tr>
<td>3.5 Partitioned Graph Topology</td>
<td>33</td>
</tr>
<tr>
<td>3.6 Summary</td>
<td>34</td>
</tr>
<tr>
<td>4 Evaluation Environment</td>
<td>35</td>
</tr>
<tr>
<td>4.1 Technical Environment</td>
<td>36</td>
</tr>
<tr>
<td>4.2 Evaluation Dataset</td>
<td>38</td>
</tr>
<tr>
<td>4.3 Data Loader</td>
<td>39</td>
</tr>
<tr>
<td>4.3.1 Batch Loading</td>
<td>39</td>
</tr>
<tr>
<td>4.3.2 Parallel Loading</td>
<td>40</td>
</tr>
<tr>
<td>4.4 Queries</td>
<td>40</td>
</tr>
<tr>
<td>4.4.1 Definition: Query #1</td>
<td>41</td>
</tr>
<tr>
<td>4.4.2 Definition: Query #2</td>
<td>42</td>
</tr>
<tr>
<td>4.4.3 Definition: Query #3</td>
<td>44</td>
</tr>
<tr>
<td>4.5 Summary</td>
<td>44</td>
</tr>
<tr>
<td>5 Evaluation: Scalability and Data Loading</td>
<td>47</td>
</tr>
<tr>
<td>5.1 Scalability</td>
<td>49</td>
</tr>
<tr>
<td>5.1.1 Scalability: Graph Topology Structures</td>
<td>49</td>
</tr>
<tr>
<td>5.1.2 Scalability: Graph Properties Structures</td>
<td>51</td>
</tr>
<tr>
<td>5.2 Batch Data Loading</td>
<td>54</td>
</tr>
<tr>
<td>5.3 Parallel Data Loading</td>
<td>57</td>
</tr>
<tr>
<td>5.4 Summary</td>
<td>60</td>
</tr>
<tr>
<td>6 Evaluation: Queries</td>
<td>61</td>
</tr>
<tr>
<td>6.1 Evaluation: Query #1</td>
<td>62</td>
</tr>
<tr>
<td>6.2 Evaluation: Query #2</td>
<td>63</td>
</tr>
<tr>
<td>6.3 Evaluation: Query #3</td>
<td>64</td>
</tr>
<tr>
<td>6.4 Summary</td>
<td>66</td>
</tr>
<tr>
<td>7 Related Work</td>
<td>67</td>
</tr>
<tr>
<td>7.1 Graph Structures Scalability</td>
<td>67</td>
</tr>
<tr>
<td>7.2 Loading Techniques</td>
<td>68</td>
</tr>
<tr>
<td>7.3 Query Performance</td>
<td>68</td>
</tr>
<tr>
<td>8 Conclusion and Future Work</td>
<td>69</td>
</tr>
<tr>
<td>8.1 Conclusion</td>
<td>69</td>
</tr>
</tbody>
</table>
Contents

8.2 Future Work ............................................................... 70

Bibliography ................................................................. 73
List of Figures

2.1 An example of a graph $G$ [Har69] ........................................ 6
2.2 An example of different graph types [RN10] .............................. 7
2.3 A diagram of a property graph [RN10] ................................. 8
2.4 An example of an RDF graph that consists of 9 triples [NALZ14] ... 10
2.5 Logical representations of the topology of directed graph $G$ [CLRS09] 11
2.6 Universal table and emerging schema representations of property graph $G$. 13
2.7 A nested key-value store composed of rows and columns [LH11] .... 16

3.1 Physical representations of the topology of vertex-labeled directed graph $G$. 23
3.2 An attributed directed graph $G$ [PLB14] ................................. 26
3.3 Universal table physical representation of graph $G$ shown in (Figure 3.2) 27
3.4 Emerging schema physical representation of graph $G$ shown in (Figure 3.2) 29
3.5 Nested key-value store physical representation of graph $G$ in (Figure 3.2). 30
3.6 Physical representation of a partitioned adjacency list of graph $G$. .... 33

4.1 LDBC - SNB data model [AAB+17] ................................. 37
4.2 Query #1 pattern [AAB+17] ........................................ 42
4.3 Query #2 pattern [AAB+17] ........................................ 43

5.1 Effect of increasing data size on the scalability of graph topology structures. 48
5.2 Effect of increasing data size on the scalability of graph properties structures. 52
5.3 Effect of increasing batch size on the loading time of graph structures .... 55
5.4 Degree of parallelism effect on the loading time of parallel adjacency list and parallel nested key-value store. ........................ 58
List of Figures

6.1  Response time of Query #1. ........................................... 62
6.2  Response time of Query #2. ........................................... 63
6.3  Response time of Query #3. ........................................... 65
List of Tables

4.1 The generated datasets size and parameters .......................... 38

5.1 Evaluation results of graph topology structures scalability .... 48

5.2 Evaluation results of graph properties structures scalability ... 52

5.3 Evaluation results of batch loading time of graph structures in seconds ... 55

5.4 The change in the number of buffer flushes and the percentage of cache misses with the different batch sizes .................. 56

5.5 Evaluation results for time of parallel loading of the parallel adjacency list and the parallel nested key-value store in seconds ... 58
List of Code Listings

4.1 Function used for batch loading of data files. 39
1. Introduction

Graph databases are characterized by being able to store and query a network of connected data with a performance that exceeds other non-graph databases (e.g., relational databases). Graph databases are flexible in modeling graph data, allowing for the evolve of the model by adding new vertices, edges, or properties as demanded without compromising the stability or performance of the system. Native graph databases differentiate themselves from non-native graph databases by using data structures that are specifically designed for the storage of graph data, in contrast to the use of other kinds of databases as the back-end storage system which is the case in the non-native graph databases [RWE13].

In a survey performed by (Sahu et al.), many participants have reported the use of graph databases to store and process graph data that is huge in size (more than 1 billion edge). The participants in the survey have reported a wide variety of computations that they perform on the stored data. The computations the users performed ranged from simple traversal queries and up to complex analytical queries. However, the survey participants have reported the scalability of the database to manage bigger sizes of data and perform more complex computations as their top challenge. Loading graph data or performing traversal queries are considered a very time consuming operations when done on a graph with large size [SMS+17].

The scalability and query performance issues of a graph database are indicators of the disability of the database to handle the growing data size. A major part of the performance of a graph database lies in the ability of its storage system to accommodate graph data with larger size without hindering the accessibility of the data. The design choices of the storage system include the proper choice of a graph data structure for the storage of graph data. Each graph data structure is representing graph data in a distinct way and hence the difference in the performance of each graph data structure in storing new data or reaching existing ones.
Few work has been done on evaluating the alternative choices of graph data structures which will aid graph database designers in choosing the proper graph data structure that fits the user requirements. Current work done in this area has only evaluated a subset of the available graph data structures. In their work, \textit{(Wheatman et al.)} have evaluated only adjacency list and compressed sparse row (CSR) performance in loading and query performance leaving out other graph data structure like adjacency matrix \cite{WX18}. Other work done by \textit{(Then et al.)}, has evaluated various techniques of parallel graph loading \cite{TKKN16}.

Both work \cite{WX18, TKKN16} have presented evaluation of loading and querying of graph topology structures which are used to represent the edges between the graph vertices. We found no previous work on evaluating the performance of graph properties structures such as (universal table, emerging schema, or nested key-value store) against each other. Graph properties structures are used to represent the properties associated with each graph element (vertices and edges) in the property graph model. The property graph model (PGM) is the most widely implemented graph data model in graph databases. Also, we found no work that covers the evaluation of graph data structures in the presence of a multi-graph data. The multi-graph property allows more than one edge between the same two vertices. Multi-graphs and graph properties are two important characteristics of the property graph model \cite{RWE13}. Lastly, we found no work that evaluates the performance of graph data structure when loaded in batches of data neither the impact of the batch-size on the performance of the loading process.

In this thesis, we aim to present a comprehensive evaluation of the major graph data structures that have been developed specifically for the storage and processing of graph data. Our contribution in this thesis is summarized in the following points:

- **Graph Data Structures:**
  We study the major available graph data structures from two perspectives. First, we study the graph structures logical design and the characteristics of each graph structure. Next, we present our physical design of the different graph structures which we have constructed using the data structures offered as part of the \textit{Standard Template Library} (STL) of the \texttt{C++} programming language. For evaluation purposes, we implement all the data structures for in-memory processing of data with no involvement of disk persistent storage.

- **Scalability of Graph Structures:**
  We present an extensive evaluation of the scalability of the different graph structures. We evaluate the scalability of a graph structure by measuring the amount of memory storage as well as the time taken to store graph data. We use a different sizes of a graph dataset generated by the \textit{LDBC} bench-marking framework \cite{Bon13}. 
• Evaluation of Loading Techniques:
We present an evaluation of two techniques for graph data loading. First, we evaluate a batch loading technique and the impact of the batch-size on the performance of the loading process. Next, we evaluate a parallel loading technique and the change in data loading time that loading the data in parallel could introduce in comparison to sequential loading.

• Evaluation of Query Execution:
We present the definition of a set of queries which compute centrality or perform pattern-matching on the loaded graph data. We evaluate the time taken to execute each of the given queries on the data loaded in each of the graph structures.

The thesis is consisted of the following chapters:

• Background:
In (Chapter 2), we present the necessary background knowledge concerning topics covered in this thesis.

• Physical Design of Graph Data Structures:
In (Chapter 3), we present the set of evaluation questions we are going to answer in this thesis as well as the physical design of the graph data structures, we are going to evaluate.

• Evaluation Environment:
In (Chapter 4), we introduce the details of every component in the evaluation environment which we will use to evaluate our implemented graph data structures.

• Evaluation: Scalability and Data Loading:
In (Chapter 5), we present the first part of the evaluation results for the experiments we conducted on the graph data structures concerning the scalability of the graph structures and the performance of data loading.

• Evaluation: Queries:
In (Chapter 6), we present the second and last part of the evaluation results for the experiments we conducted to evaluate the performance of the graph structures in query execution scenarios.

• Related Work:
In (Chapter 7), we present work by other researchers which we see as related to the work done in this thesis.

• Conclusion and Future Work:
In (Chapter 8), we draw a conclusion of the thesis as well as suggesting research points that can be a further extension of our work.
1. Introduction
2. Background

In this chapter, we present the necessary background knowledge concerning topics covered in this thesis. This chapter is consisted of the following sections:

- **Graphs:**
  In (Section 2.1), we make a definition for a graph according to the graph theory and discuss the different graph types.

- **Graph Data Models:**
  In (Section 2.2), we discuss the property graph model (PGM) and the resource description framework model (RDF), as two types of logical graph data models that are mostly used by state of the art graph databases.

- **Graph Data Structures:**
  We present the logical design of the data structures used by graph databases for the storage of graphs in (Section 2.3).

- **Data Structures in C++:**
  We present necessary background knowledge on a set of C++ data structures in (Section 2.4).

- **Summary:**
  Lastly, we make a summary of the main topics we discussed in this chapter in (Section 2.5).

2.1 Graphs

In this section, we discuss graphs and graph types. In (Section 2.1.1), we adopt a clear definition for graphs. A definition that is based on the graph theory. In (Section 2.1.2), we state the different types of graphs. A graph type is a factor that must be taken into consideration in the storage and retrieval methods of the graph.
2. Background

2.1.1 What is a Graph?

Graph theory is a mathematical topic that is focused on the study of graphs. A graph $G = (V,E)$ is defined as a finite nonempty set $V$ of vertices along with the set $E$ of edges. The set $E$ consists of unordered pairs of vertices in $V$, where an edge $x \in E$ is defined as a pair of vertices $x = \{u,v\}$ \cite{Har69}.

We define any two vertices $u \in V$ and $v \in V$ that forms any of the edges in $E$ as adjacent vertices. Similarly, two edges $x \in E$ and $y \in E$ are adjacent if they are formed of two pair of vertices where the two pair are sharing one common vertex. The adjacencies of a vertex $u$, is the set $K \subset V$ of vertices, where for each vertex $a \in K$, there is an edge $s \in E$ and $s = \{u,a\}$ \cite{Har69}.

In (Figure 2.1), we are showing an example of a graph $G$. In graph $G$, vertices $u$ and $v$ are adjacent while vertices $u$ and $w$ are not. Similarly, edges $x$ and $y$ are adjacent while edges $x$ and $z$ are not. The adjacencies of vertex $u$ in the graph is the set of vertices $\{p,v,q\}$ \cite{Har69}.

2.1.2 Graph Types

Graphs come in many types and shapes according to how rich they are with information. In (Figure 2.2), we show an example of different graph types. A short explanation of each graph type is included in the below list. Some of the below mentioned graph types can be be used together to form one graph \cite{RN10}.
2.1. Graphs

- **Simple Graph**: a graph that permits no loops and only binary edges are allowed.

- **Multi-graph**: a graph that allows the existence of more than one edge connecting the same two vertices.

- **Pseudo Graph**: a graph with reflexive edges

- **Weighted Graph**: a graph where a weight is assigned to edges to show the relationship strength.

- **Semantic Graph**: used in semantic networks to model the relationships between concepts.

- **Half-edge Graph**: an edge that is connected at one of its two ends to a vertex and on the other end connected to nothing.

- **Hyper-graph**: a graph that permits an edge to connect more than one vertex.

- **Directed Graph**: a graph where each edge is defined by an ordered pair of vertices (one of them is a source vertex and the other is a target vertex).

- **Undirected Graph**: a graph where edges are denoting symmetric relationships between edges.

- **Edge-labeled Graph**: a graph where edges are identified using a unique edge label or id.
2. Background

- **Vertex-labeled Graph**: a graph where vertices are identified using a unique vertex label or id.

- **Edge-attributed Graph**: a graph where descriptive properties are assigned to edges.

- **Vertex-attributed Graph**: a graph where descriptive properties are assigned to vertices.

- **Resource Description Framework (RDF) Graph**: a graph where vertices and edges are identified by Uniform Resource Identifiers (URI). The RDF is a standard that was issued by the World Wide Web consortium.

2.2 Graph Data Models

In Section 2.1, we presented the definition of graph and introduced the difference between the various graph types. In this section, we discuss two logical graph data models. First, we present the property graph model (PGM) in Section 2.2.1. Next, we present the resource description framework model (RDF) in Section 2.2.2.

![Figure 2.3: A diagram of a property graph](RN10)
2.2.1 Property Graph Model

Several graph database systems are supporting the property graph model. The property graph model is characterized with being directed, labeled, attributed and multi-graph. In Figure 2.3, we show an example diagram of a property graph model.

In property graph model, an edge is defined as an ordered pair of vertices. The first vertex in the pair is the source vertex of the edge and the second vertex in the pair is the target vertex of the edge.

Vertices and edges in a property graph model are labeled. A vertex label is an identifier for the vertex while an edge label is used to define the relationship type of an edge.

A vertex in the property graph model is described by properties (or attributes). The properties describing a vertex are a set of key-value pairs. An example of a property describing a vertex in Figure 2.3 is “name” = “alberto” property which is describing vertex “4”. Similarly, an edge can also be described by a set of key-value properties. An example of a property describing an edge in Figure 2.3 is “start” = “2007” property which is describing edge “attends”.

Property graph model is supporting also the creation of multi-graphs. In a multi-graph, more than two edges can be defined using the same pair of vertices and with the same direction, given that the edges have different labels [RN10, RWE15].

2.2.2 Resource Description Framework Model (RDF)

The World Wide Web consortium (W3C) has developed the Resource Description Framework (RDF) as a foundation for metadata exchanging and processing across the web. RDF is used mostly in building semantic webs. The RDF standard is used to express the metadata of the web documents. RDF is well supported by a model of triples. A triple is consisting of a resource, a property, and a value. The RDF model is characterized with being a vertex-labeled, edge-labeled, and directed graph [NALZ14].

A resource is anything that can be uniquely identified using a Unique Resource Identifier (URI). A web page is an example of a resource identified by its Unique Resource Locator (URL). A property is used to define a binary relationship between a resource and a value. A value itself can be a resource or a string of characters. A triple is considered as an RDF statement that is defined by the three elements (resource, property, value) [LS99, BLFM05].

In Figure 2.4, we show an example of an RDF graph that is consisting of 9 triples. The graph is describing the city of Leipzig and its mayor. An example of a triple in the graph is (“Leipzig”, “locatedIn”, “Saxony”) with “Leipzig” representing the resource, “locatedIn” representing the property, and “Saxony” representing the value in the triple.
Figure 2.4: An example of an RDF graph that consists of 9 triples [NALZ14].

2.3 Graph Data Structures

In (Section 2.2), we presented two logical graph data models. First, we presented the property graph model (PGM) in (Section 2.2.1). Next, we presented the resource description framework model (RDF) in (Section 2.2.2). In [PV17], the authors (Paradies et al.) categorized the data structures used to store graph data into two categories. First category is for the graph data structures used by graph databases for solely storing a directed graph topology. Second category is for the graph data structures used by graph databases for solely storing graph properties. In this section, we adopt the categorization presented in [PV17] and present the necessary background information of a selected subset of graph data structures. We first discuss three data structures (Adjacency Matrix, Compressed Sparse Row (CSR), Adjacency List) for graph topology in (Section 2.3.1). Next, we present three data structures (Universal Table, Emerging Schema, Nested Key-Value Store) for storing graph properties in (Section 2.3.2).

2.3.1 Graph Topology

The graph topology storage structures are meant to store the topology of a graph. We mean by the topology of a graph, the vertices of the graph identified by their labels and the directed edges connecting those vertices, leaving out the edge-labels, vertex-properties and edge-properties.

In this section, we introduce three data structures that can be utilized by graph databases for the storage of a graph topology. In (Section 2.3.1.1), we present the adjacency matrix as the first storage structure for graph topology. Next, we present the compressed sparse row (CSR) in (Section 2.3.1.2). Lastly, we present the adjacency list in (Section 2.3.1.3).
2.3. Graph Data Structures

(a) Directed graph $G$.

(b) Adjacency matrix representation of $G$.

(c) CSR representation of $G$.

(d) Adjacency list representation of $G$.

Figure 2.5: Logical representations of the topology of directed graph $G$ [CLRS09].

2.3.1.1 Adjacency Matrix

The adjacency matrix is the preferred method of storing graph $G = (V,E)$ when the graph is dense (i.e. $|E| \approx |V|^2$) or in case there is a need to get a fast response on checking for two vertices $u$ and $v$, if there is an edge tying them to each other.

For a directed graph $G = (V,E)$, the adjacency matrix $A$ representing the graph is a $|V| \times |V|$ matrix and the vertices of the graph are labeled with a sequence of numbers $1, 2, ..., |V|$. For a cell $a_{ij}$ in matrix $A$ where $1 \leq i \leq |V|$ and similarly $1 \leq j \leq |V|$, a value of “1” is assigned to the cell if $(i, j) \in E$, otherwise, a value of “0” is assigned to the cell [CLRS09].

In (Figure 2.5(a)), we show an example of a vertex-labeled directed graph $G$ that is consisted of 6 vertices and 8 edges. In (Figure 2.5(b)), we show the adjacency matrix logical representation of $G$ [CLRS09].

The memory required for storing an adjacency matrix of a graph is $O(|V|^2)$ which is not affected by the number of edges $|E|$ in the graph [CLRS09].
2.3.1.2 Compressed Sparse Row (CSR)

Sparseness of an adjacency matrix \( A \) can be solved by storing only the non-zero elements of the matrix (i.e. recording only the existence of an edge and discarding information about the absence of an edge between two vertices). The compressed sparse row (CSR) is a compact storage format that stores only non-zero elements of a sparse matrix. The compressed sparse row (CSR) format is storing the information about the non-zero elements of the matrix in two vectors with contiguous memory locations \((\text{col\_ind}, \text{row\_ptr})\). We store the column indices of the non-zero elements in the matrix in \( \text{col\_ind} \) ordered by their position precedence in the matrix when scanning the matrix in a row-wise left-to-right traversal. In \( \text{row\_ptr} \), we store only the indices of the elements in \( \text{col\_ind} \) that are located first in their respective rows in the matrix [DDRvdV00, PV17].

In (Figure 2.5(c)), we show the compressed sparse row (CSR) that is representing the adjacency matrix in (Figure 2.5(b)) and consequently representing the vertex-labeled directed graph \( G \) in (Figure 2.5(a)) [DDRvdV00].

Instead of storing \( |V|^2 \) elements in the adjacency matrix, the compressed sparse row (CSR) format is consuming much less storage by storing only \( nnz + |V| + 1 \) elements, where \( nnz \) is the number of non-zero elements in the matrix [DDRvdV00]. Although the compressed sparse row (CSR) format is offering a contiguous memory allocation of the graph data compacted in two vectors, a manipulation by adding or removing elements to CSR is expensive. Because the order of elements stored in the two vectors \((\text{col\_ind}, \text{row\_ptr})\) must always be maintained, a manipulation to the CSR data implies a reorganization of the elements in the vectors to maintain their elements order.

2.3.1.3 Adjacency List

In adjacency matrix we faced the problem of storing redundant and unnecessary information regarding the absent edges. Although the compressed sparse row (CSR) format has offered a solution to this problem by storing only the necessary information, it raised a different issue which is the expensive manipulation of the data.

The adjacency list \( \text{Adj} \) of a graph \( G = (V,E) \) is composed of a set of \( |V| \) pairs. Each pair is consisted of a vertex \( u \) and a vector \( K \) that contains the adjacencies of \( u \) [vR98].

Adjacency lists is memory efficient in storing sparse graphs by storing only information that indicates the existence of an edge. Adjacency lists keeps the adjacencies vector \( K \) of each vertex ordered for efficient look-up operations. Adding or removing a vertex to/from \( K \) requires a reorganization of the elements in the vector to maintain the elements order. The overhead of reorganizing the elements in \( K \) is proportional to the size of \( K \) which is less when compared to the similar overhead in CSRs that is proportional to \( |V| \) [vR98].

In (Figure 2.5(a)), we show an example of a vertex-labeled directed graph \( G \) that is consisted of 6 vertices and 8 edges. In (Figure 2.5(d)), we show the adjacency list logical representation of \( G \) [CLRS09].

The memory required for storing an adjacency list of a graph is \( O(|V| + |E|) \) [CLRS09].
2.3. Graph Data Structures

(a) An attributed directed graph $G$ [PLB14].

(b) Vertex universal table of $G$ [PLB14].

(c) Edge universal table of $G$ [PLB14].

(d) Emerging Schema - vertex column groups of $G$.

(e) Emerging Schema - edge column groups of $G$.

Figure 2.6: Universal table and emerging schema representations of property graph $G$. 
2.3.2 Graph Properties

In (Section 2.3.1), we introduced three graph topology data structures. The three data structures are storing information only regarding the graph topology, leaving out the storage of information regarding the properties of the graph to the graph properties data structures.

In this section, we introduce three data structures that can be utilized by graph databases for storing the properties of vertices and edges for a given graph. In (Section 2.3.2.1), we present the universal table. Next, we present the emerging schema approach in (Section 2.3.2.2). Lastly, we present the nested key-value store in (Section 2.3.2.3).

2.3.2.1 Universal Table

Universal tables is a method of storing the vertices and edges properties in only two tables. Each object (vertex or edge) is represented by a single row in the respective universal table. A column in the universal table is representing a distinct object’s property. A null value in in the universal table means that the property represented by the referenced column is not applicable to that object stored in the respective row [PV17].

Vertices and edges can have a high number of distinct properties, where only a few of those properties are applicable to a specific vertex or edge. The in-applicability of all properties to all vertices or edges leads to a degree of sparseness in universal table. Sparseness can be a problem in storing large graphs by consuming more memory. However, the storage of an object (vertex or edge) in a single row allows for a join-free method to extract an object’s properties [PV17].

In (Figure 2.6(a)), we show an example of an attributed directed graph G that is composed of 4 vertices and 5 edges. In (Figure 2.6(b)), we show a vertex universal table representing G, where each single vertex in G is represented by a single row and each distinct vertex property is represented by a separate column. Similarly, in (Figure 2.6(c)), we show an edge universal table of G, where each single edge in G is represented by a single row and each distinct edge property is represented by a separate column. Question marks in the tables denotes absence of a value (i.e. a null value) [PV17].

2.3.2.2 Emerging Schema

In (Section 2.3.2.1), we discussed a simple method of storing an attributed graph using only a pair of universal tables, a vertex universal table and an edge universal table. The universal table method is subject to a drawback as it tends to be more sparse as the number of distinct properties increase.

Emerging schema is a method that applies vertical partitioning on the properties of objects (vertices or edges) in a way that reduces the sparseness of the data stored and at the same time doesn’t require the joining of many tables to extract a single object’s properties. The emerging schema method is exploiting the inherent schema of the data to create a set of column groups for vertices and another set of column groups for edges.
Each column group is consisted of multiple columns where the first column is always representing the unique object’s label. In addition to the object’s label column, properties that co-occur frequently together are clustered into the same column group. An object can be represented by a row in one or more of the column groups. An object is represented by a row in a specific column group, only if the object has at least one property represented by a column in the respective column group. Objects with no descriptive properties are grouped together into a default column group [PV17].

The clustering of properties into column groups can be done by applying machine learning clustering algorithms such as k-means on the data in the universal table [Mac67, PV17]. The input to the clustering algorithm is the data stored in the universal table. Each column in the universal table is considered an \( n \) dimension point provided to the clustering algorithm, where \( n \) is the number of rows in the table and the value for each dimension is “1” in case a non-null value and “0” in case of a null value.

Clustering is applied once on the vertex universal table and once on the edge universal table. A single object’s (vertex or edge) row is constructed from the resulted column groups by joining them over the object-label column.

In (Figure 2.6(a)), we show an example of an attributed directed graph \( G \) that is composed of 4 vertices and 5 edges. In (Figure 2.6(d)), we present the vertex column groups for \( G \) that is produced by the emerging schema method. Similarly, in (Figure 2.6(e)), we present the edge column groups for \( G \) that is produced by the emerging schema method. Question marks in the column groups denotes absence of a value (i.e. a null value) [PV17].

Even though graph \( G \) is attributed, it is not explicitly defining an identifying label for each object. For storing the graph data in a universal table, an identifying label must be defined for each vertex and edge. The property ”id” that describes the vertices is appearing to have a unique value for each vertex and hence we used it to act as the vertex-label. For edges, we found three properties that we can use to uniquely identify each edge. The three properties are the ”id” property of the source vertex, the ”id” property of the target vertex, and the ”type” property of an edge. The concatenation of the three properties is acting as the edge-label.

### 2.3.2.3 Nested Key-Value Store

In this section, we are presenting another approach for solving the sparseness problem of universal tables we discussed in (Section 2.3.2.1).

Graph properties are composed of vertices-properties and edges-properties. The properties of an object (vertex or edge) are consisted of key-value pairs that describe the object. In a given vertex-attributed and edge-attributed graph \( G = (V, E) \), for each vertex \( u \in V \) and edge \( x \in E \), a set \( P \) of key-value pairs exists. Each key-value pair \( (k, v) \in P \), is consisted of a property \( k \) that describes the object and a value \( v \) that is assigned to \( k \) [LH11].
Nested key-value stores are storing an object’s properties in a two level nested key-value stores. The first level of the store is representing rows, where each object is stored in a row. The object’s row is represented by a key-value pair, where the key is the object-label and the value is itself a second level of a key-value store. The key-value store on the second level is to represent columns, where each object’s property is stored in a column. Each column is represented by a key-value pair, where the key is the name of the property describing the object and the value is a string of characters.

Sparseness problem of universal tables is eliminated in the nested key-value store by storing only the values for the properties that are applicable for a given vertex or edge. In this way, the size of the second level of key-value store for each row is equal to the number of non-empty properties describing the respective vertex or edge. We store the properties of the vertices in one nested key-value store separated from another nested key-value store that we use for the storing the edges properties.

Storing the property name together with its value in the nested key-value store, makes any modifications to the property name needs to be applied to each row in the entire nested key-value store. Also, as the sparseness of the data represented in the universal table is decreased, the memory requirements to store the property name redundantly in every row will increase.

In Figure 2.7, we show a diagram of a nested key-value store that is composed of rows and columns. On the first level of the store, rows are stored, while columns are stored in the second level of the store.

The nested key-value store is searched for an object’s row using the object-label \(k\) and the value returned is the object’s columns \(c\). The object’s columns \(c\) are searched for a specific column or property to return the value \(v\) of that column.
2.4 Data Structures in C++

In (Section 2.3), we discussed the logical design of graph data structures. In this section, we present necessary background knowledge on data structures offered as part of the Standard Template Library (STL) of the C++ programming language. In (Chapter 3), we will use the data structures presented in this section as the building blocks for the physical design of graph data structures.

We will start with introducing the (std::vector) data structure in (Section 2.4.1). Next, we present the (std::map) data structure in (Section 2.4.2). In (Section 2.4.3), we present the (std::unordered_map) data structure. Lastly, we present the (std::pair) data structure in (Section 2.4.4).

2.4.1 std::vector

The (std::vector) data structure is a sequence container that is offered as part of the STL library in the C++ programming language. Sequence containers are storing their elements in contiguous memory locations [Jos12]. Elements stored in a (std::vector) are atomic values of the same type. The order of the elements in the (std::vector) is the same as the order of inserting them into the container if they are appended to the end of the (std::vector). Another way to control the order of the elements is to insert them at exact positions in the (std::vector). The value of the element doesn’t affect the order of the element in the container [Jos12].

The (std::vector) data structure is usually implemented as a dynamic array that can grow and shrink in size. Appending elements to the end of the (std::vector) is very fast as it takes a constant time with a complexity of $O(1)$. Inserting element at any other position than the end of the (std::vector) takes more time, as all the following elements need to be displaced to free room for the inserted element. The complexity of the insertion at a position other than the end is linear $O(n)$. Accessing an elements in a (std::vector) by position is so fast as it takes a constant time $O(1)$, while the complexity of finding an element in a (std::vector) using the element’s value is linear to the size of the (std::vector) [Jos12].

2.4.2 std::map

The (std::map) data structure is an associative container that is offered as part of the STL library in the C++ programming language. Elements in an associative container are accessed using their values rather than their position in the container [Jos12].

An element in a (std::map) container is consisted of a key-value pair where the key of each pair is unique in the entire (std::map). The key and value which together are forming an element in a (std::map) can be of different types. Elements in a (std::map) are sorted by their keys according to the compactor of the stored elements data type. The order of insertion has no effect on the order of the elements in the container.
Storing the elements in a (std::map) sorted by their keys allows for range queries to be performed on the elements stored in the container [Jos12].

The (std::map) data structure is usually implemented as a binary search tree. Insertion of a new element or finding an existing one in a (std::map) has a logarithmic complexity $O(log(n))$ where “n” is the number of elements in the (std::map). Modifying the key of an element in a (std::map) can’t be done as it will invalidate the order of the elements in the container. Instead, the element has to be removed and inserted again with the new key [Jos12].

### 2.4.3 std::unordered_map

The (std::unordered_map) data structure is an associative container that is offered as part of the STL library in the C++ programming language. Elements in an associative container are accessed using their values rather than their position in the container [Jos12].

Similar to the (std::map) data structure, an element in a (std::unordered_map) container is consisted of a key-value pair where the key of each pair is unique in the entire (std::unordered_map). The key and value which together are forming an element in a (std::unordered_map) can be of different types. In contrast to the (std::map) data structure, elements in a (std::unordered_map) don’t have any defined order. The order of insertion has no effect on the order of the elements in the container nether the elements’ values [Jos12].

The (std::unordered_map) data structure is usually implemented as a hash-table. The hash-table is consisted of buckets where each element in the container is stored in a specific bucket depending on a hash value that is determined by a special hash function. Finding an element by its value in a (std::unordered_map) container has a constant time complexity of $O(1)$. A feature of a good hash function is that it generates hash values to all the available buckets in the container. Growing the size of a (std::unordered_map) data structure requires a rehashing to all the elements that are already stored in the container. Using the hash function, a rehashing of an element is done by generating a new hash value for the element to determine the element’s new location in the container [Jos12].

### 2.4.4 std::pair

The (std::pair) data structure is a simple container that is offered in the C++ programming language. A (std::pair) structure is offering the storage of two values of different types as a single unit. The two values stored in a (std::pair) have a concrete order. A value is accessed in the (std::pair) by specifying the value’s position in the structure whether it is the first or the second [Jos12].
2.5 Summary

In this chapter, we made a formal definition of a graph according to the graph theory and discussed the different graph types. We presented two logical graph models that are most commonly used by state of the art graph databases, the property graph model (PGM) and the resource description framework model (RDF). We presented several storage structures used by graph databases to store graph data and we differentiated between structures used to store graph topology and structures used to store graph properties. We presented a set of data structures that are offered by the C++ programming language which we will use as the building blocks for the physical design of graph data structures.

Aiming to evaluate and benchmark the different graph storage structures, we implemented a set of graph storage structures by their two types (graph topology structures and graph properties structures). In the next chapter, we present the details of the implementation with a focus on the physical design of the graph storage structures we implemented.
3. Physical Design of Graph Data Structures

In [Chapter 2], we presented the necessary background knowledge concerning topics covered in this thesis. In this chapter, we dive deep into the main topic of this thesis by presenting the set of evaluation questions we are going to answer in this thesis as well as the physical design of the graph data structures, we are going to evaluate. This chapter is consisted of the following sections:

- **Evaluation Questions:**
  First, we introduce a set of evaluation questions, we intend to answer in this thesis in [Section 3.1].

- **Graph Topology Structures:**
  In [Section 3.2], we present the physical design of a set of data structures that can be used by graph databases for the storage of a graph topology.

- **Graph Properties Structures:**
  We present the physical design of a set of data structures that can be used by graph databases for the storage of a graph properties in [Section 3.3].

- **Parallel Graph Structures:**
  In [Section 3.4], we present the physical design of a Parallel version of the adjacency list and the Nested Key-Value Store graph data structures.

- **Partitioned Graph Topology:**
  Next, we present a method for partitioning the graph topology in order to support an edge-labeled multi-graph in [Section 3.5].

- **Summary:**
  Finally, we provide a summary of the main topics we discussed in this chapter in [Section 3.6].
3. Physical Design of Graph Data Structures

3.1 Evaluation Questions

A set of data structures has been developed over time for the purpose of efficient storage and processing of graph data. Although, all the graph data structures can perform the same tasks, their performance is highly dependent on the characteristics of the data as well as the kind of computation performed on this data [PV17].

In a survey performed by (Sahu et al.), the authors mentioned a list of challenges for processing a graph. On top of the list of challenges came scalability. The scalability of loading, updating and performing computations on a large graph is the most prevalent issue challenge for graph processing systems [SMS+17].

In this thesis we are going to answer a set of evaluation questions that focuses on the effect of the use of a specific data structure for storing graph data on the performance. Following are the concrete set of evaluation questions we aim to answer in this thesis:

1. How do the execution time for loading data, and the memory footprint, scale for the different graph data structures when they need to accommodate larger data sizes (i.e. higher number of vertices and/or edges)?

2. What is the impact of loading the data in large versus small batches, for the different graph data structures, on the performance on this task?

3. What change in the data loading time, could processing the data in parallel introduce in comparison to sequential processing?

4. Given a set of queries, where each query computes centrality or performs pattern-matching on a graph, what is the effect of the graph data structure choice on the query response time?

For answering the above set of questions, we implemented the graph data structures previously introduced in (Section 2.3). We used C++ as the programming language for our implementation. We utilized the standard C++ data structures library (Standard Template Library (STL)) for building up the graph data structures (for more information on C++ or STL, see [Jos12]).

In (Section 3.2, Section 3.3, Section 3.4, and Section 3.5), we present the detailed physical design for the different implemented graph data structures. We implemented all the data structures for in-memory processing of data with no involvement of disk persistent storage.

In (Chapter 4 and Chapter 5), we present the evaluation results of tests we executed on our implemented graph data structures. Also, we present answers for the above mentioned evaluation questions based on the findings from the evaluation tests.
3.1. Evaluation Questions

(a) Vertex-labeled directed graph \( G \) [CLRS09].

(b) Auxiliary vertex-label look-up index.

(c) Adjacency matrix physical representation of \( G \).

(d) CSR physical representation of \( G \).

(e) Adjacency list physical representation of \( G \).

Figure 3.1: Physical representations of the topology of vertex-labeled directed graph \( G \).
3.2 Graph Topology Structures

In (Section 3.1), we introduced a set of evaluation questions that we aim to answer in this thesis. In this section, we present the physical design of graph topology data structures we implemented, which we later will execute a set of evaluation tests on to find answers for the evaluation questions. We designed all the graph topology structures in this section to represent a graph that is characterized by being labeled and directed.

We used the C++ programming language in the implementation of the data structures. We utilized data structures from the C++/STL library as the base for designing and implementing the graph topology data structures. We implemented all the data structures for in-memory processing of data with no involvement of disk persistent storage [Jos12].

Following, we present the physical design of three graph topology data structures (adjacency matrix, compressed sparse row (CSR), and adjacency list) in (Section 3.2.1, Section 3.2.2, and Section 3.2.3) respectively.

3.2.1 Adjacency Matrix

In (Section 2.3.1.1), we presented the necessary background knowledge on adjacency matrices. We discussed the logical design, suitable usage scenarios, and the memory requirements of the adjacency matrix data structure. We will present in this section our physical design of adjacency matrix.

We constructed the adjacency matrix physically using the (std::vector) data structure in C++. We represent each row in the adjacency matrix using a single (std::vector), with all of the (std::vector)’s that represent the rows, have the same size equal to the number of vertices forming the graph. We grouped all the (std::vector)’s that are representing the adjacency matrix rows into another (std::vector).

In a vertex-labeled graph, each vertex is labeled with a unique label. The vertex label doesn’t have to be a numeric label, however the label can be formed using a string of characters. The adjacency matrix logical design however, is assuming a numeric labeled vertices. To solve this problem, we attached an auxiliary structure to the adjacency matrix. The purpose of the auxiliary structure is to serve as a look-up index by mapping each unique vertex-label to a unique number that represents the vertex’s both row-and-column index in the adjacency matrix. The auxiliary index structure is utilizing the (std::map) in C++.

In (Figure 3.1(c)), we show an example of the adjacency matrix physical representation of the vertex-labeled directed graph $G$ shown in (Figure 3.1(a)). The 6x6 adjacency matrix in the figure is constructed using a set of (std::vector)’s grouped into another (std::vector). In (Figure 3.1(b)), we show the auxiliary vertex-label look-up index that maps each vertex-label in $G$ to a unique number representing the vertex’s both row-and-column index in the adjacency matrix.
3.2.2 Compressed Sparse Row (CSR)

In Section 2.3.1.2, we presented the necessary background knowledge on the compressed sparse row (CSR) structure. We discussed the logical design, suitable usage scenarios, and the memory requirements of CSR. We will present in this section our physical design of the compressed sparse row (CSR).

For constructing the compressed sparse row (CSR), we utilized the \(\text{std::vector}\) data structure for the physical design of CSR. The compressed sparse row (CSR) physical design is consisted of two \(\text{std::vector}\)’s. The first \(\text{std::vector}\) is representing the row\(_{ptr}\) structure and the second \(\text{std::vector}\) is representing the col\(_{ind}\) structure.

CSR inherits the same issue that adjacency matrix faces with a vertex-labeled graph, where each vertex is not limited to numeric labels but also can be labeled using a string of characters. We solve this issue in the same way we solved it in adjacency matrix by attaching an auxiliary structure to the CSR. The purpose of the auxiliary structure is to serve as a look-up index by mapping each unique vertex-label to a unique number that represents the original vertex’s both row-and-column index in the adjacency matrix. As the vertex-label and the column index in the adjacency matrix are no-longer equal, we chosen to directly store the vertex-label rather than the column index in the col\(_{ind}\) structure. The auxiliary index structure is utilizing the \(\text{std::map}\) in C++.

In (Figure 3.1(d)), we show an example of the compressed sparse row (CSR) physical representation of the vertex-labeled directed graph \(G\) shown in (Figure 3.1(a)). The CSR physical structure in the figure is constructed using one \(\text{std::vector}\) representing the row\(_{ptr}\) structure and another \(\text{std::vector}\) representing the col\(_{ind}\) structure. In (Figure 3.1(b)), we show the auxiliary vertex-label look-up index that maps each vertex-label in \(G\) to a unique number representing the original vertex’s both row-and-column index in the adjacency matrix.

3.2.3 Adjacency List

In Section 2.3.1.3, we presented the necessary background knowledge on adjacency lists. We discussed the logical design, suitable usage scenarios, and the memory requirements of the adjacency list data structure. We will present in this section our physical design of the adjacency list.

Adjacency list logical design is storing the topology of a graph \(G = (V, E)\) as a set of \(|V|\) pairs, where each pair is consisted of a vertex \(u\) as the key and the value is a vector \(K\) that contains the adjacencies of \(u\). We constructed the adjacency list using the \(\text{std::map}\) data structure in C++.

Each key-value pair \((u, K)\) in adjacency list is represented physically by a key-value pair in a \(\text{std::map}\). Each key-value pair in the \(\text{std::map}\) is consisted of an atomic value representing a unique vertex-label of a vertex \(u\) mapped to a \(\text{std::vector}\) of vertex-pointers. Each vertex-pointer in a \(\text{std::vector}\) is pointing to its respective vertex key in the adjacency list \(\text{std::map}\).
In (Figure 2.5(d)), we show an example of the adjacency list physical representation of the vertex-labeled directed graph $G$ shown in (Figure 3.1(a)). The adjacency list in the figure is constructed using a `std::map` of 6 key-value pairs. An example of a key-value pair is the second key-value pair which is consisted of the vertex-label of vertex $b$ and its adjacencies vector that contains only one pointer to the key of the fifth pair in the `std::map` (i.e. a pointer to vertex $f$).

### 3.3 Graph Properties Structures

In (Section 3.2), we presented the physical design of graph topology structures. In this section, we proceed with the introduction of our physical design for more graph data structures. We present the physical design of graph properties. We designed all the graph properties structures to support all the characteristics of the property graph model.

We continue using the C++ programming language in the implementation of the data structures. We utilized data structures from the C++/STL library as the base for designing and implementing the graph topology data structures. We implemented all the data structures for in-memory processing of data with no involvement of disk persistent storage (for more information on C++ or STL, see [Jos12]).

We will use the attributed directed graph $G$ in (Figure 3.2), to showcase an example of the physical representation of the properties describing each vertex or edge in $G$. Even though graph $G$ is attributed, it is not explicitly defining an identifying label for each object. The property "id" that describes the vertices is appearing to have a unique value for each vertex and hence we used it to act as the vertex-label. For edges, we found three properties that we can use to uniquely identify each edge. The three properties are the "id" property of the source vertex, the "id" property of the target vertex, and the "type" property of an edge. The concatenation of the three properties is acting as the edge-label.

Following, we present the physical design of the three graph properties data structures (universal table, emerging schema, and nested key-value store) in (Section 3.3.1, Section 3.3.2, and Section 3.3.3) respectively.

![Figure 3.2: An attributed directed graph $G$ [PLB14]](image-url)
3.3. Graph Properties Structures

(a) Vertex universal table physical representation.

(b) Vertex header table.

(c) Edge universal table physical representation.

(d) Edge header table.

Figure 3.3: Universal table physical representation of graph $G$ shown in Figure 3.2.

3.3.1 Universal Table

In Section 2.3.2.1, we presented the necessary background knowledge on universal tables. We discussed the logical design, suitable usage scenarios, advantages, and disadvantages of the universal table data structure. In this section, we will present our physical design of the universal table.

We constructed the universal table physically using the (std::map) data structure in C++. We represent each row in the universal table using a single key-value pair in the (std::map). We use the label of the object (vertex or edge) as the row key, and a (std::vector) that contains the rest of the values to represent the row value. Vertex universal table is physically represented by one (std::map), while the edge universal table is represented by another (std::map). The header of the universal table is stored in a separate data structure of type (std::unordered_map), where each property name is linked to the property’s index in each of the (std::vector)’s in the respective universal table.
In (Figure 3.3), we show the universal table physical representation of the attributed directed graph $G$ shown in (Figure 3.2). The graph $G$ is represented physically using a vertex universal table in (Figure 3.3(a)) and an edge universal table in (Figure 3.3(c)), which are storing the values of the attributes describing the graph’s vertices and edges. The header of the vertex universal table and edge universal table are stored in a vertex header table in (Figure 3.3(b)) and an edge header table in (Figure 3.3(d)) respectively.

### 3.3.2 Emerging Schema

In (Section 2.3.2.2), we presented the necessary background knowledge on emerging schema. We discussed the logical design, suitable usage scenarios, advantages, and disadvantages of the emerging schema data structure. In this section, we will present our physical design of the emerging schema.

Emerging schema is consisting of a set of column groups formed by applying a clustering algorithm on the columns of the universal table, resulting in the grouping of the columns with frequent co-occurring values together into the same column group. We constructed an emerging schema column group physically using the (std::map) data structure in C++ similar to the physical design of a universal table. We represent each row of a column group using a single key-value pair in the (std::map). We use the label of the object (vertex or edge) as the row key, and the rest of the values are placed in a (std::vector) that represents the row value. If the object-label is composed of more than one column, the values are concatenated to form a single key. Next, we assign each column group a unique column-group-id. Finally, all the key-value pairs formed from the column-group-id and its respective column group are placed into a single (std::unordered_map) data structure.

We used the k-means clustering algorithm to vertically partition the universal table columns into a set of column groups [Mac67]. A Vertex emerging schema is physically represented by one (std::unordered_map), while the edge emerging schema is represented by another (std::unordered_map). The information concerning which column group a property belongs to, and the column index of the property inside the column group is stored in separate data structure of type (std::unordered_map), where each property is linked to a pair of values comprises the property column group and the property index inside the column group respectively.

In (Figure 3.4), we show the emerging schema physical representation of the attributed directed graph $G$ shown in (Figure 3.2). The graph $G$ is represented physically using a vertex emerging schema in (Figure 3.4(a)) and an edge emerging schema in (Figure 3.4(c)), which are storing the values of the properties describing the graph’s vertices and edges in column groups based on their frequency of co-occurrence. The emerging schema vertex header table in (Figure 3.4(b)) is storing for each property, the column-group-id of the column group containing the property, followed by the property index in the respective column group. The column-group-id together with the property position are grouped into a (std::pair) data structure. Similarly, the emerging schema edge header table is shown in (Figure 3.4(d)).
3.3. Graph Properties Structures

(a) Vertex emerging schema physical representation.

(b) Emerging schema vertex header table.

(c) Edge emerging schema physical representation.

(d) Emerging schema edge header table.

Figure 3.4: Emerging schema physical representation of graph $G$ shown in Figure 3.2.
3.3.3 Nested Key-Value Store

In (Section 2.3.2.3), we presented the necessary background knowledge on nested key-value store. We discussed the logical design, suitable usage scenarios, advantages, and disadvantages of the nested key-value store data structure. In this section, we will present our physical design of the nested key-value store.

In nested key-value stores, we store each object (vertex or edge) along with only the attributes that are applicable to this object (i.e. where a value exists). We store each object physically in the nested key-value store as a row. Each row is represented by a key-value pair, where the object-label is the row key and the row value is a container of type `std::unordered_map` that contains the set of property-value pairs that describe the object. Objects with no descriptive properties are represented with a row that has an empty `std::unordered_map` as a row value.

In (Figure 3.5), we show the nested key-value store physical representation of the attributed directed graph $G$ shown in (Figure 3.2). The graph $G$ is represented physically using a vertex nested key-value store in (Figure 3.5(a)) and an edge nested key-value store in (Figure 3.5(b)), which are storing the values of the properties describing the graph’s vertices and edges.

![Figure 3.5: Nested key-value store physical representation of graph $G$ in Figure 3.2.](image)
3.4 Parallel Graph Structures

Multiprocessor computers are offering the possibility to execute multiple instructions simultaneously with a purpose of performing computing tasks in a more efficient way [CLRS09]. The physical design of the graph topology data structures discussed in (Section 3.2) and the graph properties data structures discussed in (Section 3.3), allows for only sequential operations to be done on the data structures.

Multiple data access for read and write simultaneously can lead to a data race problem. Data race is defined in the C++ standard as “two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other”. To allow for parallel operations on the data structures, we used the (std::shared_timed_mutex) primitive provided in the C++ programming language to synchronize the access to the data structures and protect them from simultaneously being modified by multiple threads [Jos12].

The (std::shared_timed_mutex) primitive is providing the following four main functions in its interface [Jos12]:

- **lock()**
  Locks the mutex in exclusive mood, blocks the execution if the mutex is already locked.

- **lock_shared()**
  Locks the mutex in shared mood, blocks the execution if the mutex is already locked in exclusive mood.

- **unlock()**
  Unlocks a mutex that is currently in an exclusive lock mood.

- **unlock_shared()**
  Unlocks a mutex that is currently in a shared lock mood.

In this section, we present a parallel version of the adjacency list data structure in (Section 3.4.1) as an example of a parallel graph topology structure, as well as a parallel version of the nested key-value store in (Section 2.3.2.3) as an example of a parallel graph properties structure. The parallel data structures are allowing for multiple data readers and writers performing operations concurrently making use of the (std::shared_timed_mutex) primitive to protect against a data race.

3.4.1 Parallel Adjacency List

In (Section 3.2.3), we discussed our physical design of adjacency list, on which we can operate only using a single thread. In this section, we extend the physical design of the adjacency list in (Section 3.2.3) to support multiple threads to operate over the data structure simultaneously in parallel. We will call the adjacency list discussed in (Section 3.2.3) a single thread adjacency list.
We extended the physical design of the single thread adjacency list with a global `std::shared_timed_mutex` primitive, which we use to protect the `std::map` of the adjacency list against concurrent writes from multiple threads.

Any thread that needs to read any data from the adjacency list `std::map` is required to acquire a lock on the global `std::shared_timed_mutex` in shared mood using the `lock_shared()` function. Acquiring the lock on the `std::shared_timed_mutex` in shared mood, allows multiple threads to read data from the `std::map` and stops any thread from acquiring an exclusive lock on the global `std::shared_timed_mutex`.

Alternatively, a thread that needs to write any data to the adjacency list `std::map` is required to acquire a lock on the global `std::shared_timed_mutex` in exclusive mood using the `lock()` function to stop any other thread from reading or writing to adjacency list.

Once a thread finishes reading or writing to the adjacency list, the thread unlocks the global `std::shared_timed_mutex` using the `unlock()` function in case the thread has holds an exclusive lock on the mutex or using the `unlock_shared()` in case the thread holds a shred lock on the mutex. This allows other threads to be able to acquire lock on the mutex.

Both, the adjacency list along with the attached `std::shared_timed_mutex` allow for parallel access to the data structure by multiple threads.

### 3.4.2 Parallel Nested Key-Value Store

In [Section 3.3.3], we discussed our physical design of the nested key-value store which allows only for a single thread to operate on the data. In this section, we extend the physical design of the nested key-value store in [Section 3.3.3] to support multiple threads to operate over the data structure simultaneously in parallel.

Similar to the extension we made to the single thread adjacency list in order to support parallel access, we extended the physical design of the single thread nested key-value store with a global `std::shared_timed_mutex` primitive, which we use to protect the `std::map` of the nested store against concurrent writes from multiple threads. We used one `std::shared_timed_mutex` primitive to manage the access to the vertex nested key-value store and another `std::shared_timed_mutex` primitive to manage the access to the edge nested key-value store.

Any thread that needs to read any data from the nested store `std::map` is required to acquire a lock on the respective `std::shared_timed_mutex` in shared mood using the `lock_shared()` function, allowing other threads to simultaneously read data from the nested store and stops any thread from acquiring an exclusive lock on the `std::shared_timed_mutex`.

Alternatively, a thread that needs to write any data to the nested store `std::map` is required to acquire a lock on the respective `std::shared_timed_mutex` in exclusive mood using the `lock()` function to stop any other thread from reading or writing to nested store.
Once a thread finishes reading or writing to the nested store, the thread unlocks the respective \texttt{(std::shared\_timed\_mutex)} using the \texttt{unlock()} function in case the thread holds an exclusive lock on the mutex or using the \texttt{unlock\_shared()} in case the thread holds a shared lock on the mutex.

Using the nested key-value store along with the appropriate lock on the attached \texttt{(std::shared\_timed\_mutex)}, allows multiple threads to access the data structure in parallel.

### 3.5 Partitioned Graph Topology

In (Section 3.3), we presented the physical design of three graph properties structures. The physical design of the graph properties structures allows for the storage of a graph that is in the form of a property graph model (i.e. directed, labeled, attributed, and multi-graph). In (Section 3.2), we presented the physical design of three graph topology structures which allows for the storage of a directed and labeled graph. For consistency of the physical design between the graph topology and graph properties structures, we present in this section a method to extend the physical design of the graph topology structures to support the storage of a multi-graph.

We introduce a graph partitioning technique as an extension to the physical design of the graph topology structures. In a multi-graph multiple edges could exist between the same two vertices, where edge is identified by an edge-label assigned to it. We use the edge label assigned to each edge in the graph as the partitioning key. In a partitioned graph topology each edge-label is linked to a graph topology structure and together they offer the storage of a multi-graph.

![a directed labeled multi-graph](image1.png)

(a) A directed labeled multi-graph $G$.  

![partitioned adjacency list physical representation](image2.png)

(b) Partitioned adjacency list physical representation.

Figure 3.6: Physical representation of a partitioned adjacency list of graph $G$. 
We designed the partitioning technique physically using a \(\texttt{std::unordered_map}\) data structure. The \(\texttt{std::unordered_map}\) is storing a set of key-value pairs, where the key is an edge-label and the value is a graph topology structure. A graph topology structure that is linked to an edge-label “\(x\)” is storing information regarding only the vertices with at least one incoming or outgoing edge labeled with “\(x\)”.

In Figure 3.6, we show an example of a physical representation of a partitioned graph topology structure. We present in Figure 3.6(b) the partitioned adjacency list physical representation of the directed labeled multi-graph \(G\) shown in Figure 3.6(a).

### 3.6 Summary

In this chapter, we presented a set of evaluation question that are focusing on evaluating the performance of graph data structures in different usage scenarios. We introduced our physical design for the graph topology structures as well as the graph properties structures. We presented a parallel version of the adjacency list and the nested key-value store structures as an example of parallel graph structures that allows for parallel access by multiple threads. We presented an extension to the physical design of the graph topology structures that allows the structures to store a multi-graph.

In next chapter, we present the evaluation environment we used to evaluate our implementation for the different graph structures.
4. Evaluation Environment

In [Chapter 3], we introduced the physical design of a set of graph data structures. We complemented the implementation of the graph data structures with other components in order to set up a complete environment for evaluating the behaviour of the graph data structures in different scenarios.

In this chapter, we introduce the details of every component in the evaluation environment which we will use to evaluate our implemented graph data structures. This chapter is consisted of the following sections:

- **Technical Environment:**
  In [Section 4.1] we detail the specifications of the technical environment that we use to test the performance of loading and querying the graph data structures.

- **Evaluation Dataset:**
  In this section, we present the dataset we used to evaluate the performance of the graph data structures [Section 4.2].

- **Data Loader:**
  Next, we present the data loader component that we utilize for loading the data into the graph data structures [Section 4.3].

- **Queries:**
  In [Section 4.4], we define a set of queries that we perform against the data which we loaded into the graph data structures.

- **Summary:**
  Lastly, we provide a summary for what we discussed in the chapter [Section 4.5].
4. Evaluation Environment

4.1 Technical Environment

In this section, we list the specifications of the technical hardware and software we used to implement, test and evaluate the graph data structures.

Following are the details of the programming language we used in implementation:

- Programming language: C++
- Compiler: g++
- Compiler version: 5.5.0

The specifications of the operating system for the test and evaluation machine are below:

- Operating System: Linux
- Distribution: Ubuntu
- Version: 18.04 LTS
- Code Name: Bionic Beaver

Lastly, we list the hardware specifications of the test and evaluation machine below:

- Hardware Provider: Microsoft Azure Cloud
- Machine Code: E16S_V3
- Processor: Intel(R) Xeon(R) CPU E5-2673 v4
- Processor Speed: 2.30GHz
- Number of Cores: 16
- Main Memory: 128 GiB
Figure 4.1: LDBC - SNB data model [AAB+17].
4. Evaluation Environment

4.2 Evaluation Dataset

For evaluating the implemented graph data structures, we used a dataset that we generated using the Linked Data Benchmark Council (LDBC) - Social Network Benchmark (SNB) framework \cite{Bon13,AAB+17}.

The Linked Data Benchmark Council (LDBC) is an independent non-profit company which focuses on developing of new benchmarks for large-scale graph and RDF data management, establishing an industry-neutral entity for developing graph and RDF benchmarks, auditing benchmark results, and publishing audited results\footnote{For more information on the Linked Data Benchmark Council - Social Network Benchmark framework see: http://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf}.

The LDBC - Social Network Benchmark (SNB) framework is providing a data generator that generates a synthetic social network data. The data generator produces networks with described schema with distributions and correlations similar to those expected in a real social network. The schema represents a realistic social network, including people and their activities in the social network during a period of time. In Figure 4.1, we show a UML diagram for the data model of the LDBC - SNB generated data \cite{AAB+17}.

We used the LDBC - SNB data generator to generate datasets with different sizes. All the generated datasets are of the same schema shown in Figure 4.1. The size of the dataset generated is depending on two parameters provided to the data generator: the number of persons and the number of years. We generated datasets of different sizes by changing the parameter value of the number of persons while fixing the number of years parameter value. The generated datasets size and parameters are shown in Table 4.1. We assigned each generated dataset a scale factor. A scale factor is an indicator of the value of the “number of persons” parameter we used to generate the dataset. For example, a dataset with scale factor 0.25 was generated with a “number of persons” parameter of 2500 and a dataset with scale factor 0.5 was generated with a “number of persons” parameter of 5000. Similarly, we assigned scale factors to the other generated datasets \cite{AAB+17}.

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th># of Vertices</th>
<th># of Edges</th>
<th># of persons</th>
<th># of Years</th>
<th>Start Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>542,040</td>
<td>2,634,240</td>
<td>2,500</td>
<td>3</td>
<td>2010</td>
</tr>
<tr>
<td>0.5</td>
<td>1,244,969</td>
<td>6,393,902</td>
<td>5,000</td>
<td>3</td>
<td>2010</td>
</tr>
<tr>
<td>1</td>
<td>2,856,994</td>
<td>15,530,170</td>
<td>10,000</td>
<td>3</td>
<td>2010</td>
</tr>
<tr>
<td>2</td>
<td>6,509,804</td>
<td>36,771,980</td>
<td>20,000</td>
<td>3</td>
<td>2010</td>
</tr>
<tr>
<td>4</td>
<td>14,737,719</td>
<td>84,895,633</td>
<td>40,000</td>
<td>3</td>
<td>2010</td>
</tr>
</tbody>
</table>

Table 4.1: The generated datasets size and parameters
The evaluation dataset is consisted of vertex files and edge files. A vertex file contains data for all the vertices labeled with the same vertex-label. Each line in a vertex file is consisted of a comma-separated list of all the properties describing one vertex. Similarly, an edge file contains data for all the edges labeled with the same edge-label. Each line in an edge file is consisted of all properties describing one edge.

4.3 Data Loader

In [Section 4.2], we presented the dataset we are using to evaluate the performance of the different graph data structures. In this section, we present the data loader component that we used to load the evaluation dataset into the graph data structures.

In [Section 4.3.1], we introduce the data loader that we developed and used to load the evaluation dataset in batches. In [Section 4.3.2], we introduce the parallel loader that we developed and utilized to load the evaluation dataset in parallel.

4.3.1 Batch Loading

We developed the batch data loader in order to pre-process the evaluation dataset, divide the dataset into batches, and load the batches into the graph data structures. Since the vertices files are containing no information regarding the relationships between the vertices, we load the vertices files contents only into the graph properties structures. Meanwhile, we load the edges files into both: the graph topology structures and the graph properties structures.

```c++
void batchLoad(list dataFiles, int batchSize, 
               GraphStruct graphStructure) 
{
    Buffer buffer(batchSize);

    for (auto file : dataFiles) 
    {
        for (auto dataLine : file) 
        {
            GraphElement x = parse(dataLine); 
            x.setLabel(file.name); 
            buffer.insert(x);

            if (buffer.isFull()) 
            {
                graphStructure.consume(buffer); 
                buffer.clear();
            }
        }
    }
}
```

Listing 4.1: Function used for batch loading of data files.
In [Listing 4.1], we show a simplified version of the function that we developed to batch load the data files into the graph data structures. First, the function is creating a buffer structure with a size equal to the batch size. Next, the batch loader function is iterating over all the files. In (Line 12), the (parse) function is used to parse each of the data lines in each and every file. The (parse) function is converting a data line into a (GraphElement) object that contains the list of values in the data line. A (GraphElement) object can be casted into a vertex or an edge object depending on the file being loaded if it is a vertices file or an edges file. The name of the file is used as the label of the (GraphElement) object. Next, the (GraphElement) object is inserted into the buffer. The function is checking the buffer to see if it has reached its full capacity. If the buffer is full, the contents of the buffer is consumed by the graph structure and the buffer is cleared.

### 4.3.2 Parallel Loading

We developed a parallel data loader in order to load the evaluation dataset in parallel. The parallel data loader is creating multiple threads that loads a data file in parallel, therefore the parallel data loader is only valid for use along with the parallel versions of the graph data structures presented in [Section 3.4].

A data file is divided into a number of partitions equal to the number of threads. Each thread loads one partition of a data file in parallel with other threads. Each thread is loading its part of the file in the same way the batch data loader is loading the data into the graph data structures.

Each thread uses a buffer for temporary storage of data. The buffer size is the same across all the threads. Each thread iterates over the data lines located in its assigned partition of the file. The thread parses the data line to construct a vertex or an edge object depending on the file being loaded if it is a vertices file or an edges file. The name of the file is used as the label of the object. Next, the object is inserted into the buffer. The thread is checking the buffer to see if it has reached its full capacity. If the buffer is full, the contents of the buffer is consumed by the graph structure and the buffer is cleared.

### 4.4 Queries

In [Section 4.3], we presented the data loader component which we utilized in order to load the evaluation dataset into the graph data structures. In this section, we present the definition of a set of queries as another component of the evaluation environment. We will use the set of queries presented in this section to evaluate the graph data structures performance in executing each query.

We present the definition of three different queries in [Section 4.4.1] [Section 4.4.2] and [Section 4.4.3]. We will define each of the steps required to execute each query and the format of the output that we expect to receive.
The two queries presented in (Section 4.4.1 and Section 4.4.2) are pattern-matching queries. A pattern matching query is defined as a set of patterns that needs to be matched against the graph elements (vertices and edges) in order to retrieve all elements that match the given pattern.

The query presented in (Section 4.4.3) is a degree centrality query. In this query we are not matching the graph elements against given patterns, but we are computing the a degree centrality of each vertex in a graph. The degree centrality of a vertex is computing how many edges (incoming and outgoing) the vertex has. The output of the query is a summary report of the degree centrality of all the vertices in the given graph.

4.4.1 Definition: Query #1

The query we present in this section (Query #1) was introduced as part of the business intelligence workload in the LDBC - Social Network Benchmark (SNB) specification. The query is given the code (BI 1) in the specification [AAB+17].

Query #1 is an example of a simple pattern-matching query that requires no traversals between the graph elements. We present in (Figure 4.2), the pattern which the query matches against all graph elements loaded from the evaluation dataset in (Section 4.2). The query is taking one parameter "$date" which is a date parameter. The query checks all the vertices in the graph that have a vertex-label "Message" and selects only those having the value of their "creationDate" property less than the given date "$date" [AAB+17].

The set of messages found are then grouped by three levels of grouping [AAB+17]:

1. By the year part of the "creationDate" property.

2. Per each year, group by the message type "isComment". The "isComment" property computed for each message is set to "true" if the message is a comment, "false" otherwise.

3. Per each year-message type group, group by the "lengthCategory" of each a message. The "lengthCategory" of a message is computed based on the "length" of each message as following:
   
   - "0": 0 <= "length" < 40
   - "1": 40 <= "length" < 80 (one liner)
   - "2": 80 <= "length" < 160 (tweet)
   - "3": 160 <= "length" (long)
The result of the query after grouping will be consisted of the following attributes:

1. “year”: Year part of the “creationDate” of a message.
2. “isComment”: “true” if the message is a comment, “false” if a post.
4. “messageCount”: Total number of messages in a group.
5. “averageMessageLength”: Average “length” of messages in a group.
6. “sumMessageLength”: Sum of “length” of all messages in a group.
7. “percentageOfMessages”: “messageCount” of a group as a percentage of all messages created before the given date “$date”.

Lastly the result set is sorted in the following order:

1. year: Descending.
2. isComment: Ascending.
3. lengthCategory: Ascending.

### 4.4.2 Definition: Query #2

The query presented in this section (Query #2) was introduced as part of the business intelligence workload in the LDBC - Social Network Benchmark (SNB) specification. The query is given the code (“BI 18”) in the specification.

In contrast to Query #1 which we presented in Section 4.4.1, Query #2 is an example of a more complex pattern-matching query that requires traversals between the graph elements in order to fulfill the query requirements. We present in Figure 4.3, the pattern which the query (Query #2) matches against all graph elements loaded from the evaluation dataset in Section 4.2. The query is taking three parameters as part of the input. The “$date” parameter which is a date, the “$lengthThreshold” parameter which is a number, and the “$languages” parameter which is a list of language-codes.
The query checks all the vertices in the graph that have a vertex-label “Person” and computes how many messages they made “messageCount”. The “Person” created a “Message” is identified as the target vertex of the edge with a label “hasCreator” of each “Message” [AAB+17].

The query counts a “Message” only if the message’s properties match the following conditions [AAB+17]:

- The value of the message’s “content” property is not empty.
- The value of the message’s “length” property is less than the value of the “$length$Threshold” parameter.
- The value of the message’s “creationDate” property is less than the value of the “$date$” parameter.
- The message’s language-code is one of the language-codes provided in the “$languages$” parameter.

- The “language” property of a “Post” identifies its language-code.
- The language-code of a “Comment” is defined in [AAB+17] as “that of the ”Post“ that initiates the thread where the ”Comment“ replies to”. The Post and Comments in between the “Comment” and the root “Post” of the thread do not have to meet any condition except for their “language”.

Next, we group the result set by the “messageCount” computed attribute, so that for each unique “messageCount”, the number of persons who created the same “messageCount” of messages are counted [AAB+17].

The final result set of the query will be consisted of the following attributes [AAB+17]:

1. “messageCount”: Number of messages a “Person” has created.
2. “personCount”: Number of persons who created the same number of messages “messageCount”.

Lastly the result set is sorted in the following order [AAB+17]:

1. personCount: Descending.
2. messageCount: Descending.
4.4.3 Definition: Query #3

In [Section 4.4.1 and Section 4.4.2], we introduced the definition of two pattern-matching queries. In this section, we present the definition of the last query. Query #3 is different from the other two queries (Query #1 and Query #2) in that Query #3 is computing a degree centrality report rather than performing a pattern-matching on the given graph.

A degree centrality of a vertex is the number of edges going out from the vertex as well as the number of edges going in to the vertex [New08]. We extend this definition to include the edge-label that identifies each edge in a multi-graph. We define the degree centrality of a vertex in a directed labeled multi-graph as the number of edges going out from the vertex as well as the number of edges going in to the vertex per an edge-label.

We compute the degree centrality report by first computing per each vertex in a graph, per each edge direction “edgeDirection”, and per each label of an edge “edgeLabel” the number of edges “edgeCount”. Next, we group by the result set by the three attributes (edgeDirection, edgeLabel, and edgeCount) in order to compute the “vertexCount” attribute. The “vertexCount” computed attribute denotes the number of vertices that have the exact same number of edges “edgeCount” with the same edge “edgeLabel” in the same edge direction “edgeDirection”.

The final result set of the query will be consisted of the following attributes:

1. “edgeDirection”: The direction of an edge from vertex perspective. “out” for an outgoing edge, “in” for an incoming edge.
2. “edgeLabel”: The edge-label identifying an edge in a multi-graph.
3. “edgeCount”: The number of edges linked to a vertex.
4. “vertexCount”: The number of vertices.

Lastly the result set is sorted in the following order:

1. edgeDirection: Ascending.
2. edgeLabel: Ascending.
3. edgeCount: Ascending.

4.5 Summary

In this chapter, we presented all components in the evaluation environment which we will use to evaluate our implemented graph data structures. We introduced the specifications of the technical environment that we use to test the performance of loading and querying the graph data structures.
We presented the dataset we used to evaluate the performance of the graph data
structures as well as the data loader component that we utilize for loading the evaluation
dataset into the graph data structures. Lastly, we presented the definition of a set of
queries that we perform against the data which we loaded into the graph data structures.

In next chapter, we present the first part of our evaluation results, with which we will
answer the evaluation questions number 1, 2, and 3 stated in Section 3.1.
5. Evaluation: Scalability and Data Loading

In (Chapter 4), we presented the details of every component in the evaluation environment which used to evaluate our implemented graph data structures.

In this chapter, we present the first part of the evaluation results for the experiments we conducted on the graph data structures in order to evaluate their performance. All the experiments we conducted are using the evaluation dataset which we presented in (Section 4.2) in order to test the performance of the different graph data structures. We aim to exploit the evaluation results which we present in this chapter in order to give answers for the evaluation questions (1, 2, and 3) which we stated in (Section 3.1). We present the second part of the evaluation results which we utilize to answer the evaluation question number 4 in (Chapter 6). This chapter is consisted of the following sections:

- **Scalability:**
  In this section, we present the results of the experiments we conducted to evaluate the scalability of the graph data structures (Section 5.1).

- **Batch Data Loading:**
  Next, we present results of the experiments we conducted to evaluate the effect of batch size on the performance of loading the graph data structures in (Section 5.2).

- **Parallel Data Loading:**
  In (Section 5.3), we present the results of the experiments we conducted in order to evaluate the improvement that loading data in parallel may introduce in comparison to sequential loading of data into the graph data structures.

- **Summary:**
  Lastly, we provide a summary for what we discussed in the chapter. (Section 5.4)
5. Evaluation: Scalability and Data Loading

(a) Change in loading time.

(b) Change in memory footprint.

Figure 5.1: Effect of increasing data size on the scalability of graph topology structures.

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>Loading Time (Second)</th>
<th>Memory Footprint (MiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Adjacency List</td>
<td>CSR</td>
</tr>
<tr>
<td>0.25</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>0.5</td>
<td>10</td>
<td>70</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
<td>499</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>3,390</td>
</tr>
<tr>
<td>4</td>
<td>159</td>
<td>30,263</td>
</tr>
</tbody>
</table>

Table 5.1: Evaluation results of graph topology structures scalability.
5.1 Scalability

In this section, we present the set of evaluation results for the experiments we conducted in order to evaluate the scalability of the different graph data structures. We evaluate the scalability of a graph data structure using two factors. Firstly, we evaluate the time taken to load the evaluation dataset into each of the graph data structures. Secondly, we measure the amount of memory consumed by each graph data structure in order to store the loaded data. We measure both the loading time and memory footprint when loading each scale factor of the evaluation dataset.

In (Section 5.1.1), we present the evaluation results concerning the scalability of the graph topology structures. We present the evaluation results concerning the scalability of the graph properties structures in (Section 5.1.2).

5.1.1 Scalability: Graph Topology Structures

In this section, we present the evaluation results regarding the scalability of the graph topology structures (adjacency list, compressed sparse row (CSR), and adjacency matrix).

- **Experiment Setup:**
  In this experiment we used the batch data loader to load the evaluation dataset into the different graph topology structures. We fixed the batch size to 1000 record per batch. For each graph topology structure, we measured the loading time and memory footprint when loading each of the scale factors of the evaluation dataset. We executed each test for five to twenty times and recorded the loading time and memory footprint of each run. We took the average of the measured figures after removing the most top and least figures.

- **Expected Result:**
  We expect adjacency list to outperform adjacency matrix and compressed sparse row (CSR) and score the least loading time. We expect the loading time of adjacency matrix to be the largest among the three structures.

  In regard to the memory footprint, we expect the memory footprint of adjacency matrix to be a lot bigger than the memory footprint of the adjacency list and the compressed sparse row (CSR). We expect the memory footprint of CSR to be smaller than that of adjacency list.

- **Observation:**
  We observed that the loading time of adjacency list is increasing with an average rate of (2.5x) as we increase the scale factor of the loaded data. A higher average rate (7.3x) was recorded for the increase in loading time while we increase the scale factor of the loaded data into the compressed sparse row (CSR). We recorded a higher time taken in loading the CSR structure than that taken in loading the adjacency list across all the scale factors. The loading time of CSR was on average (54) times the loading time of adjacency list.
In regard to the memory footprint of stored data, we observed that the memory footprint of adjacency list is increasing with an average rate of (2.315x) as we increase the scale factor of the loaded data. A slightly higher average rate (2.329x) was recorded for the increase in memory footprint while we increase the scale factor of the loaded data into the compressed sparse row (CSR). In contrast to the difference in loading time between the adjacency list and the CSR structure, we recorded a higher memory footprint taken in storing the loaded data in the adjacency list than the memory footprint of storing the loaded data in the CSR structure across all the scale factors. The memory footprint of adjacency list was on average (1.56) times the memory footprint of CSR.

Concerning the adjacency matrix, we failed to load any of the scale factors of the evaluation dataset except for the smallest (0.25) scale factor. Loading the evaluation dataset with the (0.25) scale factor has consumed around 80 GiB. We ran out of memory trying to load the evaluation dataset with the (0.5) scale factor due to memory limitation of the machine used in running the evaluation experiments which is 128 GiB. However, the time taken to load the evaluation dataset with the (0.25) scale factor was (16) times the corresponding loading time of adjacency list and (6) times the loading time of CSR. Meanwhile, the memory footprint of the data stored in adjacency matrix was (425) times the corresponding memory footprint of adjacency list and (669) times the memory footprint of CSR.

In (Figure 5.1), we show two graphs depicting the scalability of adjacency list and compressed sparse row in terms of loading time and memory footprint. In (Figure 5.1(a)), we show a chart with the change in the loading time of each scale factor of the evaluation dataset into the adjacency list and the compressed sparse row (CSR). In (Figure 5.1(b)), we show a chart with the change in the memory footprint of each scale factor of the evaluation dataset into the adjacency list and the compressed sparse row (CSR). In (Table 5.1), we present a table containing the exact figures we recorded for each of the scalability experiments we executed on the adjacency list, the compressed sparse row (CSR), and the adjacency matrix.

**Explanation:**
The (`std::vector`) data structure forces a data movement operation to occur for all the data following an inserted element in any place other than the end of the vector in order to free room for storing it. Both, compressed sparse row (CSR) and adjacency list are utilizing the (`std::vector`) data structure to store their data. However, as the loaded elements into the adjacency list are splitted up on several (`std::vector`) structures, the accumulative number of elements that needs to be displaced each time a new element comes is far fewer than in CSR, which in return causes the CSR structure to take more time in loading the data than the adjacency list.

The (`std::map`) data structure that is used by adjacency list and CSR has a complexity of (O(log(n))) for insertion which means as the number of elements stored in the data structure increases the insertion time will also increase.
The increase in insertion time into the `std::map` participates into increasing the average rate of loading time to more than (2.0x) for both: adjacency list and CSR.

In regard to the memory footprint, the `std::vector` data structure which is utilized by compressed sparse row (CSR) and adjacency list to store their data is consuming extra memory in order to manage the container, besides the memory already consumed for storing the elements themselves. By using only two `std::vector` data structures in CSR the additional memory overhead is small and negligible to the memory consumed by the stored elements. On the contrary, the number of `std::vector` data structures used in adjacency list is equal to the number of vertices in the graph which makes explains the more memory consumed to store the data in comparison to the CSR structure.

The adjacency matrix memory consumption is much higher in comparison to CSR and adjacency list due to the memory requirements of adjacency matrix which is \( n^2 \) bits, where \( n \) is the number of vertices in the graph. Also, the loading time of adjacency matrix is relatively high due to the need to expand each of the `std::vector` data structures of adjacency matrix every time a new element needs to be included into the matrix.

- **Conclusion:**
  To answer question number (1) from the evaluation questions listed in (Section 3.1), the evaluation results of the scalability of the three graph topology structures (adjacency list, compressed sparse row (CSR), and adjacency matrix), are showing that the CSR structure is more suitable for use-cases where memory saving is the most important factor. However, adjacency list is offering a much better performance in terms of loading time with an average of (1.56) times the memory consumption of CSR. Lastly, adjacency matrix huge memory consumption makes it a poor scalable structure and makes it only useful for use-cases where the dataset is tiny.

### 5.1.2 Scalability: Graph Properties Structures

In this section, we present the evaluation results regarding the scalability of the graph properties structures (universal table, emerging schema, and nested key-value store).

- **Experiment Setup:**
  In this experiment we used the batch data loader to load the evaluation dataset into the different graph properties structures. We fixed the batch size to 1000 record per batch. For each graph properties structure, we measured the loading time and memory footprint when loading each of the scale factors of the evaluation dataset. We executed each test for twenty times and recorded the loading time and memory footprint of each run. We took the average of the measured figures after removing the most top and least figures.
5. Evaluation: Scalability and Data Loading

(a) Change in loading time.

(b) Change in memory footprint.

Figure 5.2: Effect of increasing data size on the scalability of graph properties structures.

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>Universal Table Loading Time (Second)</th>
<th>Emerging Schema Loading Time (Second)</th>
<th>Nested Key-Value Store Loading Time (Second)</th>
<th>Universal Table Memory Footprint (MiB)</th>
<th>Emerging Schema Memory Footprint (MiB)</th>
<th>Nested Key-Value Store Memory Footprint (MiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>7</td>
<td>15</td>
<td>7</td>
<td>492</td>
<td>463</td>
<td>612</td>
</tr>
<tr>
<td>0.5</td>
<td>17</td>
<td>38</td>
<td>17</td>
<td>1,180</td>
<td>1,108</td>
<td>1,469</td>
</tr>
<tr>
<td>1</td>
<td>43</td>
<td>94</td>
<td>43</td>
<td>2,836</td>
<td>2,649</td>
<td>3,526</td>
</tr>
<tr>
<td>2</td>
<td>106</td>
<td>230</td>
<td>108</td>
<td>6,702</td>
<td>6,243</td>
<td>8,317</td>
</tr>
<tr>
<td>4</td>
<td>263</td>
<td>557</td>
<td>267</td>
<td>15,448</td>
<td>14,358</td>
<td>19,140</td>
</tr>
</tbody>
</table>

Table 5.2: Evaluation results of graph properties structures scalability.
5.1. **Scalability**

- **Expected Result:**
  We expect nested key-value store and universal table to record the same loading time of the datasets. We expect the loading time of emerging schema to be the larger than the loading time of nested key-value store and universal table.

  In regard to the memory footprint, we expect the memory footprint of universal table to be the largest. We expect the memory footprint of emerging schema to be smaller than that of nested key-value store.

- **Observation:**
  We observed that the loading time of all the three graph properties structures (universal table, emerging schema, and nested key-value store) is increasing with the same average rate of (2.4x) as we increase the scale factor of the loaded data.

  We recorded that almost the same time was taken in loading the universal table and the nested key-value store with a slight advantage for the universal table when loading the evaluation dataset with scale factors (2 and 4). The loading time of emerging schema was on average (2.17) times the loading time of universal table.

  In regard to the memory footprint of stored data, we observed that the memory footprint of all the three graph properties structures (universal table, emerging schema, and nested key-value store) is increasing with the same average rate of (2.4x) as we increase the scale factor of the loaded data which is the same average rate we recorded for the increase in the loading time. The emerging schema structure has recorded the lowest memory footprint among the three graph properties structures across all the evaluation dataset scale factors. In the second place came the universal table with a memory footprint that on average equals (1.07) times the memory footprint of emerging schema. The highest memory footprint was recorded for the nested key-value store with an average of (1.33) times the memory footprint of emerging schema and (1.24) times the memory footprint of universal table.

  In (Figure 5.2), we show two graphs depicting the scalability of universal table, emerging schema, and nested key-value store in terms of loading time and memory footprint. In (Figure 5.2(a)), we show a chart with the change in the loading time of each scale factor of the evaluation dataset into the three graph properties structures and in (Figure 5.2(b)), we show a chart with the change in the memory footprint. In (Table 5.2), we present a table containing the exact figures we recorded for each of the scalability experiments we executed on the universal table, emerging schema, and nested key-value store.

- **Explanation:**
  The (**std::map**) data structure that is used by universal table, emerging schema and nested key-value store has a complexity of \(O(\log(n))\) for insertion which means as the number of elements stored in the data structure increases the insertion time will also increase. The increase in insertion time into the (**std::map**) participates into increasing the average rate of loading time to more than (2.0x) for the three graph properties structures.
The loading time in universal table and nested key-value store is almost equal because both data structure utilize the \texttt{(std::map)} data structure as their main container for storing the graph properties. The clustering operation that takes place after loading the data in emerging schema is causing the more loading time that emerging schema is taking in comparison to universal table and nested key-value store.

In regard to the memory footprint, nested key-value store has recorded the largest memory footprint among all the three graph topology structures due to the memory that is used to store the properties names in each and every row in addition to the properties values. The emerging schema structure has benefited from the vertical partitioning of the data into a set of column groups to record less memory footprint than that of universal table. The vertical partitioning used in emerging schema is reducing the memory consumption by eliminating most of the null values and not storing them.

**Conclusion:**

To answer question number (1) from the evaluation questions listed in ([Section 3.1]), the evaluation results of the scalability of the three graph properties structures (universal table, emerging schema, and nested key-value store), are showing that the emerging schema structure is more suitable for use-cases where memory saving is the most important factor, thanks to the vertical partitioning method it is using. However, universal table and nested key-value store are offering a better performance in terms of loading time.

### 5.2 Batch Data Loading

In this section, we present the evaluation results regarding the effect of batch size on the loading time of graph structures. Due to the scalability issue with adjacency matrix which we explained in ([Figure 5.1]), we were not able to consider adjacency matrix in this test.

**Experiment Setup:**

In this experiment we used the batch data loader to load the evaluation dataset into the different graph structures. We loaded the data into each graph structure with batch sizes set to (1, 10, 100, 1000, 10000, and 100000). We used the evaluation dataset with scale factor (1) in this test. For each graph structure, we measured the loading time when loading the evaluation dataset using each of the of the batch sizes. We executed each test for ten to twenty times and recorded the loading time of each run. We took the average of the measured figures after removing the most top and least figures.

**Expected Result:**

For all graph structures we expect the loading time to decrease as we increase in batch size of the loaded data.
5.2. Batch Data Loading

(a) Change in loading time of graph topology structures.

(b) Change in loading time of graph properties structures.

Figure 5.3: Effect of increasing batch size on the loading time of graph structures.

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>Graph Topology Structures</th>
<th>Graph Properties Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Adjacency List</td>
<td>CSR</td>
</tr>
<tr>
<td>1</td>
<td>26.641</td>
<td>496.502</td>
</tr>
<tr>
<td>10</td>
<td>24.476</td>
<td>491.992</td>
</tr>
<tr>
<td>100</td>
<td>25.087</td>
<td>495.490</td>
</tr>
<tr>
<td>1000</td>
<td>25.247</td>
<td>499.597</td>
</tr>
<tr>
<td>10000</td>
<td>25.855</td>
<td>505.502</td>
</tr>
<tr>
<td>100000</td>
<td>27.046</td>
<td>514.883</td>
</tr>
</tbody>
</table>

Table 5.3: Evaluation results of batch loading time of graph structures in seconds.
• **Observation:**
  We observed that by increasing the batch size, the change in loading time of all the graph structures was too small. The loading time for the graph topology structures started to decrease at first and recorded the lowest loading time when the batch size was set to (10) and then started to increase to record the highest loading time at (100000). Similar observation was made for graph properties structures as the loading time started to decrease at first and recorded the lowest loading time when the batch size was set to (100) and then started to increase to record the highest loading time at (100000).

  In (Figure 5.3), we show two graphs depicting the change in loading time of the graph structure as an effect of the change in the batch size. In (Figure 5.3(a)), we show a chart with the change in the loading time that corresponds to each batch size of the data loaded into the graph topology structures and in (Figure 5.3(b)), we show a chart with the change in the loading time that corresponds to each batch size of the data loaded into the graph properties structures. In (Table 5.3), we present a table containing the exact figures we recorded for each of the batch loading experiments we executed on the graph structures.

• **Explanation:**
  Our expectation that the loading time will decrease as we increase the batch size was based on the number of times the batch loading procedure in (Listing 4.1) will need to push the buffer contents into the graph structure and then clear the buffer. The results was not fully consistent with our expectation.

  We used the *(perf)* tool that is part of the *(ubuntu)* operating system to further analyze the performance of the tests from hardware perspective\[1\].

<table>
<thead>
<tr>
<th>Batch size</th>
<th># Buffer Flushes</th>
<th>Cache Misses (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15,530,170</td>
<td>10.848</td>
</tr>
<tr>
<td>10</td>
<td>1,553,017</td>
<td>10.923</td>
</tr>
<tr>
<td>100</td>
<td>155,302</td>
<td>11.639</td>
</tr>
<tr>
<td>1000</td>
<td>15,531</td>
<td>13.462</td>
</tr>
<tr>
<td>10000</td>
<td>1,553</td>
<td>18.542</td>
</tr>
<tr>
<td>100000</td>
<td>156</td>
<td>43.622</td>
</tr>
</tbody>
</table>

Table 5.4: The change in the number of buffer flushes and the percentage of cache misses with the different batch sizes.

\[1\] For more information on the *(perf)* tool see: http://manpages.ubuntu.com/manpages/bionic/man1/perf.1.html
5.3 Parallel Data Loading

We observed in the statistics collected using the \texttt{(perf)} tool that the percentage of cache misses to the total number of cache references was increasing by the increase in the batch size. The copy of the buffer contents into the graph structure is the main source of the cache misses. The bigger the buffer size gets, the more it can’t fit totally into the cache memory and the more cache misses we get.

The loading with smaller batch sizes has a better cache utilization with an average of only (10.84 \%) when we set the batch size to (1). However, the number of times we flush the buffer contents into the graph structure is the highest (15.5 million times on a batch size of (1)). On the contrary, the loading with larger batch sizes has a worse cache utilization with an average of (43.62 \%) when we set the batch size to (100000). However, the number of times we flush the buffer contents into the graph structure is the lowest (156 times on a batch size of (100000)).

The effect of the percentage of cache misses and the number of times we flush the buffer are contradicting each other. This leads to the decrease in loading time at the lower batch sizes when the cache misses are at its lowest and the increase in loading time at the larger batch sizes when the number of times we flush the buffer contents is the smallest.

We show in \textbf{(Table 5.4)}, the change in the number of times we flush the buffer as well as the percentage of cache misses caused by the change in batch size.

\begin{itemize}
  \item \textbf{Conclusion:}
    To answer question number (2) from the evaluation questions listed in \textbf{(Section 3.1)}, the evaluation results of the batch loading of the graph structures, are showing that there is a minimal effect for the batch size on the loading time. The minimal effect of batch size is because all the data is main memory resident and no disk storage is involved. The batch size setting need to be carefully tuned to reach the lowest loading time. In our experiments, we found the batch size (10) is providing the lowest loading time for the graph topology structures and the batch size (100) is providing the lowest loading time for the graph properties structures.
\end{itemize}

5.3 Parallel Data Loading

In this section, we present the evaluation results regarding the effect of parallel loading on the loading time of parallel adjacency list and parallel nested key-value store.

\begin{itemize}
  \item \textbf{Experiment Setup:}
    In this experiment we used the parallel data loader to load the evaluation dataset into the parallel adjacency list and the parallel nested key-value store structures. We executed the loading with a parallelism degree of (1, 2, 4, 8, and 16). We fixed the buffer size of the parallel loader to (1000). We loaded the evaluation dataset with scale factor (1). We executed each test for twenty times and recorded the loading time of each run. We took the average of the measured figures after removing the most top and least figures.
\end{itemize}
5. Evaluation: Scalability and Data Loading

(a) Change in loading time of parallel adjacency list.

(b) Change in loading time of parallel nested key-value store.

Figure 5.4: Degree of parallelism effect on the loading time of parallel adjacency list and parallel nested key-value store.

<table>
<thead>
<tr>
<th>Parallelism Degree</th>
<th>Parallel Adjacency List</th>
<th>Parallel Nested Key-Value Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32.109</td>
<td>49.638</td>
</tr>
<tr>
<td>2</td>
<td>25.205</td>
<td>29.524</td>
</tr>
<tr>
<td>4</td>
<td>23.632</td>
<td>22.439</td>
</tr>
<tr>
<td>8</td>
<td>24.784</td>
<td>24.324</td>
</tr>
<tr>
<td>16</td>
<td>33.492</td>
<td>28.941</td>
</tr>
</tbody>
</table>

Table 5.5: Evaluation results for time of parallel loading of the parallel adjacency list and the parallel nested key-value store in seconds.
• **Expected Result:**
  We expect the loading time of parallel adjacency list and parallel nested key-value store to decrease as we increase in batch size of the loaded data.

• **Observation:**
  We observed that by increasing the parallelism degree, the loading time of parallel adjacency list and parallel nested key-value store decreases just until we reach the parallelism degree of (4). On parallelism degree (4), the two parallel graph structures scored their least loading time. However, the loading time of both graph structures has started to increase once we set the parallelism degree higher than (4).

  With the parallelism degree set to (4), the loading time of parallel adjacency list was less by (6 to 16%) than the corresponding loading time of the sequential version of adjacency list. The parallel nested key-value store has scored a loading time that is less by (47 to 52%) than the corresponding loading time of the sequential version of nested key-value store, also with the parallelism degree set to (4).

  On parallelism degree (1), both: the parallel adjacency list and the parallel nested key-value store recorded a loading time that is higher than the corresponding loading time of the sequential version.

  In (Figure 5.4), we show two graphs depicting the change in loading time of parallel adjacency list and parallel nested key-value store as an effect of the change in the parallelism degree of the parallel loader. In (Figure 5.4(a)), we show a chart with the change in the loading time that corresponds to each degree of parallelism when loading the evaluation dataset with scale factor (1) into the parallel adjacency list and in (Figure 5.4(b)), we show a similar chart for the change in the loading time of the parallel nested key-value store. In (Table 5.5), we present a table containing the exact figures we recorded for each of the parallel loading experiments we executed on the parallel adjacency and the parallel nested key-value store.

• **Explanation:**
  The observed behaviour of decrease in the loading time as we increase the parallelism degree only up to the parallel degree of (4) is due to the maximum ability of the physical machine processor to execute only (4) threads in parallel. Increasing the number of threads to more than (4) has caused the loading time to increase due to the overhead of synchronization required to be done by the operating system between the threads in order to execute the instructions of all threads.

• **Conclusion:**
  To answer question number (3) from the evaluation questions listed in (Section 3.1), the evaluation results are showing that the parallel loading of data into the parallel adjacency list and the parallel nested key-value store has decreased the loading time by (6 to 16%) for the parallel adjacency list and (47 to 52%) for the parallel nested key-value store over their sequential counterparts.
The maximum decrease in loading time was achieved when the parallelism degree was set to (4). The correct choice of parallelism degree should be configured according to the parallelism capabilities of the processor.

5.4 Summary

In this chapter, we presented the first part of our evaluation results, with which we answered the evaluation questions number 1, 2, and 3 stated in [Section 3.1]. We presented the loading time and memory footprint scalability test for the different graph structures. We presented the effect of batch size on the loading time of graph structures. We showed the possible improvement that parallel loading of data can introduce over sequential loading.

In next chapter, we present the second and last part of our evaluation results, with which we will answer the evaluation question number 4 stated in [Section 3.1].
6. Evaluation: Queries

In (Chapter 5), we presented the first part of the evaluation results for the experiments we conducted on the graph structures in order to evaluate their performance. With the first part of the evaluation results, we managed to give answers for the evaluation questions (1, 2, and 3) which we have stated in (Section 3.1) concerning the scalability, batch loading, and parallel loading of the graph structures.

In this chapter, we present the second and last part of the evaluation results for the experiments we conducted on the graph structures. The evaluation experiments which we present in this chapter are aiming to evaluate the performance of the graph structures in query execution. All the experiments we conducted are using the evaluation dataset which we presented in (Section 4.2). We aim to exploit the evaluation results which we present in this chapter in order to give answers for the evaluation question number (4) which we have stated in (Section 3.1) concerning the effect of graph structure choice on the query response time. This chapter is consisted of the following sections:

- **Evaluation: Query #1:**
  In section (Section 6.1), we present the evaluation results concerning the performance of the graph data structures to execute Query #1.

- **Evaluation: Query #2:**
  In section (Section 6.2), we present the evaluation results concerning the performance of the graph data structures to execute Query #2.

- **Evaluation: Query #3:**
  In section (Section 6.3), we present the evaluation results concerning the performance of the graph data structures to execute Query #3.

- **Summary:**
  Lastly, we provide a summary for what we discussed in the chapter. (Section 6.4)
6. Evaluation: Queries

6.1 Evaluation: Query #1

In this section, we present the evaluation results for the query response time of Query #1 when executed on the data stored in the different graph structures. We defined Query #1 in [Section 4.4.1]. Query #1 is an example of a simple pattern-matching query that requires no traversals between the graph elements.

• **Experiment Setup:**
In this experiment we used the batch data loader to load the evaluation dataset into the graph structures. We fixed the batch size of the batch loader to (1000). Query #1 requires only as an input the properties of each vertex, therefore we loaded the evaluation dataset only into the graph properties structures. We loaded the evaluation dataset with scale factor (4). We loaded the data into each graph properties structure and executed the query for twenty times and recorded the query response time of each run. We took the average of the measured figures after removing the most top and least figures.

• **Expected Result:**
We expect the response time of Query #1 on universal table to be the lowest and on emerging schema to be the highest.

• **Observation:**
We observed that universal table has recorded the lowest response time for Query #1 with only (13.901 seconds). Also, we observed that the nested key-value store came in the second place with a response time of (15.285 seconds) and at the last place came the emerging schema structure with (15.293 seconds) not far from the nested key-value store.

In [Figure 6.1], we show a chart depicting Query #1 response time when executed on the different graph properties structures.

![Figure 6.1: Response time of Query #1.](image-url)
• Conclusion:
To answer question number (4) from the evaluation questions listed in (Section 3.1),
the evaluation results are showing that the universal table structure has the lowest
response time when executing a simple pattern-matching query that requires no
traversals between the graph elements. Universal table response time for Query #1
is less by (9%) than the response time of emerging schema and nested key-value
store.

6.2 Evaluation: Query #2

In this section, we present the evaluation results for the query response time of Query #2
when executed on the data stored in the different graph structures. We defined Query #2
in (Section 4.4.2). Query #2 is an example of a complex pattern-matching query that
requires traversals between the graph elements in order to fulfill the query requirements.

• Experiment Setup:
In this experiment we used the batch data loader to load the evaluation dataset
into the graph structures. We fixed the batch size of the batch loader to (1000).
Query #2 requires as an input both: the properties of each vertex as well as the
topology of the graph, therefore we loaded the evaluation dataset into every possible
combination of a graph properties structure and a graph topology structure. We
loaded the evaluation dataset with scale factor (4). We excluded the adjacency
matrix out of this test due to the scalability issue we faced when we tried to
load the evaluation dataset into the adjacency matrix (see Section 5.1.1). We
loaded the data into each graph structure and executed the query for twenty times
and recorded the query response time of each run. We took the average of the
measured figures after removing the most top and least figures.

Figure 6.2: Response time of Query #2.
• **Expected Result:**
We expect the response time of Query #2 on the combination of graph structures where the compressed sparse row (CSR) is part of to be lower than the response time of those where the adjacency list is part of. Also, we expect the response time of Query #2 on the combination of graph structures where the universal table is part of to be lower than the response time of those where the emerging schema or the nested key-value store is part of.

• **Observation:**
We observed that the combination of compressed sparse row (CSR) along with nested key-value store has recorded the lowest response time for Query #2 with only (15 seconds). Also, we observed that the combination of adjacency list along with emerging schema has recorded the highest response time (18.819 seconds).

The combination of graph structures where the nested key-value store is part of has recorded lower response time for Query #2 than the response time of those where the universal table or the emerging schema is part of. Also, the combination of graph structures where the compressed sparse row (CSR) is part of has recorded lower response time for Query #2 than the response time of those where the adjacency list is part of.

In [Figure 6.2](#), we show a chart depicting Query #2 response time when executed on the different combinations of graph properties structures and graph topology structures.

• **Conclusion:**
To answer question number (4) from the evaluation questions listed in [Section 3.1](#), the evaluation results are showing that the combination of compressed sparse row (CSR) along with nested key-value store has the lowest response time when executing a complex pattern-matching query that requires traversals between the graph elements in order to fulfill the query requirements.

### 6.3 Evaluation: Query #3

In this section, we present the evaluation results for the query response time of Query #3 when executed on the data stored in the different graph structures. We defined Query #3 in [Section 4.4.3](#). Query #3 is computing the performing a degree centrality report for the graph topology.
6.3. Evaluation: Query #3

• **Experiment Setup:**
  In this experiment we used the batch data loader to load the evaluation dataset into the graph structures. We fixed the batch size of the batch loader to (1000). Query #3 requires only as an input the properties of each vertex, therefore we loaded the evaluation dataset only into the graph topology structures. We loaded the evaluation dataset with scale factor (4). We excluded the adjacency matrix out of this test due to the scalability issue we faced when we tried to load the evaluation dataset into the adjacency matrix (see [Section 5.1.1]). We loaded the data into each graph structure and executed the query for twenty times and recorded the query response time of each run. We took the average of the measured figures after removing the most top and least figures.

• **Expected Result:**
  We expect the response time of Query #3 on adjacency list to be lower than that of compressed sparse row (CSR).

• **Observation:**
  We observed that adjacency list has recorded a slightly lower response time for Query #3 with only (162 seconds). The response time of the compressed sparse row (CSR) was slightly higher than the response time of adjacency list and recorded a response time of (165 seconds).

  In [Figure 6.3](#), we show a chart depicting Query #3 response time when executed on adjacency list as well as compressed sparse row (CSR).

• **Conclusion:**
  To answer question number (4) from the evaluation questions listed in [Section 3.1], the evaluation results of executing a degree centrality query are showing that the response times of the adjacency list and the compressed sparse row (CSR) were very close with a small advantage for the adjacency list structure. Adjacency list response time for Query #3 is less by (2%) than the response time of CSR.
6.4 Summary

In this chapter, we presented the second and last part of our evaluation results, with which we answered the evaluation question number (4) stated in [Section 3.1]. We presented the difference in response time of three different query when executed on a different graph structures.

In next chapter, we present other work that is similar or related to our work in this thesis.
7. Related Work

In (Chapter 6), we presented the second and last part of the evaluation results for the experiments we conducted on the graph structures in order to evaluate the performance of the graph structures in query execution. In this chapter, we present an overview of other research that is related to the work we did in this thesis. This chapter is consisted of the following sections:

- **Graph Structures Scalability:**
  In section (Section 7.1), we present other research that have been done to evaluate the scalability of graph data structures.

- **Loading Techniques:**
  In section (Section 7.2), we present other research that have been done to evaluate the performance of graph data loading techniques.

- **Query Performance:**
  In section (Section 7.3), we present other research that have been done to evaluate the performance of query execution on data in the graph data structures.

### 7.1 Graph Structures Scalability

The scalability of a graph data structure is determined by its ability to process larger graphs. Scalability is measured by the amount of time and memory taken to load, modify, or perform computation on a graph with larger size. In a research done by (Wheatman and Xu), the authors presented a scalability evaluation for the loading, updating, and querying of a packed version of the compressed sparse row (CSR) structure against basic CSR, adjacency list, and other variants of adjacency list [WX18]. King et al. have presented a scalability evaluation for a dynamic version of CSR for (GPU) processing [KGKM16].
7.2 Loading Techniques

Then et al. have presented evaluation of a set of strategies that can be utilized for the parallel loading of graph data into graph structures. The authors have tested their proposed parallel loading strategies on an implemented version of the compressed sparse row (CSR) and map of neighbour list [TKKN16].

7.3 Query Performance

Abadi et al. have presented an evaluation for the performance of query execution on a set of graph properties structures that included the emerging schema structure as well as other graph properties structures [AMMH07]. Wheatman and Xu as well as Blandford et al. have presented an evaluation for the performance of query execution on a set of graph topology structures that included the compressed sparse row structure and the adjacency list structure among others [WX18, BBK04].
8. Conclusion and Future Work

In [Chapter 7], we presented an overview of other research that is related to our work in this thesis. In this chapter, we make a conclusion of the thesis as well as proposing points where our research can be extended in a future work. This chapter is consisted of the following sections:

- **Conclusion:**
  In section ([Section 8.1]), we draw a conclusion of the work presented in this thesis.

- **Future Work:**
  In section ([Section 8.2]), we propose how future work can extend the work we have done in this thesis.

8.1 Conclusion

The emergence in the use of specialized graph databases has imposed many challenges to the graph database designers in making their design choices. The graph structures utilized by graph databases for the storage and processing of graph data are contributing to the overall performance of the database, therefore choosing the proper graph structure that will best fits the user needs is of important weight. In this section, we conclude the main points we presented in this thesis on assessing the performance of graph structures.

- **Graph Structures Scalability:**
  The scalability of the three graph topology structures (adjacency matrix, compressed sparse row, and adjacency list) is contrasting. Adjacency matrix is the worst of the three in scalability specially from memory consumption point of view. Adjacency list takes the least time to load graph data, on the other hand compressed sparse row (CSR) makes the least memory footprint. Therefore, adjacency
list is most recommended when loading time is crucial. On the other hand, CSR is the graph structure to use whenever a limitation in memory exists.

Due to its vertical partitioning based technique, the emerging schema method consumes the least memory among the three graph properties that we tested. However, universal table and nested key-value store are taking an almost equal loading time which is less than the loading time of emerging schema. Therefore, we recommend the emerging schema method in scenarios where memory saving has a higher weight than loading time. Both universal table or nested key-value store are more suitable for faster data loading than emerging schema.

- **Loading Techniques:**
  Changing the batch size in the batch loading strategy doesn't make a significant influence on the loading time of any of the graph structures. On the contrary, the parallel loading strategy improves the performance of data loading in both adjacency list and nested key-value store especially when the degree of parallelism is configured to match the maximum degree of parallelism offered by the processor.

- **Query Performance:**
  In regard to simple pattern-matching queries that require no traversals between the graph vertices, universal table shows the best query performance with least query response time. However, the combination of compressed sparse row (CSR) along with nested key-value store gives the best performance in executing complex pattern-matching queries that require traversals between the graph vertices. Lastly, adjacency list has small advantage ahead of CSR in regard to the performance of degree centrality queries.

### 8.2 Future Work

Despite our comprehensive evaluation of the major available graph data structures concerning their scalability, loading techniques, and query performance; their still exist points which we didn’t cover and can be considered in future work. We summarize those points in the following list.

- **Data Manipulation Workload:**
  We have tested the performance of the graph structures in terms of the time taken in loading a graph data. The performance of graph structures in workloads that include manipulation of data by changing, deleting, or inserting values is a factor that influences the choice of a graph structure over another. Evaluating the performance of graph structures under a data manipulation workload need to be considered in a future work.
8.2. Future Work

• **Concurrency:**
  A typical graph database will offer access to multiple users simultaneously. The concurrent access by users needs special handling by the database to schedule the operations of the users in a way that returns the expected result to the user and at the same time maintains the consistency of the database. Future work may consider the performance of the graph structures in performing concurrent operations requested by multiple users.

• **Reachability, and Analytical Queries:**
  In this thesis, we have examined the performance of the graph structures in the execution of two pattern-matching queries in addition to a degree centrality summarizing query. Other categories of queries haven’t been covered in this thesis. Reachability queries are testing the possibility of reaching one vertex from another via a path of connected edges. Analytical queries are complex queries that perform statistical or analytical computation over the graph data. An example of an analytical query is the PageRank algorithm [AG18].
Bibliography


[Har69] Frank Harary. Graph theory, 1969. (cited on Page xiii and 3)


