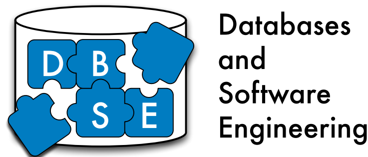
Otto-von-Guericke-Universität Magdeburg

Faculty of Computer Science



Master's Thesis

An Evaluation of the Design Space for Scalable Data Loading into Graph Databases

Author:

Jingyi Ma

February 23, 2018

Advisors:

M.Sc. Gabriel Campero Durand

Data and Knowledge Engineering Group

Prof. Dr. rer. nat. habil. Gunter Saake

Data and Knowledge Engineering Group

Ma, Jingyi:
An Evaluation of the Design Space for Scalable Data Loading into Graph Databases
Master's Thesis, Otto-von-Guericke-Universität Magdeburg, 2018.

Abstract

In recent years, computational network science has become an active area. It offers a wealth of tools to help us gain insight into the interconnected systems around us. Graph databases are non-relational database systems which have been developed to support such network-oriented workloads. Graph databases build a data model based on graph abstractions (i.e. nodes/vertexes and edges) and can use different optimizations to speed up the basic graph processing tasks, such as traversals. In spite of such benefits, some tasks remain challenging in graph databases, such as the task of loading the complete dataset. The loading process has been considered to be a performance bottleneck, specifically a scalability bottleneck, and application developers need to conduct performance tuning to improve it.

In this study, we study some optimization alternatives that developers have for load data into a graph databases. With this goal, we propose simple microbenchmarks of application-level load optimizations and evaluate these optimizations experimentally for loading real world graph datasets. We run our tests using *JanusGraphLab*, a *JanusGraph* prototype.

Specifically, we compared the basic loading process with bulk/batch transactions and client vs. server side loading. Furthermore, we considered partitioning the data and loading it using different clients. Additionally, we evaluate the potential benefits of different partitioning strategies. For these optimizations, we report performance gains changing hours-long loading processes into minutes-long. As a result we provide a novel tool for data loading in a scalable manner using different configurations, and a simple to use configuration for loading data with different schemas, into the JanusGraph database.

Finally, we summarize best practices for loading data into graph databases. We expect that this work can help readers to understand better how to optimize loading processes in a graph database and inspire them to contribute to this research.

Acknowledgements

By submitting this thesis, my long term association with Otto von Guericke University will come to an end.

First and foremost, I am grateful to my advisor M.Sc. Gabriel Campero Durand for his guidance, patience and constant encouragement without which this may not have been possible.

I would like to thank Prof. Dr. rer. nat. habil. Gunter Saake for giving me the opportunity to write my Master's thesis at his chair.

It has been a privilege for me to work in collaboration with the Data and Knowledge Engineering Group.

I would like to thank my family and friends, who supported me in completing my studies and in writing my thesis.

Declaration of Academic Integrity

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

Magdeburg, February 23rd 2018

Jingyi Ma

Contents

List of Figures **xiv**

1 Introduction **1**

2 Background **7**

2.1 Models . . . 8
2.1.1 Graph Model . . . 8
2.1.2 Property Graph Model . . . 10
2.2 Solutions for Graph Management . . . 11
2.2.1 Relational Databases . . . 13
2.2.2 Graph Databases . . . 17
2.2.2.1 Graph Databases with Native Storage . . . 21
2.2.2.2 Graph Databases with Non-Native Storage . . . 21
2.2.3 Large Scale Graph Processing Systems . . . 23
2.2.3.1 General Large Scale Processing Systems . . . 24
2.2.3.2 Graph-Specific Systems . . . 27
2.3 Challenges in Graph Processing . . . 28
2.4 Loading Tasks for Big Data . . . 29
2.4.1 Traditional Databases . . . 29
2.4.2 Loading Large Scale Data . . . 30
2.4.3 Loading in Graph Workloads . . . 30
2.5 Optimization Alternatives for Loading Data into Graph Databases . . . 33
2.5.1 Batch/Bulk Loading . . . 33
2.5.2 Parallel and Partitioning . . . 34
2.5.2.1 Partitioning Strategies . . . 34
2.6 Summary . . . 36

3 Microbenchmarking Application-Level Optimizations for Loading Graph Data **37**

3.1 Evaluation Questions . . . 38
3.2 JanusGraph . . . 39
3.3 JanusGraphLab . . . 43
3.3.1 Architecture of JanusGraphLab . . . 43
3.3.2 Data Loading in JanusGraphLab . . . 46

3.4 Measurement Process 47
3.5 Testing Framework 49
3.6 Experiment Environment 49
3.6.1 Backend Configuration 50
3.7 Datasets 51
3.7.1 Wikipedia Requests for Adminship (with text) 51
3.7.2 Google Web Graph 52
3.8 Summary 54

4 Case Study: Client vs. Server Side and Batching 55
4.1 Evaluation Questions 56
4.2 Loading Process from Client/Server Side 56
4.3 Batch Loading in JanusGraph 59
4.4 Microbenchmark 59
4.5 Best Practices 61
4.6 Summary 63

5 Case Study: Partitioning and Parallel Loading 65
5.1 Evaluation Questions 66
5.2 Partitioning and Parallel Loading 66
5.3 Partitioning and Parallel Loading, from the Publisher/Subscriber Perspective 67
5.4 Microbenchmarks 68
5.4.1 Parallel and Partitioning Loading without Batching 69
5.4.2 A Closer Look into Load Balancing with the Partitioning Strategies 72
5.5 Best Practices 78
5.6 Summary 79

6 Case Study: Combination of Batching, Partitioning and Parallelizing 81
6.1 Evaluation Questions 81
6.2 Microbenchmarks 82
6.2.1 Batch size = 10 82
6.2.2 Batch size = 100 84
6.2.3 Batch size = 1000 85
6.3 Best Practices 86
6.4 Summary 87

7 Conclusion and Future Work 89
7.1 Summary: Best Practices for Loading Scalable Data into Graph Databases 90
7.2 Threats to Validity 91
7.3 Concluding Remarks 92
7.4 Future Work 93

A The Impact of Edge-Ordering on Load Times with JanusGraph 95

A.1 Edge Loading with Different Sort Strategies 95
A.1.0.1 Sort Strategies 95

Bibliography **99**

List of Figures

1.1 Graph ETL and Data Analysis Pipeline 2

2.1 There are numerous types of graphs. Many of the formalisms described can be mixed and matched in order to provide the modeler with more expressiveness ([RN10]). 11

2.2 A property graph is a directed, labeled, attributed, multi-graph. The edges are directed, vertexes/edges are labeled, vertexes/edges have associated key/value pairs meta data (i.e. properties), and there can be multiple edges between any two vertexes. 12

2.3 A property graph, abstract from Wiki-RfA dataset. 12

2.4 A property graph, abstract from Google-Web dataset. 13

2.5 Evaluation of database models. Rectangles represent models, arrows represent influences, and circles represent theoretical developments. On the left hand side. there is a time-line in years[AG08]. 14

2.6 Neo4j Node and Relationship *Store File* Record Structure ([IR15]) 22

2.7 The Four Building Blocks of Column Family Storage ([IR15]) 22

2.8 Kafka APIs 26

2.9 Source File Examples: Edge Lists, from SNAP Astro-Physics Collaboration Dataset 31

2.10 Source File Examples: One Vertex And All Edges Configuration, from SNAP Wiki-Top Categories Dataset 31

2.11 Source File Examples: Implicit Entities, from SNAP Amazon Movie Reviews Dataset 32

2.12 Source File Examples: Encoded Properties, from SNAP Ego-Twitter Dataset 32

3.1 JanusGraph Data Model 41

3.2 Cassandra Physical Data Model 42
3.3 An Architectural Overview of the JanusGraphLab 44
3.4 Working Flow for Loading Data into JanusGraph using JanusGraphLab . 48
3.5 Topology for the Wiki-Vote Dataset (An Earlier Version of Wiki-RfA) . . 52
3.6 Topology for the Google Web Dataset 53

4.1 Client vs. Server-side Management of the Data Loading 58
4.2 Effect of Batch-Loading the Edges 61
4.3 Speedups from Batch-Loading the Edges 62

5.1 Loading Data with Apache Kafka 68
5.2 Loading Time Using Partitioning and Parallel approaches 71
5.3 Speedups from Loading Time using Partitioning and Parallel Approaches 72
5.4 Loading Time Using Different Partitioning Strategies without Batching . 73
5.5 Speedups from Load Using Different Partitioning Strategies without Batching 74
5.6 Measures to Compare Load distributions [PGDS+12]. 75

6.1 Loading Time Using Different Partitioning Strategies with Batch size = 10 83
6.2 Loading Time Using Different Partitioning Strategies with Batch Size = 100 84
6.3 Loading Time Using Different Partitioning Strategies with Batch Size = 1000 85

A.1 Average Response Time for Loading Edge Data with Different Sorts (Wiki-RfA) 96
A.2 LRU Cache Model Used to Estimate the Cost of A Sorted Load 97

1. Introduction

In this chapter, we will present the motivation behind the thesis, describe goals and outline its organization.

Our Big Data era can be defined by two key issues: the ever-expanding dataset size and an increase in data complexity. In order to solve these problems, new database models, such as **graph databases** and, in general, **NoSQL systems**, have been put into use more and more. Naturally, graphs can be used to simulate many interesting problems. For example, they can represent social networks, biological networks and road networks. Apart from the " blackboard-friendly" representation that graphs can provide, they can be used for different kind of data analysis, namely network analysis. In recent years these have been fundamental for some developments (e.g. the investigation of the Panama papers[1]). There has also been a growing interest in heterogeneous network analysis, a field that seeks to extract more information across diverse graph sources that model different kinds of relations in one dataset [YS12]. Based on these aspects, the interest in graph databases has been growing in recent years. Paired with this development there is a large amount of work related to benchmarking and standardizing graph technologies, such as research from LDBC ([LDBa]).

As an illustration, we include data from a recent survey ([SMS+17]). The paper is a survey from a team from the the University of Waterloo, conducted using almost 90 participants that represented developers of graph applications from different domains. They listed several popular software products for processing graphs (see Table 1.1). And they provided a list of work fields which people work in for graph processing tasks (see Table 1.2).

[1] https://neo4j.com/blog/icij-neo4j-unravel-panama-papers/

Technology	Software	#	Users
Graph Database System	ArrangoDB	40	233
	Caley	14	
	DGraph	33	
	JanusGraph	32	
	Neo4j	69	
	OrientDB	45	
RDF Engine	Apache Jena	87	115
	Sparksee	5	
	Virtuoso	23	
Distributed Graph Processing Engine	Apache Flink (Gelly)	24	39
	Apache Giraph	8	
	Apache Spark (GiraphX)	7	
QueryLanguage	Gremlin	82	82
GraphLibrary	Graph for Scala	4	97
	GraphStream	8	
	Graphtool	28	
	NetworkKit	10	
	NetworkX	27	
	SNAP	20	
Graph Visualization	Cytoscape	93	116
	Elasticsearch (X-Pack Graph)	23	
Graph Representation	Conceptual Graphs	6	6

Table 1.1: Graph-related Software Products

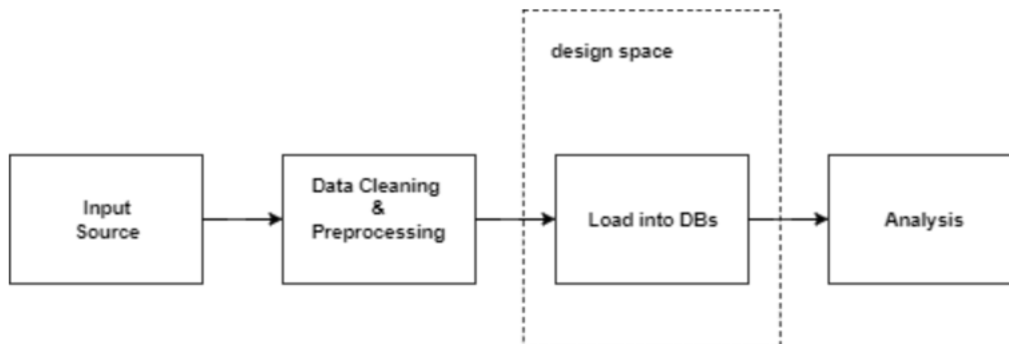


Figure 1.1: Graph ETL and Data Analysis Pipeline

Field	Total	R	P
Information & Technology	48	12	36
Research in Academia	31	31	0
Finance	12	2	10
Research in Industry Lab	11	11	0
Government	7	3	4
Healthcare	5	3	2
Defense & Space	4	3	1
Pharmaceutical	3	0	3
Retail & E-Commerce	3	0	3
Transportation	2	0	2
Telecommunications	1	1	0
Insurance	0	0	0
Other	5	2	3

Table 1.2: Work Fields for Graph Applications

An aspect mentioned in this survey as the biggest challenge faced by developers is scalability, and from this the most important problem is data loading. With the rate at which data is produced and gathered, the data loading process becomes more complex and requires careful engineering so as not to become a bottleneck in the whole graph ETL and data analysis pipeline. From Figure 1.1 we can observe a generic graph ETL and data analysis pipeline. In fact, optimizing the loading, such that it could be so fast as instant loading ([Mue13]), is essential for enabling analysis on more fresh data, a factor which could have potential valuable business impact in several domains.

In traditional disk-based relational database systems, loading performance depends on different factors. Here we list several factors and challenges we faced for good loading performance.

1) Studies ([ADHW99]) have experimentally shown that, at a low level, the CPU, memory and I/O system are the fundamental performance factor.

2) The exploitation of task and data parallelism (i.e., exploiting more cores, SIMD instructions or, in the case of clustered databases, more nodes) for speeding up the loading process.

3) Internal data representation (vectors or chunks) of the database and input file formats, which can influence the parsing process and the granularity of the parallelism possible.

4) Reasonably other factors could include the scope of the transactions (i.e. how many inserts per transaction) or domain-specific input data characteristics (e.g. compressed formats, access patterns for checking dependencies).

To our knowledge, though a significant body of work has considered graph query languages and general query execution, there seems to be a limited number of studies addressing the graph data loading process.

The work ([TKKN16]), focuses on optimizations at the level of DBMS design for loading a graph into an in-memory database. They propose to decompose the process into 1) Parsing (in which the vertex data and identifiers are loaded into memory). 2) Dense vertex identification (in which, for improving memory use, vertex identifiers are sorted based on their density or number of edges connected). 3) Relabeling, which is a form of in-memory dictionary encoding such that dense vertexes have smaller identifiers than large ones. 4) Finally writing the in-memory data structures that represent the graph (e.g. the authors consider compressed sparse rows and a map of neighbor lists). There is also existing offerings by major graph database vendors in terms of tools that automate the loading process (e.g. Neo4j's importer).

We believe that important research work is needed to establish and experimentally characterize the design space for the database application-level choices for performing the data loading. Among these choices we can highlight some like the effects of graph partitioning strategies, batching of transactions for the insertions, the overhead from combining loading with indexing, and others. Perhaps such a study is missing, not because of any specific complexity, but because of the recency of the technologies and the slow standardization process. In this Master thesis we aim to address this research gap.

In this study we propose the goal of optimizing the loading process from an application perspective (i.e., without changing the database internals). Based on previous research we establish several optimization choices and we study their impact using a commercial graph database **JanusGraph**, a distributed message bus service: Apache Kafka, and real world datasets, which are representative of diverse graph topologies. We designed several microbenchmarks carefully for the evaluation. In pursuing this study, our goal is to give developers a reliable list of best practices about application-level load optimizations for a given database. From our work we offer a novel, readily available server-side tool for scale-out loading of graphs into a graph database. To our knowledge this is a first-of-its-kind tool for the JanusGraph database, offering configurable options and support for ad-hoc graph schemas. In using and developing this tool we have found that a server-side implementation is necessary, and batching the loading process and parallelizing it can improve the performance notably. We also find that partitioning strategies can lead to differently balanced loads, and that EE

(i.e., distributing the edges in a round robin fashion) is a good match for a base case when not using batching. In addition we find that combining these optimizations is challenging, and not due to imbalances, but to causes which need further study and consideration. As a result we show that these loading alternatives have different influences on different real-world datasets, and have a practical importance for everyday usage of these databases. We can't give the reader a "one-size-fits-all" optimization for diverse application scenarios, but we provide several best practices to guide the search for the best strategy. With this work we aim to improve the understanding on the optimization alternatives for general loading tasks in a mainstream graph database, contributing to the goal of instant loading for graph databases and to foster more research in this field.

Our work is organized in the following way:

- **Introduction:**
  We present the motivation behind the thesis, describe goals and outline its organization. (Chapter 1)
- **Background:**
  We present an overview of the theoretical background and state of the art relevant to the current research work. (Chapter 2)
- **Microbenchmarking:**
  We define a set of core questions to evaluate our optimized graph loading alternatives and use these questions to structure our experiments. Then we introduce JanusGraphLab(Section 3.3) - our experimental prototype for microbenchmarking application-level graph loading optimizations. (Chapter 3)
- **Case Study: Client vs. Server Side and Batching:**
  We evaluate client vs. server side loading, and batched approaches to the loading process. (Chapter 4)
- **Case Study: Partitioning and Parallel Loading:**
  We present our evaluation on the performance of loading using partitioning and parallel strategies in JanusGraph. (Chapter 5)
- **Case Study: Combination of Batching, Partitioning and Parallelizing:**
  We document the experimental results of a simple microbenchmark to evaluate the combination of loading optimization alternatives: batching, partitioning and parallelization. (Chapter 6)
- **Conclusion and Future Work:**
  We conclude our studies in this chapter. (Chapter 7)

2. Background

In this chapter, we present an overview of the theoretical background and state of the art relevant to the current research work. Since our work is on loading data into graph databases, and the area of graph data management is still in development, in this chapter we do not attempt to provide a comprehensive survey on graph data management. Instead we aim to provide sufficient information for understanding the context of our research, and to present with care the main ideas necessary for understanding our research questions and focus. We outline the whole chapter as follows:

- **Models:**
  We begin by introducing the field of graph models in Section 2.1. This is necessary since these models form the backbone for graph databases.

- **Solutions for Graph Data Management:**
  Next we present the context in which graph technologies are being developed, in Section 2.2. Graph applications are generally managed in three ways: 1. with relational databases (Section 2.2.1), 2. with graph databases (Section 2.2.2) and 3. with Large Scale Graph Processing Systems (Section 2.2.3).

- **Loading Task:**
  An introduction to the loading tasks inside graph management systems. (Section 2.4)

- **Optimizations:**
  Based on the previous sections where we introduce graph data management and loading tasks, we present our main research focus: loading tasks optimization for graph databases. (Section 2.5)

- **Summary:**
  We conclude this chapter by summarizing our studies in Section 2.6.

2.1 Models

2.1.1 Graph Model

Why graph data?
Large graph datasets are everywhere. In fact a recent survey on the current uses of graph technologies ([SMS+17]) has found that the majority of practitioners use graphs composed of billions of nodes. Such scale of datasets is no longer only an issue of a few big Internet companies, but it is becoming the common use case of everyday practitioners. Today there are a lot of companies and research institutes which are finding benefits from using graph data, related to different real-world networks. Among them:

- social networks (e.g., LinkedIn, Facebook)
- scientific networks (e.g., Uniprot, PubChem)
- scholarly networks (e.g., MS Academic Graph)
- knowledge graphs (e.g., DBPedia)
- Others ([SMS+17])

Mathematically, a graph is a pair (N,E) where N is a finite set of nodes (also called vertexes) and

$$E \subseteq N \times L \times N$$

is a set of edges between nodes, with labels drawn from some domain L. The labels represent different relationships between nodes. For example in a social network the label could represent the friendship connection between two given users. Depending on the application, we can also extend graphs with node labels

$$\lambda : N- > L$$

In turn these node labels can represent types of nodes or identifiers of specific nodes. For example in a social network node types could be *users* or *institutions* to which groups of users might belong; on the other hand identifiers could be any single if assigned to a node. We can further extend graphs such that E is a finite multi-set and nodes/edges carry complex structured information. This information are called attributes or properties. The kind of information within a graph entity can be thought of as the schema of that specific entity. For example, we have a **property graph** if we extend graphs with node and edge properties

$$\lambda : N \bigcup E \times P- > Val$$

for P a set of property names and Val a domain of atomic values[1] ([FVY17]). Continuing with the example of the social network, node properties could be the name and email

[1] In more formal definition property graphs also need to be directed and multi-graphs, as explained in the next sub-section.

addresses of users; edge properties could include the date in which a given friendship relationship was created in the system.

A model is a representation of some aspect of reality. Graphically, an object in a network can be denoted by a dot (i.e. a node/vertex) and a relationship can be denoted by a line (i.e. an edge). A structure formed by dots and lines is known as a graph–the mathematical term for a network ([RN10]).

Models are important for discussing graphs, specifically graph database models, because a clear model provides a way to standardize the features that can be expected from using a given graph.

In addition, the discussion of graph data management usually needs to be decomposed into two aspects (as suggested by [AG17]), the first is graph database models, which refers to principles that ideally should guide the design and specific implementation of systems; the second are the graph management systems themselves, which process queries and need to develop solutions for data management challenges. In this section we focus on the models.

Data models for databases can be characterized by three basic components, data structures, query and transformation languages, and integrity constraints. Based on this, graph database models present a structure where data and the operations follow graph models and integrity constraints can be defined. These characteristics make graph database models easy to apply for representing unstructured data. An important aspect of these models is that in databases built on them the separation between schema and data (i.e., the specific instances to store) is less strict or evident than in the classical relational model ([AG17]).

Regarding the operations, we explain a bit more: when we discuss operations that follow graph models we refer to path queries, adjacency queries, neighborhood searches, subgraph extraction, pattern matching, connectivity/reachability queries, and analytical queries (such as community detection). Regarding integrity constraints, some examples are schema-instance consistency, identity (e.g., the same edge or the same id cannot be used twice) and referential integrity (e.g., every edge needs to have two connected vertexes), and functional and inclusion dependencies. Since neither operations nor integrity constraints are necessarily tied to a theoretical graph model (unlike the join operation which is tied to the relational model), it is very database-specific what operations and constraints will be supported.

In the next subsection we discuss one popular graph database model: the property graph model. We should note that neither operations nor integrity constraints form part of the following discussion.

2.1.2 Property Graph Model

In practice, a graph is rarely composed of only vertexes and edges. For instance, sometimes it's useful to have a name associated with a vertex, a weight and direction associated with an edge, etc. From primitive dots and lines various bits and pieces can be added to expose a more flexible, more expressive graph. Figure 2.1 shows a collection of different graph types ([RN10]).
And here we list some commonly used graph types:

- **directed graph:** orders the vertexes of an edge to present edge orientation. The line connecting vertexes has an arrow at the end.
- **multi graph:** there are multiple edges between the same two vertexes.
- **weighted graph:** used to represent strength of ties or transition probabilities.
- **Vertex/Edge labeled graph:** Vertexes can be labeled (e.g. identifier) or the edges can be labeled to represent the way in which two vertexes are related (e.g. friendships).
- **Vertex/Edge attributed graph:** 1. Attributes as non-relational meta data appended to vertexes. 2. Non-relational meta data to an edge.
- **Semantic graph:** It models cognitive structures such as the relationship between concepts and the instances of those concepts. Unlike the other definitions given in this list, this type of graph can be generalized to constitute a graph database model.
- **Half-edge graph:** A unary edge (i.e. an edge that "connects" one vertex)
- **Pseudo graph:** It's used to denote a reflexive relationship.
- **Hyper graph:** An edge could connect an arbitrary number of vertexes. This type of graph can also be used as a basis for a graph database model.
- **RDF graph:** A Resource Description Framework (RDF) graph restrict the vertex/edge labels to Uniform Resource Identifiers (URIs). RDF graphs can be semantic graphs. Through its use of URIs for referring to real-world entities, this model is connected to the semantic Web.

Another concept related to this classification is that of homogeneous or heterogeneous networks. Those with a single type of relationship and modeling conventions are called homogeneous. Heterogeneous networks, on the other hand, express more than one relationship and can have differing conventions for defining node types and properties.

For our study we focus on the **property graph model**, which is supported by most graph systems. The property graph model, also known as "property graph", is a **directed,**

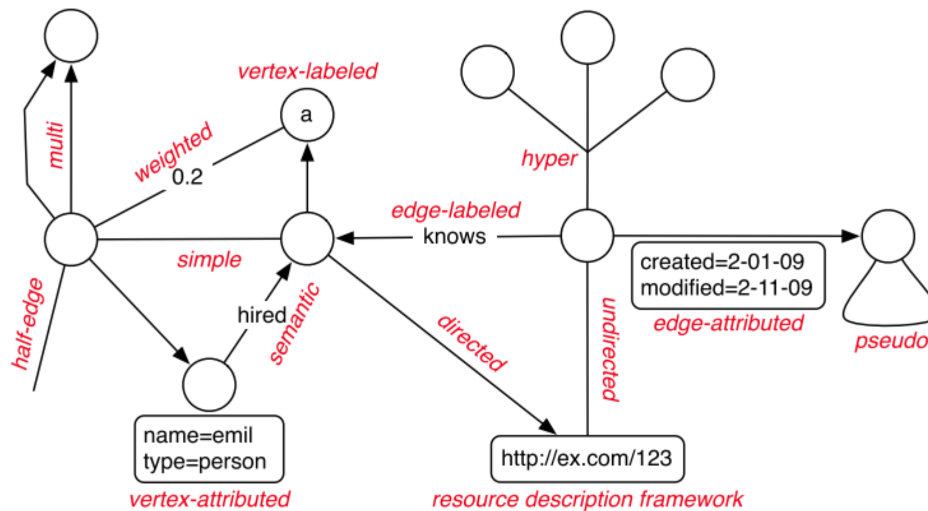


Figure 2.1: There are numerous types of graphs. Many of the formalisms described can be mixed and matched in order to provide the modeler with more expressiveness ([RN10]).

labeled, attributed, multi-graph. Graphs of this form allow for the representation of labeled vertexes, labeled edges, and attribute meta data (i.e. properties) for both vertexes and edges. Figure 2.2 gives an example of property graph. The high expressiveness of the property graph, which can express also RDF, makes the property graph one of the most popular graph data type.

Related to our experimental datasets. Here we give two more property graph examples in the form of the Wiki-RfA dataset (Figure 2.3) and Google-web dataset (Figure 2.4). Wiki-RfA dataset describes votes information for administrator in Wikipedia. Nodes represent Wikipedia members and edges represent votes to grant a member administrator rights. Nodes and edges both have attributes. Google-Web dataset describes google web graph. Nodes represent web pages and directed edges represent hyperlinks between them.

With this we conclude our presentation on graph models, which are necessary to introduce the fundamentals for graph data management. Next we overview existing solutions for graph data management.

2.2 Solutions for Graph Management

The volume and diversity of graph data are growing these days. So the management and analysis of huge graphs force developers to build powerful and highly parallel systems. The flexible and efficient management and analysis of “big graph data” not only holds promise, but also faces numerous challenges for suitable implementations in order to

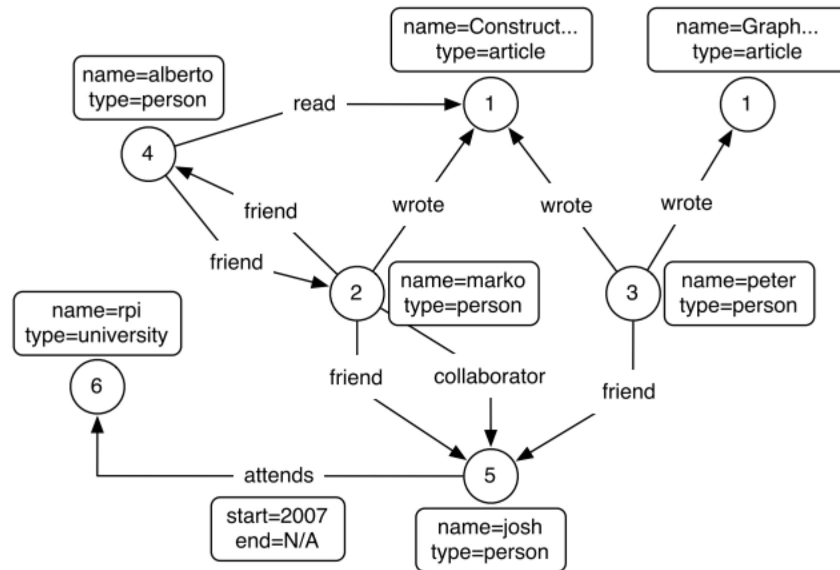


Figure 2.2: A property graph is a directed, labeled, attributed, multi-graph. The edges are directed, vertexes/edges are labeled, vertexes/edges have associated key/value pairs meta data (i.e. properties), and there can be multiple edges between any two vertexes.

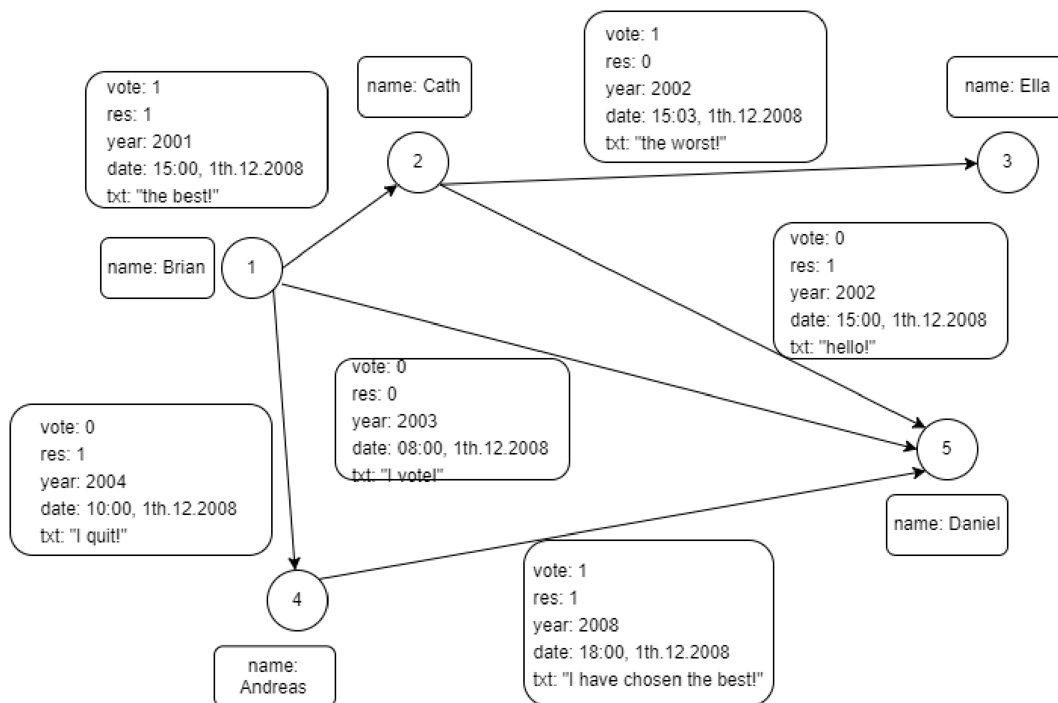


Figure 2.3: A property graph, abstract from Wiki-RfA dataset.

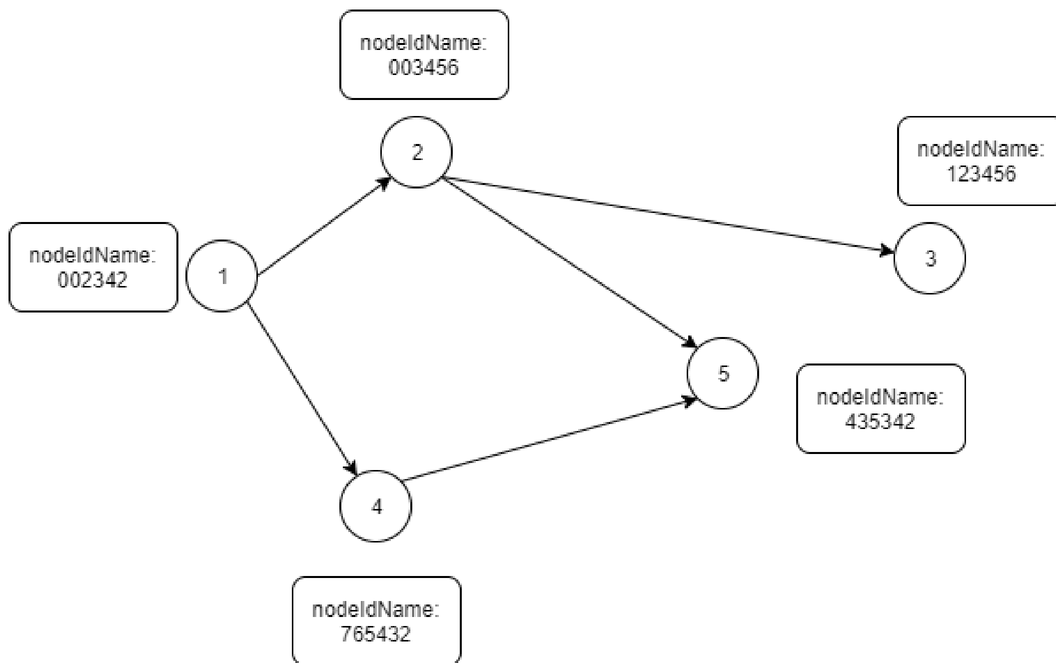


Figure 2.4: A property graph, abstract from Google-Web dataset.

meet the requirements for efficient analytics and storage, such as powerful graph data models, powerful query and analysis capabilities, high performance and scalability, persistent graph storage and transaction support, ease of use and so on. In this section, we will introduce graph management solutions from different systems ([JPNR17]). Our presentation is generally structured after the work of Angles and Gutierrez ([AG17]).

2.2.1 Relational Databases

The main idea of the **relational data model** is to provide database users with a logical model where real world entities can be modeled either as a row of attributes in a table with a predefined schema, or in a more complex way, as a set of relations between tables. A key aspect of this model is the use of set-based logic. Each row represents an entity (or part of it) of the real world within a table. Each columns inside table represents attributes (or properties) of these entities ([Wie15]). Join operations, ACID guarantees with transactional consistency and SQL form some of the key ingredients of working with relational databases. Through this data model and efficient developments, relational databases have become the most popular commercial databases in the last decades.

With the interest in graph management and in consideration of the maturity of relational technologies, research has naturally been devoted to employing relational databases for managing graph data. Research in this topic generally attempts to argue that specialized graph databases and engines for graph analytics may not be necessary, and that RDBMSs constitute a reasonably good alternative ([XD17]).

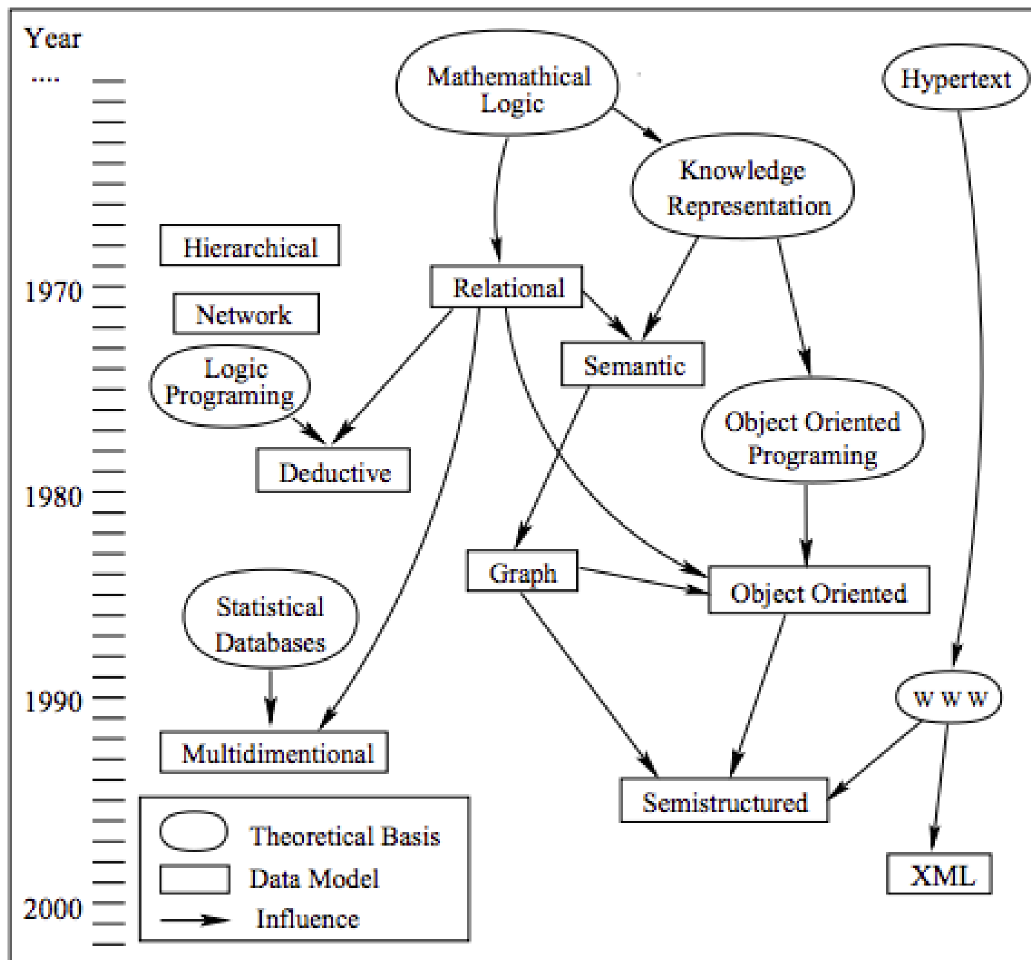


Figure 2.5: Evaluation of database models. Rectangles represent models, arrows represent influences, and circles represent theoretical developments. On the left hand side. there is a time-line in years[AG08].

In terms of data models, the relationship between the relational model and graph models has an interesting history, as described in the following Figure 2.5. Namely, database models influenced by graph theory appeared in close proximity to the relational model. In fact, the E/R model (as discussed in [Che76]) is considered to be a semantic model influenced by graph theory. Some early graph database models appeared in the 1980s, in parallel with Object-Oriented models, both motivated by observed difficulties in modeling complex objects with the relational model. In turn, both of these models had an influence on the emergence of semi-structured and XML models. More recent models used in document databases like JSON or in key-column value stores like the column family model stem from this evolutionary branch.

The approach of using RDBMS to manage graphs has advantages and challenges. On the one hand, various real scenarios depend on RDBMS. Relational data has latent graph structures and the latent graph structures are amiable for graph analysis. In addition, graph-relational queries are everywhere and an RDBMS is useful to process the relational constructs of these queries. On the other hand, the main challenge to leverage relational databases in processing graph queries is the mismatch between the relational and the graph models ([HKJ+17]).

In a recent paper ([HKJ+17]) authors classify the existing approaches into two large groups, by focusing on how the query engine manages the graph. Specifically the propose to divide systems as: a) **Native-Relational core (NRC)**, where the graph is encoded as a relational table and graph queries are translated to SQL queries and the results are translated to a graph model; and b) **Native-Graph core (NGC)**, where graph queries are not translated, instead a graph is extracted from the relational database (with SQL graph extraction queries) and the query is processed with a native graph query engine either in memory or in a separate graph database. Some limitations of the first approach are:

- It is limited, per design, to the subset of supported graph queries.
- Since graph queries are supported by SQL steps (for example, using self-joins), the operations might be less efficient than when running over a native graph representation.
- In the cases of automated graph-to-relational schema mappings, the generated schema might be difficult to comprehend.

Among the limitations of the second approach are:

- Schema changes on the relational database might involve re-extraction of the graph.

NRC	**NGC**
Vertexica(2014,2015)	Ringo(2015)
SAP HANA Graph Story/Graphite	G-SQL(2016)
SQLGraph(2015)	GraphGen(2015)
EmptyHeaded (2017)	GQ-Fast(2016)
GRFusion (2017)	GRFusion (2017)

Table 2.1: Classification of the Existing System Studies Based on NRC and NGC

- Difficulties for integrating results of queries that move across both models.

In this section we provide a brief overview on studies to support graph analysis within RDBMSs. We structure our discussion by following the aforementioned classification suggested in recent work ([HKJ+17]). Since the comparison of relational and graph technologies is not the core of our work, we limit our study to presenting summarily this small selection of systems. For more information we refer the reader to the cited papers. To the extent of our knowledge, there is currently no comprehensive survey covering this topic. Table 2.1 organizes the work we review in this section based on the discussed classification:

Vertexica[JRW+14][JMCH15], Emptyheaded[ALT+17], Graphite and SQLGraph[SFS+15] are useful tools to map graph vertex-centric queries into SQL queries in an RDBMS, using VoltDB, SAP HANA and a non-disclosed RDBMS. To support this, in all systems except Emptyheaded, the graph has to be stored in specific ways inside the database schema. SQLGraph uses a JSON extension of the database to store the data for each entity and the relational storage for the adjacency information. SQLGraph can be specially noted because it also supports TinkerPop Gremlin, a graph native query language. Graphite uses a format called the universal table, wherein one table is assigned to vertexes and other to edges. Since Graphite uses the in-memory database SAP HANA, the system is optimized with dictionary compression, special indexes and tuned algorithms for traversals. Vertexica employs the same universal table mapping of Graphite, but adds another table to manage the message passing involved in vertex-centric graph processing.

After mapping the queries to SQL these systems perform several optimizations, with the authors of SQLGraph reporting better results for using their system with relational databases, against using native graph databases like Neo4j and Apache Titan (now JanusGraph).

In the case of EmptyHeaded the data representation is based on separate columns instead of a table. The optimizations in EmptyHeaded are in query processing to find optimal plans and exploit SIMD, making it the first worst-case-join system to be able to run graph queries without index-free-adajacency, in a competitive manner.

In contrast to these systems Ringo[PSB+15] use graph processing engines to execute graph queries efficiently after mapping data from a relational database for this engine. They also map the data back to a relational model. G-SQL[MSX+16] and

GQ-Fast[LMPS16] are concerned with the problem of using graph frameworks to accelerate SQL queries, like multi-way joins. GRFusion[HKJ+17] offers the possibility to combine the approaches by building on the ideas of Vertexica and adding more concepts for cross-model queries and optimizations.

2.2.2 Graph Databases

Why graph databases? Graphs can model objects and their relationships intuitively. Road, biological and social networks have high-connected data structures. To analyze these kind of data, graph models are a valuable choice. Also, to extract business value from the network perspective, analytics on big graphs become increasingly important. We could discover the role of actors in social networks, to identify interesting patterns in biological networks and to find important publications in a citation network. Further common uses include: Master data management, fraud detection, cyber-security, recommendation systems and identity and access management. In response to these trends, graph data management solutions have recently sparked significant interest.

A graph database management system (henceforth, a graph database) is an online database management system with Create, Read, Update, and Delete (CRUD) methods that expose a graph data model. Graph databases are generally built for use with transactional systems. Accordingly, they are normally optimized for transactional performance, and take transactional integrity and operational availability into account ([RWE13]). Graph database systems are based on a graph data model. And they represent data in graph structures and provide graph-based operators like neighborhood traversal and pattern matching. Table 2.2 provides us a overview of recent graph database systems ([JPNR17]).

Table 2.2: A Comparison of Recent Graph Databases ([JPNR17])

	DataModel			Scope		Storage		
	RDF/S-PARQL	PGM/Ti-nkerPop	Gene-rics	OLTP/Q-ueries	Analy-tics	Appr-oach	Replic-ation	Partit-ioning
Apache Jena TBD	x/x			x		native		
AllegroGraph	x/x			x		native	x	
MarkLogic	x/x			x		native	x	x
Ontotext GraphDB	x/x			x		native	x	
Oracle Spatial and Graph	x/x			x		native	x	
Virtuoso	x/x			x		relational	x	x
TripleBit	x/x			x		native		
Blazegraph	x/x	x/x		x	x	native RDF	x	x
IBM System G	x/x	x/x	x	x	x	native PGM wide column store	x	x
Stardog	x/x	x/x		x	o	native RDF	x	
SAP Active Info. Store		x/-		x		relational		
ArangoDB		x/x		x		document store	x	x
InfiniteGraph		x/x		x		native	x	x
Neo4j		x/x		x		native	x	
Oracle Big Data		x/x			x	key value store	x	x
OrientDB		x/x		x		document store	x	x
Sparksee		x/x		x		native	x	
SQLGraph		x/x		x		relational		
Titan(JanusGraph)		x/x		x	o	wide column store, key value store	x	x
HypergraphDB			x	x		native		

Supported data models

There are two major data models that are supported by recent graph databases: property graph model (PGM) and the resource description framework (RDF). RDF and the related query language SPARQL are standardized. But for PGM , its query language is Apache TinkerPop or Cypher, to date this is only industry driven. Some examples of databases supporting RDF are AllegroGraph and RDF3x. Some examples of databases for the PGM are Sparksee (formerly DEX), Neo4j, JanusGraph, and Trinity. For our experimental system we focus on PGM and a detailed property graph model, as has been introduced in Section 2.1.2.

In paper ([AAF+17]) authors provide a more recent list of commercial systems, most offered as cloud storage or analytical services, supporting the property graph model. Here we include this list, to give more context and examples: AgensGraph, AmazonNeptune, ArangoDB, IBM's BlazeGraph, Microsoft's CosmosDB, DataStax EnterpriseGraph, Oracle PGX, SAP HANA Graph, OrientDB, Stardog and Tigergraph.

There are also several graph databases that support generic graph models. "General graph model" means the graph databases support arbitrary user-defined data structures (from simple scalar values or tuples to nested documents). And most graph processing systems support the generic graph data models. But the usage of generic graph data models could be advantages and disadvantages. On the one hand, generic graph models are most flexible . On the other hand, such generic graph data models supported systems cannot provide built-in operators related to vertex or edge data. Because the existence of certain features like type labels or attributes are not part of the data model ([JPNR17]).

Application Scope:

There are two major areas for graph data management: OLTP (online transactional processing) and OLAP (online analytical processing). OLTP are focusing on processing data transactions. OLAP are focusing on graph analytics, such as data mining. Both of them are used for different business applications. Most graph databases focus on OLTP workload, i.e., CRUD operations for vertexes and edges and transactions and query processing. Graph databases such as **JanusGraph**, **Neo4j**, **ArangoDB** and **OrientDB** specialize in OLTP area. Graph analytics systems like **Pregel**,**Giraph**, and **PowerGraph** specialize in OLAP area, in spite of the capabilities to have distributed storage using memory grids, they are not considered to be graph databases but processing frameworks ([PZLz17]), which we discuss in a subsequent section. In recent year a new application scope called HTAP (Hybrid Transacion/Analytical Processing) is being considered for graph data management, according to authors[PV17]. This combines OLAP with OLTP uses. Graph processing systems like LLAMA, SAP HANA Graph and PGXISO server this case. To date there is no native HTAP graph database, with

SAP HANA Graph being the closest thanks to it's native graph engine and non-native storage.

Some of considered graph databases already show build-in support for graph analytics such as the execution of graph algorithms that may involve processing the whole graph, i.e.,to calculate the pagerank of vertexes or to detect frequent substructures. IBM System G and Oracle Big Data support build-in algorithms for graph analytics, such as pagerank, connected components or k-neighborhood. The current version of TinkerPop provides the virtual integration of graph processing systems in graph databases. For example, from the user perspective graph processing is part of the database system but data is actually moved to an external system ([JPNR17]).

Storage Techniques

The graph database world is populated with both technology designed to be "graph first," known as native, and technology where graphs are an afterthought, classified as non-native. Graph databases with **native graph storage** have storage designed specifically for the storage and management of graphs. They are designed to maximize the speed of traversals during arbitrary graph algorithms. **Non-native graph storage** uses a relational database, a columnar database or some other general-purpose data store rather than being specifically engineered for the uniqueness of graph data. Most of graph databases we listed in the table are using native storage approach. **Adjacency lists** are a typical technique of graph-optimized approach,i.e., storing edges redundantly attached to their vertexes.

Some systems implement the graph database based on the data models such as relational and document stores. IBM System G and Titan (now is JanusGraph) provide multiple storage options. About half of the listed systems has some support for partitioned storage and distributed query processing. Systems with non-native storage provide no specific partitioning strategies for graphs, e.g. OrientDB treats vertexes as typed documents and implements partitioning by type-wise sharding.

Query Language supported

In paper ([Ang12]), four operators specific to graph databases query languages: adjacency, reachability, pattern matching and aggregation queries, have been introduced. Adjacency queries mean to determine the neighborhood of a vertex. Reachability means to identify if and how two vertexes are connected. Pattern matching need no specific starting point and can be applied to the whole graph. They enable very expressive kinds of queries. This characteristic makes pattern matching an important operator. Also, there are combinations of reachability and pattern matching queries in the form of Regular Path Queries, these use regular expressions to indicate the pattern of a path that should

be matched. Finally, aggregation is used to find aggregated, scalar values from graph structures.

Either SPARQL for RDF, and TinkerPop Gremlin and OpenCypher (previously Neo4j's Cypher) for the property graph model, are being supported by most of the recent graph databases ([JPNR17]). Also it is fairly common for these databases to have query languages of their own, without too much formalizations.

In paper ([PV17]), existing languages are divided in pattern-matching based and traversal-based. From the first case examples are Gradoop, OpenCypher, Socialite and PGQL. From the second class examples are GEM and GraphScript, graph query languages of SAP HANA [RPBL13], Gremlin, GreenMarl and GraphiQL.

A new type of graph query language, called path property graph model has recently been proposed by the LDBC council [AAF+17] for capturing the core of traversal and pattern-matching languages, adding more composability and the concept of having paths as first-class citizens.

2.2.2.1 Graph Databases with Native Storage

Databases with native graph storage have storage designed for the storage and management of graphs. The way in which graphs are stored is one key aspect of the design of a graph database. And a native graph storage format supports rapid traversals for arbitrary graph algorithms. Below we will give an example to show how data is natively stored in Neo4j.

Neo4j

Neo4j stores graph data in different *store files*, which contains the data for a specific part of the graph (e.g. there are separate stores for nodes, relationships, labels and properties). The division of storage responsibilities facilities high performing graph traversals. Particularly the topological structure is separated from property data. For example, the node store file stores node records and each node store is a fixed-size record (each record has nine bytes in length). Correspondingly, relationships are stored in he relationship store file. Like the node store, the relationship store also consists of fixed-sized records. Fixed-sized records facilitate the computation of any individual record's location by knowing its ID ([IR15]). The division of storage responsibilities, particularly the separation of graph structure from property data facilitates performant graph traversals. We can observe the structure of nodes and relationships physically stored on disk as shown in Figure 2.6.

2.2.2.2 Graph Databases with Non-Native Storage

Non-native graph storage use a relational database, a column database or some other general-purpose data store rather than being specifically engineered for the uniqueness if graph data. In previous sections we already discussed some forms of non-native storages when using relational databases, below we introduce two popular and general non-native storage choices, and then we introduce one non-native graph database example, JanusGraph.

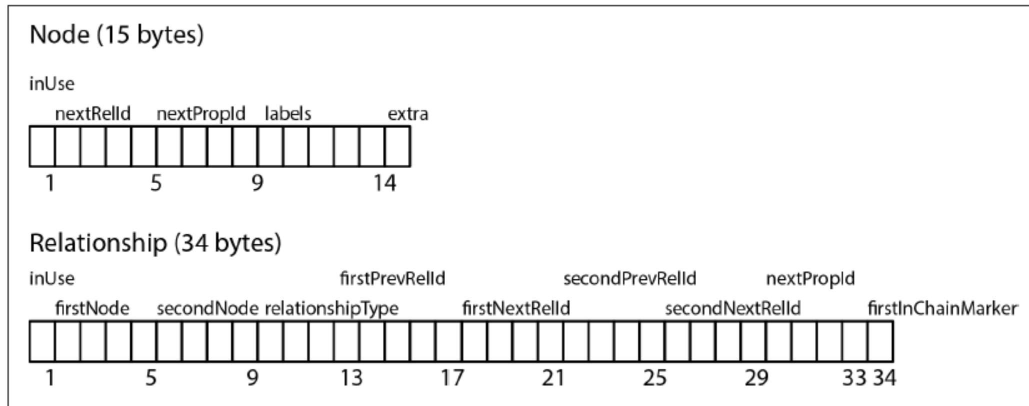


Figure 2.6: Neo4j Node and Relationship *Store File* Record Structure ([IR15])

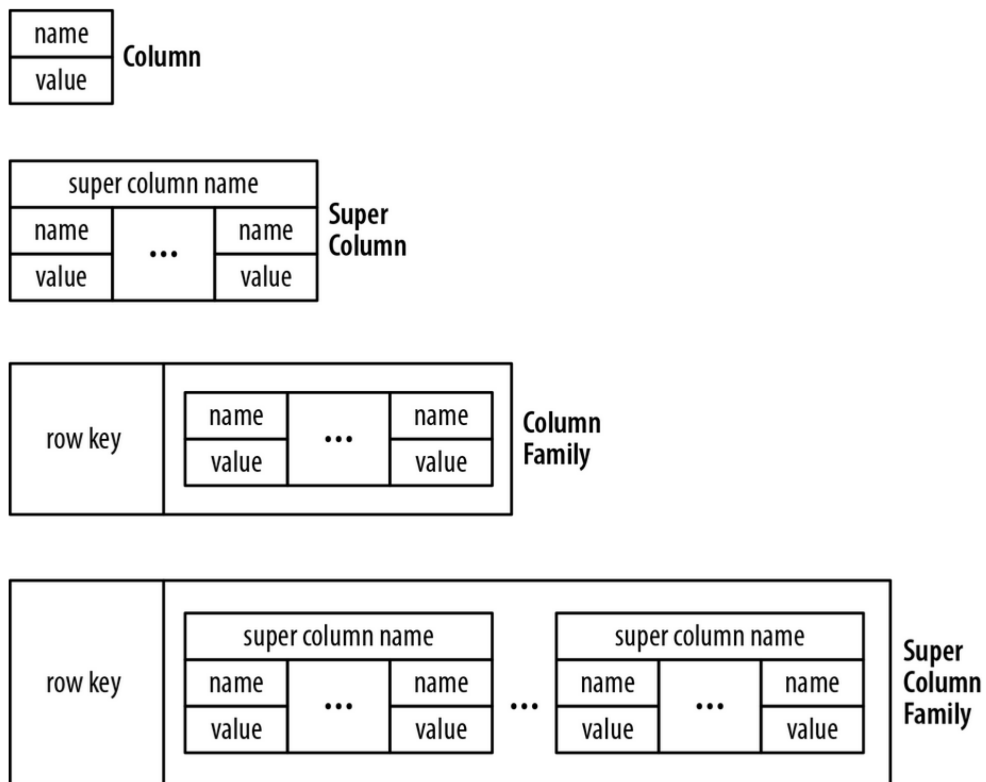


Figure 2.7: The Four Building Blocks of Column Family Storage ([IR15])

Non-Native Storage Choices: Column Family Stores

Column family stores are modeled on Google's Big Table. The data model is based on a sparsely populated table whose rows can contain any columns where the keys provide natural indexing, and the names of columns according to family groups are also used to partition the data. There are four building blocks of family storage (see Figure 2.7). Storage unit *Column* consists of a key-name-value pair. Storage unit *Super Column* consists of any number of columns, which gives a name to a stored set of columns. Columns are stored in rows, by keys. A group of columns that, for distribution reasons, is considered to be a single entity, is called a column family. When a group contains super columns, it is known as a super column family. The way in which data is stored, makes column family databases distinguished from document and key-value stores. Column family databases consist also of its own operational characteristics. Apache Cassandra, for example, which is based on a Dynamo-like infrastructure, is architected for distribution, scale, and failover and is a column family stored database.

Non-Native Storage Choices: Key Value Stores

Key-value stores constitute one of the most basic and primitive storage systems. Items are stored as key, value pairs, just like in a common map structure or hash table from programming languages. Values can usually have any type. Not many operations are supported, save from basic CRUD ones.

Some examples of KVSs include Redis, Voldemort, etcd and Memcached.

JanusGraph

JanusGraph is a non-native scalable graph database optimized for storing and querying graphs. It's also a transactional database that can support thousands of concurrent users executing complex graph traversals in real time[2]. Not like Neo4j, which has a specific storage model for storing graph data, JanusGraph's data storage layer is pluggable. Implementations of the pluggable storage layer are called storage backends. JanusGraph supports for various storage backends: Apache Cassandra, Apache HBase, Google Cloud Bigtable and Oracle BerkeleyDB[3].

In our prototypical implementation we employ Apache Cassandra as a storage backend to support a non-native graph database, JanusGraph.

2.2.3 Large Scale Graph Processing Systems

Compared to graph databases, large scale graph processing systems are less concerned with storage and transactions, instead they focus on analytical tasks at scale. They are also different from databases because they do not take requests and short-lived queries, they rather take some amount of input data, process it (using user defined functions or jobs) and return an output. In this section we present an overview of such systems. For this we start with general large scale processing systems and then we discuss graph-specific systems.

[2]http://janusgraph.org/

[3]http://docs.janusgraph.org/latest/storage-backends.html

2.2.3.1 General Large Scale Processing Systems

Large scale processing systems aid researchers by providing support for distributed and parallel computation, using a simple set of abstractions such as Map and Reduce concepts, or Spark's Dataframes. These frameworks undertake the tasks of managing communication, resource management, job-to-node placement and fault-tolerance keeping the developer unaware of these complex tasks.

The granularity of processing (either if it happens on small batches, or on large ones) and how synchronization happens between jobs are two factors that allow to distinguish large-scale processing frameworks. In this small section we discuss 3 types of large scale processing (i.e., traditional batch processing, bulk synchronous processing and streaming), and then we give two examples of systems. Our presentation is based on [Kle17]. We hope that with this presentation we provide readers with a clear understanding of frameworks for scaling out the computation, helping our further discussion on large scale graph processing systems.

Map Reduce is one example of a system that uses traditional batch processing. In this framework jobs can be scheduled as either Map or Reduce jobs. Map jobs run independently in parallel, while reduce jobs require synchronization and introduce dependencies. Map jobs and Reduce jobs are alternated, finally forming a computation graph that branches out and collects the branches in iterations. An aspect of this style of processing is that jobs end after an iteration, and they must either store their data or send it to the next job in the data flow.

An extension and a more general approach than Map Reduce is called bulk synchronous processing ([Val90]). In this approach the same ideas of the traditional case remain and it is still a batch processing approach, but jobs keep alive, retain their data, and can even be invoked after their first execution. Some large scale graph processing systems such as Pregel and Apache Giraph also utilize this approach. Spark is originally a bulk synchronous processing system, using mini-batches rather than large ones, now it also supports streaming.

Streaming is the third approach that we discuss. Unlike batch processing, where the analysis is expected to be done offline, the design for stream processing is somewhere between online and offline, so it is sometimes called near-real-time or rearline processing ([Kle17]). While in traditional batches, data needs to enter the job before it begins and the jobs need some coordination, in streaming the whole logic is moved to message passing. Stresm jobs keep alive and become message processors, operating on events shortly after they happen: they send and receive messages, acting according to them. Process synchronization is not implicit in the model, instead it is managed by how the user designs the flow of messages. Furthermore, to ease the communication, there are

no specific client-to-client calls. Messages are sent to a distributed message bus, and all clients can read them according to their configurations. As a result of this design, clients of the message bus can be classified as publishers (who send message to a given mailbox) and subscribers (who subscribe to a given mailbox), hence these systems are often called publish-subscribe frameworks. Apache Kafka and Twitter's Storm are examples of streaming large scale processing systems.

Another aspect of streaming that has to be mentioned is time: the clients can be implemented with a logic that includes window operations. These are operations that instead of happening over all the data, happen over a window of the data in the most recent time intervals. As a result a novel kind of analysis is provided. Systems like Flink and Spark Streaming are examples of streaming systems that focus on windowed/streaming analytics.

A further distinction in how large scale processing happens is related to how the complete process is represented for the the system. Here there are dataflow systems, like Spark, Tez and Flink that see the process as a complete workflow, instead of systems like Map Reduce that break it into independent sub jobs. This different perspective allows these systems to consider more optimizations than non-dataflow systems, like reducing intermediate variables and network costs.

Further approaches exist to support large scale processing, like timely dataflow, but we will not discuss them in this section.

Spark

Apache Spark is a system that employs bulk synchronous processing and streaming, it is optimized by storing all the data in memory.

One of the central abstractions of Spark are Resilient Distributed Datasets (RDDs), which provide for fault tolerance. Spark runs as a cluster.

Apache Kafka

Apache Kafka is a scalable publish-subscribe messaging system with its core architecture as a distributed commit log (i.e., the message bus). It runs as a cluster on one or more server. Kafka maintains streams of messages in "topics". Each record of message consists of a key, a value, and a timestamp. These messages can be used to store any object and get passed around in byte arrays (Apache Software Foundation 2017). Kafka includes 8 APIs (see Figure 2.8) and is used in building real-time streaming data applications that collect data between systems or transform the streams of data. In Kafka, a stream processor takes continual streams of data from input topics, performs processing on the

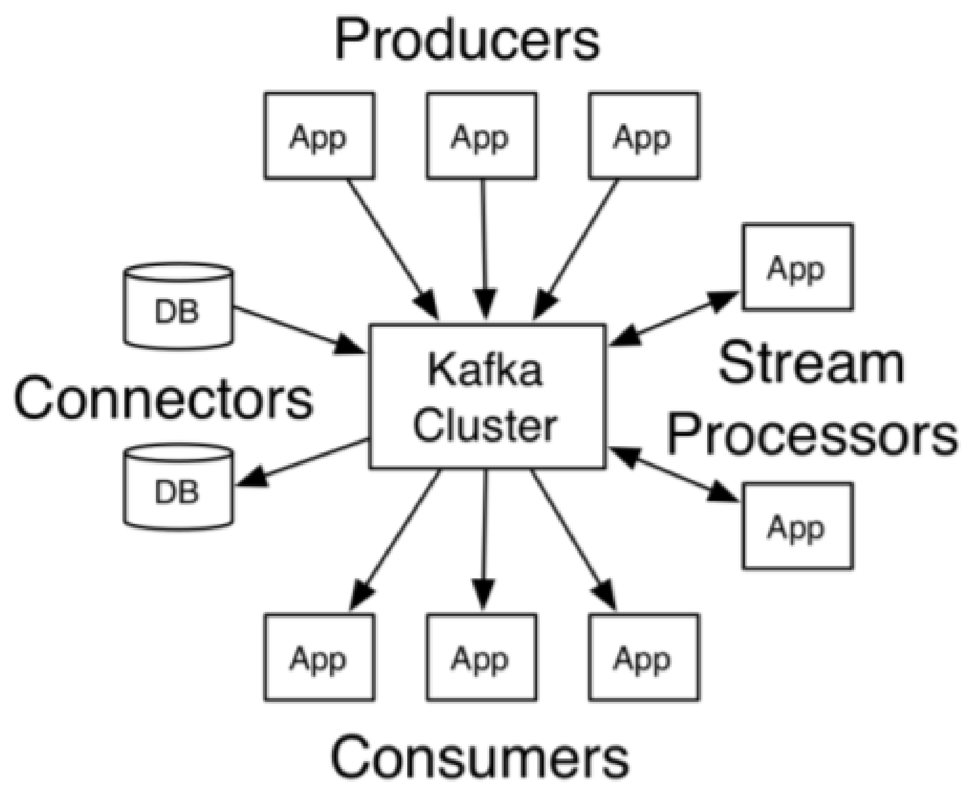


Figure 2.8: Kafka APIs

data, and produces continual streams of data to output topics. Zookeeper is always needed together with Kafka. It's a product also developed by Apache, specialized in managing configuration for distributed synchronization.

In our work we will use Apache Kafka for creating a scalable message passing solution, to distribute edge partitions during graph data loading. In the next subsection we move our discussion from general large scale processing to specialized systems for large scale graph processing.

2.2.3.2 Graph-Specific Systems

For example, many algorithms for graph analytics such as page-rank, triangle counting or connected components, need to iteratively process the whole graph, when the dataset is large the computational efficiency might be improved by scaling out the computation. Graph databases support querying graphs but usually cannot process a large graph in a scalable and iterative way. Also, when storage is not needed graph databases might not be the best tool for this task. As a solution the use of large scale processing systems, and the specialization of them for graphs, becomes a good alternative.

There is a general architecture of a distributed graph processing framework. The architecture uses a **master node** for coordination and a series of **worker nodes** for the actual distributed processing. A graph is given as input and partitioned among all worker nodes, typically using hash or range-based partitioning on vertex labels.

Vertex-centric approach has been recommended by Google Pregel in 2010 ([MAB+10]). Since then many frameworks have adopted or extended it. To realize such Pregel-like systems, a **vertex compute function** has to be written. This function include three steps: 1. read all incoming messages, 2. update the internal vertex state (i.e. its value), 3. send information to its neighbors, along the edges of the graph (i.e., the edges are only for communication and they do not participate actively in the computation). This is supported in an iterative manner with bulk synchronous processing, as discussed before. As a result a vertex remembers it's state in memory from one iteration to the next, so the function only needs to process new incoming messages. If no message is sent in some parts of the graph then no work needs to be done. In this approach there is also fault-tolerance and communication happens in fixed iterations with the synchronization guarantee that on each iteration all messages from previous iterations were delivered. For performance messages are batched and optimizations are introduced to better use locality. Systems like Pregel and Giraph also have the optimization of running in memory only.

In terms of the parallel execution, the partitioning happens on vertexes and it is responsibility of the framework to decide on it. A vertex does not need to know on which partition or physical machine it is executing, it only sends messages to other vertexes by their ids, and it s up to the framework to partition the vertexes and manage the communication. Ideally the partitioning such happen such that vertexes that

communicate frequently are colocated in the same partition. Given that this is complex, in practice the common approach is to distribute the vertexes arbitrarily through a random hash partition [Kle17]. Edge cuts are also a possibility for partitioning.

Some examples of these systems are Apache Giraph (the open source implementation of Pregel), Flink's Gelly and Spark's GraphX API.

Graph/partition-centric model doesn't let the compute functions execute on each vertex. A compute function in a Graph/partition-centric model takes all vertexes managed by a single worker node as input. These functions are then executed using the BSP (Bulk Synchronous parallel) model. In real-world scenario, graph processing is often a single step inside the whole data transformation pipeline. Recent graph processing frameworks like Apache Spark and Apache Flink provide graph processing libraries to make this task more easily. Apache Giraph is an example of iterative graph processing system, built for high scalability[4].

With the presentation of large scale graph processing systems we conclude our presentation of graph data management solutions. Next we discuss challenges in graph processing and focus on the task of loading data into graph databases.

2.3 Challenges in Graph Processing

From the survey ([SMS+17]), we present these top three challenges user faced when processing graphs. These problems do not only refer to large scale graph processing, but to working with graphs in general:

- *Scalability:* Scalability becomes the most pressing challenge. The specific scalability challenges has been mentioned include inefficiencies in loading, updating, and performing computations, such as traversals, on large graphs. Among them we choose loading of scalable graphs as our study topic.

- *Visualization:* Perhaps even more surprising is the visualization of graphics becomes one of the top 3 graph processing challenges. visualization is the most popular non-query task. And user perform on their graphs. For these users visualization is really a challenge.

- *Query Languages and APIs:* Query languages and APIs is another common graph processing challenge. The specific challenges include expressibility of query languages, compliance with standards, and integration of APIs with existing systems. And we have found, there are a lot of existing researches dedicated to this area.

[4]http://giraph.apache.org/

Today user faces a lot of challenges when processing graphs as introduced before. We are inspired by this and want to focus our study on optimizing the scalable graph loading process. Below we present some researches on loading tasks for scalable data. And we choose JanusGraph database instead Neo4j, because it's a scalable graph database and Neo4j is not.

2.4 Loading Tasks for Big Data

Traditional data loading has been thought of as a "one time deal". Loading can be processed off-line and out of the critical path of query execution[DKA+17]. Nowadays the rate in which data is produced and gathered breaks out the assumption of "one time deal" and "off-line". Before using the powerful functions from Traditional database management systems (DBMS), the user must do several tedious steps and answer several questions by databases. Should the database use a column-store, a row-store or some hybrid format? What indexes should be created? All of these make data loading process time-consuming and create high cost. Loading data into databases has been considered as a performance bottleneck.

In this paper we aim to optimize the process of **loading big data into graph databases**. A shortly introduction about loading loading tasks for big data has been presented below. Next we discuss on loading for graph databases, covering the limited number of studies we could find on the topic.

2.4.1 Traditional Databases

Data from science and scientific worlds increase rapidly. The recent advances of cost-effective storage hardware enable storing huge amount of data. But the ability to analyze and process them is behind. **Extracting** value out of gathered data is a traditional requirement of loading it into a operational database. In data warehouse scenarios, **ETL** involves: 1. **E**xtracting data from outside sources. 2. **T**ransforming it to fit operational needs. 3. **L**oading it into the target databases ([IC03]). With the advent of big data age, new needs have followed. The requirement of reduce of the "query time" force data loading to be a fast operation. The requirements of high availability ask to minimize batch loading windows. And with the ever-increasing growth of data needs data loading to be a scalable operation, which can enable a parallel loading. Parallel loading means a massive amounts of data could be in a single or different machines in parallel in a short amount of time loaded. But traditional data loading methods can't meet the demands completely. Then we need new contributives in this area.
Over the past few years, most of the researches is about finding innovative techniques to avoid or accelerate data loading.([Mue13] [CR14]) The big data community realize the importance of parallel loading process through multi-core processors and modern hardware storages. They consider the time spent to load data into the system as a part of the benchmark metric and make it as the "Big Data Top 100" benchmark ([Bar13]).

2.4.2 Loading Large Scale Data

Commercial analytical database systems suffer from a high "time-to-first-analysis": before data can be processed, it must be **modeled** and **schematized** (a human effort), **transformed into the database's storage layer** and optionally **clustered** and **indexed** (a computational effort [AAS]). Loading process for large scale graph is more complicated as traditional DBMS. Following I'll give some descriptions about loading process in recent large scale data analysis systems.

Hadoop
The hadoop system is the most popular open-source software framework. It's a distributed file system used for distributed storage and processing of dataset of big data using the MapReduce programming model.

We can use two ways to load data into Hadoop's distributed file system (HDFS). 1. Use Hadoop's command-line file utility. We use this method to upload files stored in the local file system into HDFS. 2. Create a custom data loader program using Hadoop's internal I/O API. Then use this data loader to upload local data ([PPR+09]).

DBMS-X
DBMS-X is a parallel SQL DBMS from a major relational database vendor. It stores data in a row-based format.

There are two steps for lading data into DBMS-X. 1. Execute the LOAD SQL Command in parallel in each node in the cluster. This command reads data from local file system and insert its content into a particular table in the database. 2. After the complement of first step, execute an administrative command to reorganize the data on each node. In this step, data is compressed on each node in parallel. Index of each table has been built and so on ([PPR+09]).

2.4.3 Loading in Graph Workloads

The initial loading and construction of the graph data structures has been seen as a performance bottleneck for many graph workloads. However this topic is difficult to analyze in the literature and, according to researchers, is usually neglected in system evaluations ([TKKN16]).

The graph loading process has been characterized as divided into several steps. Namely, previous work on in-memory graphs has divided the process in: parsing vertex data and loading vertexes (when available), parsing edge data and then relabeling vertex identifiers for an improved memory footprint ([TKKN16]).

```
# Directed graph (each unordered pair of nodes is saved once): CA-AstroPh.txt
# Collaboration network of Arxiv Astro Physics category (there is an edge if authors coauthored at least one paper)
# Nodes: 18772 Edges: 396160
# FromNodeId    ToNodeId
84424   276
84424   1662
84424   5089
84424   6058
```

Figure 2.9: Source File Examples: Edge Lists, from SNAP Astro-Physics Collaboration Dataset

```
Category:Laboulbeniomycetes; 1941 1942 1943 
1977 1978 1979 1980 1981 1982 1983 1984 1985
2019 2020 2021 2022 2023 2024 2025 2026 2027
2063 2064 2065 2066 2067 2068 2069 2070 2071
```

Figure 2.10: Source File Examples: One Vertex And All Edges Configuration, from SNAP Wiki-Top Categories Dataset

For our study we observe that the loading process can be more complex than this previous study suggests, depending on the characteristics of the source data. For example, if the data is given as a simple edge list Figure 2.9, then one pass could be needed to collect the vertex ids and insert them, keeping a mapping between the database provided id and the id in the dataset. Finally a second pass could be performed to read the edges and load them. We assume this to be the standard load, and we see that it leaves some space for optimizations, namely in how the edges might be loaded. If, alternatively, the loading proceeds sequentially, then perhaps less optimizations are possible.

We further observe that for the standard load, a large number of files can be given, with all following the same format, this gives some changes to the process. Partitions of the files are also possible when the data is large.

On occasions the dataset might already make assumptions on how data is to be loaded, by specifically organizing each vertex with the collection of vertex ids to which it has connected edges Figure 2.10. This can also be changed into a standard load, but requires special considerations.

On other configurations, there might be no explicit edge relations and the user can model how the data should be loaded for analysis Figure 2.11. Once more this could be changed into a standard load, but the case requires consideration and support from a data loading tool.

In other instances more processing might be needed to add to entities their properties, as the properties are given encoded and further files have to be consulted to fill the missing data. Figure 2.12 gives an example on how these properties might be encoded when a vertex is given. This extra steps could be scheduled after the standard load, however they should be considered by loading tools and for optimization purposes.

Based on these observations we propose that the loading process can be more generally described as follows:

```
product/productId: B00006HAXW
review/userId: A1FRPGQYQTAOR1
review/profileName: Les
review/helpfulness: 9/9
review/score: 5.0
review/time: 1062979200
review/summary: Marvelous, just Marvelous!
review/text: The people who have reviewed this DVD before me have said it all! It is just great!
bought DOO WOP 50 and DOO WOP 51. What a wonderful nostalgic trip to the past! You will never reg
```

Figure 2.11: Source File Examples: Implicit Entities, from SNAP Amazon Movie Reviews Dataset

```
137779150 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
212266188 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
168683422 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
153996235 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0
52283878 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0 1 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0
52012019 0 0 0 0 1 1 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1
288932529 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 1 0 0 1 0 0
120178497 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0
147418979 1 0 0 0 1 1 1 1 1 1 1 0 1 0 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
140741639 0 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 0 0 0 1 1 0 0 0 1 0 1 0 1 1 0 0 1 0 0 1 0 1 1 1 0 1 0 0 0 1 0 0 1 0 0 0 0
51999405 1 0 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 0 1
164998160 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 1 0 0
52284349 0 0 0 0 1 1 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 0 0 1 1 1 0 1 0 1 0 1 0
154070583 0 0 1 1 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 1 1 0 1 1 0 0 0 0 1 1 0 0 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 1 0 0 1 0 1 0 0 0 1 0
```

Figure 2.12: Source File Examples: Encoded Properties, from SNAP Ego-Twitter Dataset

1. *(Optional) Schema and index definition:* Though most graph databases are schema-less, it is often good practice to define the graphs with a specific schema and indexes. This, in turn, can enable optimizations from the database. Headers from csv files, or alternative text files, can be used to convey this information on data load time.

2. *Parsing for key data:* First, graph datasets are normally not in a system-specific binary format. They have their own format, like JSON, CSV or TXT. These datasets must be parsed to extract the actual graph data, i.e. vertexes, edges and their related properties. However on a first stage not all data needs to be loaded, and just the ids might be sufficient to arrange the process.

3. *(Optional) Dictionary encoding and relabeling of vertex identifiers:* Then, the properties can be dictionary encoded, so as to reduce their memory footprint. Furthermore, original vertex identifiers from the dataset can be relabeled to numbers in a dense range. This means that the ids of vertexes with many edges can be encoded with less bits than the ids of vertexes with few edges, leading to less memory footprint than when the encoding is done the other way around, or when all vertexes are assigned the same number of bits. This relabeled graph data can then used to construct the final in-memory representation of the graph. ([TKKN16])

4. *Loading of key vertex data and creation of mappings:* In this step the vertexes can be loaded, and the mapping from the id given in the dataset and the id given by the database (which enables faster access) can be stored.

5. *Optimizations over edges, including partitioning, and loading:* At this stage optimizations can be decided, such as partitioning the edge list, and then the loading can happen.

6. *Loading of additional data:* Here encoded properties could be loaded.

These steps can be repeated for different node or relationship types, if necessary.

In each phase of the data loading process and in the organization of the complete process, there are several possible improvements. There might be also several trade-offs for different scenarios. This motivates our work, as we seek to help developers by improving the data loading into graph databases, and developing a tool to support this.

To our knowledge there are limited tools developed for loading data efficiently into graph databases. Neo4j importer is one such tool that enables batch imports of CSV files into Neo4j[5]. This tool is offered as a bash script that is packaged with Neo4j. The tool consumes files for nodes and relationships, each separately. The files must follow a specific format with header names and other characteristics. Among the options that this tool offers are configurations for allowing duplicates or missing data. To our knowledge, as of the date of this publication, optimization choices save for memory sizes, are not made available to end users. Options to load encoded data or to load data without standard source file characteristics (e.g. implicit entities) are not considered.

Some optimization alternatives that we consider in this work will be introduced in the next section.

2.5 Optimization Alternatives for Loading Data into Graph Databases

There might be many ways to speed up data loading. In this section we introduce the optimization approaches that we selected for loading data into graph databases. These approaches are inspired from different data systems, not limited to graph databases.

2.5.1 Batch/Bulk Loading

With the growing of collected information data loading has been turned into a bottleneck analysis tasks. To improve the data loading performance numerous approaches has been researched. Bulk/Batch loading techniques are typically used during the load phase of the Extract, Transform and Load (ETL) process. They can process massive data volumes at regular time-intervals. "Right-Time ETL (RiTE)" ([GK10]) is a middle-ware system, which provides ETL processes with INSERT-like data availability, but with bulk-load speeds. Instead of loading entire input data directly into the target table on a server. A so called "server-side", in-memory buffer is used to hold partial rows of this

[5]https://neo4j.com/docs/operations-manual/current/tools/import/

table before they are materialized on disk.

Inspired with the related work from other data process systems. We create bulk/batch loader for graph databases. For graph database JanusGraph We implemented a bulk/-batch loading class in order to optimize the process of data loading. Bulk/batch loading class has been created to load more data in each transaction. In contrast to loading small amounts of data per transaction, bulk loading is able to add more data through individual transactions. JanusGraph provides us a lot of configuration options and tools to acquire massive graph data. To make loading process more efficient. "*In many bulk loading scenarios it is significantly cheaper to ensure data consistency prior to loading the data then ensuring data consistency while loading it into the database.*" The storage.batch-loading configuration option exists because of this observation. This conclusion is from JanusGraph team. According with this they create **storage.bath-loading** configuration option specific for JanusGraph to improve the time performance.

2.5.2 Parallel and Partitioning

Parallel loading could be a means to improve the utilization of CPU resources available ([DKA+17]) and hence reduce the whole loading time.

Traditionally, to distribute graph computation over multiple machines in a cluster, the input graph should be partitioned before computations start. The vertexes and edges need to be assigned to individual machines at first. This is, in fact a key process for distributed graph computations, requiring methods such as random hash partitioning or algorithms to find the minimum edge cut[SK12]. One commonly used algorithm and system for partitioning is Metis[KK95]. Good graph partitioning algorithms are useful for many reasons. First, real world networks are not random. Edges have a great deal of locality which could be exploited to reduce inter-machine communication. Second, partitioning is related to distributing the computation time. Hence, when the distribution is skewed, the uneven partitions could lead to a longer computation time and waste in resource usage.

A survey on graph partitioning is presented by Bulucc et. al.[BMS+16].

For the task of loading there is a significant difference with respect to traditional partitioning approaches. Namely that the complete graph is not available in such a way that it could enable computing a large algorithm over the graph. Instead the loading process must partition the graph with incomplete information, deciding for the location of a vertex or an edge, or a group of them, as it processes them. In spite of the limited information there is still the goal of finding a balanced partition that can also reduce communication costs during the loading process. Hence this can be defined as a streaming graph partitioning problem[SK12].

2.5.2.1 Partitioning Strategies

By parallelizing the batch loading across multiple ports, the load time can be reduced. But how should we decide for a large graph on how to load sub-sections of it separately?

To address these issues we implemented a parallel loading class in order to load the partitioned data in parallel. First we use some conventional strategies to partition the graph into several subgraphs. Second we load the subgraphs in parallel. This process is referred to as Partitioning in this paper.

Partitioning Strategy selection becomes increasingly important for variety domains. There is no one-size-fits-all partitioning strategy. In [VLSG17], the author demonstrates that the choice of partitioning strategies depends on several items, such as the degree of the distribution, the type and duration of the application and the cluster size.

Authors have proposed[SK12] the use of different heuristics for streaming graph partitioning, such as balanced (assigning a vertex to a partition with minimal size), chunking (assuming some order in the stream, divide the stream into chunks and distribute them in a round-robin fashion), hashing items, deterministic greedy (assigning an entity to the partition where it has more items, e.g. a vertex to where it has more edges, this can be further parametrized to include penalties to large partitions), next to buffer-based ones. Authors find that these simple heuristics can bring important benefits over random cases and also reduce the edge-cuts, improving distributed graph processing[SK12]. The best performing variant is the parametrized deterministic greedy[SK12].

For our study it is not clear if the partitioning strategies will bring a benefit to the loading task.

Since our research aims to consider the impact of partitioning on the time of data loading, we have picked in a reasonable manner 4 different partition strategies to arrange our experiments in order to find the influence of these partitioning strategies ([KKH+17]). These strategies were selected due to their suitability for specifically distributing the edges, since it is not clear to us if reducing edge cuts will be important or not for loading.

- E/E Strategy
  This strategy uses round-robin (RR) algorithm. It distributes edges to partitions in a lightweight way. It allocates many or all outgoing edges of one vertex to multiple partitions.

- V/V Strategy
  A graph can be partitioned by vertexes. V/V strategy uses vertexes and balances the amount of vertexes for each partition. This strategy distributes all outgoing edges of a vertex to a single partition. It uses also round-robin algorithm.

- BE Strategy
  This strategy partition the graph by vertexes and meanwhile balances the amount of edges per partition. This strategy requires to sort the vertexes according with the number of outgoing edges in a descending order. And then iterates over this sorted list and allocate all outgoing edges from one vertex to the currently smallest

partition. It balances the edges across the partitions. Thereby all outgoing edges of a vertex belong to the same partition.

- DS Strategy
  DS Strategy basically extends BE Strategy. It's a approximation for handling skewed data. To ease the pressure of highly connected vertexes DS strategy allocate the edges equally to the partitions. For the vertexes that have significantly more edges, this strategy separate the edges and distribute them in different partitions.

- 80/20 Rules
  This rule can be summarized as: for many events, roughly 80 percent of the effects come from 20 percent of the causes. Mathematically, the 80/20 rule is roughly followed by a power law distribution for a particular set of parameters. Many natural phenomena show this distribution empirically. We are inspired to this rule and combine DS Strategy with it together. After we sort the vertexes according with the number of outgoing edges in a descending order. Then we find out the first 20 percent vertexes and distribute the outgoing edges from these vertexes to different partitions. And for the rest 80 percent vertexes, the outgoing edges from a vertex should be allocated to the same partition.

2.6 Summary

In this section we presented the relevant related work which is the basis of our work in optimizing of data loading into graph database process. We introduced graph data models and solutions for graph data management, then we gave a brief overview on the limited state-of-the art regarding the loading task, for more insights we discussed some practical aspects on how the process might be affected by source file characteristics, we presented our proposal for steps to understand the loading process and we presented two optimizations that could be considered for the loading task.

In the next chapter we introduce our evaluation questions, the prototype that we develop to study them and the experimental settings.

3. Microbenchmarking Application-Level Optimizations for Loading Graph Data

In this chapter, we introduce the precise evaluation methods that we seek to use in our research. The outline for this chapter is as follows:

- **Evaluation Questions:**
  First we provide several evaluation questions that we aim to address in our study. (Section 3.1)
- **JanusGraph:**
  We introduce JanusGraph, a distributed graph database, which we selected for our experiments. (Section 3.2)
- **JanusGraphLab:**
  We introduce JanusGraphLab, a prototype that we developed for microbenchmarking optimizations in loading graph data. (Section 3.3)
- **Measurement Process:**
  We introduce our measurement methodology. (Section 3.4)
- **Testing Framework:**
  A description of our testing framework is provided in (Section 3.5).
- **Evaluation Environment:**
  A quick listing of the defining characteristics from the execution environment of our tests are discussed in (Section 3.6).

- **Datasets:**
  We describe the real-world datasets we used for the tests in (Section 3.7).

- **Summary:**
  We conclude the whole chapter in (Section 3.8).

3.1 Evaluation Questions

Graph databases are used in various fields for a diverse number of tasks, with graph computations and machine learning tasks being some of the most used ([SMS+17]). Some examples of the former tasks are finding connected components, neighborhood queries and discovering shortest paths. Some examples of machine learning tasks done in graph databases are clustering and graphical model inference; these support analysis cases like community detection, recommendations, link prediction and influence maximization.

In a recent survey of practitioners, scalability is identified as the biggest problem when working with graph databases, followed by the need to accelerate graph visualization and limitations in query languages, among others ([SMS+17]). The specific scalability challenges that practitioners mention in the survey are inefficiencies in loading and updating graph data, next to some computations like traversals, on large graphs.

In our study we aim to improve the scalability challenge of loading data into graph databases. The loading process is a crucial aspect in working with graphs. In fact, according to the kind of analysis being performed on the graph, it might be possible that the time for loading and building the graph exceeds the time of computation.

Concretely, in our study we propose to answer the following set of research questions, by utilizing a state of the art graph database, JanusGraph, and investigating basic application-level loading optimization alternatives made available by the database:

1. Which is the best place to put the loading logic, at client or at server side?
2. What is the effect of batching when loading graphs of different topologies?
3. What are the opportunities and limitations in parallelizing and distributing the data loading?
4. What is the influence of partitioning strategies for loading a dataset in parallel loading?
5. What are the best practices for integrating publisher/subscriber framework into the data loading process?

6. In deciding for batching, parallelization, which factors can be determined statically and which are dependent on changing topologies? For the topology dependent factors: what is the tipping point for making other decisions? Can these optimizations be implemented in an adaptive manner; If so, how to model the optimization function for real-world cases (where network latency and different replication strategies could also affect the performance)?

To address these questions, we carry out experimental research with JanusGraphLab, a prototype that we develop based on JanusGraph. We implement both data loading and microbenchmarking functionalities into JanusGraphLab, to test the possible optimization methods of the data loading tasks under evaluation. We conduct our experiments with real-world public datasets/benchmark data. At the end, we collect the performance data of the loading tasks and analyze the experimental results. According to the analysis of the experimental observations, we give some suggestions for improving data loading tasks, encapsulated in the form of best practices.

3.2 JanusGraph

JanusGraph is a scalable graph database with non-native storage[1]. Graphs, which contain billions of vertexes and edges distributed across multi-machine clusters, can be stored and queried with JanusGraph. This is also supported with transactional consistency and failure stability. We can also use JanusGraph as a traditional database to execute complex graph traversals in real time.

It supports Apache Cassandra, Apache HBase and Oracle Berkeley DB Java Edition as its storage backends. In order to speed up and enable more complex queries, it uses ElasticSearch, Apache Solr and Apache Lucene as its indexing backends. These are specifically useful for full-text queries.

We chose JanusGraph Version 0.1.1 (May,11,2017) for our tests.

JanusGraph contributes to compact graph serialization, rich data modeling, and efficient query execution. Here we present several concepts pertinent to JanusGraph.

We selected JanusGraph as it represents a non-commercial open-source graph database, with ongoing development work. Another reason was that we did not found a similar loading tool in this database, and thus we were motivated to contribute to this community.

An alternative could've the open-source commercial database Neo4j and adding our work to the specific Neo4j csv importer. We would've liked to select this, but we found it easier and more impacting, at least for this current project, to work from scratch on JanusGraph than to add our contributions to an existing tool, with the involved effort

[1] http://janusgraph.org/

of understanding a pre-existing codebase. We also made the choice in the observation that the codebase of the tool has had limited developments in the last 2 years[2].

- **Storage Backends**
  JanusGraph stores graphs in an **adjacency list** format[3], also called *index-free adjacency*. This means, JanusGraph stores each graph as a collection of vertexes, each containing its adjacency list. All of the incident edges (incoming and outgoing) and the edge properties are stored inside the adjacency list of the connected edges. Each edge has to be stored twice - once for each end vertex of the edge. The storage requirements are, thus, doubled ([KG14]).

  In JanusGraph, each adjacency list is stored as a sparse row in the underlying storage backend. A vertex id (64 bit) is assigned uniquely to every vertex by JanusGraph. The vertex id is the key which points to the row containing the vertex's adjacency list. JanusGraph stores each vertex property, edge and edge property as an individual cell in the row. (See Figure 3.1) If the storage backend supports key-order, the adjacency list will be ordered by vertex id. JanusGraph can improve its performance through properly assigning of the vertex id. For example, vertexes, which are frequently co-accessed, are given ids with a small absolute difference. In this way the vertexes will be co-located by the underlying storage[4].

  The adjacency list format speeds up traversals inside JanusGraph, since the filtering criteria of a traversal can be applied in the storage space of each vertex, without accessing another one.

  In our experiment we picked Cassandra as our storage backend, which supports **column-stores**. The understanding of the data model inside JanusGraph and Cassandra, with the related transactional behaviour, inspire us to create a batch/bulk loading method.

  Apache Cassandra[5]
  is a distributed database with scalability and high availability. In our application we use the **remote server mode** to let JanusGraph and Cassandra work together. Logically, Cassandra and JanusGraph are separated into different machines. The Cassandra cluster contains the primary graph data itself, and any number of JanusGraph instances maintain socket-based read/write access to the cluster, in

[2]For reference, the GitHub repository of the Neo4j importer tool: https://github.com/jexp/batch-import

[3]http://docs.janusgraph.org/latest/data-model.html

[4]http://docs.janusgraph.org/latest/data-model.html

[5]Janhttp://docs.janusgraph.org/latest/cassandra.html

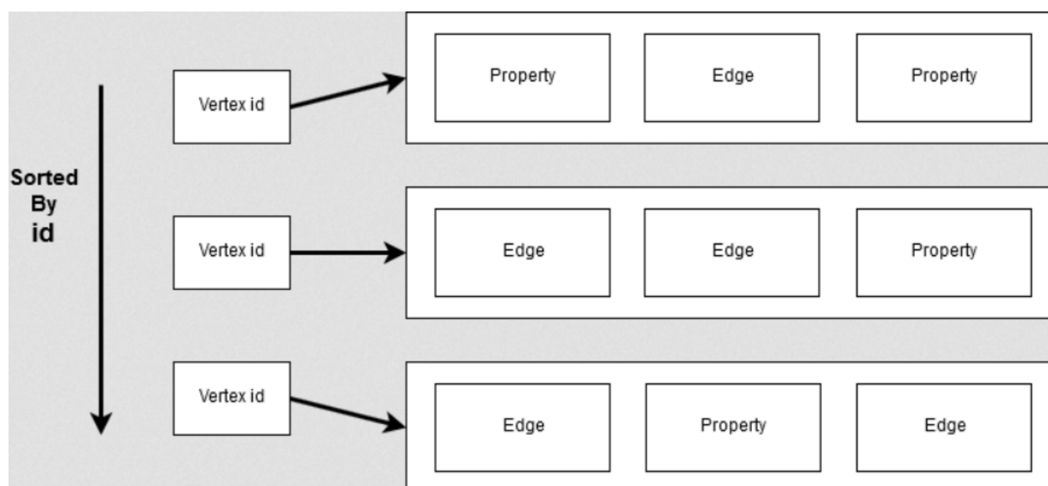


Figure 3.1: JanusGraph Data Model

Column Name	**Column Type**	**Value**
Name	String	Tom

Table 3.1: "Author" Vertex Resource Represented in Cassandra Data Model

addition to specific client-level caches.

To run JanusGraph over Cassandra there are two steps we should set up: 1. Download Cassandra, unpack it and set filesystem paths in `conf/cassandra.yaml` and in `conf/log4j-server.properties` . 2. Start Cassandra by running `bin/cassandra -f` on the command line in the directory where Cassandra was unpacked.

Apache Cassandra Data Model
At first we introduce the Cassandra physical data model. Figure 3.2 represents the Cassandra physical data model used in our experiments for wiki-RfA dataset. Wiki-RfA is a physical structure, which holds a set of column families. Each Column family (CF) stores a specific resource. For example, "Author" contains all user names. "Rfa" contains all vote information. Each vote has two names, the voter and the user running for the elections. And each vote has vote, result, data, year and is typically accompanied by a short comment. (See Figure 3.2) Each row of a column family consists of a unique row key and columns of data of the same resource type (See Table 3.1 and Table 3.2).

For our tests we selected **Apache Cassandra 2.1.1** as a storage backend.

- **Indexing Backends**[6]
  JanusGraph uses indexing to improve its performance. There are two types of

[6]http://docs.janusgraph.org/latest/indexes.html

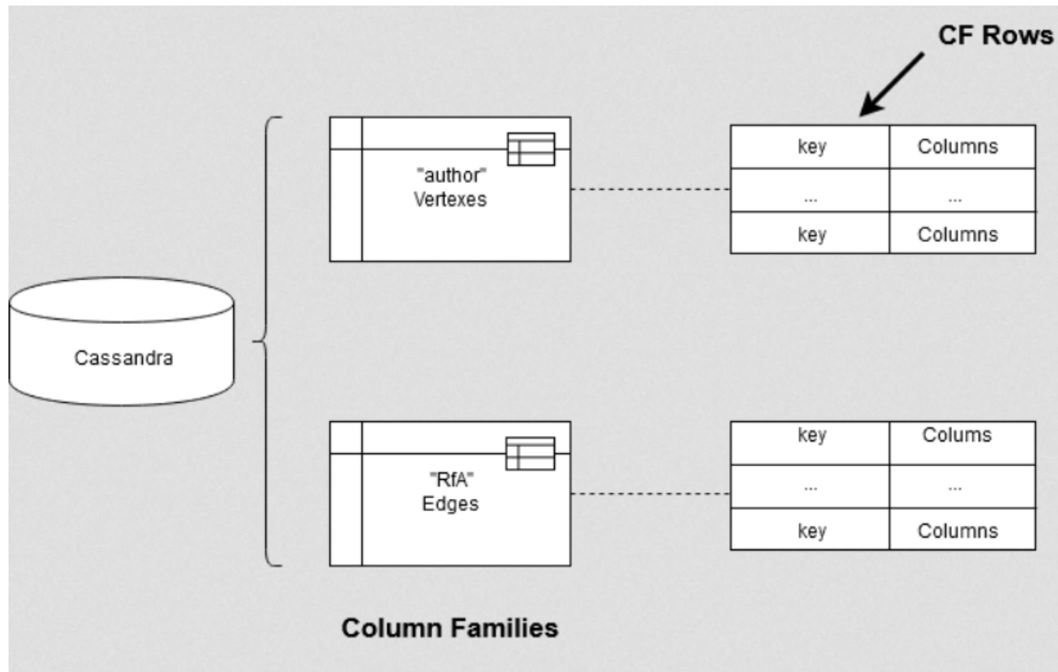


Figure 3.2: Cassandra Physical Data Model

Column Name	**Column Type**	**Value**
Source Name	String	Maria
Target Name	String	Tom
Vote	int	-1
Result	int	1
Year	String	2008
Date	String	19:53, 25 January 2008
Txt	String	hello

Table 3.2: “RfA” Edge Resource Represented in Cassandra Data Model

indexing to speed up query process: **graph indexes** and **vertex-centric indexes**.

Graph indexes are global index structures over the whole graph. After providing the selective conditions, graph indexes allow user to retrieve vertexes or edges by their properties efficiently. Inside graph/global indexes, JanusGraph distinguishes two types of indexing: **composite** and **mixed** indexes. Composite indexes allow the retrieval of vertexes or edges by one or a (fixed) composition of multiple keys. Mixed indexes allow the retrieval of vertexes or edges by any combination of property keys. Composite indexes are fast and efficient but limited to equality lookups. Mixed indexes give us more flexibility and support additional condition predicates beyond equality. It supports multiple condition predicates in addition to equality. In [Dur17], the authors evaluate how different indexing types influence the query performance. Each indexing type has its strengths and weaknesses. JanusGraph supports multiple index backends, such as ElasticSearch, Apache Solr and Apache Lucene. We configured ElasticSearch inside our experimental system, though it was not used in our evaluation.

Apache ElasticSearch[7] is a distributed, RESTful search and analytics engine capable of solving a growing number of use cases. JanusGraph uses ElasticSearch as an index backend to enable several ElasticSearch features. For example, feature **Full-Text** supports all Text predicates to search for text properties that matches a given word, prefix or regular expression.

We picked ElasticSearch 1.5.2 as our full-text indexing backend.

3.3 JanusGraphLab

In order to evaluate our application-level optimizations for loading data into graph databases, such as batching, parallel loading and to address our evaluation questions (Section 3.1) we implemented **JanusGraphLab**.

Given the characteristics of the implementation, JanusGraphLab is not simply a prototypical implementation, but instead it is a functional tool that enables developers to configure efficiently, in a scalable manner the loading process, such as to accelerate it. We aim to make JanusGraphLab publicly available.

3.3.1 Architecture of JanusGraphLab

JanusGraphLab processes different microbenchmarks, which represent various aspects of a JanusGraph application. It explains the options for configurations. It guides us how JanusGraph connects and uses the backends. The connection details are hidden from

[7] http://docs.janusgraph.org/latest/elasticsearch.html

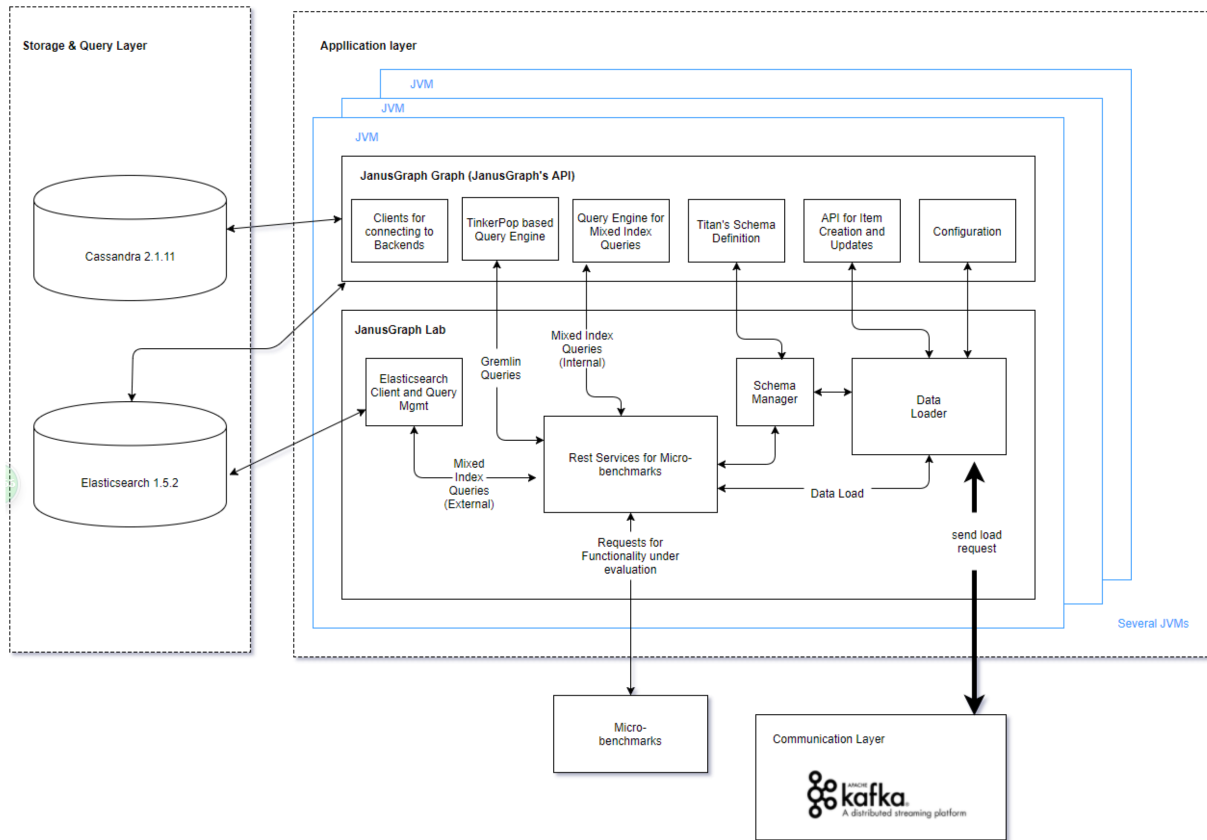


Figure 3.3: An Architectural Overview of the JanusGraphLab

the users to make this connection work more easy to use. It implements the creation of a JanusGraph schema, the loading of data and other required functionality with JanusGraph embedded into the application.

We sketch a high-level structure diagram to help us better to understand the components inside JanusGraphLab, in Figure 3.3. From Figure 3.3 we can see the core components and interactions between them. Firstly, look at this "JanusGraph" section. As a graph database JanusGraph supports us to use it as an **embedded API** for the graph functionality. Going to the JanusGraphLab section, it provides the following functionalities, based on its use of JanusGraph:

- **Schema Management[8]:**
  To define a JanusGraph graph schema we need edge labels, property keys and vertex labels used therein. A JanusGraph schema can either be explicitly or implicitly defined. We prefer an explicit way to make our application more robust and to improve collaborative software development. And for each real-world dataset referred to our experiments we need to adapt them with a specific data schema. Schema Management Module helps the schema definition process. It also helps the query process by supporting REST APIs that need information about the current schema. In our prototype we pass the schema as a configuration file.

- **Data Load**
  Before loading the data to server we should extract them from source data. According with our needs we can sort and partition them with several strategies. After that we can upload the transformed data into JanusGraph clusters. We specifically implemented two kinds of functionalities, on the one side some functions for individual item creation (used for client side tests), on the other functions that for generic loading as a complete process (used for the server side tests). The latter included Kafka distribution of partitions and the partitioning strategies themselves.

- **RESTful Functionality for Micro-benchmarks:**
  JanusGraphLab was developed to provide central **RESTful services** based on JanusGraph, ElasticSearch and Data Load and Schema Management Modules. We use these functionalities to accomplish our tests defined in Micro-benchmarks module.

- **ElasticSearch Client and Queries Management:**
  We could also create a stand-alone ElasticSearch project for querying the full-text backend, using a fixed pool of ElasticSearch clients to reduce connection overheads, though implemented and ported within the context of our thesis, this component was not used in our evaluation.

[8] http://docs.janusgraph.org/latest/schema.html

- **Documentation:**
  We use this functionality to document our prototype and make it easy to understand and contribute to.

- **Micro-benchmark Applications:**
  We start our tests from this module, an alternative to this could be the use of scripts (which we also use). The different micro benchmarks will be introduced in detail in the following chapters. At their essence the microbenchmarks only perform the loading requests with different parameters, number of repetitions and datasets, collecting the results and saving them to a file for further analysis. The core optimizations for the loading process are included in the Data Load Module and made available via REST APIs.

3.3.2 Data Loading in JanusGraphLab

Finally for the purpose of our presentation, JanusGraphLab provides several alternatives for loading graph data. Here are some points we want to mention for the loading process.

- Default Transactional Loading Process[9]:
  1. Open a JanusGraph graph instance to load one dataset. `JanusGraphFactory` gives a set of static `open` methods. These `open` methods take configurations as their arguments and return a graph instance. This is the same process on client or server side.
  2. Write a helper class to load a specific graph. Inside the helper class `graph.newManagement()` helps to create schema, `graph.newTransaction()` helps to open a transaction. These two methods can do the following steps to the newly created graph before returning it.

  - Create a collection of global and vertex-centric indexes on the graph.
  - Add all vertexes to the graph with their properties
  - Add all edges to the graph with their properties. Notice that edges are loaded after loading vertexes.

- Bulk/Batch Loading [10]
  As one of our optimization strategies, we developed **bulk/batch loading** features. Compared with the default transactional loading approach mentioned: **one edge one transaction**, bulk loading adds much more graph data into each JanusGraph transaction: **more edges one transaction**. It reduces the numbers of opening and closing of the transactions, which saves the time of the whole loading process.

[9]http://docs.janusgraph.org/latest/getting-started.html
[10]http://docs.janusgraph.org/latest/bulk-loading.html

- Parallel Loading
  Our approach for loading a graph in parallel, starts by decomposing the data from one graph into multiple subgraphs using one of our partitioning strategies (Section 2.5.2.1). Then these subgraphs can be loaded independently in parallel across multiple machines. **Apache Kafka** helps us to accomplish these tasks by acting as a message passing layer sending the parallel chunks. The stream processing work mode from Kafka helps these parallel sub loading processes to work together and collect information about each other.

- Working Flow
  Figure 3.4 gives us an overview of the data loading process in JanusGraphLab. From this flow chart we can observe the whole working flow for loading data into JanusGraph using JanusGraphLab.

Apache Kafka[11]
Apache Kafka is a distributed streaming platform. It provides publish-subscribe framework and has been considered as a distributed commit log. It lets you publish and subscribe to streams of records, and enable to store streams of records in a fault-tolerant way. It also allows you to process the streams of records as they occur. It has a distributed publisher/subscriber architecture where the information is classified by topics. Kafka helps to process off-line and online tasks by providing a mechanism for parallel loading and it is able to partition real-time consumption over a cluster of machines ([CY15]).

In our experimental prototype JanusGraphLab we use Kafka to build a real-time streaming data pipeline, which helps us to load data in parallel.

3.4 Measurement Process

In our evaluation we focus on the response time of loading functionality for each request (i.e. the time for parsing the file and other, implementation-specific features, are not in consideration). Our decision of focusing in measuring the core tasks, is based on the observation that these are less likely to be affected by implementation subtleties as other tasks.

We use the Java system timer to record the transaction time inside the central application in the RESTService for the Microbenchmarking Module. And we output the time measurement for analyzing our optimization methods.

We use the Java System timers `System.nanoTime()` because of their sufficient precision. `System.nanoTime()` is designed to measure elapsed time, and unaffected by any of

[11] https://kafka.apache.org/

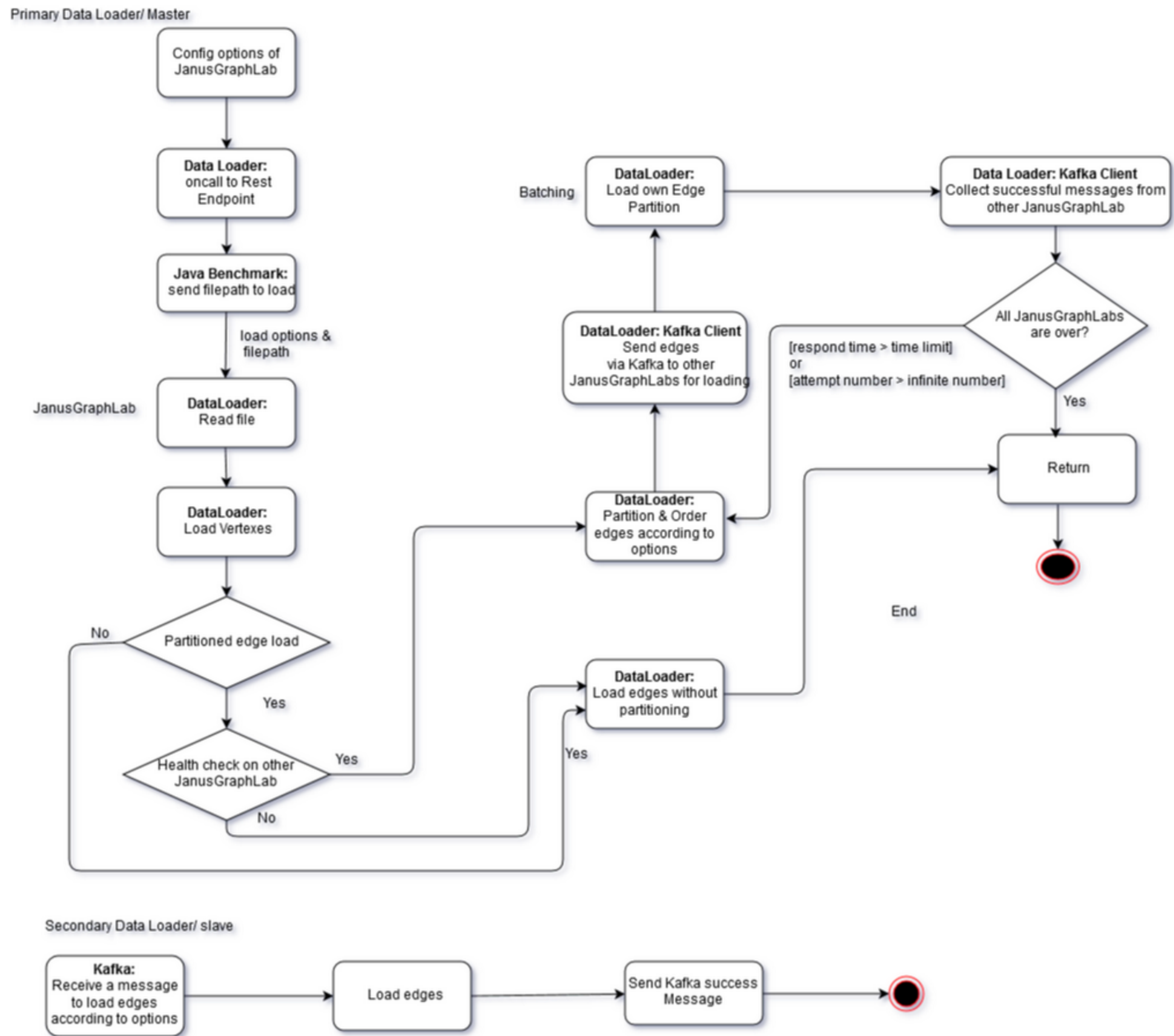


Figure 3.4: Working Flow for Loading Data into JanusGraph using JanusGraphLab

small system clock corrections. And it's simple and efficient to use for our experiments. Although it's very convenient to use system timer. It has also limitations. We record the time at the moment using nanosecond for the whole transaction. But for recording more sub processes or detailed time intervals, it's difficult to realize. In future work we might consider double-checking our results by using more advanced benchmarking solutions.

We focus on recording the time for loading edges. Compared to the data size of edges, vertexes in our experiment datasets are relative much smaller and, commonly, with less properties. That's why we focus on loading edges and on optimization strategies for edges. Before loading edges we start one transaction to load all vertexes and necessary informations. We only accumulate the **partitioning time** of edges, **batching process** (communicating and waiting through Apache Kafka) and **opening/closing transactions time** as the edge loading duration.

3.5 Testing Framework

For **automated testing** we wrote down Linux-based scripts. For each test configuration we repeat 10 times of the loading process and output the test results in several files for our use. Using the collected test results we evaluate each optimization method. Below is an outline of the test script.

1.Start ElasticSearch
2.Start Zookeeper
3.Start Kafka
4.Start a 10 times Loop here

******Loop Begin******
5.Start Cassandra
6.Run Application in the Background. And we set the arguments here: such as number of partitions, partition strategies and batch size...
7.Run Data Loader and we redirect the output to the specific file.
8.After data loaded, we kill the application based on the process number.
9.Delete ElasticSearch Data
10.Delete Cassandra Data
******End Loop******

3.6 Experiment Environment

Here we introduce our experiment set-ups for the real datasets. Our experiments were executed on a commodity multi-core machine composed of 2 processors(8 cores in total) with 251 GB of memory. The application was running on Ubuntu 16.04 and java-1.8.0-openjdk-amd64.
Detailed information of these processors is listed below.

- Product name: Intel(R) Xeon(R) CPU E5-2609 v2 @ 2.50GHz
- Number of Cores: 4
- Number of Threads: 4
- Processor Base Frequency: 2.50 GHz
- Cache: 10 MB SmartCache
- Bus Speed: 6.4 GT/s QPI

And the operation system information:

- LSB Version: core-9.20160110ubuntu0.2-amd64:core-9.20160110ubuntu0.2-noarch:security-9.20160110ubuntu0.2-amd64:security-9.20160110ubuntu0.2-noarch
- Distributor ID: Ubuntu
- Description: Ubuntu 16.04.2 LTS
- Release: 16.04
- Codename: xenial

Most of small business companies that use JanusGraph, Cassandra and ElasticSearch, do so on commodity hardware or cloud-based offerings. We arranged our experiments on commodity hardware to represent this specific scenario.

3.6.1 Backend Configuration

On JanusGraph User Guide page there are pages of configuration introductions for different use cases. We decided on the following configurations which are suitable for our experiments:

- JanusGraph
  We left most configurations with their default values. We only changed two configurations:
  1. `cache.db-cache`, we set this value to false to disable JanusGraph's database-level cache, which is shared across all transactions.
  2. `query.fast-property`, we set it to false to disable pre-fetching all properties on first singular vertex property access. It can reduce backend calls on subsequent property access for the same vertex at the expense of retrieving all properties at once. But the alternative approach can be expensive for vertexes with many properties.

<table>
<tr><th>Dataset</th><th>Type</th><th>Nodes</th><th>Edges</th><th>Data Format</th></tr>
<tr><td rowspan="8">wiki-RfA</td><td rowspan="8">directed, labeled</td><td rowspan="8">10,835</td><td rowspan="8">159,388</td><td>SRC:Guettarda</td></tr>
<tr><td>TGT:Lord Roem</td></tr>
<tr><td>VOT:1</td></tr>
<tr><td>RES:1</td></tr>
<tr><td>YEA:2013</td></tr>
<tr><td>DAT:19:53, 25 January 2013</td></tr>
<tr><td>TXT:”’Support”’ per [[WP:DEAL]]:</td></tr>
<tr><td>clueful, and unlikely to break Wikipedia.</td></tr>
<tr><td rowspan="2">web-Google</td><td rowspan="2">directed</td><td rowspan="2">875,713</td><td rowspan="2">5,105,039</td><td>23434362345 3452346234645</td></tr>
<tr><td>9809809750986 34583748</td></tr>
</table>

Table 3.3: Summary of Datasets used in our Experiments

- Cassandra
  We run Cassandra with the default properties for the 2.1.11 version. Among these properties,Cassandra defines a maximum heap usage of 1/4 of the available memory.

- Apache Kafka
  We used Apache Kafka 2.11-1.0.0

- Zookeeper
  1. the directory where the snapshot is stored. dataDir=/tmp/zookeeper
  2. the port at which the clients will connect clientPort=2181
  3. disable the per-ip limit on the number of connections since this is a non-production config
  maxClientCnxns=0

3.7 Datasets

We tackle our evaluation questions by running the data loading process on real-world datasets. We have selected two datasets from different areas, with different sizes, in order to make our tests more diverse.

3.7.1 Wikipedia Requests for Adminship (with text)

To become a Wikipedia administrator, contributors should submit a request for adminship (RfA). Then any Wikipedia member can cast a supporting, neutral,or opposing vote. This dataset collects the complete set of votes from 2003 (when the Wiki-RfA practice started) until 2013. It contains 11,402 users (voters and votes), which form 189,004 distinct voter/votee pairs. And if the same user ran for election several times, the same voter/votee pair may contribute several votes.This dataset is a directed, signed network. Nodes represent Wikipedia members and edges represent votes. There is also a rich

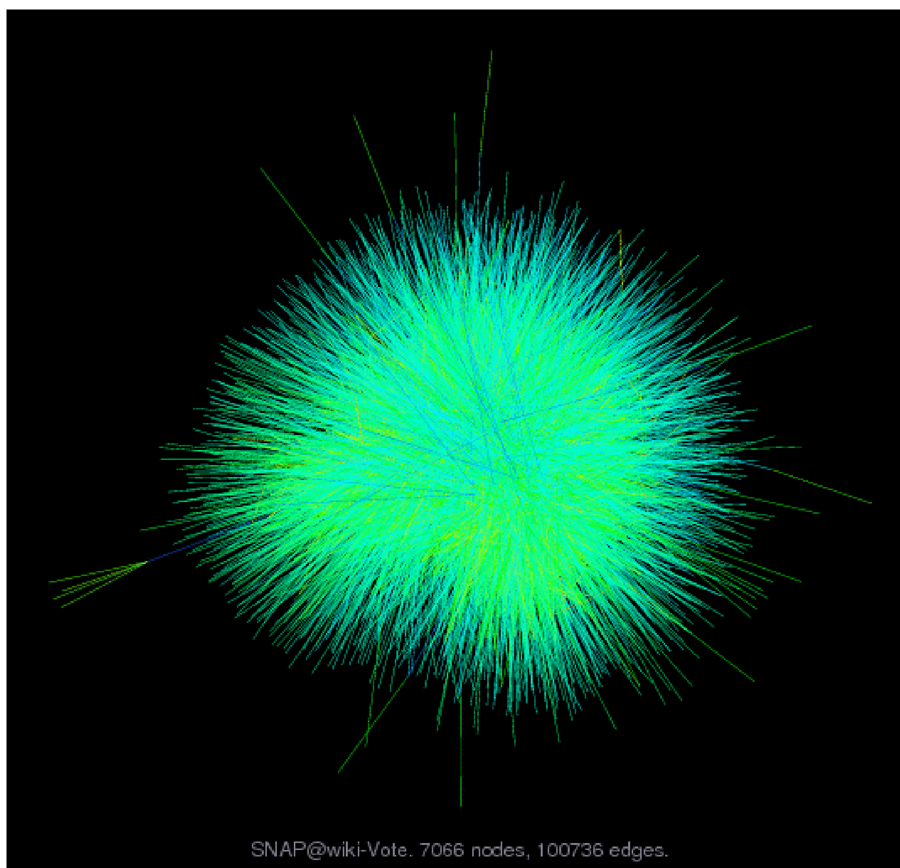


Figure 3.5: Topology for the Wiki-Vote Dataset (An Earlier Version of Wiki-RfA) [12]

textual component in RfAs since each vote is combined with a short comment. For example, a typical positive comment reads, "I've no concerns, will make an excellent addition to the admin corps" ([wik]).

Wiki-RfA is an example of a real-world temporal signed network, since edges represent either positive, negative or neutral votes, and the network presents a time dimension that specifies the order in which votes are cast. In terms of topology, Wiki-RfA can be classified as a social media network, this is a kind of network similar to a social network (i.e., it can also be considered to be based on a social network), with the same scale-free properties and short paths, but that can be shaped by the affordances of the interaction platform [KALB12]. Figure 3.5 represents a view of the network with voters and votees acting as hubs. We can expect this network to follow the topology of a social network, with short paths between all nodes.

3.7.2 Google Web Graph

We choose the Google-Web graph as a representative of information networks. In this graph, nodes represent web pages and directed edges represent hyperlinks between them.

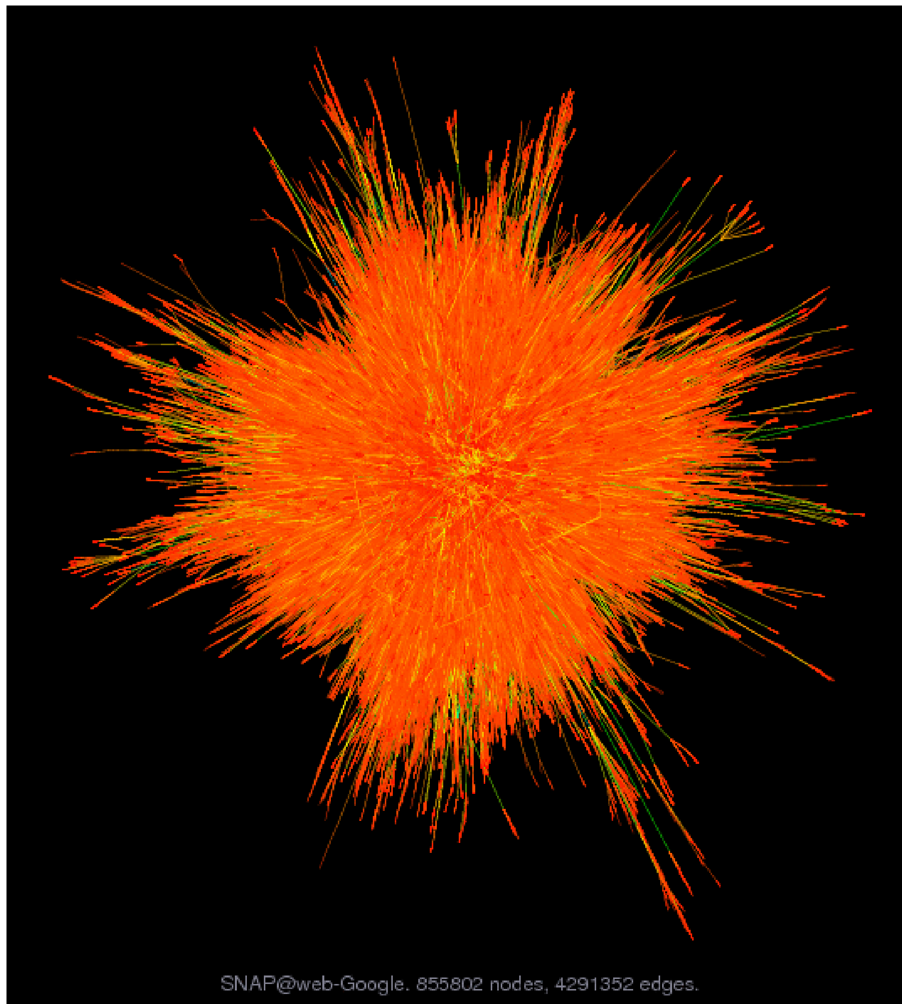


Figure 3.6: Topology for the Google Web Dataset
13

The expected topology for this graph is to find a strongly connected central component with expanding incoming and outgoing groups of vertexes, with links to and fro this central component, and finally with a large amount of weakly connected components at the edges. This structure should resemble a bow-tie ([CF12]). Figure 3.6 displays the topology of the provided dataset. The strongly connected component remains but the bow-tie structure (with longer paths for some groups of nodes) is not entirely visible. This could be due to the small scale of the dataset.

Google released this dataset in 2002 as part of its programming contest[goo].

Table 3.3 gives us more size and type details for each dataset.

3.8 Summary

In summary, in this chapter we proposed a list of evaluation questions to guide or research. These questions cover part of the design space for graph data loading, starting with the basic concern of client vs. server side loading, covering as well the effect of batching and parallelizing the process.

In this chapter we also discussed the basic characteristics of the technologies we used for our work (JanusGraph, Kafka, Cassandra), and we presented JanusGraphLab, our prototypical implementation for evaluating the impact of loading optimizations. Apart from it's role as an evaluation framework, JanusGraphLab can, de facto, be used as a tool to support developers in scaling out and accelerating their graph loading tasks.

To conclude the chapter we disclosed our measurement process, the testing framework and environment, next to the datasets that we selected. In terms of topological characteristics we observed that the Google Web Graph should present longer paths when compared to the Wiki-RfA, since the latter can be considered to be based on a social network.

In the next chapter we present the tests and results for evaluating the first of our research questions.

4. Case Study: Client vs. Server Side and Batching

In this chapter we ask, what is the right place for loading graph data, considering first if there are fundamental performance differences between carrying out the load process from client vs. server side. This evaluation aims to guide our research, in determining whether it is worthwhile to have optimizations being considered at the server side (i.e., closer to the database) or more simply at the client side. Then we introduce our evaluation on the performance impact of batching for loading tasks. This is a strategy for reducing the number of transactions, and resulting requests to the database. We start by researching a basic load process, which we will call the 'baseline'. This process loads all the vertexes, and opens one transaction per vertex where all JanusGraph performance enhancements (background optimizations, etc.) are left enabled. Next we propose a targeted microbenchmark that studies exclusively the batch/bulk loading process. We first load all vertexes and subsequently the edges in a batched manner. In this study we compare 4 loading processes with different batch size for each: a) Batch 1: 1 edge per transaction, b)Batch 10: 10 edges per transaction, c)Batch 100: 100 edges per transaction and e)Batch 1000: 1000 edges per transaction.

This chapter is organized as follows:

- **Evaluation Questions:**
  We list evaluation questions in this section. (Section 4.1)

- **Load from Client vs. Server Side:**
  We introduce and evaluate two options for loading graph data into JanusGraph classified by the location of the loading logic. The results of this evaluation motivate our later work. (Section 4.2)

- **Batch/Bulk Loading in JanusGraph:**
  We describe how batch loading works in our JanusGraphLab experimental prototype. (Section 4.3)

- **Microbenchmarks:**
  We present our design and implementation of the microbenchmarks for our evaluation, the results from our tests, and a discussion (Section 4.4). This section covers specifically the tests for batch/bulk loading.

- **Best Practices:**
  We provide a short list of the best practices that can be learned from the evaluations discussed in this chapter. (Section 4.5)

- **Summary:**
  We conclude with a brief summary of the chapter. (Section 4.6)

4.1 Evaluation Questions

1. Which is the best place to put the loading logic, at client or at server side?

2. What is the effect of batching when loading graphs of different topologies?

These are the first two of the evaluation questions we defined for our study (Section 3.1).

4.2 Loading Process from Client/Server Side

As discussed in Section 2.4.3, the loading process involves several steps according to the source files. The main steps we proposed where loading of vertexes and loading of edges; each of these involved parsing the files, creating in-memory mappings for ids, ordering the input items, determining the load granule (i.e., transaction size or batch size) and distributing/parallelizing the process itself.

Considering that database operations can be performed as client or server codes (with the first one being passed to the systems as a series of http, websocket, language client or CLI requests, and the latter being passed as a single script to be executed on the server side), the first question in designing a loading tool for a graph database is to determine which of these options is the best for launching the process.

In this section we introduce the key ideas of loading from client/server side, then we present test results that allowed us to decide on the best location for our tool design.

- Loading process from Client Side

  For loading graph data into an online graph database like JanusGraph, we need transactions. These help ensure an amount of consistency between reads and writes in the management of the database. The specific consistency promises that transactions can deliver depend, in the end, on the underlying non-native storage used by JanusGraph. Thus, in the case of Cassandra transactions are configured by default to ensure BASE consistency (i.e., Basically Available, Soft State, Eventual Consistency)[1]. For providing transactions, interactions with the database need to be managed via a client (which takes a transaction at a time) or a server (which can take a script with a series of transactions to perform).

  At the beginning of our experimental work, we asked to open the transactions from an HTTP client in the Micro-benchmarking Module. Each time we needed to load something we ran the Micro-benchmarking Module, which in turn carried out all the process and created individual REST requests (for creating elements) to send the client (i.e., JanusGraphLab, with the embedded JanusGraph process). We call this mode of processing as a loading process from client side, since all logic was performed on the Micro-benchmarking Module, and the JanusGraphLab only received individual request to insert specific items, without having access to temporary data structures used in the loading or other parts of the process. In this mode, all steps of the loading process are undertaken on the application side (Micro-benchmarking Module) and only the individual write operations are sent as individual requests to the database client (JanusGraphLab).

- Loading process from Server Side

  A second mode of processing was found after refactoring our code, placing the loading process inside a single REST endpoint made available in JanusGraphLab to the Micro-Benchmarking Module. Specifically we redesigned the implementation to provide a single endpoint to input a file, or set of files, in addition to a list of parameters (such as batch size or partitioning strategy), to JanusGraphLab. On receiving this, JanusGraphLab was provided with functionality to create a complete representation of the transactions in the request (either as a groovy script, or simply interacting with the embedded JanusGraph client, who also creates a groovy scripts from the requests), finally the request was executed by JanusGraph both in the embedded client and resourcing to the backends. We call this second

[1] JanusGraph offers options to enforce higher consistency levels through the use of locks. We did not use this in our evaluations, instead relying on the eventual consistency of the storage. Some issues are reported by the database developer team, such as the possible appearance of ghost vertexes, however these do not affect our study since we consider only loading and not updates. More information in: http://docs.janusgraph.org/latest/eventual-consistency.html

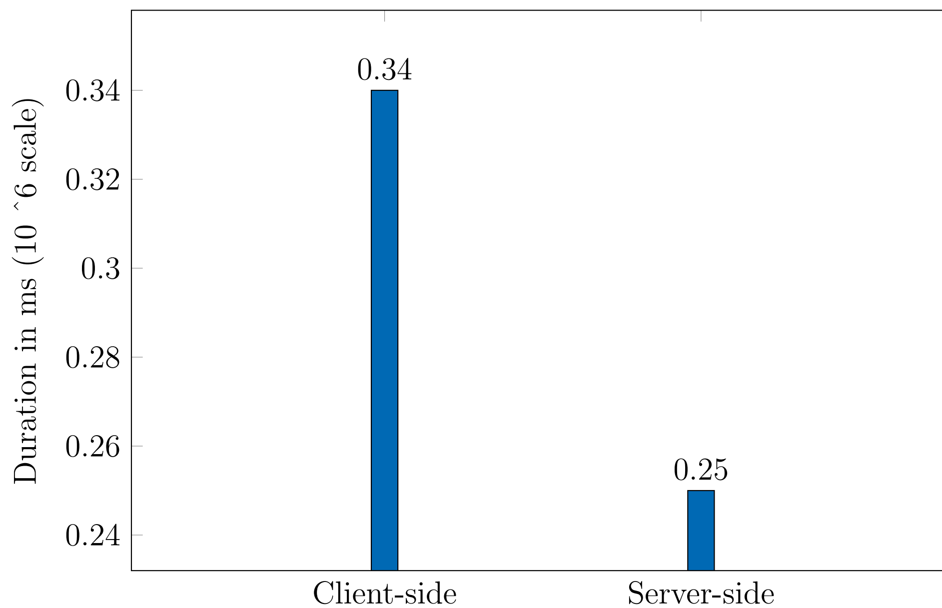


Figure 4.1: Client vs. Server-side Management of the Data Loading

mode of execution, server side loading.

A noteworthy aspect is our use of scripting which might reduce overheads from interpretation. In our current implementation we only employ scripting for creating the items in the graph. This covers the batching optimizations we discuss later, but it does not cover partitioning.

Here we give some test results from our evaluation in distinguishing both modes of loading. Figure 4.1 represents the average time performance over 10 runs of loading data from Client/server side. We used Wiki-Rfa dataset. The average loading time from client side is 339283.2ms (5.65 minutes). The average loading time from server side is 245320.4265ms (4.08 minutes). And the average speed up is 1.38x.

From this evaluation we observe that even for a relatively small dataset, and without adopting any optimization, there is an evident distinction between loading in client side vs. server side, leading at least moderate speedups. Thus we answer experimentally to the first of the evaluation questions we assigned to this chapter.

As a result from our experiment, we decided for the rest of our studies to develop the loading functionality in the server side, as much as possible.

4.3 Batch Loading in JanusGraph

JanusGraph provides several configuration options to make ingesting large amounts of graph data more efficient. As mentioned in Section 2.5.1, we aim to contribute to these alternatives by developing a bulk/Batch loading process.

JanusGraph provides transactional loading by default, where small amounts of data are loaded per transaction. Bulk/Batch loading enables us to add large amounts of data to individual transactions.

In our JanusGraphLab experimental prototype we develop batch loading functionality. Below are the steps we follow.

1. Set `batch-loading=true`. We run JanusGraph with beneficial configurations for bulk loading. Enabling batch loading disables JanusGraph internal consistency checks in a number of places and disables locking to improve the performance. We leave other related configurations as default.

2. Before loading edges we upload all of the vertexes with their properties to the graph (i.e., we create the vertexes with their properties). We also create a map between the vertex ids used in the dataset (i.e. provided unique identifiers) and the internal vertex ids assigned by JanusGraph (i.e. `vertex.getId()`) which are 64 bit long ids. Keeping such a map is done building upon the insight from previous research on Apache Titan (i.e. the database product on which JanusGraph is based), that retrieval of data is better when performed using the database identifiers than when performed using individual indexed identifiers. In fact this was observed for random access of different number of vertexes rather than for individual access, with authors reporting 45 to 59x speedups from accessing items by their identifiers ([Dur17]).

3. The last step adds all the edges using the vertex map. The edge loading process uses the JanusGraph internal vertex id for retrieval of vertexes from JanusGraph.

From these steps both the vertex creation and edge creation can be performed in batches of configurable sizes. This is basically achieved by adding more statements to create items within a transactional scope (i.e. between `tx.start()` and `tx.commit()`).

4.4 Microbenchmark

To answer the second evaluation question assigned to this chapter (i.e., What is the effect of batching when loading graphs of different topologies?), we designed a set of tests and named them as "**Batching without Partitioning**". In this section we focus on the

influence of batch approaches, and ignore parallelization and partitioning optimizations.

Here we compare the loading performance with vs. without batching strategies. The core idea of the batch/bulk loading is that for each transaction we load more objects instead of a single object. In our experiments we only considered batching strategies for loading all edges. This was first because the size of edges is bigger than the size of vertexes, secondly because in our observation this process took more time than the loading of vertexes (which, compared to loading edges which need to re-load the vertexes to which they will attach the edges, can be done without retrieving items).

In our tests we evaluated the response time of a) batch size = 1, b) batch size = 10, c) batch size = 100, d) batch size = 1000. For these tests we used the Wiki-RfA dataset and the Web-Google dataset, as defined in Section 3.7. Batch sizes describe the number of edges in each loading transaction. Here we want to evaluate the influence of batch size on the total load time. Batch size of 1 means that the loading task doesn't use any batching strategy (baseline). The results we report are the average for 10 repetitions.

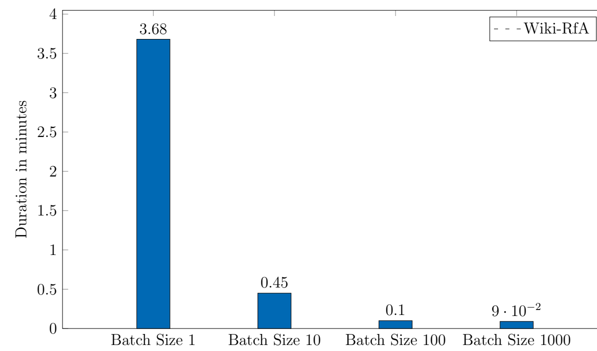
Figure 4.2 shows the time taken to load all edges with different batch sizes. Each sub chart shows the test results from one dataset. It can be seen among the two charts that batching approaches reduce the loading time significantly. The bigger the batch size, the faster the loading process is observed to be. However, when batch sizes are increased exponentially, the loading time does not decrease in the same scale. There seems to be diminishing returns from the increases in batch sizes. In fact, beyond a certain extent, the time improvement of performance from increased batch sizes becomes smaller. If the batch size is very big, it might even increase the overall time of the loading task. From our test results, the threshold of batch size where the best performance is achieved is 100.

We speculate that a possible explanation for the decreasing gains from batching could be that more data per transaction deteriorates the use of transaction caches, breaking temporal and spatial locality that appear on small transactions. A further aspect that should be considered is that large transactions could also lead to more costly distributed transactions. This was not studied here, since we did not employ multi-node backends.

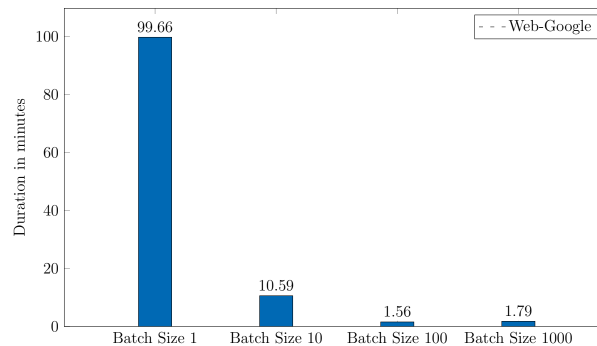
Figure 4.3 shows the speedups for loading tasks with batching strategies. The speedups are calculated based on the time of loading process without the batching strategy(baseline).

$$Speedup = T_1/T_n \tag{4.1}$$

T1 is the baseline. –Loading without batching.
Tn is the loading time with batch size equals n.



(a) Effect of Batch Loading the Edges (Wiki-RfA)



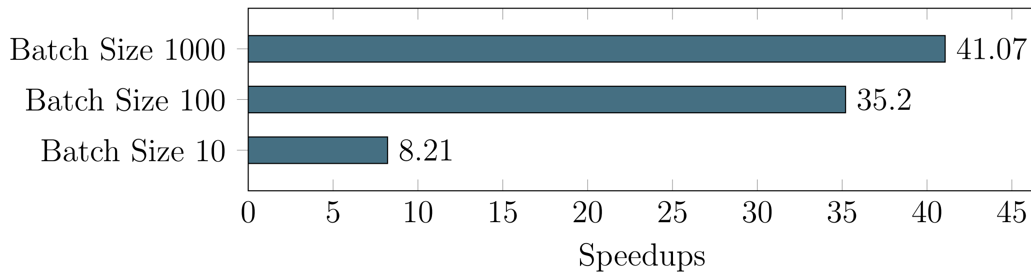
(b) Effect of Batch Loading the Edges (Web-Google)

Figure 4.2: Effect of Batch-Loading the Edges

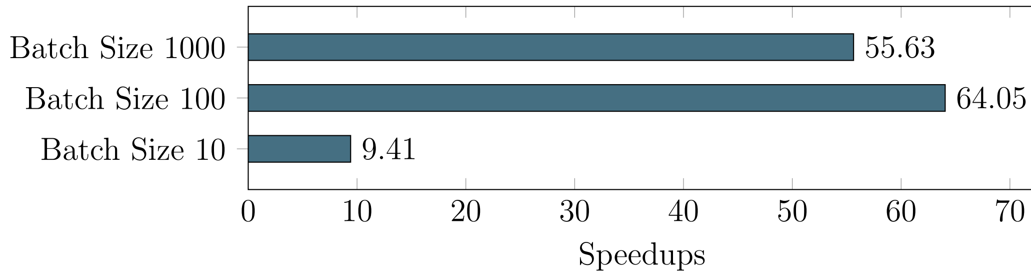
The effect of batching, which was seen to yield a performance improvement of 10x over the non-batched approach, is clearly visible from Figure 4.3. Batching improves the performance by a large margin, but the benefits were not equal over all datasets. For example, the 'Web-Google' dataset shows a reduced speedup. The one interesting thing to note is that in dataset "Web-Google" the speedup is reduced, when batch size equals 1000, while the same is not evident in Wiki-RFA. From this we can speculate that the batch size is not the only factor that affects loading performance and that topology characteristics, affecting in turn transaction cache usage, might also have an impact. Specifically, Wiki-RFA represents a more connected network than Web-Google, thus there might be more chances of reusing data already in the transaction cache, reducing loading costs. Further studies would be needed to verify these possible cases.

4.5 Best Practices

Following the test results in this chapter we define some best practices regarding batching approaches and the server vs. client side question. These best practices are a direct answer to the questions mentioned in Section 4.1.



(a) Speedups from Batch-Loading the Edges (Wiki-RfA)



(b) Speedups from Batch-Loading the Edges (Web-Google)

Figure 4.3: Speedups from Batch-Loading the Edges

1. Observed from the test results we should load graph data from server side because of its better performance. It is a best practice for developers to write the load logic directly from server side. Temporal structures, such as the mapping between unique identifiers and JanusGraph identifiers, can be more useful when managed from server than from client side. Also, reducing the number of requests can bring performance gain by lessening the communication and interpretation costs of individual requests. In our work we aim to offer a graph loading tool that works on server side and supports developers in their different data loading scenarios.

2. For developers it's a best choice to load graph data using a batching approach. Compared with the traditional loading process (one object, one transaction), a batch loading process loads several objects (vertexes and edges) inside a single transaction leading to decreased transaction overheads. The performance improvement is evident, however it decreases with large batch sizes. In our study we report a case where by moving from a batch size of 1 to 100, the loading process moves from 100 minutes to close to 1.5 minutes.

3. The data size is relevant for the performance improvement to be expected from batching. When the loading time is relatively long, the impact of batch on loading time is even more noticeable. Thus developers that expect substantial loading times should consider batching as a useful optimization. Our experiments support this observation for non-distributed cases. Further tests are needed to cover distributed cases.

4. For different topology of graphs, the efficiency of the batching approach seems to work the same for the cases when the gains from larger batch sizes do not decrease. For our two test datasets, loading time is reduced close to linearly.

5. When the gains from larger batch sizes decrease, topology characteristics might have an impact on the transaction cache use, with more connected topologies performing better. Further studies are needed to assess the latter assumption.

6. The limitation is, larger batches do not always guarantee better performance. We could intuitively observe from Figure 4.3(b), when batch size equals 1000, the speedup is reduced. In summary, finding a **suitable batch size** is the key to improve load performance. From the test results we recommend choosing batch size between 100 and 1000, or on the same order of magnitude. Before choosing a batch size, we advice developers to test how the batch size influences their own data load, perhaps on a sample load, before proceeding on the complete dataset.

4.6 Summary

In this chapter we evaluated client vs. server side loading, and batched approaches to the loading process. First we introduced the motivation for our load optimization work by considering client vs. server side loading. We have found that the different places to put the load logic can influence the loading performance. In the limited context of our evaluation we found that server side loading achieves better results. Then we introduced batch loading implementation details: loading several objects inside a transaction. Next we described the microbenchmarks that we designed to compare the loading time with different batch sizes. Next we discussed our test results. Stemming from these we were able to propose some best practices for application developers to adopt batch loading approaches for loading tasks with JanusGraph.

The following chapter proposes other microbenchmarks we created. Specially we consider partitioning and parallel loading.

5. Case Study: Partitioning and Parallel Loading

In our studies so far we have considered batching, which consisted on fitting more data inside a single transaction, in order to reduce the number of transactions employed in the loading process. In this chapter we consider how to organize the process with parallel transactions by partitioning the data into parallel chunks and running the loading for each chunk in separate requests to the backend. Contrasted to the previous experiments, with this approach we do not seek to reduce the number of transactions but to schedule them in such a way that some of them can be performed simultaneously, thus possibly reducing the overall runtime.

This chapter is outlined as follows:

- **Evaluation Questions:**
  We start by recapitulating the evaluation questions that guide this chapter. (Section 5.1)
- **Partitioning and Parallel Loading:**
  We describe the basic concepts for our implementation of partitioned and parallel loading (Section 5.2), discussing too how we included a publisher/subscriber framework into the data loading process (Section 5.3).
- **Microbenchmarks:**
  We answer the evaluation questions regarding experimental analysis and results. (Section 5.4)
- **Best Practices:**
  For clarity, we collect the findings of this chapter in a list of best practices. (Section 5.5

- **Summary:**
  To conclude we summarize the work in this chapter. (Section 5.6)

5.1 Evaluation Questions

Considering the partitioning and parallel loading steps that we mentioned in Section 5.2, we recapitulate the evaluation questions which motivate this chapter.

3. What are the opportunities and limitations in parallelizing and distributing the data loading?
4. What is the influence of partitioning strategies for loading a dataset in parallel loading?
5. What are the best practices for integrating publisher/subscriber framework into the data loading process?

5.2 Partitioning and Parallel Loading

As mentioned in (Section 2.5.2), parallelization might accelerate the loading process. To achieve this it is necessary to determine a strategy to partition the loading task. One straightforward possibility is to partition the dataset into groups of items that can be inserted separately.

We have selected several partitioning strategies among those discussed in Section 2.5.2.1. The partitioning strategies are mainly applied to the edges, since the datasets chosen in this study have a larger number of edges than vertexes, i.e., the partition strategies will have more impact on the edges. We focus on partitioning edges, after inserting all vertexes in a sequential manner. We followed the steps below to load our graph data in parallel:

1. Add all vertexes with their properties to the stored graph, and create a map. This map has the mapping information to translate dataset-specific vertex ids to the JanusGraph-assigned internal vertex ids.
2. Partition all edges into several small edge lists using one partitioning strategy.
3. Insert each of these these edge lists/partitions in parallel. Each edge list loading process runs as a separate script in a separate JanusGraph client. When one partition finishes its task, it will send a message acknowledging the completion of the task. When all partitions are inserted, the received the original loading request returns. We use Apache Kafka to arrange the cooperation. When we add these edges, it's necessary to send each client the map from JanusGraph internal ids. Currently we serialize and send this map as a Kafka message (since even for large graphs, this is kept under 1 MB), for future versions of our loader we will consider to distribute this map using a clustered key value store, like Redis.

5.3 Partitioning and Parallel Loading, from the Publisher/Subscriber Perspective

In this section we briefly discuss how we introduce a streaming publish/subscribe framework (Apache Kafka) into the data loading process.

In order to support parallel loading the first task is to start a set of JanusGraph instances, all of which are connected to the same clustered backend. For our work, we assign an environmental variable that acts as a unique identifier of each instance.

Instances play 2 roles in our framework, on the one hand they can take general API requests (e.g. for searching through the graph), on the other hand they can perform background collaborative tasks, such as parallel graph loading.

In order to support the dual roles, we divide each JanusGraph into 2 threads, one in the background who is subscribed to a Kafka topic, waiting to receive messages and programmed to perform certain background/collaborative actions upon the reception of given messages. The main thread, in the foreground, is capable of receiving and processing the REST requests from our microbenchmarks.

In our experiments we start several instances, yet it is only one who takes the initial loading request and manages the process, as described in Section 5.2 and Section 5.3.

First of all the thread in the foreground of this instance loads the vertexes and then partitions all edges in smaller edge lists, in order to schedule for the load to happen in parallel. Then, by having a pre-programmed list of ids, the thread sends the requests for loading the edges into the separate Kafka topics that correspond to the background threads of the selected instances (both the topics and the instances are identified by their environmental variables).

Upon reception of a request/message, the background thread in each of the instances will insert the edges, according to provided parameters, and return a message acknowledging the the load task was completed.

On the other side, at the site of the original request, once the main thread has sent all the messages, this same thread can load one partition (such that parallelism is not lost). Subsequently this thread waits to collect the acknowledgements from all the instances (also via a dedicated Kafka topic). To conclude, the main thread returns a message indicating the success in the loading process.

As we can see from the diagram Figure 5.1, each partition is assigned to a thread. For the first Partition, it produces all edges, total number of edge partitions to specific

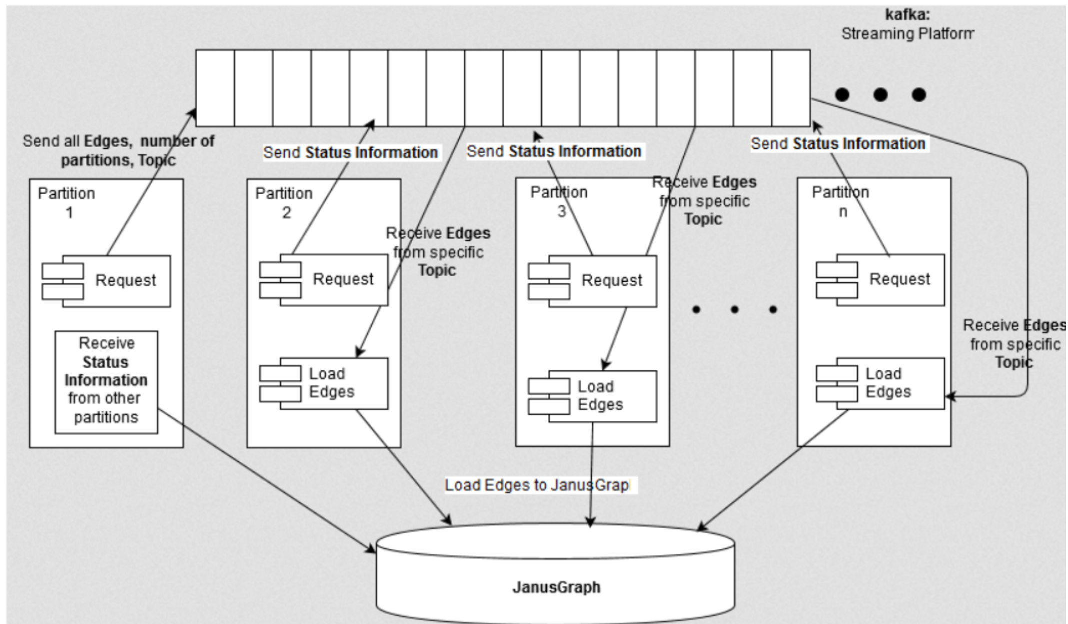


Figure 5.1: Loading Data with Apache Kafka

topic. And it consumes status information (like "finished") from other partitions. From the second partition, they consume small partitioned edge list and load the edges to JanusGraph. They also produce their own status information to specific topics. The diagram gives us an overview of loading process with Apache Kafka.

Though our evaluation reports results for a single multi-core processor. The use of Kafka enables our prototype to scale out to a clustered architecture.

Since we do not perform tests on alternative implementations for including the Publish/-Subscribe framework into the loading process, this design constitutes our answer to the last evaluation question in this chapter. We judge our design to be reasonable and to encompass good practices like allowing for a scalable design, with a single master process managing the load. Some limitations of our design include the overheads introduced by network usage in Kafka (specifically considering the large messages for sending the edge partitions and the maps), the always alive nature of the background thread (with an approach like server-less computing, this thread could only be created when needed). Future work could consider improving these limitations through exploring server-less computing, better network usage (e.g. exploiting RDMA), compression of the messages, and a distributed setting for sharing the map across the processes.

5.4 Microbenchmarks

The parallel and partitioned loader uploads each partitioned edge list on its own loading thread. In order to gain clearer insights into the specific benefits from scaling out the

parallel load, we designed tests for evaluating our parallel and partitioned approaches without including batching (Section 5.4.1). We also compared loading processes with 4 different partitioning strategies as mentioned in Section 2.5.2.1.

In our tests we evaluate the response time of: a) baseline, load without any partitioning strategy, b) partition 2, we separate all edges to two smaller edge lists, c) partition 4, we separate all edges into four edge lists, d) partition 8, we separate all edges to eight small edge lists. For these tests we used the Wiki-RfA dataset and the Web-Google dataset (introduced in Section 3.7). And we select 4 basic partitioning strategies: a)VV, b)EE, c)BE and d)DS.

In what follows we recapitulate the key ideas from the strategies Section 2.5.2.1.

- **E/E Strategy**
  Using a round-robin (RR) algorithm to distribute the large edge list into small edge lists. This strategy separates many or all outgoing edges from one vertex to multiple partitions. It spreads the edges completely, balancing the load evenly.
- **V/V Strategy**
  In contrast to the E/E strategy, this one assigns edges from one vertex to a single partition. This can lead to imbalances. It uses a round robin strategy.
- **BE Strategy**
  It's a balanced E/E strategy. It balances the amount of edges per partition and attempts to keep all outgoing edges from one vertex in the same partition.
- **DS Strategy**
  It's a combination of BE strategies and the 80/20 rules introduced in chapter 2. It extends the BE strategy for handling highly skewed data Section 2.5.2.1, easing the pressure introduced to the system by the existence of highly connected vertexes. In this situation, for specific vertexes, a large number of connected edges may exist (this would create large imbalances for the V/V strategy). The DS strategy sorts each vertex by the number of edges it contains. Then edges from the top 20% vertexes (we call it popular vertexes) are assigned to different partitions using a round-robin strategy. Edges from the rest 80% vertexes are assigned using the V/V strategy.

5.4.1 Parallel and Partitioning Loading without Batching

In this section we present the results of our studies on the influence of partitioning and parallel approaches, without considering batch loading. We present the results of parallel loading with different partition strategies. These studies are our response to the three evaluation questions that guide this chapter.

Figure 5.2 shows the average duration time through several runs of loading all edges of a graph dataset. The response times, when considering partitioning are also the average

from the different strategies. It can be seen from the chart that partitioning and parallel approaches consistently reduce the loading time. The more partitions are processed in parallel at the same time, the shorter is the overall loading time. However, the loading time does not decrease proportionally to the growth in the number of partitions. When the number of partitions increases to a certain extent, the improvement of time performance becomes smaller.

Figure 5.3 records speedups for parallel and partitioning loading without any batching.

$$Speedup = T(baseline)/T(parallelloading) \tag{5.1}$$

The number of partitions represents the number of work threads. Our choices of number of the working threads are related to the CPU inside. Multiples of 2 are better for full usage of the eight core CPU. But, reinstating our observations, even if the the partition number increases exponentially, the speedups don't grow with this trend. Although the parallel processing improves the performance of the loading tasks, the overheads added due to threading and communication (i.e., more Kafka clients) limit the speedups for relatively short loading tasks. For this reason, when the loading time is relatively short, the rate of increase of the speedup is declining. Thus, a careful balance is required for determining the best number of partitions according to the size of the loading task.

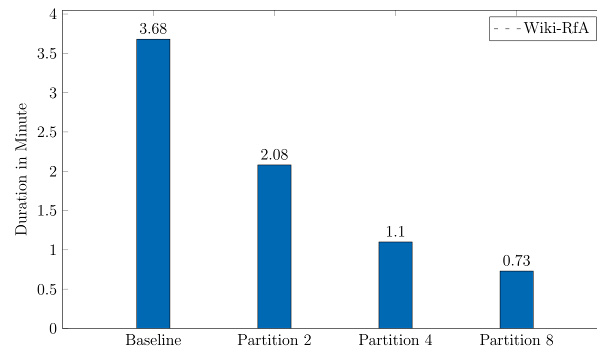
Regarding the evaluation of the specific strategies, we picked up 4 partitioning strategies: VV, EE, BE, DS. And the baseline is the loading time without partition strategies.

Figure 5.4 records loading time for these partitioning strategies. We can observe from 5.4(a) that loading processes using Strategy VV spent more time than using other strategies. Strategy VV is not balanced and in some situation it can lead to lower performance.

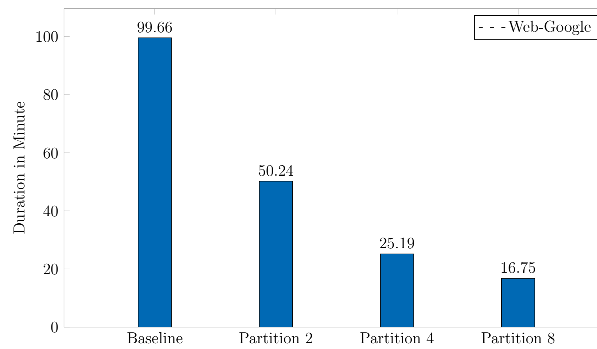
In spite of the small difference for VV, we found that, overall, these basic partitioning strategies have little impact on loading time. Loading time using different strategies doesn't translate to large differences in performance.

However we speculate that for different topologies of graph datasets, and for scaled-out architectures (where the loading is distributed but coordinated) the influence of these strategies might be different. We believe that there is no "one-size-fits-all" partitioning strategy for all scenarios.

The specific speedups from the strategies are presented in Figure 5.5. We can observe once again that speedups almost doubled with the growing number of partitions. However, when the loading time is quite short (i.e., for smaller datasets), the impact of partitioning and parallel loading approach on the performance of the entire loading process becomes

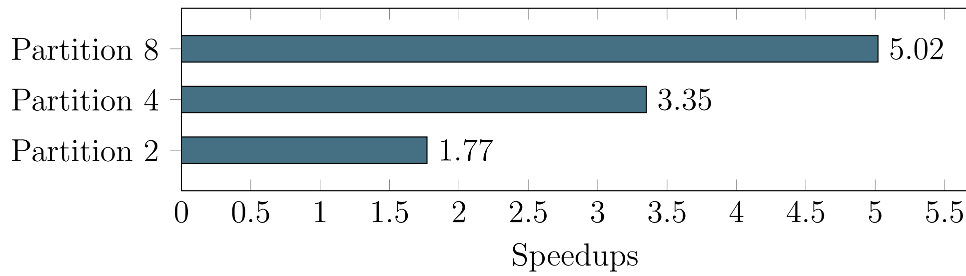


(a) Loading Time Using Partitioning and Parallel Approaches (Wiki-RfA)

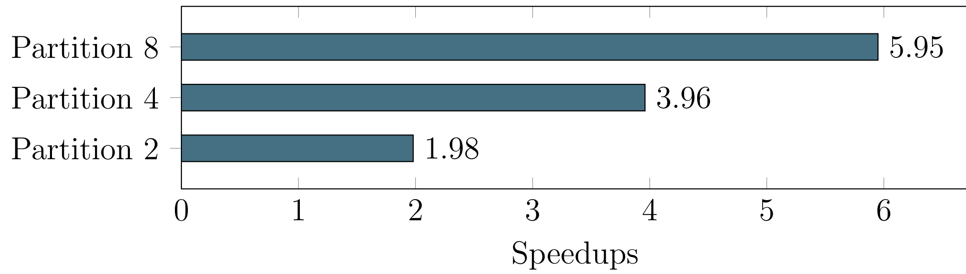


(b) Loading Time Using Partitioning and Parallel Approaches (Web-Google)

Figure 5.2: Loading Time Using Partitioning and Parallel approaches



(a) Speedups from Parallel and Partitioning the Loading of Edges (Wiki-RfA)



(b) Speedups from Parallel and Partitioning the Loading of Edges (Web-Google)

Figure 5.3: Speedups from Loading Time using Partitioning and Parallel Approaches

smaller, leading to smaller speedups. For the different partitioning strategies, the conclusion is the same: the distinctions of partitioning strategies have little performance impact.

5.4.2 A Closer Look into Load Balancing with the Partitioning Strategies

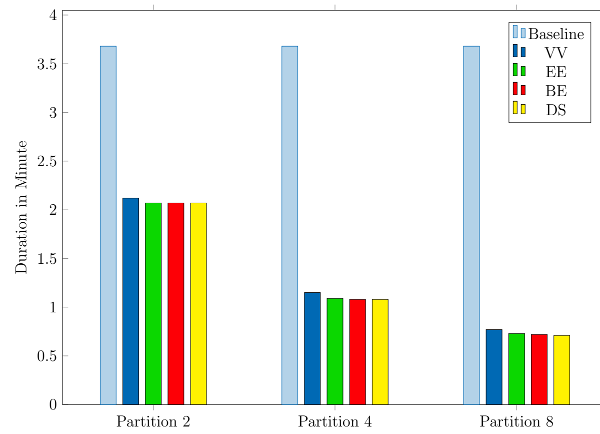
In this section we provide a bit more insights into the results that we observed, namely, the lack of distinctive behaviors from the partitioning strategies (apart from VV).

In the literature there is already an awareness that the performance of load balancing (i.e., the same problem that partitioning strategies need to consider when distributing the edges), and specially dynamic load balancing, cannot be adequately modeled through statistical measures alone ([PGDS+12]). Instead, advanced cost models are needed.

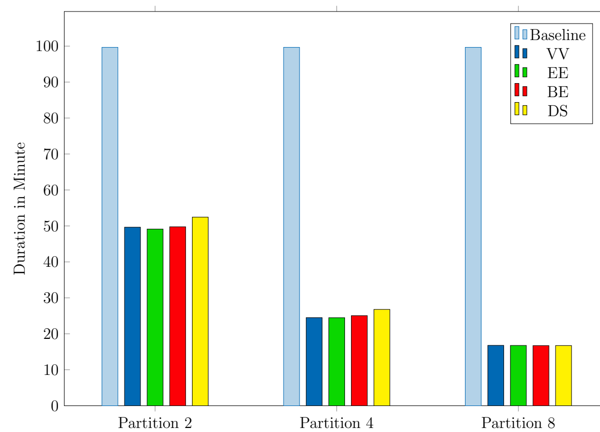
Some measures proposed, but deemed insufficient, include the *Percent Imbalance Metric (PIM)*, λ, as defined below:

$$\lambda = (\frac{L_{\max}}{\bar{L}} - 1) * 100\% \tag{5.2}$$

Here $L_{\max}$ is the maximum load, and $\bar{L}$ is the average load.

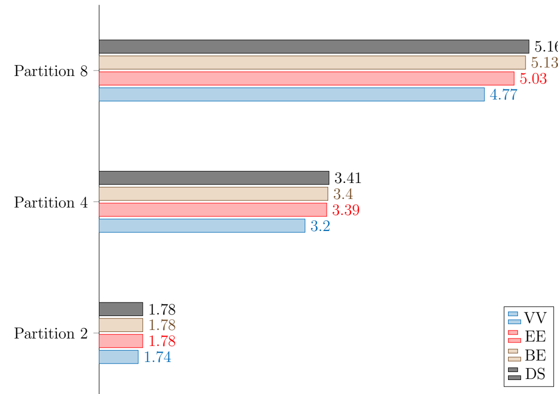


(a) Loading Time using Different Partitioning Strategies (Wiki-RfA)

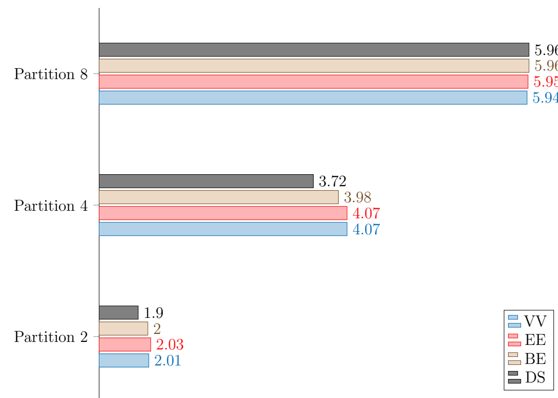


(b) Loading Time Using Different Partitioning Strategies (Web-Google)

Figure 5.4: Loading Time Using Different Partitioning Strategies without Batching

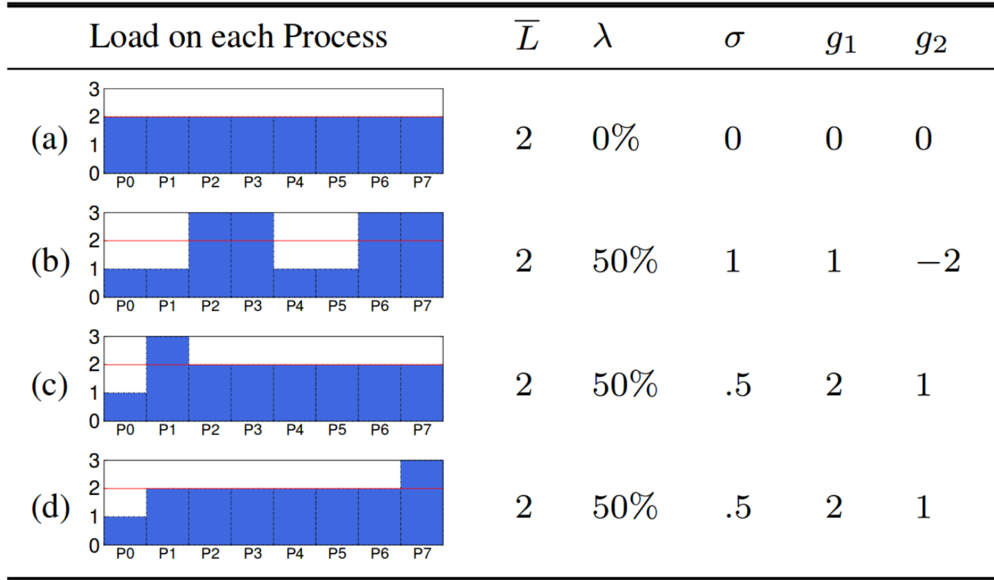


(a) Speedups from Load Using Different Partitioning Strategies (Wiki-RfA)



(b) Speedups from Load Using Different Partitioning Strategies (Web-Google)

Figure 5.5: Speedups from Load Using Different Partitioning Strategies without Batching



	Load on each Process	$\overline{L}$	λ	σ	g_1	g_2
(a)		2	0%	0	0	0
(b)		2	50%	1	1	-2
(c)		2	50%	.5	2	1
(d)		2	50%	.5	2	1

Figure 5.6: Measures to Compare Load distributions [PGDS+12].

Other more general measures include the statistical moments, as defined below:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(L_i - \bar{L})^2} \tag{5.3}$$

$$g1 = \frac{\frac{1}{n}\sum_{i=0}^{n}(L_i - \bar{L})^3}{(\frac{1}{n}\sum_{i=0}^{n}(L_i - \bar{L})^2)^{(3/2)}} \tag{5.4}$$

$$g2 = \frac{\frac{1}{n}\sum_{i=0}^{n}(L_i - \bar{L})^4}{(\frac{1}{n}\sum_{i=0}^{n}(L_i - \bar{L})^2)^2} - 3 \tag{5.5}$$

The last two moments are the skewness and kurtosis, respectively.

Figure 5.6 gives some examples of the limitations of these measures ([PGDS+12]), mainly that they alone are not enough to characterize some cases. For example cases c and d (should order and locality of partitions be distinguishing factors) might not be considered to be the same. In spite of these observed limitations for tasks roughly similar to ours, we judged that these measures were adequate to provide a simple representation of the distribution. In Table 5.1 and Table 5.2 we show how the edges are partitioned for the experiments that we report, corresponding to the Wiki-RfA and Google-Web datasets, respectively.

To interpret these measures, it is necessary to remember the following:

Table 5.1: Number of Edges per Partition and Statistical Measures for Different Partitioning Strategies, Wiki-RfA

Partition Id	VV	EE	BE	DS
2 Partitions				
1	101896	99138	99138	99138
2	96379	99137	99137	99137
Absolute Difference	5517	1	1	1
4 Partitions				
1	48835	49569	49569	49569
2	45775	49569	49569	49569
3	53061	49569	49569	49569
4	50604	49568	49568	49568
PIM	7.04	0	0	0
Skewness	-0.26	-2	-2	-2
Kurtosis	-0.02	4	4	4
8 Partitions				
1	24945	24785	24785	24785
2	23729	24785	24785	24785
3	25566	24785	24785	24785
4	22952	24784	24784	24784
5	23890	24784	24784	24784
6	22046	24784	24784	24784
7	27495	24784	24784	24784
8	27652	24784	24784	24784
PIM	11.57	0	0	0
Skewness	0.35	0.64	0.64	0.64
Kurtosis	-1.07	-2.24	-2.24	-2.24

Table 5.2: Number of Edges per Partition and Statistical Measures for Different Partitioning Strategies (Google-Web)

Partition Id	VV	EE	BE	DS
2 Partitions				
1	2552753	2552520	2552520	2552520
2	2552286	2552519	2552519	2552519
Absolute Difference	467	1	1	1
4 Partitions				
1	1274053	1276260	1276260	1276260
2	1277953	1276260	1276260	1276260
3	1278700	1276260	1276260	1276260
4	1274333	1276259	1276259	1276259
PIM	0.2	0	0	0
Skewness	0.07	-2	-2	-2
Kurtosis	-5.4	4	4	4
8 Partitions				
1	638983	638130	638130	638130
2	637731	638130	638130	638130
3	640390	638130	638130	638130
4	637399	638130	638130	638130
5	635070	638130	638130	638130
6	640222	638130	638130	638130
7	638310	638130	638130	638130
8	636934	638129	638129	638129
PIM	0.35	0	0	0
Skewness	-0.33	-2.83	-2.83	-2.83
Kurtosis	-0.07	8	8	8

- PIM represents the severity of the load imbalance.
- Regarding skewness, positive skewness means that a relatively low number of partitions has a higher than average load, negative skew means that a relatively low number of partitions has a lower than average load. A normal distribution would have a skew of 0.
- Regarding kurtosis, high values indicate that more of the variance is originated from outlier values (i.e., partitions with very different numbers than average), while lower kurtosis values correspond to relatively frequent but modestly-sized variations.

The first observation that we can make is that, for all the cases that we evaluate, the performance of EE, BE and DS, at least in terms of the generated partitions (and not considering the time to calculate the partition itself) is the same. This explains the similar speedups observed for these strategies. It also highlights that small differences in behavior, like for DS, are most likely due to measurement errors or overheads, but not due to algorithmic differences. In general these strategies achieve a better performance, with a consistent PIM value of zero indicating no imbalance, relatively high kurtosis signaling that the variance observed is due to a small number of outliers, and finally negative skewness meaning that some few partitions has a lower number than average. We expect that with increasing number of partitions, on similar datasets, we should see more negative skewness, larger kurtosis and a zero PIM value for these strategies. Given the equivalence of them over the selected datasets, we would suggest that EE would be the best choice among them, due to it's simplicity.

VV consistently displays some imbalance. This is most notable when using 2 partitions and the Wiki-RfA dataset (with an imbalance that accounts for roughly 5% of the largest partition). For other cases the effect is not that large, and is reduced with increasing number of partitions. This explains the effect that we saw in the average response time: with increasing number of partitions, the gap between VV and the other strategies decreased, and it was more marked for 2 partitions than for any other case.

With this we conclude our study on partitioning without batching. In the next section we summarize the best practices we can infer from our observations.

5.5 Best Practices

Following our evaluations, we can propose a list of best practices regarding the use of partitioning and parallel loading. These are the direct answer that we give to the evaluation questions in this chapter Section 5.1.

1. Using partitioning and parallel loading approaches bring beneficial improvements to the response times. Parallelizing the loading processes becomes a consistently good choice to load graph data into JanusGraph.

2. When we want to load data concurrently by partitioning, it is not always the case that duplicating the number of partitions in a single processor will provide double times acceleration. Finding a suitable number of partitions becomes the first step. This should be adequate, first, to the computing framework. From our observation the number of working treads between 4 and 8 is a good choice for datasets with comparable sizes to the datasets tested in this work. The performance impact of the number or working treads is related to the whole loading time. The shorter the loading time, the less the performance impact.

3. Basic Partitioning strategies seem to have little impact on time performance. In our evaluation we find that load imbalance does not play a large role in the resulting runtime. This, of course, is a consequence of the datasets that we used in our evaluation, which did not give occasion to large imbalances. Based on this we cannot rule out that some special strategies will significantly reduce loading time for datasets with very different characteristics (i.e., a group of outliers with very highly-connected vertexes). We suggest developers to put partitioning strategies on the second position when considering optimizations, and to use the EE strategy by default. Many alternative partitioning strategies exist than the ones we studied. There might not be a "one-size-fits-all" strategy, but at least developers should mind that strategies should come second to finding the right number of partitions, which depends on the compute resources. It takes time to find a "perfect" match.

4. Adding a publisher/subscriber framework eases the parallel working process and can make the complete loading task scale to more compute power. Using such frameworks help developer to care less about the network interactions. In our prototypical implementation we show an example of how this could be done. In our evaluation we only show gains for local executions, distributed cases are possible with our implementations, but we decided not to test them.

5.6 Summary

This chapter presented our evaluation on the performance of loading using partitioning and parallel strategies in JanusGraph. Instead of loading sequentially object by object (edge by edge), this chapter aimed to provide experimental insights on parallelizing the loading process, such that different processes could load in parallel several data partitions, using a publisher/subscriber framework (from Apache Kafka).

The next chapter is concerned with combined optimization alternatives: using batching, partitioning and parallel loading approaches together. The following chapter will conclude our evaluations in this study.

6. Case Study: Combination of Batching, Partitioning and Parallelizing

In this chapter we survey some opportunities that arise from the combination of batching, partitioning and parallelizing the loading process. In previous chapters we've evaluated some benefits and limitations of these approaches individually, here we expand the functionality under consideration, by combining them. We implement a mixed data loader in our experiment prototype JanusGraphLab. In this data loader we use both batching and partitioning-parallel loading.

We outline this chapter as follows:

- **Evaluation Questions:**
  We establish the evaluation questions that motivate this chapter. (Section 6.1)
- **Microbenchmarks:**
  We answer the evaluation questions regarding experimental analysis and results. (Section 6.2)
- **Best Practices:**
  We collect the findings of this chapter in a list of best practices. (Section 6.3)
- **Summary:**
  To conclude we summarize the work in this chapter. (Section 6.4)

6.1 Evaluation Questions

6. In deciding for batching, parallelization, which factors can be determined statically and which are dependent on changing topologies? For the topology dependent

factors: what is the tipping point for making other decisions? Can these optimizations be implemented in an adaptive manner; If so, how to model the optimization function for real-world cases (where network latency and different replication strategies could also affect the performance)?

6.2 Microbenchmarks

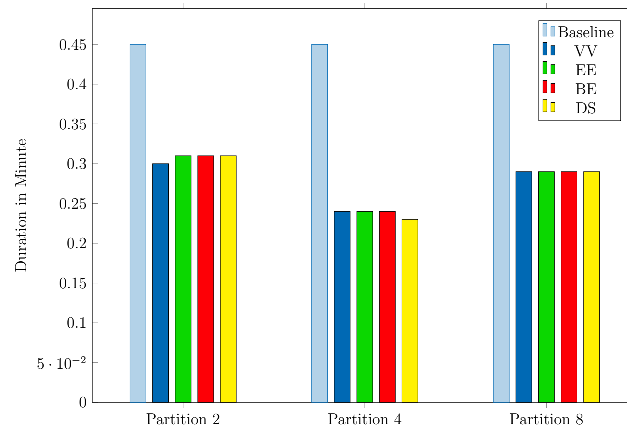
For our evaluation, we compare the impact of partitioning strategies and parallelization on loading performance under different batch sizes (10,100,1000). As baseline we use a local execution with the same batch size. We do not report results for 8 partitions with Google-Web since we found large variations in the measurements, which we believe were produced by congestion from sending a large amount of heavy messages with Kafka.

6.2.1 Batch size = 10

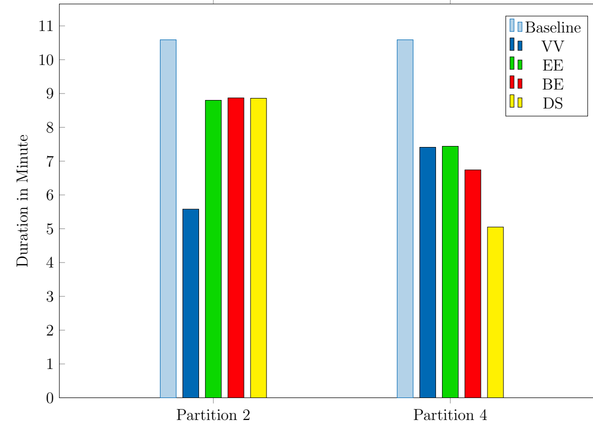
Figure 6.1 records the loading time using partitioning strategies when the batch size equals 10. We can observe that, parallelization reduces the loading time in general, when compared to the baseline. But when the number of working treads (partitions) increase to some degree, it stops to reduce the loading time, even increasing it. We can see from 6.1(a) that when the number of partitions grows to 8, the loading time is longer than the loading time when the partitions were 4. The working principle of parallelization in a single processor, is that different processes share resources, introducing overheads, from scheduling and context switching, apart from communication costs. When the loading time is short enough (as occurs when batch sizes grow, which we proved experimentally for Wiki-RfA in Chapter 4), more partitions could lead to degraded performance, making the process more affected by overheads than benefiting from the parallelization.

The small performance gains for VV with two partitions is unexpected, since this could have had a slightly worse performance due to imbalances. One explanation might be that transaction commits for distributed transactions could have an effect, thus this strategy, which keeps edges close to the connected vertexes, could fare better. However to validate this claim more careful profiles are necessary.

Using Google-Web we also observe that partitioning consistently introduces improvements over the baseline. For different strategies we observe that each of them has a different impact on the loading performance. For example 6.1(b) we can observe that the loading time using VV strategy when the number of partitions equals 2 is the lowest. In fact, this is one of the most interesting observations from our study here. This case is not due to imbalances, since, as discussed in Section 5.4.2, the only significant imbalance appears for VV with the Wiki-RfA dataset. We speculate once again that the difference observed (specifically between VV and the others) is related to transaction processing and lock management, but we did not evaluate these aspects. As for the differences between EE, DS and BE, since they produce the same partitions we can only propose that these could be explained by system aspects, such as communication overheads.

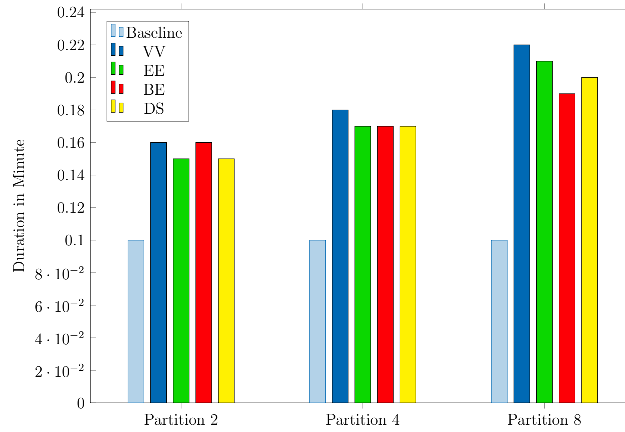


(a) Loading Time Using Different Partitioning Strategies When Batch Size Equals 10 (Wiki-RfA)

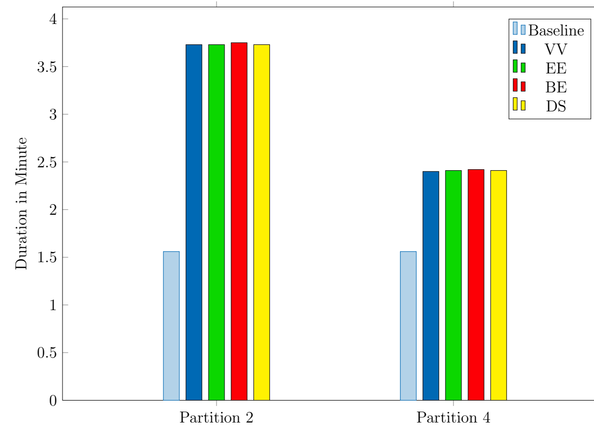


(b) Loading Time Using Different Partitioning Strategies When Batch Size Equals 10 (Web-Google)

Figure 6.1: Loading Time Using Different Partitioning Strategies with Batch size = 10



(a) Loading Time Using Different Partitioning Strategies When Batch Size Equals 100 (Wiki-RfA)



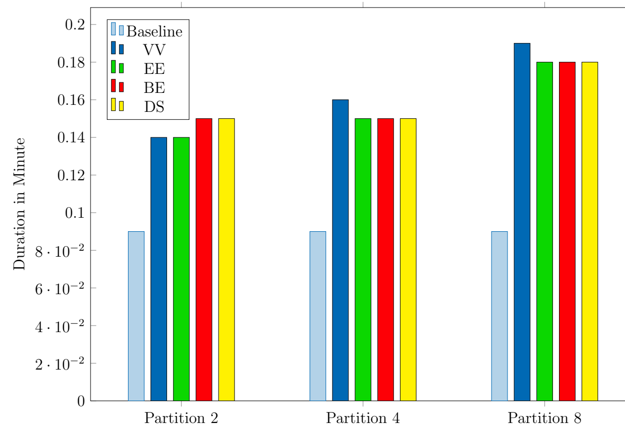
(b) Loading Time Using Different Partitioning Strategies When Batch Size Equals 100 (Web-Google)

Figure 6.2: Loading Time Using Different Partitioning Strategies with Batch Size = 100

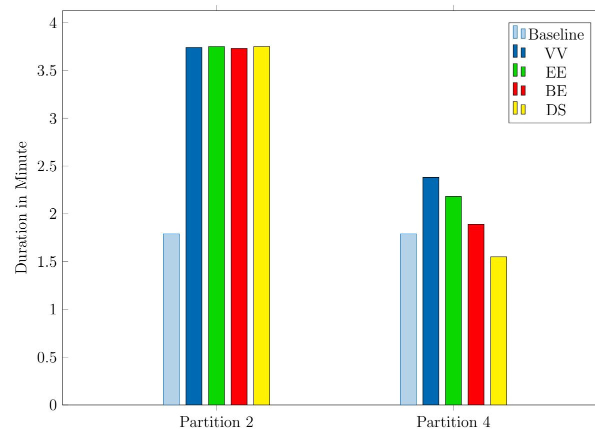
To summarize the results for this batch size, we observe that for small batch sizes partitioning still brings gains and, given the smaller task and the fact that it introduces distributed commits over more items, it appears to blur a bit the imbalance differences between VV and the rest of the strategies, making VV a better choice on some cases.

6.2.2 Batch size = 100

Figure 6.2 records the loading time using partitioning strategies and parallelization approach when batch size equals 100. In all of our previous tests, partitioning and parallelization had a good impact on the loading performance. But in 6.2(a) we observe that all results are worst than the baseline. Also for the Wiki-RfA dataset, with the reduced size of the task, with the increase of partition numbers, the loading time increases. This strengthens our previous conjecture, that overheads rule out gains when



(a) Loading Time Using Different Partitioning Strategies When Batch Size Equals 1000 (Wiki-RfA)



(b) Loading Time Using Different Partitioning Strategies When Batch Size Equals 1000 (Web-Google)

Figure 6.3: Loading Time Using Different Partitioning Strategies with Batch Size = 1000

the loading time is short. When the general loading time is short enough, more partitions can lead to degraded performance.

In this case some small differences in the strategies appeared, but since VV did not behave particularly different and the rest are equal in partitions, we did not consider these differences to be specially informative of underlying problems and we deem that they might be due to overheads external to the strategies themselves.

For the Google-Web dataset, we observe that more partitions can still lower the time. Unfortunately this still does not outperform the baseline.

6.2.3 Batch size = 1000

Figure 6.3 presents the times of partitioning and parallel loading with batch size equals 1000. As expected, from the trend that we report in this chapter, when the loading time

is much shorter, partitioning and parallelization degrade the performance rather than improve it. We see this for the Wiki-RfA dataset. Furthermore, the loading performance remains different with different partitioning strategies, which we speculate is due to either transaction commit or communication costs and not to imbalances.

Finally, there is a performance gain for Google-Web, when we move the partitioning strategies from 2 to 4 partitions.

To summarize the results: The combination of partitioning and batching was only better than the baseline for a batch size of 10, with maximum performance gains of 2x and 1.5x over all strategies for both datasets in 4 and 8 partitions respectively. Thus, the gains are sublinear. For other batch sizes, there were no improvements over the baseline. We believe that the core factor leading to this situation is that the overheads for message passing dominate the performance when the batch sizes are larger (i.e., when the tasks to perform are few). This argument is also sustained with the observation that, when not comparing against the baseline, more partitions consistently improve the performance for Google-Web, no matter the batch size, as opposed to Wiki-RfA (where the task is shorter).

Regarding the differences in strategies we report one interesting case: VV for Google-Web with 2 partitions, which outperforms all cases. From our studies we know that this gain does not come from a better load balance, instead we speculate that it might be due to a good reduction in transaction commit overheads for distributed transactions, produced by the fact that the strategy assigns to a partition with a given vertex all the edges that connect to it. However further studies are needed to understand better if this is the case.

For all other cases we observe mixed results regarding the strategies, and there is no clear sense of one being better than others.

With these observations we conclude the discussion of the experiments in this chapter. Next we summarize our findings in a series of best practices.

6.3 Best Practices

Following the tests in this chapter we are prepared to define some best practices regarding the combination of batching, partitioning and parallelization. These best practices are a direct answer to the questions that spawned our evaluations for this chapter.

1. Batch loading is the best choice for loading graph data into JanusGraph. The combination of batching, partitioning and parallelization can actually lead to degraded performance, when the loading time is relative short (as happens with batch sizes larger than 10). Subtle transaction commit overheads might also be at play, at least in distinguishing VV from the other strategies, but we did not measure this. In any case, developers should select batching first, and then consider if adding partitioning can help

or not. Further studies considering distributed cases could be of interest to see if our insights need to be revised for that different setting.

2. When combining batching and partitioning, there is no clear winner for a strategy to pick. Without using batching we can recommend to start with EE, but when using batching there is no clear starting point and studies would be needed. Nonetheless the consideration should be with VV against others, with distinctions between EE, BE and DS only being considered if they produce different partitions.

3. We performed our study using datasets from 2 different domains, in the expectation to evaluate if the differences in topology (i.e., in how nodes are connected) have an impact in loading optimizations. In our limited study we found that topologies do not seem to play the biggest role, and issues like the amount of data to be loaded are more impactful. An adaptive process (or at least a process that adapts to data characteristics) might still be needed due to subtle differences in performance when combining the optimizations. For selecting among them, batch sizes would be the foremost choice, followed by a careful combination with partitioning, with a partition number fit to the computing resources and then partitioning strategies adapted to the use case. More specifically, we advice developers to choose batch loading approaches at first and then create several performance tests using samples from the dataset. By evaluating the loading time generated in these performance tests they could made informed choices on batch sizes, number of partitions and partitioning strategies (also the choice would be VV against others, unless the others generate different partitions).

6.4 Summary

In this chapter we document the experimental results of a simple microbenchmark to evaluate the combination of loading optimization alternatives: batching, partitioning and parallelization, unlike other chapters in this study where we have focused on previous mentioned optimizations in isolation. With this chapter we conclude our evaluation on some core issues that programmers must tackle for developing graph applications with JanusGraph. In the next chapter we wrap-up our study by summarizing the best practices derived from our findings in Chapter 4, Chapter 5 and Chapter 6. We also propose future work to advance the research program we conducted in this project.

In the next chapter we conclude our work by summarizing the best practices that we found, giving threats to validity to our evaluations and proposing future work.

.

7. Conclusion and Future Work

The goal for this study was to evaluate several optimizations and design choices for scalable data loading into graph databases. With the aim to evaluate the optimized loading alternatives we developed an experimental prototype that can already be used to load data in a scalable and configurable manner into a graph database. In fact, research ([DJP+18]).

For our implementation we selected an open-source distributed graph database: **JanusGraph**. To carry out our evaluations we developed JanusGraphLab, a prototypical application that uses JanusGraph as an embedded graph database. Among the optimized application design choices we mainly researched traditional and a batch/bulk loading process in JanusGraph. Furthermore we considered adding a Provider/Consumer framework to implement a parallel loading process with small partitioned edge lists. Lastly, we researched (included as an Appendix), how edge ordering influences the loading performance. What are the strength and limitations? In order to complete our evaluations, we use JanusGraphLab to collect experimental confirmation on the influence of JanusGraph clusters on the loading process. All of our experiments use real-world datasets with different topologies.

In this chapter we conclude our study, as follows:

- **Summary: Best Practices for Loading Scalable Data into Graph Databases**
  We summarize the best practices that we infer from our experimental studies. (Section 7.1)
- **Threats to Validity**
  We disclose some threats that may influence the validity of our evaluations. (Section 7.2)

- **Concluding Remarks**
  Briefly explain this research. (Section 7.3)
- **Future Work**
  We complete this project by proposing future work to expand our learning experience. (Section 7.4)

7.1 Summary: Best Practices for Loading Scalable Data into Graph Databases

1. *Server-side loading is required:* Writing the loading logic in server side is much more efficient and performs better than writing the logic on the client side. Through this, less requests are sent and temporary data structures can be managed better. Using groovy scripts to send to JanusGraph is one useful alternative for doing batch requests to create items, however similar functionality can be achieved using a native language client.

2. *Batch/Bulk loading is the best choice for loading graph data into JanusGraph:* In contrast to loading objects (vertexes and edges) one by one, uploading a group of objects (vertexes or edges) per transaction saves the overall loading time. The choice of batch size depends on the size of the dataset. In our evaluation the best performance improvement is 64.05x faster (100 minutes to close to 1.5 minutes) when batch size equals 100. Furthermore we suggest that batching should be a choice considered before others, due to it's simplicity. However there are limits to this approach (too big batches might not have the failure tolerance we expect from database interactions), and performance gain do not grow in proportion to batch sizes.

3. *Parallelization approaches are useful and are the key for scalable loading:* For this use case, we implement concurrent loading processes using the publisher/subscriber framework Apache Kafka, which makes several uploading threads work on its own port in parallel. Though this made our prototype scalable, we did not evaluate in a distributed setting. From our study, the benefits of parallelization grow with increasing number of partitions. Concurrency can reach an upper limit for increased loading performance, which is also reasonably related with the amount of processing resources available. In our evaluation we observed that parallelization, when not combined with batching, can lead to best speedups of 5.96 when the partition number equals 8.

4. *Usage of Publisher/Subscriber framework can ease the concurrent loading process:* We used a Publisher/Subscriber framework from Apache Kafka to adopt our parallelization approach for concurrent loading processes. Experimental results shows the performance improvement from the parallelization approach. During our experiments we encountered some unknown problems from Apache Kafka. These need more studies.

5. *It might be possible to decide on one size fits all partitioning strategy without batching:* Apart from VV, which leads to imbalance, specially visible on small datasets and small number of partitions, there is no clear evidence for partitioning strategies leading to markedly different performance when not using batching over the datasets that we evaluated. Therefore we suggest that EE could be used by default, because of it's simplicity and the fact that it produces balanced partitions. However we cannot rule out that some special strategies will significantly reduce loading time for datasets with very different characteristics (i.e., a group of outliers with very highly-connected vertexes). We suggest developers to put partitioning strategies on the second position when considering optimizations, and to use the EE strategy by default.

6. *The combination of optimization alternatives: batching, partitioning, parallelization should be chosen properly, after loading tests:*
   We have observed that using more optimization technologies does not necessarily translate into more performance improvements. On several cases, the combination can worsen the performance, on others, it can bring benefits.In our evaluation tests, when batch size equals 1000, more use of partitioning and parallelization strategies can only reduce the loading efficiency. Taken together, batching becomes the best optimization factor. It's easy to use for loading graph data in a sequential and local manner, but for distributed loading scenarios parallelization is needed and then it has to be combined in the best way possible with batching. In our tests we haven't covered a distributed scenario. We will describe it in the future work section. Apart from selecting batch sizes, parallelization always depends on partitioning strategies.
   The combination of the best partitioning strategy and batch sizes is challenging and requires consideration from the developer. In our studies we did not find a clear winner, and we also find that load imbalance is not the single factor determining performance (in fact, EE, BE, DS all produce the same number of partitions in our study). We speculate that communication and transaction commit costs might be a part of the performance determining factors which we saw made the strategies lead to different results, specifically VV vs. the others. Therefore we strongly recommend that a best practice is to test different strategies and batch sizes over small data samples in order to determine the best configuration. A careful look at the partition sizes, to determine if the strategies lead to the same results is useful in this step. Once the best configuration is found, it could be used for the complete dataset.

7.2 Threats to Validity

In this section we disclose some items that we believe may threaten the validity of our evaluations.

- External threats:
  Datasets, technologies, hardware, configuration and implementation choices might

have affected in some ways the capacity of our results to be replicated in other cases. The loading time of graph data we recorded in our experiments is also related to the network status to some extent, specifically with port connections to the backends and processes. To limit this threat we report tests for extensive repetitions. Future studies should also consider other datasets, and specifically more synthetic datasets such as to create challenges that evaluate the effect of partitioning strategies.

- *High-level optimization only:*
  Our optimization approaches are basically inspired from application-level aspects. From the point of view of developers, we propose some effective strategies for optimization. We haven't considered much about the low-level implementation details, such as choices of optimized in-memory data structures, memory management or SIMD acceleration. We ignore impact from low-level factors, such as network speed, CPU, virtualization overheads and I/O performance.

- *Real-world dataset only:*
  Researchers usually provide experiments both using real-world datasets and specific system-generated datasets, such as LDBC[ldbb]. Real world data is more authentic, but sometimes system-generated datasets are needed to verify the effectiveness of certain features. As mentioned already in this list, arguably synthetic datasets might be more useful for the optimization choices we study.

- **Limitation in product versions:**
  The version of JanusGraph we used in our experiments may not be representative of the database in general. Some functionality may still be improved by the community. Furthermore, we used Version 2.1.11 of Cassandra, which doesn't include some newest storage improvement solutions.

- **Problem caused by using Apache Kafka:** In our studies we found some cases when Apache Kafka would randomly stay in an infinite loop, or take a long time to pass messages. We made sure that in the tests that we report, this phenomena did not occur.

7.3 Concluding Remarks

Usually loading the data is only considered as a first step before analysis can proceed. In this thesis we considered that loading in itself was an interesting area for analysis, and as a result we focused our work in considering several choices available to developers for improving the loading performance of graph database applications. Instead of comparing with optimization alternatives for different graph databases (such as Neo4j etc.), we narrowed down our scope to a specific graph database: **JanusGraph** and aimed to study the impact of optimizations.

In order to study these alternatives, we propose a series of research questions about the process of optimized graph data loading. While this may be a very limited focus and few people have studied it, the graph data loading process is considered a performance bottleneck for interacting with graph databases, so we think that improving this process can have a significant impact on the performance of applications. There are mainly three optimization alternatives we researched: **batching**, **Partitioning**, and **parallelization**.

To address the evaluation questions, we created **JanusGraphLab**, a prototype of a graph database application that uses JanusGraph as an embedded database and supports Elastic-search and Cassandra as its indexing and storage backends. Within JanusGraphLab, we implemented different functionalities to execute carefully designed microbenchmarks to answer our evaluation questions.

The findings of our experiments using these microbenchmarks are included in Chapter 4, Chapter 5 and Chapter 6. From these findings we concluded a set of best practices for developers, summarized in Section 7.1.

The results of our work may have practical direct relevance to the application developers who use JanusGraph. The recommended best practices for other graph databases may not be directly transferred. However, our proposed core evaluation questions and microbenchmarks can be an effective study approach to determine best practices for other graph databases.

Regarding our experiment results, we have found that our optimization approaches can reduce loading time from hours to minutes. Our study provides developers some suggestions for improving the performance of the loading process, we also offer a tool developed for JanusGraph, which seems to be the first of its kind in enabling scalable data loading with configurable options for speeding up the process and for having ad-hoc schemas. We seek to make our tool publicly available.

In some futuristic movies the loading of large and complex data is portrayed as an instantaneous and effortless task. With our current study and the first version of our tool we aim to contribute towards making such visions a reality for graph databases, helping the end users.

7.4 Future Work

Ours was an early study into graph data loading, carried out within the limited time-frame of a Thesis project. Future work could cover several areas that we consider are interesting:

- **Evaluation of further functionality and configuration:**
In our experiments we only considered the basic data loading function. We suggest that in future work, research on data loading in more complex scenarios, such as data loading in distributed systems or loading datasets with tightly controlled conditions. Other desirable tasks in the microbenchmarks may be the data integration process and to combine loading with query processing. In addition, our research significantly lacks aspects of database configuration (such as the use of database caching in transactions). In fact, during the time of this project we also performed evaluations on the impact of configuration parameters like late materialization and database cache usage, on the loading process. Since exploring this configuration space did produce performance gains comparable to the optimizations we cover in this work, we left these tasks for future work.

- **Microbenchmarking other databases:**
JanusGraphLab can be extended to be a more platform-agnostic prototype. In this way, it can be used for other graph databases to find application-level optimizations.

- **Research on loading scenarios with more nodes/clusters:** In our microbenchmarks we researched only basic loading processes on a single machine. In future work we could consider more complicated loading scenarios, such as data loading on multi-clusters/nodes, or with implicit entities in the source data. We can take more factors into consideration to find other optimization alternatives.

- **Further research on partitioning strategies:**
Our studies on partitioning strategies might've been limited in the end by our choice of datasets. We believe that using synthetically generated data we might be able to understand better the features of the strategies. In addition, more profiling for understanding the sources for performance difference between strategies when using batching could be considered. A study could consider if edge cuts are correlated or not to performance differences. Finally, other strategies could be studied. Among them stream orders and parametrized heuristics.

- **Research on low-level optimization using modern hardware:**
In the future work it might be worthy to research hardware-level optimization alternatives with modern hardware, for example the use of GPUs. We specifically envision to use GPUs to aid in the creation of batches, carrying out the mapping between ids.

A. The Impact of Edge-Ordering on Load Times with JanusGraph

In this section we study if the sorting of edges, or their arrival order, has an impact on the overall performance. We believed that this could be the case due to how memory is managed. Specifically, since in the internals of most graph databases (incl. JanusGraph), edges are stored as part of the vertexes (supporting the so-called index-free adjacency), it might be possible that their arrival order might have an influence on the vertexes that are kept in the cache, and, as a consequence we speculated that certain orders might have a better cache usage and thus be faster than others. In this chapter we include some of our studies regarding this aspect.

A.1 Edge Loading with Different Sort Strategies

In this section we study how edge order might influence the edge loading performance. We begin by introducing several sort strategies we researched, next we present our test results.

A.1.0.1 Sort Strategies

Basic Sort Strategies

We tested response time for loading wiki-RfA data using different basic sort strategies. We also created comparative test (load without sort strategies) and call it "Baseline". Now we list some very basic sort strategies that we used in the tests here:

- **Random:** The entire edge list was shuffled randomly.
- **SrcName:** The edge list was alphabetically sorted by source name of the edge. This meant that all edges with a given vertex SrcName will be loaded sequentially.

	Baseline	**Random**	**SrcName**	**SrcName+Alt Tgt Name**	**Reverse SrcName**	**Reverse SrcName +Alt Tgt Name**
Average Load Time (ms)	508165.74	498857.28	498119	496592	498152.76	499681.04
Speedups		1.02	1.02	1.02	1.02	1.02

Table A.1: Average Response Time for Loading Edge Data with Different Sorts (Wiki-RfA)

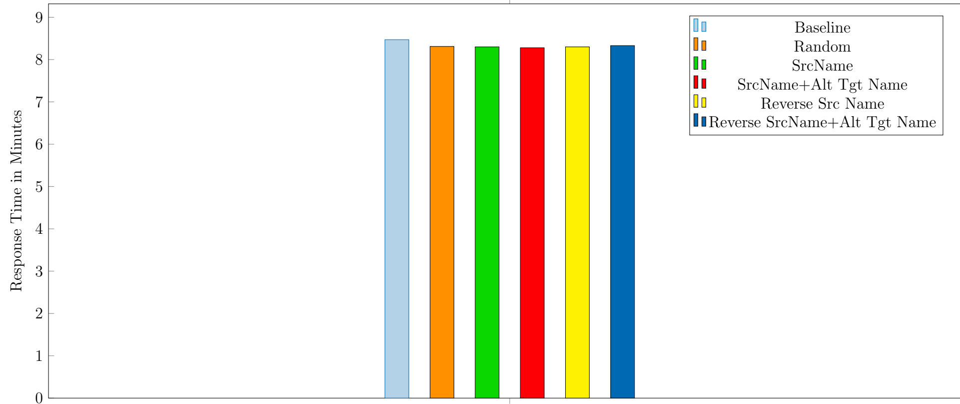


Figure A.1: Average Response Time for Loading Edge Data with Different Sorts (Wiki-RfA)

- **SrcName+Alt Tgt Name:** The edge list was alphabetically sorted by source name of the edge. And then sorted by target name of the edge. Through this, we speculated that some cache efficiency might be achieved for the non-source edges.
- **Reverse SrcName:** The edge list was alphabetically sorted by source name of the edge and reversed. This was done for comparing with the second strategy.
- **Reverse SrcName+Alt Tgt Name:** The edge list was alphabetically sorted by source name of the edge. And sorted by target name of the edge. At last the edge list is reversed. This was done for comparing with the third strategy.

Table A.1 shows the experiment results. We repeated load process with each sort strategies 10 times to ensure the accuracy of experimental results. Clearly we can observe from Figure A.1 that, there is no significant improvement of load performance using different sorts. The speedup of using different sort strategies compared with “Baseline” is 1.02x.

Furthermore we considered that the basic sort strategies are not targeted and not specific for graph usage and we did some work on a slightly more complicated fair-sorting method.

To evaluate the possible impact of sorts on a cache we developed an analytical model. Next we will introduce the **LRU Cache Model** below, which we created to evaluate

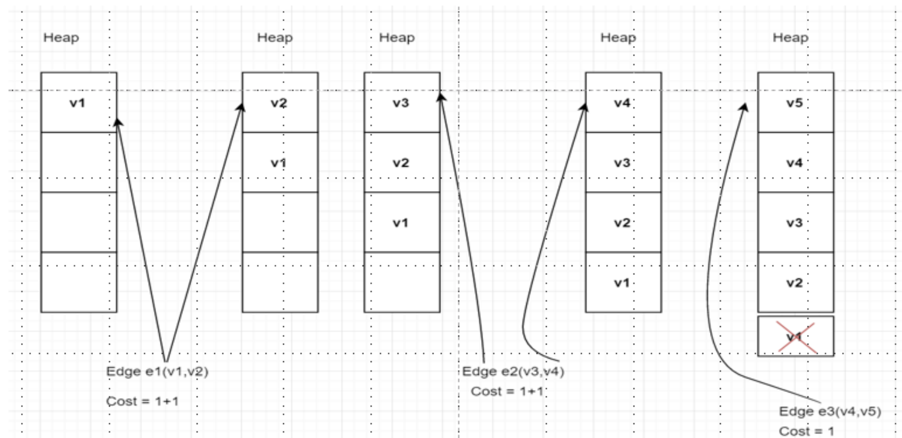


Figure A.2: LRU Cache Model Used to Estimate the Cost of A Sorted Load

the sorting of edges.

LRU Cache Model

Beyond the basic sort strategies, our goal is to find a model to evaluate analytically a sort strategy in order to determine if it could improve the performance, and then to assess experimentally if the model is valid. Normally the data loading process works under limited main memory. Since memory is generally not large enough to hold the whole data, approaches like LRU caching are used to manage the memory. For our model on sorts, we proposed a cost model to simulate the usage of the cache. Following an LRU cache model (See Figure A.2) we calculated the costs of a sorted load by simply counting the number of cache misses.

Surprisingly, we have found that the response time used for loading the edges with different sorts did not match the predictions for LRU cache models with different cache sizes and configurations; thus we concluded that the LRU model was not a proper model to explain the observations when using JanusGraph.

Future work using the same database could repeat these studies, perhaps in more memory restricted environments and with more profiling information. However, based on the simple results that we report, we do not believe that edge arrival order has an influence on the overall performance when using JanusGraph. Instead we believe, and show experimentally, that other factors (such as batching and partitioning) might play larger roles in determining the run time. The same statements cannot be mapped by default to other databases, thus we suggest that tests like the ones described in this chapter could be useful.

Bibliography

[AAF+17] Renzo Angles, Marcelo Arenas, George HL Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, Hannes Voigt, et al. G-core: A core for future graph query languages. *arXiv preprint arXiv:1712.01550*, 2017. (cited on Page 19 and 21)

[AAS] Azza Abouzied, Daniel J. Abadi, and Avi Silberschatz. Invisible loading: Access-driven data transfer from raw files into database systems. (cited on Page 30)

[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. Dbmss on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. (cited on Page 3)

[AG08] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, February 2008. (cited on Page xiii and 14)

[AG17] Renzo Angles and Claudio Gutierrez. An introduction to graph data management. *arXiv preprint arXiv:1801.00036*, 2017. (cited on Page 9 and 13)

[ALT+17] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):20, 2017. (cited on Page 16)

[Ang12] R. Angles. A comparison of current graph database models. In *2012 IEEE 28th International Conference on Data Engineering Workshops*, pages 171–177, April 2012. (cited on Page 20)

[Bar13] Chaitanya Baru. Benchmarking big data systems and the bigdata top100 list. *Big Data*, 1:60–64, February 2013. (cited on Page 29)

[BMS+16] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016. (cited on Page 34)

[CF12] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: laws, tools, and case studies. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 7(1):1–207, 2012. (cited on Page 53)

[Che76] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976. (cited on Page 15)

[CR14] Yu Cheng and Florin Rusu. Parallel in-situ data processing with speculative loading. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1287–1298, New York, NY, USA, 2014. ACM. (cited on Page 29)

[CY15] Rubén Casado and Muhammad Younas. Emerging trends and technologies in big data processing. *Concurrency and Computation: Practice and Experience*, 27(8):2078–2091, 2015. CPE-14-0259. (cited on Page 47)

[DJP+18] Gabriel Campero Durand, Anusha Janardhana, Marcus Pinnecke, Yusra Shakeel, Jacob Krüger, Thomas Leich, and Gunter Saake. Exploring large scholarly networks with hermes (forthcoming). In *International Conference on Extending Database Technology*. ACM, 2018. (cited on Page 89)

[DKA+17] Adam Dziedzic, Manos Karpathiotakis, Ioannis Alagiannis, Raja Appuswamy, and Anastasia Ailamaki. *DBMS Data Loading: An Analysis on Modern Hardware*, pages 95–117. Springer International Publishing, Cham, 2017. (cited on Page 29 and 34)

[Dur17] Gabriel Campero Durand. Best practices for developing graph database applications: A case study using apache titan. 2017. (cited on Page 43 and 59)

[FVY17] George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. Declarative graph querying in practice and theory. In *EDBT*, 2017. (cited on Page 8)

[GK10] Goetz Graefe and Harumi Kuno. *Fast Loads and Queries*, pages 31–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. (cited on Page 33)

[goo] Google web graph. https://snap.stanford.edu/data/web-Google.html. Accessed February 9, 2018. (cited on Page 53)

[HKJ+17] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. Empowering in-memory relational database engines with native graph processing. *CoRR*, abs/1709.06715, 2017. (cited on Page 15, 16, and 17)

[IC03] Jonathan G. Geiger Imhoff Claudia, Nicholas Galemmo. *Mastering Data Warehouse Design : Relational and Dimensional Techniques*. Wiley, 2003. (cited on Page 29)

[IR15] Emil Eifrem Ian Robinson, Jim Webber. Graph databases, 2nd edition new opportunities for connected data. chapter 6. O'Reilly Media, 2015. (cited on Page xiii, 21, and 22)

[JMCH15] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using vertica relational database. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1191–1200, Oct 2015. (cited on Page 16)

[JPNR17] Martin Junghanns, Andre Petermann, Martin Neumann, and Erhard Rahm. *Management and Analysis of Big Graph Data: Current Systems and Open Challenges*, pages 457–505. Springer International Publishing, Cham, 2017. (cited on Page 13, 17, 18, 19, 20, and 21)

[JRW+14] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. Vertexica: Your relational friend for graph analytics! *Proc. VLDB Endow.*, 7(13):1669–1672, August 2014. (cited on Page 16)

[KALB12] Gerald Kane, Maryam Alavi, Giuseppe Labianca, and Steve Borgatti. What's different about social media networks? a framework and research agenda. 2012. (cited on Page 52)

[KG14] Aapo Kyrola and Carlos Guestrin. Graphchi-db: Simple design for a scalable graph database system - on just a PC. *CoRR*, abs/1403.0701, 2014. (cited on Page 40)

[KK95] George Karypis and Vipin Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995. (cited on Page 34)

[KKH+17] Alexander Krause, Thomas Kissinger, Dirk Habich, Hannes Voigt, and Wolfgang Lehner. Partitioning strategy selection for in-memory graph pattern matching on multiprocessor systems. In *23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain*, August 2017. (cited on Page 35)

[Kle17] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. " O'Reilly Media, Inc.", 2017. (cited on Page 24 and 28)

[LDBa] Ldbc social network benchmark (snb) – 0.3.0. https://github.com/ldbc/ldbc_snb_docs. Accessed February 9, 2018. (cited on Page 1)

[ldbb] Ldbcouncil. http://www.ldbcouncil.org/. Accessed February 9, 2018. (cited on Page 92)

[LMPS16] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. Fast in-memory sql analytics on typed graphs. *Proc. VLDB Endow.*, 10(3):265–276, nov 2016. (cited on Page 17)

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. (cited on Page 27)

[MSX+16] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. G-sql: Fast query processing via graph exploration. *Proc. VLDB Endow*, 9(12):900–911, August 2016. (cited on Page 16)

[Mue13] Tobias Muehlbauer. Instant loading for main memory databases. *Proc. VLDB Endow.*, 6(14):1702–1713, September 2013. (cited on Page 3 and 29)

[PGDS+12] Olga Pearce, Todd Gamblin, Bronis R De Supinski, Martin Schulz, and Nancy M Amato. Quantifying the effectiveness of load balance algorithms. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 185–194. ACM, 2012. (cited on Page xiv, 72, and 75)

[PPR+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM. (cited on Page 30)

[PSB+15] Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1105–1110. ACM, 2015. (cited on Page 16)

[PV17] Marcus Paradies and Hannes Voigt. Big graph data analytics on single machines–an overview. *Datenbank-Spektrum*, 17(2):101–112, 2017. (cited on Page 19 and 21)

[PZLz17] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer zsu. Do we need specialized graph databases?: Benchmarking real-time social networking applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, GRADES'17, pages 12:1–12:7, New York, NY, USA, 2017. ACM. (cited on Page 19)

[RN10] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010. (cited on Page xiii, 9, 10, and 11)

[RPBL13] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The graph story of the sap hana database. In *BTW*, volume 13, pages 403–420, 2013. (cited on Page 21)

[RWE13] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013. (cited on Page 17)

[SFS+15] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1887–1901, New York, NY, USA, 2015. ACM. (cited on Page 16)

[SK12] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012. (cited on Page 34 and 35)

[SMS+17] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tameroezsu. The ubiquity of large graphs and surprising challenges of graph processing: A user survey. *CoRR*, abs/1709.03188, 2017. (cited on Page 1, 8, 28, and 38)

[TKKN16] Manuel Then, Moritz Kaufmann, Alfons Kemper, and Thomas Neumann. Evaluation of parallel graph loading techniques. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 4:1–4:6, New York, NY, USA, 2016. ACM. (cited on Page 4, 30, and 32)

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. (cited on Page 24)

[VLSG17] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *Proc. VLDB Endow.*, 10(5):493–504, January 2017. (cited on Page 35)

[Wie15] Lena Wiese. *Advanced Data Management:For SQL, NoSQL and Distributed Databases*. De Gruyter, 2015. (cited on Page 13)

[wik] Wikipedia requests for adminship (with text). https://snap.stanford.edu/data/wiki-RfA.html. Accessed February 9, 2018. (cited on Page 52)

[XD17] Konstantinos Xirogiannopoulos and Amol Deshpande. Extracting and analyzing hidden graphs from relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 897–912, New York, NY, USA, 2017. ACM. (cited on Page 13)

[YS12] Jiawei Han Yizhou Sun. *Mining Heterogeneous Information Networks:Principles and Methodologies*. Morgan & Claypool, 2012. (cited on Page 1)