Graph Pattern Matching on Symmetric Multiprocessor Systems

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. Alexander Krause
geboren am 17. August 1988 in Dresden

Gutachter:

Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden, Deutschland

Prof. George Fletcher
Eindhoven University of Technology
Department of Mathematics and Computer Science
Database Group
Groene Loper 5
5612 AP Eindhoven, The Netherlands

Tag der Verteidigung: 29. Juni 2020

Dresden im Juni 2020

ABSTRACT

Graph-structured data can be found in nearly every aspect of today's world, be it road networks, social networks or the internet itself. From a processing perspective, finding comprehensive patterns in graph-structured data is a core processing primitive in a variety of applications, such as fraud detection, biological engineering or social graph analytics. On the hardware side, multiprocessor systems, that consist of multiple processors in a single scale-up server, are the next important wave on top of multi-core systems. In particular, symmetric multiprocessor systems (SMP) are characterized by the fact, that each processor has the same architecture, e.g. every processor is a multi-core and all multiprocessors share a common and huge main memory space. Moreover, large SMPs will feature a non-uniform memory access (NUMA), whose impact on the design of efficient data processing concepts should not be neglected. The efficient usage of SMP systems, that still increase in size, is an interesting and ongoing research topic. Current state-of-the-art architectural design principles provide different and in parts disjoint suggestions on which data should be partitioned and or how intra-process communication should be realized. In this thesis, we propose a new synthesis of four of the most well-known principles Shared Everything, Partition Serial Execution, Data Oriented Architecture and Delegation, to create the NORAD architecture, which stands for NUMA-aware DORA with Delegation.

We built our research prototype called NEMESYS on top of the NORAD architecture to fully exploit the provided hardware capacities of SMPs for graph pattern matching. Being an in-memory engine, NEMESYS allows for online data ingestion as well as online query generation and processing through a terminal based user interface. Storing a graph on a NUMA system inherently requires data partitioning to cope with the mentioned NUMA effect. Hence, we need to dissect the graph into a disjoint set of partitions, which can then be stored on the individual memory domains.

This thesis analyzes the capabilites of the NORAD architecture, to perform scalable graph pattern matching on SMP systems. To increase the systems performance, we further develop, integrate and evaluate suitable optimization techniques. That is, we investigate the influence of the inherent data partitioning, the interplay of messaging with and without sufficient locality information and the actual partition placement on any NUMA socket in the system. To underline the applicability of our approach, we evaluate NEMESYS against synthetic datasets and perform an end-to-end evaluation of the whole system stack on the real world knowledge graph of Wikidata.

CONTENTS

AKNOWLEDGMENTS

During my time with the chair, I was concerned about me being able to finish this thesis. However, Professor Lehner encouraged me to stay on track and helped me to pursue this goal. Thus I want to thank you, Wolfgang, for being a great supervisor and for having my back, whenever it was necessary. Dirk Habich aided me with his tremendous knowledge and diligence, to make our papers and general work the best it could be. To all of my former and newly joined colleagues, who made the daily routine feel less like work but like being part of a family: Annett, Johannes 'Jay' P., Claudio, Eric, Johannes L., Michael, Mikhail, Patrick, Robert, Lucas and everybody who I have forgotten – thank you for all the enjoyable moments, the laughter, the banter and the coffee-kitchen debates.

I do also want to express my gratitude to my friends Martin, Robert, Alex and Kenny. The occasional beers, Doppelkopf evenings and gaming sessions lead to great memories and were a great way to relax after a long day of PC work. Frank, your critical voice helped me a lot to find the right balance between formaly expressing a context and its ease of understanding. My sincere gratitude goes towards Benjamin. You are the definition of a true friend and a great inspiration, you kept me going when it was the hardest.

Without any doubt, I could continue this list for pages, but I do not want to forget the most important person, my beautiful Evelyn. You cared for me, you kept me healthy and our relationship grew even stronger over these years. Your unconditional love and support allowed me to stay focused on my work and to finish this thesis eventually. Without you, I would not have been as successful. Last but not least, I do also want to express my gratitude towards my parents and my brother, Astrid, Roland and Tilo. All of you placed your trust and believes in me and encouraged me to pursue this endeavor. Without you, all of this would not have been possible. Thank you for everything.

Alexander Krause
Dresden, June, 2020

INTRODUCTION

1.1 MOTIVATION

The relational data model was first introduced in 1970 [Cod70] and its prominence lead to the trend, that traditional data processing often and heavily relied on it. As a consequence, many data domains were mapped to fit this model, e.g. temporal data [DDL02], XML documents [ACL+07] or provenance data [CLFF10]. However, graphs experience continuously increasing more interest as a general data structure, due to their expressiveness for semantic relations. Representative examples are manifold, e.g. traffic analysis in road networks, heritage analysis for anthropologists or recommender systems in video streaming platforms or online marketplaces.

> *"Graphs are omnipresent in our lives*
> *and have been increasingly used in a variety of application domains."*
> – George H. L. Fletcher [FHL18]

The most commonly known example is the internet web graph itself. A web crawl from 2012 contained already $3.5\,\mathrm{B}$ pages and $128.7\,\mathrm{B}$ hyperlinks [MVLB15], newer datasets from April 2020 feature $2.8\,\mathrm{B}$ unique pages with more than $280\,\mathrm{TiB}$ of uncompressed data[1]. Today's most prominent social network Facebook reported $1.39\,\mathrm{B}$ active users with more than $400\,\mathrm{B}$ edges in their social graph as of December 2014 [CEK+15]. Even the research for medicine is based on graph data. Here graphs are used e.g. for modeling and discovering polypharmacy side effects, where a drug-to-drug interaction network can contain 645 distinct drug and $19\,\mathrm{k}$ protein vertices, but 964 different *types* of edges, resulting in a total of $5.3\,\mathrm{M}$ interaction edges [ZAL18].

Efficiently storing and processing graphs is not trivial. Their sheer size requires suitable data models to handle this amount of data and moreover, completely or partially traversing such large graphs is also a complex task itself. These facts have thus created the necessity for specialized languages and systems, that treat graphs as first class citizens, like Ligra [SB13], Galois [NLP13], Green-Marl [HCSO12], Virtuoso [Erl12], Pregel+ [YCLN15] or Turbograph++ [KH18]; only to name a very few. Neo4j[2] is one of the most prominent graph processing systems, with both a commercial and a freely available product, that also satisfies the needs for large companies[3]. Efficient graph processing does also require an expressive query language to formulate graph specific algorithms and queries. Among others, Cypher[4], Gremlin[5], GraphQL[6], SPARQL[7] or G-Core [AAB+18] are designed to serve exactly this requirement. All these examples underline the importance of graphs and graph processing as a generally accepted and required data structure and application.

The tool box of graph algorithms contains a similar amount of examples. We have chosen graph pattern matching out of that plethora as a generally applied class of algorithms, due to its widespread applicability and relevance. Common use cases involve fraud detection [PCWF07], biomolecular engineering [OFGK00], scientific computing [TKS17], or social network analytics [OR02]. Generally, the pattern matching process receives a *pattern*, i.e. a set of vertices, that are connected through a specific set of edges, where both

[1] https://commoncrawl.org/2020/04/ [Last Accessed: 18.04.2020]
[2] https://neo4j.com/ [Last Accessed: 18.04.2020]
[3] https://neo4j.com/customers/ [Last Accessed: 18.04.2020]
[4] https://neo4j.com/developer/cypher-query-language/ [Last Accessed: 18.04.2020]
[5] https://tinkerpop.apache.org/ [Last Accessed: 18.04.2020]
[6] https://graphql.org/ [Last Accessed: 18.04.2020]
[7] https://www.w3.org/TR/sparql11-overview/ [Last Accessed: 18.04.2020]

vertices and edges can be enriched with additional information. It then tries to find all vertices in the data graph, that can be mapped to the query vertices, such that all query vertices are bound to a concrete data vertex. This process requires a graph traversal with one or multiple root vertices to start off. The efficient execution and optimization of multiple or parallel graph pattern matching instances is thus an interesting and relevant problem, that is not yet fully solved and still researched.

Solving such problems as fast as possible requires powerful hardware. To accommodate the ever-growing demand for sufficiently strong servers, hardware vendors first increased the core frequency of available processors. In addition to this, the continuously shrinking size of semiconductors lead to the development of multiprocessor systems. Both trends provide more compute power in one system, but with dark silicon as a major counterpart for increasing single core frequency, hardware vendors are putting increasingly more multiprocessors into single server machines.

A prime example for this category of servers are symmetric multiprocessor (SMP) systems. SMP systems are characterized by the fact, that each processor has the same architecture, e.g. every processor is a multi-core and all of these multiprocessors share a common and huge main memory space. Every processor is connected to its own, local main memory domain. However, any processor in an SMP system is able to access the memory domains of other processors, as if the system has one huge coherent address space. This allows us to store and process increasingly large graphs completely in-memory, while also keeping intermediate results inside the RAM. However, SMPs exhibit a *Non-Uniform Memory Access* (NUMA), where the memory access latency can vary significantly between different memory addresses. The TU Dresden just installed a new high performance compute server, which features a total of 1792 logical cores, which are provided by 32 sockets in total. Every socket provides approximately $1500\,\mathrm{GB}$ of main memory, totaling in almost $48\,\mathrm{TB}$ of shared main memory in just one system. Considering the NUMA effect is crucial for competitive performance, especially in such large systems, which has been proven by several previous works [ZCC15, PJHA10, LBKN14, KKS+14].

Current SMP systems do in fact provide sufficient compute power and main memory capacities to accommodate large graphs and perform online processing. Parallel graph traversal could be performed for either a single query and multiple root vertices or multiple queries can be processed concurrently. However, such approaches require algorithms, that are tailor-made for the underlying hardware. From a relational perspective, this has been investigated by ERIS, which was developed by the Chair of Databases with the TU Dresden. ERIS is a NUMA-aware database prototype, that is built to follow the near-memory processing paradigm, i.e. data is processed by cores, that reside on the same socket as the data. Adapting these database principles for hardware oriented graph processing, especially graph pattern matching, is yet to be investigated.

The goal of this thesis is to thoroughly examine the implications of SMP systems for near-memory graph pattern matching. Therefore, we leverage the ERIS prototype, to start off with an established database architecture and implement our graph pattern matching engine called NeMeSys on top. We integrate graph specific extensions for storing and processing graphs into ERIS and thoroughly evaluate them. We will furthermore demonstrate the interplay and applicability of all presented optimization techniques and provide an outlook for promising future research directions towards extending the graph pattern matching capabilities of NeMeSys on SMP systems.

1.2 SUMMARY OF CONTRIBUTIONS

This thesis analyzes concepts and best practices for processing GPM on NUMA-affected SMP systems. In addition, we use our proof-of-concept prototype NEMESYS to evaluate our assumptions. Our contributions can be summarized as follows:

(1) We detail on the foundations of both graph processing and current hardware trends. Based on our observations, we formulate our core idea of how to exploit the highly parallel target hardware through an asynchronous processing model. The ultimate goal is the proof-of-concept prototype NEMESYS, which allows for near-memory GPM on SMP systems.

(2) Processing GPM with an asynchronous processing model requires sophisticated data exchange or messaging techniques. We provide detailed insights on how the employed metadata for the graph's topology influences the system performance. Furthermore, we provide reasonable optimization measures to reduce the memory consumption while retaining already achieved performance gains.

(3) Parallel, asynchronous computations inherently require data partitioning or locking mechanisms. We categorize current graph partitioning approaches and provide heuristic implementations for them. Based on these categories, we analyze the influence of each approach and determine a guideline, how to fine tune the core-to-data partition ratio.

(4) The placement of data partitions in the system and their content directly influences the communication behavior. Because of the NUMA effect, socket-local communication will always be faster, than data exchange with any remote socket. We provide an optimizer, which is based on a communication cost model, that outputs partition reorganization proposals. The goal of these proposals is to maximize socket-local communication and minimizes remote messaging.

(5) We evaluate our prototype NEMESYS against Wikidata as a real world example. Our experiments are based on Wikidatas actual query logs, which allows us to evaluate our proof-of-concept on both real world data and queries. We detail on the influence of our system design aspects and how to overcome arising limitations.

1.3 OUTLINE

Figure 1.1 presents an abstract outline of this thesis, which matches in part with the previously presented summary of contributions. Introduction aside, the remaining structure is as follows. Chapter 2 provides the necessary information about graphs and graph processing. First, we present our applied data model, the edge-labeled multigraph and reason about graph pattern matching as go-to use case. Second, state-of-the-art processing models are presented and we explain our choice for asynchronous processing. The chapter is concluded with an introduction to the core concepts of Wikidata.

Chapter 3 is our second foundation chapter, which introduces the targeted hardware. Furthermore, we elaborate on the foundations of NEMESYS and the underlying ERIS implementation, which was developed earlier with the Chair of Databases at TU Dresden. Last, we formulate and provide evidences for three main challenges for GPM on SMP systems, that are solved with this thesis.

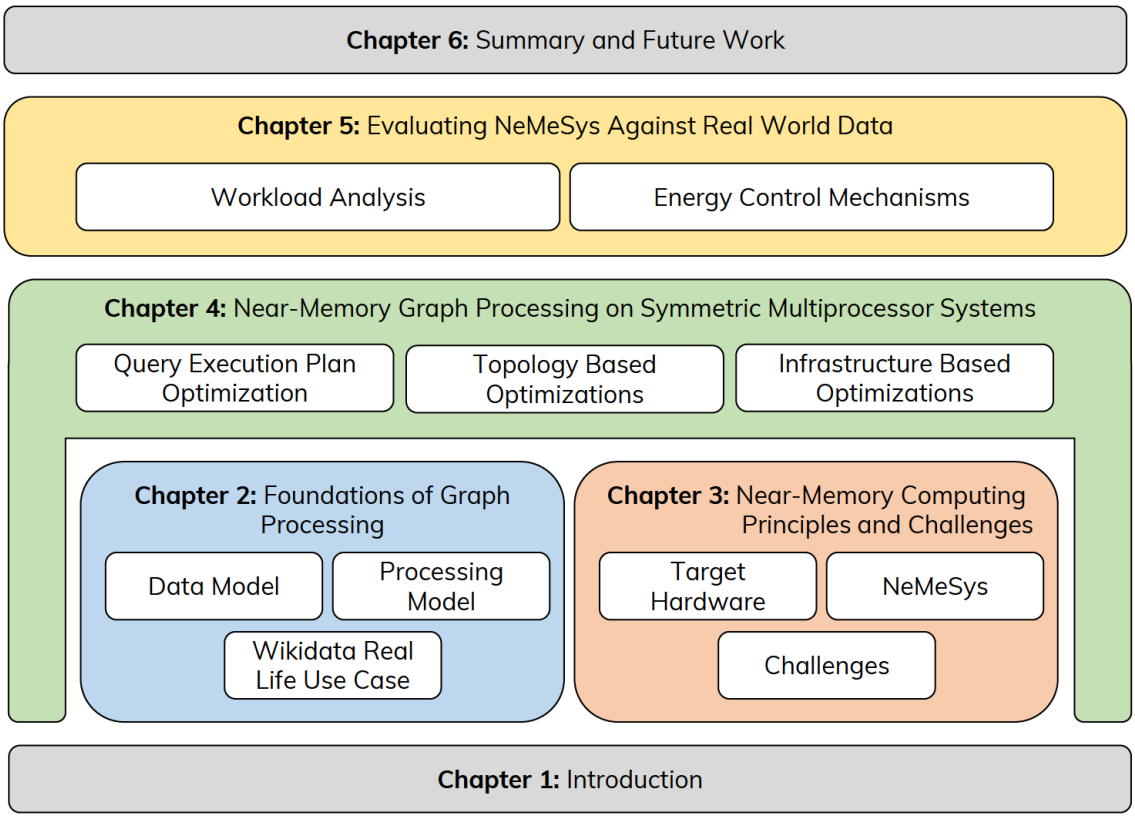


Figure 1.1: Thesis structure and outline.

Chapter 4 contains our optimization techniques. These are divided into three categories. Query execution plan optimization details about GPM statement reordering with and without information about the underlying data graph. Topology-based optimizations consider the actual data graph and leverage different graph partitioning techniques in combination with system resource allocation to improve the systems' performance. The infrastructure-based optimizations target system components like the messaging interface or the routing table, which holds vertex locality information.

Chapter 5 performs an end-to-end evaluation of the optimization techniques from Chapter 4 on Wikidata. This is done with parts of the original query logs from September 2016. To account for adaptivity, we apply existing energy-control techniques on NeMeSys. Thus we demonstrate the cooperation of the underlying ERIS implementation with our NeMeSys engine. We also present newly discovered limitations of our current implementation.

Chapter 6 concludes this thesis with a summary of this document. Ultimately, we provide an outlook for the most promising future research projects, that arose during the preparation of this thesis.

FOUNDATIONS FOR GRAPH PROCESSING

The last decades have seen a resurgence of interest in graph data management [Ang12]. With the network data model in the 1970s [TF76] and object-oriented database systems in the early 1990s, graph-based data models and graph query languages got considerable attention in research already [AG08]. However, the traction of today's graph data management efforts is unequally higher with many major IT companies and DBMS vendors on the band wagon [RKB04, FCP+11]. Among others, one major driver behind the graph concept's revival is a shift in the interest of analytics from merely reporting towards data-intensive science and discovery [HTT09]. Graph data can easily range in the size of billions of vertices and edges.

Prominent examples of large graphs are the Facebook friendship graph, the Twitter follower graph, citation networks, web link networks, road networks, supply chains, etc. (cf. [Les]). The areas of interest for the usage of graphs can further be extended to biology [SSV+17], chemistry [ITDK16], psychology [SC15], fraud detection [PCWF07], biomolecular engineering [OFGK00, TU10], scientific computing [TKS17], or social network analytics [OR02]. Fundamentally, the meaning of graphs as data structure is increasing in a wide and heterogeneous spectrum of domains, ranging from recommendations in social media platforms to analyzing protein interactions in bioinformatics [PV17].

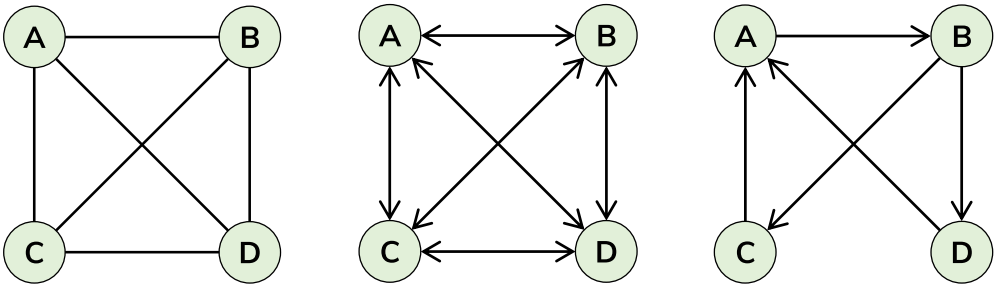
In 2017, Sahu, et al. [SMS+17] conducted a survey among researchers and practitioners to assess the ubiquity of graphs. The participant's fields of interest are wide spread and – besides *Research in Academia* – range from *Finance* over *Defence & Space* to *Telecommunications*. Mainly, the employed graph datasets were categorized into *Humans*, *Non-Human entities*, *RDF or Semantic Web* data and *Scientific* data, where everything which did not fit into the *Human* category was even further classified. The questionnaire showed, that the size of real world graphs ranges from less than $10\,\mathrm{k}$ edges to more than $10\,\mathrm{B}$ edges with raw sizes between a couple of megabytes and more than $1\,\mathrm{TB}$. Furthermore, the authors show that graph data is used by companies with more than $10\,000\,\mathrm{e}$mployees, which are not Google, Facebook or Twitter. These statements lead us to the conclusion, that the graph data format is a valid form of representation, widely used and accepted among industrial companies and research facilities.

In this chapter, we investigate the currently employed graph data models in Section 2.1 and highlight our targeted use case scenario from this domain in Section 2.2. After defining the general foundations of graphs for this thesis, we will further discuss state-of-the-art processing models in Section 2.3. Section 2.4 concludes this chapter, where we present Wikidata[1] - the knowledge graph behind Wikipedia. Inspired by the previously presented survey, we want to use this prominent real world example for graph processing as proof-of-concept for our graph processing engine, which is later introduced and extended in Chapters 3 and 4.

2.1 GRAPH DEFINITIONS AND DATA MODELS

The term *graph* has been firstly used by Sylvester in the 19th century [Syl78]. As today commonly known, the easiest graph definition is $G = (V, E)$, where V represents a set of vertices and E denotes a set of edges, which are connecting elements of V. Every vertex $v \in V$ usually represents some kind of entity, be it a person or a building and an edge $e \in E$ represents a relationship between two vertices $v_i, v_j \in V$; specifically every edge e is a pair of vertices. The *neighborhood* neigh of a vertex v_i contains every vertex v_j, that is reachable from a vertex v_i through an edge e and the degree of a vertex v_i is defined as $\deg(v_i) = |\text{neigh}(v_i)|$. This definition does not state, that the vertices in every

[1] https://www.wikidata.org/ [Last Accessed: 30.01.2020]



(a) An undirected graph.

(b) A directed graph, equivalent to Fig. 2.1a.

(c) A directed graph.

Figure 2.1: Schemas for different graph models.

edge need to be distinct and thus already allows the existence of *reflexive edges* or *loops*, i.e. (v, v) can be in E. Such edges occur often in e.g. state machines, where the vertices are states and the edges represent transitions between them. In such graphs, transitions from a state to itself, i.e. inner transitions, are often used. A reasonable example for this process is pressing the *reset* button of a vending machine, before inserting any money. This would trigger transition from the current state, which is *start*, to the *start* state itself. Without further information, such graphs are called *undirected graphs*, i.e. every edge is bidirectional. Figure 2.1a illustrates an undirected graph with $V = \{A, B, C, D\}$ and $E = \{(A, B), (A, C), (A, D), (B, C), (B, D), (C, D)\}$. Such graphs are used to express symmetric relationships, such as e.g. marriage, which is generally considered as a symmetric relationship.

As the model of undirected graphs has a limited expressiveness, an easy extension is to add directionality to edges, such that $(v_i, v_j) \in E \nRightarrow (v_j, v_i) \in E, i, j \in \mathbb{N} \wedge i \neq j$. These unidirectional edges are used to express one-sided relations, e.g. a one-way street connecting two intersections in a road network. However, undirected graphs can still be represented, as Figure 2.1b shows. In the model of directed graphs, bidirectionality has to be expressed by an explicit edge per relation. Ultimately, graphs are not limited to just contain directed or undirected edges. Figure 2.2 shows a compilation of different graph models and their respective edge types, as outlined by [RN10].

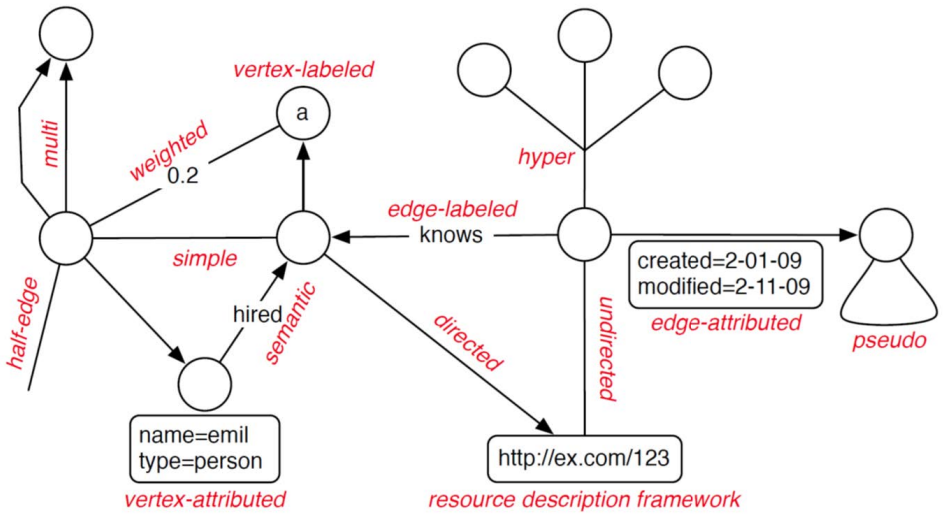


Figure 2.2: A collection of edge types, as shown in Figure 2 of [RN10].

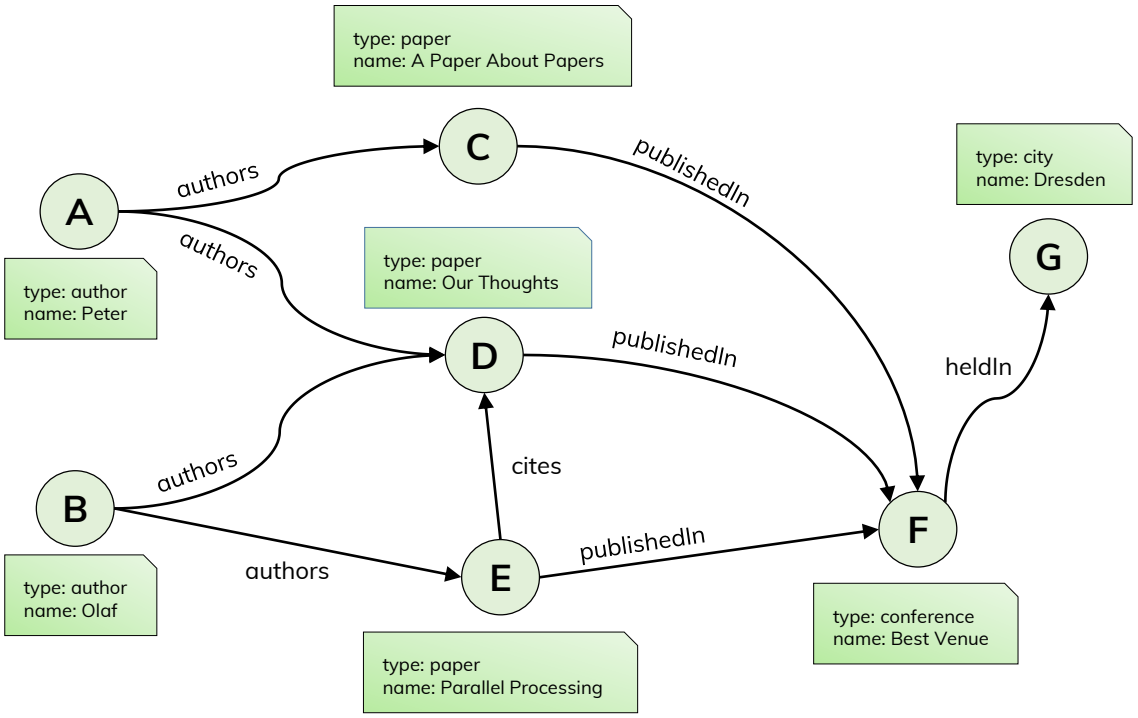


Figure 2.3: A bibliographical network using the property graph model.

Among others, the most important graph models for this thesis are the *labeled graph* and the *property graph*. As their names suggest, the graph is enriched with a variety of information, so-called properties, which are allocated with the vertices or the edges. A natural way of adding information to an entity, be it a vertex or an edge, is to add attributes or properties to them.

2.1.1 Property Graphs

In Figure 2.3, we show a bibliographical network using the *property graph model*, with properties attached to vertices. In such graphs, the vertices represent authors, papers, conferences and cities, where they have been held, etc. Edges of such a graph could be labeled with *wrote*, *heldIn* or *cited*. Through traversal operations, we could then create, e.g. a citation network and identify, which papers have been mostly cited.

According to Green et al., a property graph can be defined as $G = \langle N, R, \text{src}, \text{tgt}, \iota, \lambda, \tau \rangle$, where $N \mathrel{\widehat{=}} V$ and $R \mathrel{\widehat{=}} E$ of our previous definitions [GGL+19]. Additionally, *src* and *tgt* are sets of functions $R \to N$, that map each relationship (edge) to its respective source and target vertices. ι maps all vertices and edges to a set of key-value pairs, i.e. their properties. λ is a function that maps each vertex to a set of labels, which can also be empty and τ maps each edge to its actual type. In our example bibliographical network, vertex $\iota(B)$ would return {(type:author), (name:Olaf)} and $\tau((B, D))$ returns the label *authors*. Well known systems, that leverage this model, are Neo4j, ApacheSpark through GraphFrames [DJL+16], Oracle PGX [HDM+15] or Gradoop [JKA+17] only to name a few. One of the advantages of this graph model is, that almost all information of a certain vertex are held in one place and are directly accessible. However, storing them in an easily extensible way is not trivial, which is also shown by Green et al. [GGL+19].

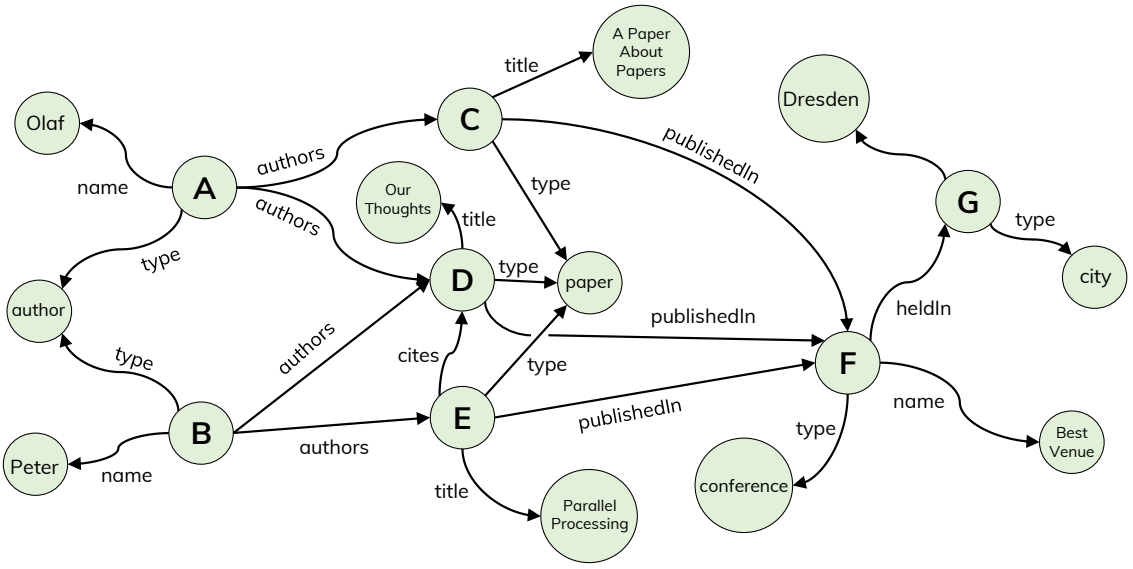


Figure 2.4: A network using the labeled graph model.

2.1.2 Labeled Graphs

A simpler representation is the *labeled graph model*. Labels are the easiest form of a property, e.g. as in the center of Figure 2.2, where the edge-label *knows* is shown. Such labels are usually used in directed graphs to enhance the semantic of a relation, as commonly used in social networks. Despite using less sophisticated data structures to store information, this model can hold the same level of detail as a property graph. In Figure 2.4, we transformed the property graph of Figure 2.3 to a labeled graph. This can be easily achieved by introducing proxy-vertices for each constant like names or types and adding the corresponding edges accordingly. At first glance, the graph looks more complicated, since its visual representation is more convoluted. However, we only store a single information per edge or vertex, which requires less effort to store or update data accordingly. Updating the type of vertex A would only need an update of the target vertex of one edge, in contrast to the property graph model, where we would have to look up all key-value pairs for A, find the appropriate entry and finally alter it.

Another advantage of the labeled graph model is, that it can be represented solely by writing triples, as they are used in the Resource Description Framework (RDF)[2]. That said, a graphs topology would be written down on a per-edge basis using triples of the form $\langle$src, tgt, label$\rangle$, where src, tgt $\in V$ and (src, tgt) $\in E$.

Within this thesis, we focus on *edge-labeled multigraphs* as a general and widely employed graph data model [OFGK00, OR02, PCWF07]. We define an edge-labeled multigraph as $G = \langle V, E, \rho, \Sigma, \lambda \rangle$, which consists of a set of vertices V, a set of edges E, an incidence function $\rho : E \to V \times V$, a set of labels Σ and a labeling function $\lambda : E \to \Sigma$ that assigns a label to each edge. Hence, edge-labeled multigraphs allow any number of labeled edges between a pair of vertices. This is especially important to represent graphs like social networks, where vertices can have multiple relationships with one specific target vertex. As briefly mentioned earlier, a prominent example for this graph data model is RDF [DMvH+00].

[2] https://www.w3.org/RDF/ [Last Accessed: 30.01.2020]

2.2 GRAPH PATTERN MATCHING

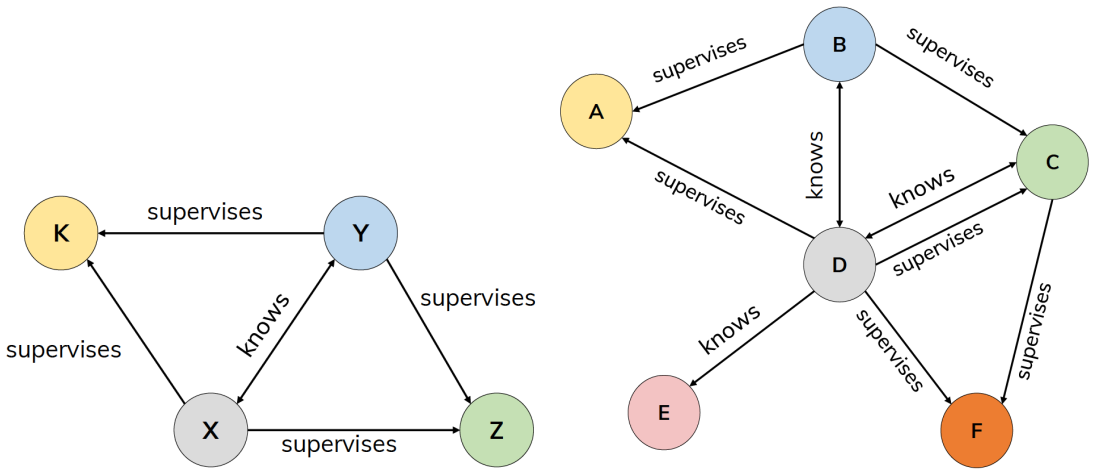
The previous section showed, that graphs can contain a plethora of information and extracting it can be done manifold. The analysis of even the biggest graphs is often done with recursive algorithms [SPSL13], whereas the Breadth First Search (BFS) has proven to be one of the most fundamental building blocks for many popular graph analysis algorithms. Such algorithms are used to determine reachability, connected components or betweenness centrality [Kin08, JRDY12, BKM+00].

In many cases, users are interested in identifying logical connections between vertices of their data graph. Thus, recognizing comprehensive patterns on large graph-structured data is a common use case and a prerequisite for a variety of application domains such as fraud detection [PCWF07], biomolecular engineering [OFGK00], scientific computing [TKS17], or social network analytics [OR02], only to name a few. Graph Pattern Matching (GPM) can therefore be considered as a crucial procedure and serves as the targeted use case for this thesis.

Over the time, a couple of graph query languages have emerged, such as Cypher, Gremlin, GraphQL, SPARQL or G-Core [AAB+18], to name the most well-known. Formulating GPM queries can be done in many of them. SPARQL is a query language for the triple oriented RDF data format and thus naturally aligns with our choice for labeled graphs and their representation as triples. Therefore we selected SPARQL for the visual representation of our example queries.

GPM queries are usually given as a subgraph of the queried data graph, which consists of vertices and edges with labels, that may occur in the original graph. Figure 2.5a shows a simple example of a GPM query. This query requests all two-sets of entities, who know each other and both supervised two distinct other entities. Considering the data graph from Figure 2.5b, the query would get only one distinct result, namely ⟨A, B, D, C⟩ which match to the query vertices ⟨K, Y, X, Z⟩ respectively.

A simple form of GPM queries are the well studied *conjunctive queries* (CQs) [Woo12]. CQs consist of a set of statements, where all statements are connected with the logical *and* operator, thus requiring every statement to be true. For edge-labeled multigraphs, a



(a) A graph pattern query.

(b) A simple data graph.

Figure 2.5: A Graph Pattern Matching example.

statement would represent any edge from E and considering the previously mentioned triple-notation, we could write down the query from Figure 2.5a as: ($\langle$ Y, K, supervises$\rangle \wedge$ $\langle$ X, K, supervises$\rangle \wedge \langle$ Y, Z, supervises$\rangle \wedge \langle$ X, Z, supervises$\rangle \wedge \langle$ Y, X, knows$\rangle \wedge \langle$ X, Y, knows$\rangle$). A greedy algorithm for GPM is to search the graph for every edge, in the order given by the user, and try to expand the found intermediate state by the following edges of the query. In addition, edges can also be expressed in an inverted form. This indicates, that instead of outgoing edges an incoming edge is desired and that this edge is to be traversed backwards. Such two-way queries are called 2CQs.

Another possibility of expressing GPM queries are *regular path queries* (RPQs). RPQs are based on regular expressions and thus allow the user to express queries, which can e.g. form a path of arbitrary length. This is especially useful, when searching in a taxonomy or when traversing social networks for e.g. n-hop friendship queries. Both approaches can be combined into CRPQs, where multiple RPQ statements are concatenated. C2RPQs are a form of CRPQs, where also inverted labels are allowed in the RPQ part. Processing RPQs of any kind is often done with a deterministic finite automaton (DFA) in the background, where the system traverses the data graph and the automaton in parallel. The DFA from Figure 2.6 shows the derived automaton for the expression knows*/(repliedTo/hasCreator)+ of an RPQ targeting a social graph, that contains users and forum postings, with the / symbol being a concatenation of path labels. While evaluating the query, a system would try to find all qualifying vertices for state s, i.e. vertices with outgoing *knows* or *repliedTo* edges. Upon following any of these edges and storing visited vertices, the automatons current state would then be updated to q_1 or f respectively. The result set would then be the current set of vertices, through whose edge traversals we can reach the state f.

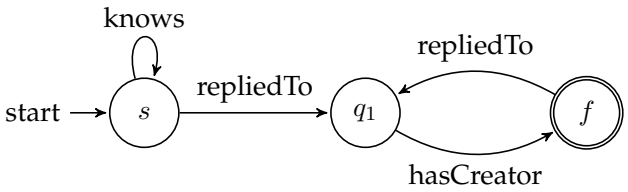


Figure 2.6: Derived automaton for the expression knows*/(repliedTo/hasCreator)+.

Processing both CQ or RPQ queries imposes a set of general challenges, which have to be thoroughly considered during query execution. A major issue is the number potential intermediate results during the evaluation of each query edge. Listing 2.1 visualizes the query from Figure 2.5a in SPARQL. For the edge predicate evaluation on line 5, we would retrieve all *uni:supervises* labeled edges from the graph of Figure 2.5b, which account for $54.5\,\%$ of all edges in the graph. Depending on the order and selectivity of edge predicates, intermediate results can easily grow beyond the size of the original data graph and thus an intelligent processing of the query is fundamental. For RPQ evaluation, the complexity is further increased with the inherent recursive matching of edges, while maintaining a list of visited vertices per state. Furthermore, the plentiful occurrence of the edge label *uni:supervises* generates a lot of potential vertices, from which we can start the actual graph traversal to complete the requested pattern. This allows for high parallelism, as each pattern could be processed individually. Handling both parallelism and a potentially high amount of intermediates is therefore a non-trivial task and needs to be covered by intelligent processing models.

Listing 2.1: SPARQL example for Fig. 2.5a.

```
1  PREFIX uni: <http://random.example.edu/>
2  SELECT ?X ?Y ?Z ?K
3  WHERE
4  {
5      ?Y uni:supervises ?K .
6      ?X uni:supervises ?K .
7      ?Y uni:knows ?X .
8      ?X uni:knows ?Y .
9      ?Y uni:supervises ?Z .
10     ?X uni:supervises ?Z
11 }
```

2.3 PROCESSING MODELS

Graph processing can require a high amount of compute resources, depending on the size of the graph and the issued queries. To satisfy this demand and enabling the inherent parallelism of GPM queries, modern servers with a high amount of multiprocessors are used, whose overall hardware aspects are further discussed in Chapter 3. This is a generally different hardware approach than before, because former systems sped up algorithms by increasing the processors core frequency, leading to a higher performance at a free lunch [BC11, Sut05]. However, because of power and thermal constraints, this free lunch is over and speedups will be only achieved by adding more parallel units [BC11], yet these parallel units have to be utilized in an appropriate way [BC11, Sut05].

The availability of different hardware settings with multiprocessors provides the general foundation for parallel GPM processing, as requested at the end of Section 2.2. However, proper utilization of the underlying hardware requires the implementation of an appropriate processing model. Today's state-of-the-art graph processing systems are usually built upon a vertex-centric programming model and are often referred to as "think like a vertex" (TLAV) systems [MWM15]. Such systems encourage the user to program from a vertex point of view. This forces the scope of an algorithm to a single vertex and a general communication pattern, since intermediate results have to be sent along the edges to neighboring vertices. Another approach are "think like a graph" systems [TBC+13]. Tian et al. argue, that providing users with more information, e.g. data partitioning and locality, leads to more overall performance and yields better optimization possibilities, including algorithm specific tuning.

2.3.1 Bulk Synchronous Processing

A traditional and widely used processing model is the Bulk Synchronous Processing (BSP) model [Val90], on which many vertex-centric systems are based [MWM15]. The overall workflow is depicted in Figure 2.7. In general, the BSP model consists of two disjunct phases. The first phase performs all processing steps and the second phase is solely reserved for communication. Both phases are then combined to global supersteps.

From the viewpoint of a GPM query, lines 5 through 10 from Listing 2.1 would each represent a global superstep. The processing phase would consist of several steps: (1) the vertex-local lookup, if an edge in the right direction with the appropriate label exists (2) the creation of a new intermediate matching state with the newly found target vertex (3) enqueuing the previously created intermediate matchings to be sent to their individual

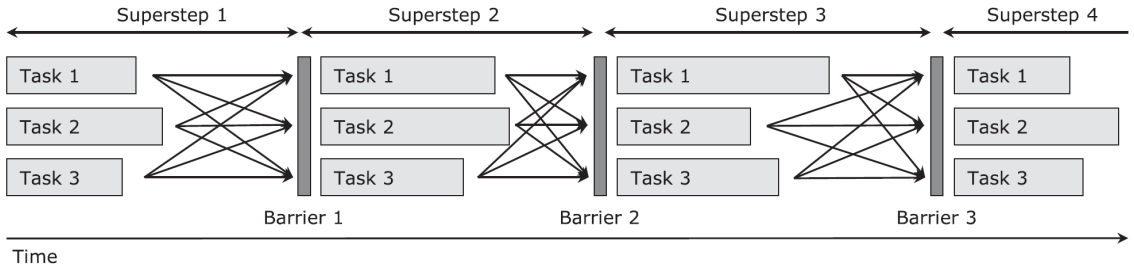


Figure 2.7: BSP execution diagram, cf. Fig. 1 from [MWM15].

target vertices. Vertices which do not produce any intermediate states during a superstep mark themselves as inactive. During the communication phase, the queues of all active vertices are checked and if not empty, the contents are moved to the respective target vertices. At the end of the communication phase, each vertex which receives any amount of intermediate states is marked as active and thus scheduled for execution in the next superstep. The procedure halts, when all vertices mark themselves as inactive.

The definition of parallel tasks could be either one task per individual vertex, i.e. one worker per vertex, since vertices are considered as first-class citizens in TLAV systems. This approach is generally not feasible, since real graphs consist of millions of vertices and even modern compute clusters do not yet features millions of processors. A more applicable approach is to partition the data into disjunct sets of vertices, where the number of such data partitions equals the number of compute resources. A widely known, BSP based, vertex-oriented system is Pregel [MAB+10].

Generally, such systems suffer from synchronization barriers, because of potential workload skew. A superstep will always only terminate, after all active vertices have concluded both of the phases. If the vertices in the individual sets are unevenly distributed in terms of total number of edges, then a few workers could concentrate the vast majority of the actual work. This can especially happen in so-called scale-free graphs, i.e. the distribution of neighboring vertices or edges per vertex follows a power law distribution. If only a few vertices contain a huge amount of edges, but they are additionally paired with more vertices from the graph to achieve an evenly distributed amount of vertices among all partitions, these overloaded partitions can become major bottlenecks.

2.3.2 Asynchronous Processing

Asynchronous processing models have been developed to overcome workload imbalances of any kind. Furthermore, asynchronicity allows to hide communication or disk access overhead via layering these operations with leftover computations. In their work, Han and Daudjee [HD15] have abstracted the BSP model to the *Barrierless Asynchronous Parallel* (BAP) model. Compared to the BSP model, BAP drastically reduces the amount of global synchronization barriers by introducing local barriers. These serve as breakpoints, where tasks can determine the next steps, i.e. mutating the graph or agreeing on a global synchronization barrier. However, the model still exhibits the processing and communication phases for individual tasks. Compared to BSP, individual tasks can have a different amount of local supersteps in BAP and since faster tasks are not as excessively stalled, the overall query response time decreases.

According to [MWM15], asynchronous computation can usually outperform synchronous approaches on CPU-bound problems, while the opposite is true for memory-bound problems. That is, since the flexibility of asynchronous computations allows to

adapt or reschedule the execution of vertex programs, according to arising workload skew. However, the authors state, that due to the asynchronous execution, the inherent messaging is unable to exploit optimization techniques like batching. This is an optimization technique, where multiple messages with the same target are combined into one large message, to better utilize the memory bandwidth by copying one large chunk of memory instead of many small chunks.

The BSP model does not naturally align with pattern matching algorithms [FNR+13]. This is caused by the usually high number of intermediate results, which is generated during the evaluation of edge predicates with low selectivity. Such intermediates have to be materialized as a potential result and transferred, which applies a lot of pressure on the communication layer of the system. This is also a major problem for power law graphs like social networks, when vertices with the highest degrees are generating plenty of intermediates. When the straggler problem, as defined by [MWM15], overlaps with a high amount of intermediates, we can expect a significant performance drop and thus anticipate any asynchronous processing model to be more suitable for GPM. That said, optimizing the communication issues would then be a high priority task, to enable this processing model not only for compute bound but also memory bound tasks.

2.4 WIKIDATA - A REAL LIFE USE CASE

Wikidata [VK14] is a free and collaborative open data platform, that has been founded in October 2012. Today, Wikidata is also known as the knowledge graph beneath Wikipedia. Like its sister project, Wikidata heavily relies on voluntary community work to create, integrate and maintain useful and relevant data. As of December 2019, Wikidata contains 890 M statements, which are equivalent to our definition of edges, and 56 M referenced items, which correspond to our vertex definition, forming one of the biggest openly available data graphs world wide featuring a plethora of languages. Wikidata is also used as a data plane for different science directions, such as life science and social studies [BWM+16, WGGM16].

The linked information is internally stored in JSON, but globally shared in RDF dumps [MKG+18]. A small excerpt of Wikidata in an RDF representation is shown in Figure 2.8 and represents a statement about the speed limit on roads in Germany outside of villages. Items, prefixed with a Q, are connected to other items via properties, which are prefixed with a P. Commonly with RDF, items and properties can be further prefixed with namespaces.

Wikidata features a globally accessible SPARQL endpoint[3]. The website provides an API, where users can post queries in the SPARQL query language, to extract any data of interest. Listing 2.2 shows an example query of the previously mentioned SPARQL endpoint.

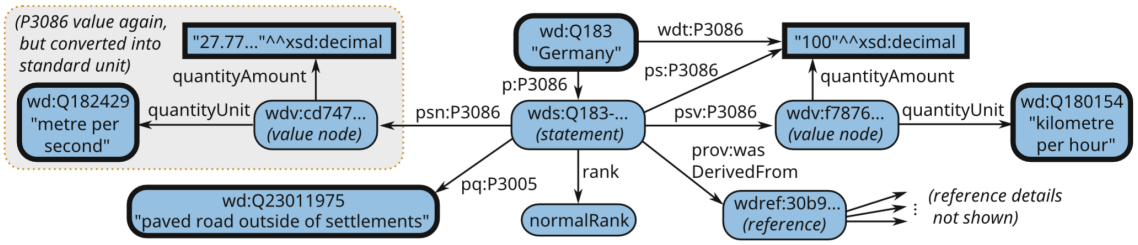


Figure 2.8: An RDF graph sample for a Wikidata statement, cf. Fig. 2 from [MKG+18].

[3] https://query.wikidata.org/ [Last Accessed: 30.01.2020]

The SPARQL query language closely follows a triple notation for every statement, similar to our definition of CQs in Section 2.2. The query requests all humans, which have no children, by combining three statements into one CQ. First, all humans are bound, followed by retrieving all child statements for every match for the variable *human*. Last, the *childStatement* variable is checked to be of the type "no value".

Previous work has shown, that this endpoint is not only used by humans, but is also heavily utilized by automated scripts or bots [BGK18]. Over twelve weeks, the authors parsed over $211\,\mathrm{M}$ SPARQL queries and categorized the workload in organic and robotic queries. We define a workload as the mix of queries, which arrives at the system over a given amount of time. A major finding was, that the organic workload accounts for only $0.5\,\%$ of all queries. Furthermore, the workload does not only consist of plain CQs as the one from Listing 2.2, but also CRPQs. In [BGK18], the authors show that CRPQs account for $44.5\,\%$ of robotic traffic and $24.4\,\%$ of organic queries respectively. The query logs for the work of [BGK18] have also been made publicly available[4].

Listing 2.2: Adjusted SPARQL example from Wikidatas SPARQL endpoint.

```
1  SELECT ?human
2  WHERE
3  {
4      #find humans
5      ?human wdt:P31 wd:Q5 .
6      #with at least one P40 (child) statement
7      ?human p:P40 ?childStatement .
8      #where the P40 (child) statement is defined to be "no value"
9      ?childStatement rdf:type wdno:P40 .
10 }
```

For Figure 2.9, we analyzed the workload logs from September 2016, which are not included in the openly available dataset. We divided the workload into four categories, which cluster queries by their length and assigned them to four quartiles, *Very Short*, *Short*, *Medium* and *Large*. The figure illustrates the arriving queries with an hourly resolution. The most interesting observation is the occurring variance in both query complexity, i.e. length, and volume over the course of the two weeks. These two workload characteristics imply the necessity for a workload adaptive processing model, which further underlines our statement from the end of Section 2.3

As previously mentioned, the SPARQL query language forms a natural match for our targeted use case, the processing of GPM with CQs. Because of Wikidatas prominence

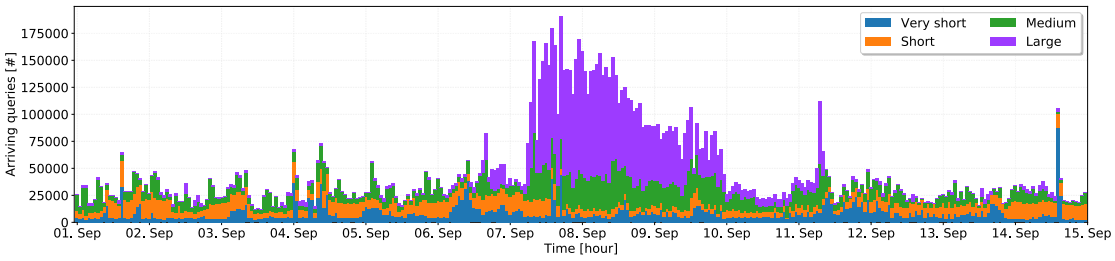


Figure 2.9: Arriving queries per hour, categorized by length in characters (quartiles), Wikidata workload from September 1st through 14th, 2016.

[4] https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en [Last Accessed: 30.01.2020]

and its openly available query logs with a tremendous amount of real life queries, we decided to use Wikidata as qualifying example for this thesis. In Chapter 5, we will evaluate our proposed query engine from Chapter 3 against this dataset and apply our optimization techniques from Chapter 4 to allow for scalable processing of graph pattern matching.

2.5 SUMMARY

Graph processing is a broad research field, which is widely studied. Representing graphs can be done in a variety of different data models, each having their individual advantages amenities and drawbacks. In this thesis, we focus on *edge-labeled multigraphs*, which represent edges of vertices as triples consisting of a source and a target vertex, enriched with a label. Despite introducing additional storage overhead for adding more structured information like properties to vertices, this graph data model excels at the ease of storing individual edges. From a data storage perspective, the triple representation allows for both row and column storage methods, where either triples are stored undissected or each source, target and label are placed in individual data columns.

The applied use case of this thesis is the processing of Graph Pattern Matching (GPM). This algorithm has a variety of application domains and thus plays a key role in analytical graph processing. Expressing query patterns can be done manifold, with conjunctive queries (CQs) and regular path queries (RPQs). Evaluating either of the two query types can lead to considerably large intermediate result sets, depending on the size of the underlying data graph, the selectivity of the pattern or possibilities to prune irrelevant vertices or edges during the graph traversal.

Graphs can contain a variety of patterns at many different locations. Searching for one specific pattern can thus yield many occurrences, where different data vertices match the given query variables. Hence we can employ parallel processing paradigms to speedup the discovery of all applicable matchings. Over the time, the Bulk Synchronous Processing (BSP) and Asynchronous Processing (AP) models arose to steer parallel computations. Synchronous approaches suffer from synchronization overhead and thus we selected asynchronous processing as the go-to approach for this thesis, since it also allows for maximum parallelism and flexibility during the query processing.

A prominent example for graph processing is given by the Wikidata knowledge graph and its publicly available SPARQL endpoint. Humans and robots can use this API to post queries, which are then processed concurrently. Parallel processing allows to spread out the computation of incoming queries among as many processors as necessary, to evenly distribute the load within the system. However, the whole data graph can not always fit into the main memory of a computer, while it additionally accommodates all intermediate results of all incoming queries. Thus, parts of the data remain on disk, which leads to slow access times, whenever these are needed.

Combining the potentially huge intermediate result space of GPM and the highly concurrent query processing of real world applications such as Wikidata leads to the conclusion, that highly parallel hardware with sufficient main memory is a desirable hardware platform. A huge and coherent main memory would allow, to keep both the data graph and intermediate results permanently inside the fast accessible main memory. In addition, many processors allow for either more intra- or inter-query parallelism. Providing more compute resources to an asynchronous processing model can further increase the ability for load balancing and timely query answers. Furthermore, the resources inside one large system can be more easily adapted than individual machines in a compute cluster, when the overall number of arriving queries increases. As long as sufficient compute power is available to overcome the largest anticipated workload spikes, we can always turn off overprovisioned processors, e.g. to save energy.

NEAR-MEMORY COMPUTING PRINCIPLES AND CHALLENGES

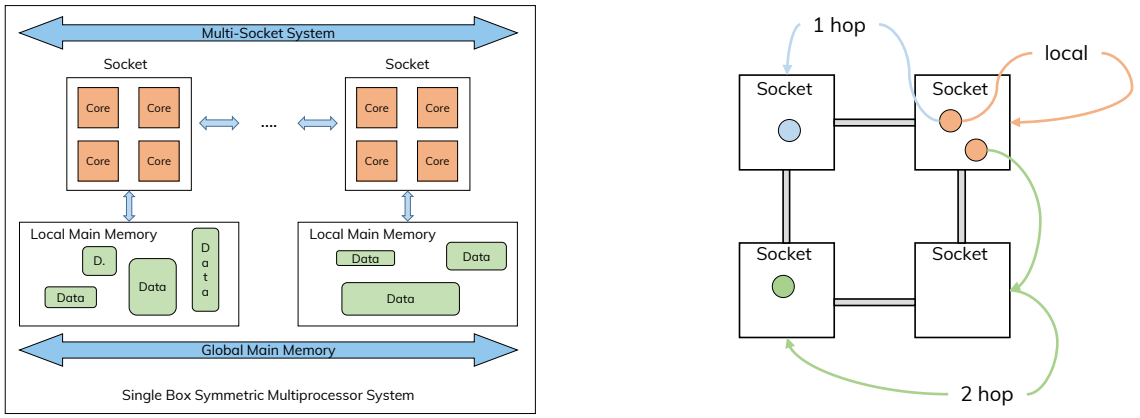
Today, building a new system requires thorough planning and the correct architecture for optimal or scalable performance. Within this chapter, we want to outline current state-of-the-art system architecture designs as well as the target hardware for our envisioned system. Section 3.1 gives an overview about the currently employed hardware for server class systems and discusses the pros and cons of standard architectures. Furthermore, we will present ERIS, which is a related research prototype, that leverages a Data-Oriented Architecture (DORA)-like architecture to allow for highly scalable processing of relational data in Section 3.2. After laying out the foundations of the target hard- and software, we will explain the principles of processing graph pattern matching on a system like ERIS in Section 3.3 That includes storing the data, query generation and processing as well as the crucial communication paths. The chapter will be concluded by a thorough analysis of arising challenges when combining the previously mentioned stack to process graph pattern matching.

3.1 HARDWARE CONSCIOUS SYSTEM DESIGN

When building a modern database system or even for data processing engines, considering the underlying hardware is a crucial aspect of the system design process. This section covers a definition of our targeted hardware, as well as state-of-the-art architectural approaches for current systems. In Section 3.2, we will present ERIS, which is a research prototype that leverages Data-Oriented Architecture (DORA) principles (cf. Section 3.1.2), to achieve high scalability. In this thesis, the term scalability refers to a proportional increase of performance, i.e. work done in a specific amount of time, proportional to the invested hardware resources. ERIS will serve as the foundation for our graph processing engine called NEMESYS, which is described in Section 3.3. Thus, parts of this section, especially the targeted hardware, share definitions and content with [Kis17].

3.1.1 NUMA-Affected Symmetric Multiprocessor Server Class Systems

Within this thesis, we are solely focusing on symmetric multiprocessor (SMP) server systems, also known as scale-up systems. That said, scale-out architectures, which consist of



(a) Schema of our target hardware.

(b) Illustration of the NUMA property.

Figure 3.1: Illustration of an SMP server system with the NUMA property.

Table 3.1: Experimental server setup.

	Physical Cores	Total Cores	Sockets	RAM
Small	32	64	4	128 GB
Medium	64	128	4	384 GB
Large	384	768	64	8 TB

multiple standalone servers, connected through any form of interconnect, are not considered. SMPs are characterized by the fact, that each processor has the same architecture, e.g. every processor is a multi-core and all multiprocessors share a common and huge main memory space. The amount of multiprocessors within one server can vary and an individual processor inside an SMP system is also usually called node or socket. In addition, every multiprocessor is composed of a number of similar CPUs, which we call cores henceforth. Furthermore, modern multiprocessors employ the simultaneous multithreading (SMT) technology, where every physical core provides multiple *logical* cores. The set of all logical cores of a physical core is called *siblings* and the term *hyperthread (HT)* is used to address all siblings of a core, except the actual physical core itself. Every socket is connected to its own, local main memory domain. However, any processor in an SMP system is able to access the memory domains of other processors, as if the system has one coherent address space.

This leads to one crucial aspect of such systems, the memory access model. Modern systems distinguish between *Uniform Memory Access* (UMA) and *Non-Uniform Memory Access* (NUMA). UMA defines, that the access of any memory address in the whole address space is performed with equal latencies. In contrast, NUMA means, that the memory access latency can vary significantly between different addresses. UMA yields limited system scalability. Since latencies are also influenced by e.g. circuit length, current SMP systems employ a NUMA model. Figure 3.1b illustrates basic NUMA access patterns. Considering a mesh-connected four-socket system, we can face up to two NUMA hops, before the actual memory access can happen. With every hop, the resulting memory access latency increases, but also the effective memory bandwidth decreases, as illustrated in Figure 3.2.

Several previous works prove, that considering the NUMA effect is crucial for competitive performance [ZCC15, PJHA10, LBKN14, KKS+14]. This underlines our hardware

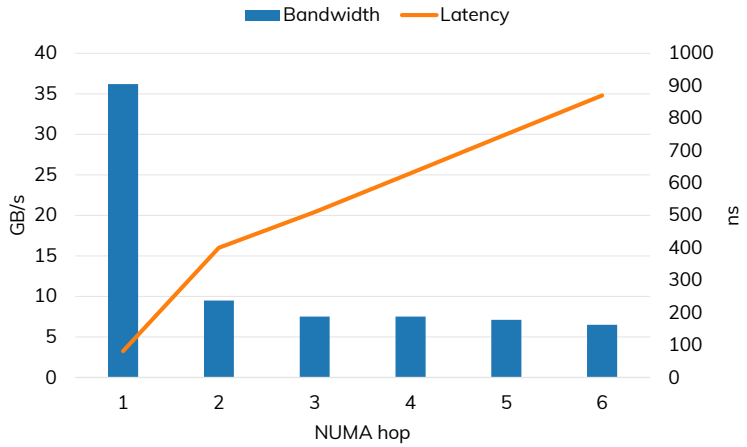


Figure 3.2: Bandwidth and latency effects per NUMA hop, sampled from Table 3.2 - SGI UV 2000/3000 of [Kis17].

focus, since handling a single box NUMA machine well will also result in higher performance, when a compute cluster consists of multiple of such machines. Therefore, we perform our experiments on the full spectrum of NUMA systems, ranging from a smaller to a larger machine. The experiments in this thesis were performed on differently sized servers, with their details being given in Table 3.1.

3.1.2 Database System Architectures for Parallel Systems

This section gives an overview about design principles for database architectures, especially those focusing the hardware of the previous Section 3.1.1. Deploying a database system on such scale-out hardware demands for good parallelization and synchronization mechanisms for optimal and scalable performance. There are two prominent usage scenarios, namely Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). OLTP refers to a high throughput scenario, where the database is required to process and complete the maximum amount of queries in the fastest possible time, whereas OLAP queries usually take a considerable amount of time to compute e.g. statistical overviews for sales purposes.

For OLTP optimized systems, Appuswamy et al. did a thorough analysis of state-of-the-art database architectures. Figure 3.3 displays an illustration of transactional processing on the four most popular designs, which are namely *Shared everything*, *Partition Serial Execution*, *Delegation* and the *Data-Oriented* [AAP+17]. More details about these four previously mentioned architectures can be found in their work.

Shared Everything (SE)

SE is an architecture which is employed by systems like Silo [TZK+13] or Hekaton [DFI+13]. Systems with this model employ a globally shared memory, where all data is stored and accessible by every database worker. Because of the general accessibility of the whole dataset to all database workers, their access must be synchronized by any form of concurrency control (CC) and/or locks and latches. These can be applied with any granularity, be it per record, table or whole databases. The SE concept discussed in [AAP+17] uses record-level locks, as depicted in Figure 3.3.

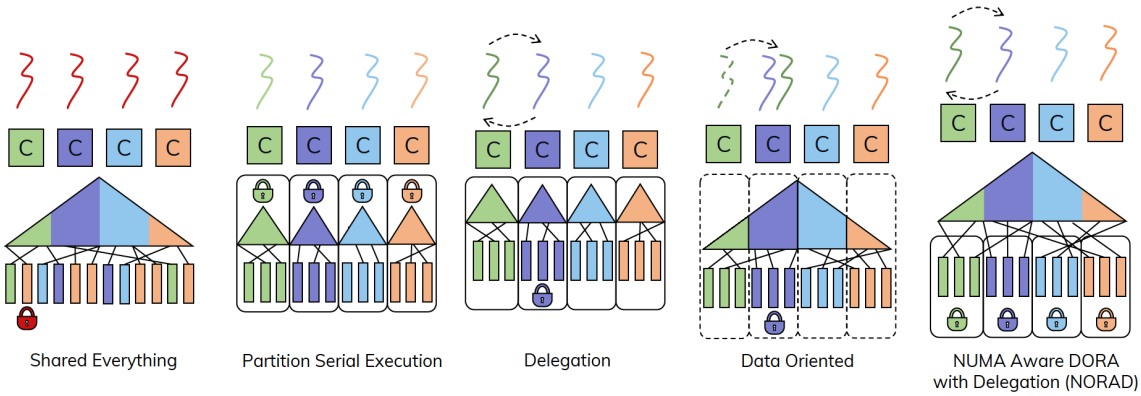


Figure 3.3: Transaction execution in different architectures (cf. Figure 1 from [AAP+17]) and our new synthesis: NUMA-aware DORA with Delegation (NORAD).

Partition Serial Execution (PSE)

Systems like H-store [KKN+08] or HyPer [KN11] use PSE. In contrast to SE, this model assumes a *shared nothing* situation, where workers can only exclusively access their respective datasets. This implies the necessity of data partitioning, where each database worker or site is provided with an exclusive data partition. According to [AAP+17], transactions are usually scheduled in a way that they are "*single site* in nature". A site is "the basic operational entity in the system", of which "a single physical computersystem [can] host one or more" entities [KKN+08]. Thus, that the systems tries to maximize local data access in terms of a single partition. In the case of a remote site access, transaction execution is serialized by locking all necessary partitions upfront and therefore guaranteeing that only one transaction or thread accesses a partition at the same time.

Delegation

Caldera [AKPA17] and Orthrus [RFA16] are examples for Delegation-based systems. This model assumes a distributed setting, even for single box SMP systems. In general, Delegation follows the same principles as PSE. However, when transactions need data from multiple partitions, the responsible worker sends an explicit *message* to the worker, which is responsible for the partition in question. Thus, a thread will always only process its own partition but needs to communicate, in order to receive all the necessary data.

Data-Oriented

The data-oriented transaction execution (i.e. DORA) approach [PTB+11] leverages the eponymous Data-Oriented architecture, so does the system PLP [PTJA11]. This approach is inspired by improving the performance of disk centric database systems. The core concept is a logical partitioning in contrast to physical partitioning. Here, worker access rights are stored in a global access table. This makes e.g. repartitioning of the data extremely easy, as only the entry for the to-be-moved records needs to be updated. In addition, DORA changes the usual thread-to-transaction assignment to a *thread-to-data* assignment. This means, that one thread is always responsible for a portion of the dataset and transactions need to switch between threads to access all necessary data.

These four architectural flavors are also thoroughly evaluated in [AAP+17]. A general outcome is, that the PSE architecture is the best performing architecture. However, we see that all four architectures have unique properties, which can and should be combined to achieve a highly scalable and performant new architecture: NUMA Aware DORA with Delegation (NORAD), as depicted in Figure 3.3. It combines the shared memory for metadata from SE, the physical data partitioning from PSE, the message passing from Delegation and the thread-to-data mapping from DORA to create a highly scalable architecture for single box SMP servers. We coined the name of this architecture after reviewing the work of Kissinger [Kis17]. In his thesis, he built the system ERIS based on the aforementioned principles.

3.2 ERIS - A NUMA-AWARE DATA MANAGEMENT SYSTEM

In this section, we will review ERIS, which is a NUMA-aware data management system. It was originally developed as a research prototype at TU Dresden, with the Chair of Databases and is solely written in C++ [Kis17]. The system employs architectural design aspects, that sum up to the NORAD architecture as coined by us and shown in Figure 3.3. In his work, Kissinger demonstrates ERIS' high scalability on modern NUMA-affected SMP servers and its variety of adaptivity mechanisms directly built-in [Kis17].

Throughout the last years and over the lifetime of its development, this system manifested as chair-internal testbed and collaborative platform. That said, we too took the chance to assess ERIS' good results in the relational environment and use it as the foundation for a graph processing engine. As outlined in Chapter 2, graph processing is an important field just like relational query processing. ERIS was developed to achieve high scalability, considerable performance and the ability to adapt to workload changes while running on the same SMP hardware, that we target ourselves. Combining these facts leads to the conclusion that a NORAD based system like ERIS should not only be able to efficiently process relational queries, but could also be used for graph processing.

Thus, we review the schematic design of ERIS in Section 3.2.1 and provide a brief set of details of its C++-programmable query engine. ERIS does also perform self-managed memory handling, which is discussed in Section 3.2.2. Furthermore, details about the message passing mechanism like message structuring and messaging infrastructure are presented in Section 3.2.3. This section is then concluded with an overview of the Energy Control Loop, which provides ERIS with high flexibility in terms of software-enabled adaptivity.

3.2.1 Architecture

ERIS is specifically tailored for single box SMP server systems, as depicted in Figure 3.1a, with its corresponding architecture being shown in Figure 3.4. The system features an

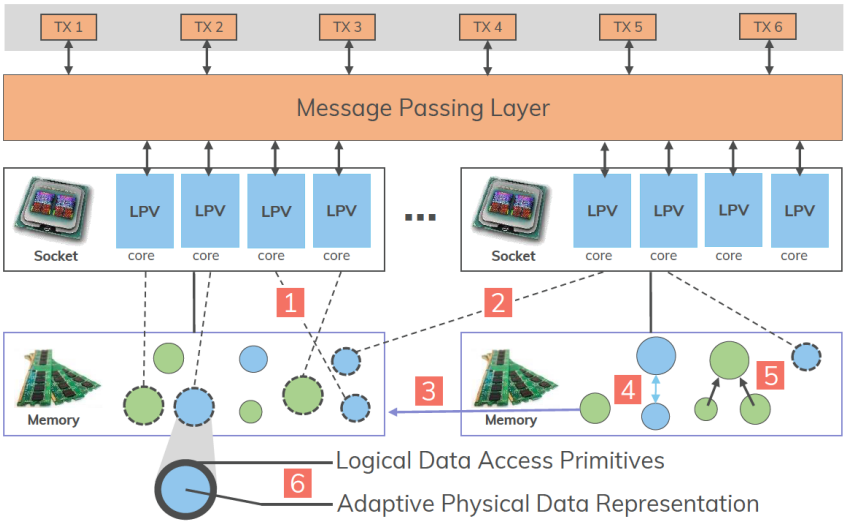


Figure 3.4: Query processing in the Living Partitions architecture, cf. Figure 3.5 of [Kis17].

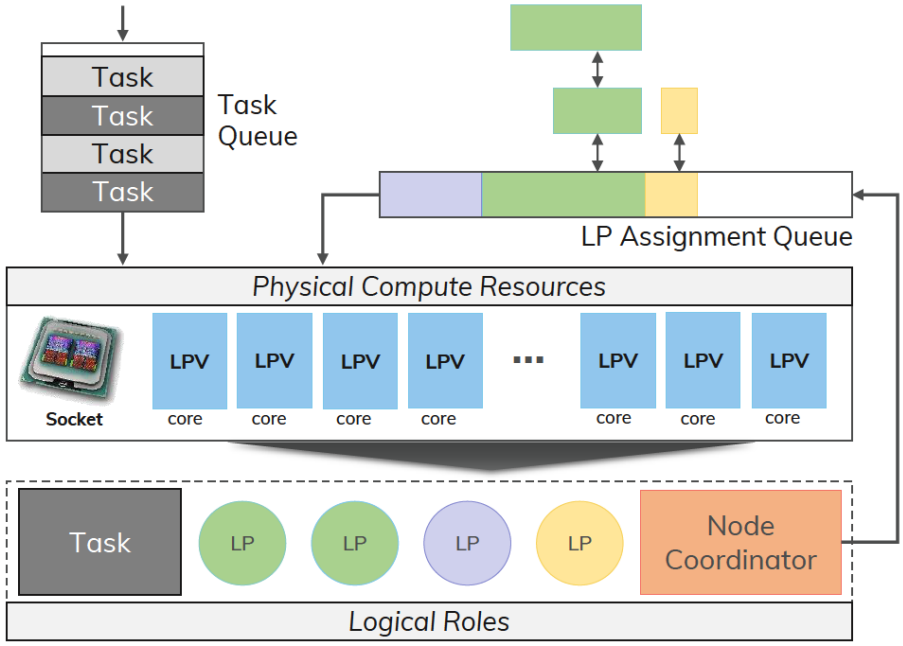


Figure 3.5: ERIS processing architecture of a single socket, cf. Figure 3.18 of [Kis17].

asynchronous execution model and can be roughly dissected into three layers: the *storage layer*, the *processing layer* and the *message passing layer*.

The *storage layer* holds data partitions, called `LivingPartitions` (LPs). When data is loaded into ERIS, it is stored within the `RelationalContainer`, which resembles a table of a database. This container is then divided into an arbitrary amount of LPs. Thus, LPs always contain a unique slice of the stored data and are placed on the local main memory of a single socket, without the usage of replication techniques. In general, an LP provides the user with general data access primitives, but allows for different physical data representations. This representation is exchangeable individually for any LP during runtime, cf (6) in Figure 3.4.

The *processing layer* contains the `LivingPartitionVitalizers` (LPVs) and `NodeCoordinators`. LPVs represent software threads, which are pinned to exactly one logical core of the system and thus ERIS can generally occupy all available compute resources. Pinning threads to cores is necessary, to avoid costly context switches and thus improves the overall system performance. The `NodeCoordinator` (NC) is a role, which every LPV can obtain while cycling through their event loop, but only exactly one LPV per socket is allowed to become the local NC at any point in time. In contrast to the DORA concept, where one worker is only allowed to access an exclusively assigned dataset, LPVs are free to choose the LP which they want to process, as long as the LP resides on the same socket as the LPV itself, cf (1) in Figure 3.4. However, under certain circumstances, an LPV may process the LP of another socket as shown with (2), which is later discussed in Section 3.2.4.

Figure 3.5 zooms in on the processing architecture of a single node. In ERIS, query processing is split into tasks, which are stored in socket-local task queues. Tasks are executed by LPVs and specify the concrete operators and logic behind a query. Algorithm 2 of [Kis17] gives a brief usage description of tasks in general. Programming tasks in ERIS is done with the ERIS/C++ interface. That is an API, which provides classes to handle e.g. transactions, dataflows or user-programmable operators, that are called `MicroOperators`.

Listing 3.1: GenericMicroOperator example.

```
1  GenericMicroOperator* gmo = new GenericMicroOperator( container,
2      [&] ( MicroOperator* mo ) -> int64_t {
3          MessageBuilder mb( // Instantiate Messaging with...
4              StorageOperation::Lookup, // DB Operation
5              container, // Targeted Container
6              mo->dataFlow, // Attached DataFlow
7              m_successor->gmo // Succeeding Operator
8          );
9
10         // Project these columns
11         mb.setDesiredAttributes( ... );
12         // Filter by those columns
13         mb.setTargetAttributes( ... );
14
15         /* Some runtime logic */
16
17         mb.add( ... ); // Create Message
18         deallocate( ... ); // Free Memory
19         return 0;
20     },
21     nullptr
22 );
```

`MicroOperators` (MOs) always target exactly one container – like the previously mentioned `RelationalContainer` – specify two C++ callback functions and can have a set of succeeding MOs. In Listing 3.1, we illustrate a basic implementation of a root operator, based on the derived class `GenericMicroOperator`. The `gmo` specifies only the first lambda function, as it does an initial `lookup` operation, see line 4. Then, we need to specify which columns (i.e. attributes) we want to filter for (line 11) and which columns to project into our result tuple (line 13). Qualifying tuples are fetched from the storage layer and are sent to a succeeding operator (line 17). The second function (line 21) is always called by the predecessor of the current operator, which handles the incoming tuples and can thus be ignored and set to `nullptr` for a root operator.

Generally speaking, a query will be composed of a set of MOs, that are encapsulated within a task. During runtime, an LPV will process tasks and execute the respective lambda functions together with the required LPs. Since LPs are independent data objects, each LPV can call the same function of a task on its individual LP without the necessity of synchronization. Due to this mechanism and the asynchronous nature of ERIS, it can easily happen that multiple stages of a query are processed at the same point in time. It is totally possible, that e.g. the root operator is still scanning and producing tuples on LPV_1, while LPV_{25} runs a function instance of the result collecting operator.

3.2.2 Memory Management

NUMA is a crucial property of our target hardware and thus ERIS implements a self-made memory management subsystem from scratch. Important aspects are high scalability, low contention risk on key resources and a good compatibility to ERIS' general architectural characteristics [Kis17]. Memory itself is hierarchical by nature, with the cache hierarchy as a prominent example. In case of a NUMA system, we extend that hierarchy by global and local memory domains. This lead to the thought of a hierarchical memory management component, as shown in Figure 3.6.

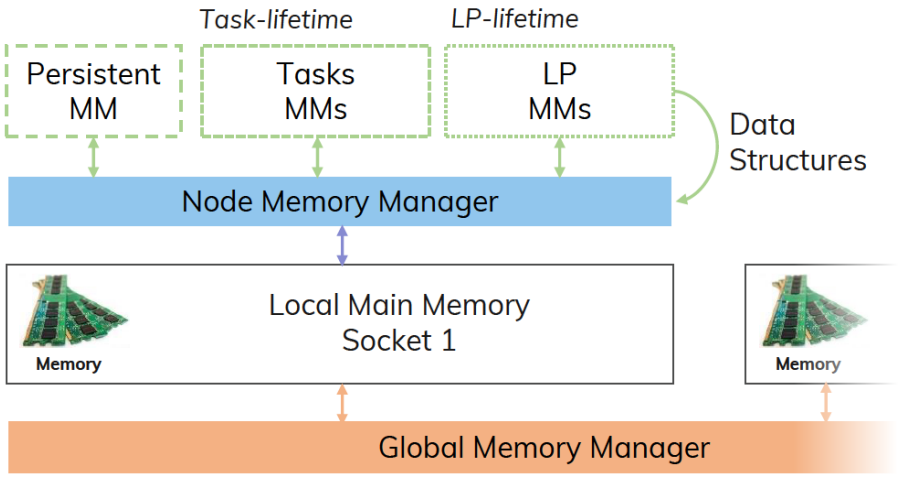


Figure 3.6: ERIS memory management, cf. Figure 3.19 of [Kis17].

ERIS works with one static global memory manager, which is responsible for allocating memory from the system and managing chunks of memory, which are allocated on the local memory areas of a certain socket. For every socket, node local memory managers are instantiated as the second layer of this hierarchy. These request memory chunks from the global manager, according to the individual memory demand of the local LPVs. Tasks and LPs are transient objects by nature, i.e. they can be created and deleted over the lifetime of the system. Thus, the last layer of memory managers consists of individual managers for each task and every LP. The third part of the last layer is a persistent memory manager, which is responsible for allocating everything related to LPVs internal routines or shared data structures. The hierarchical memory manager stack minimizes the actual amount of system calls, which effectively increases the systems performance. Since only the global manager allocates chunks from the operating system, all other managers can obtain small slices internally, which is considerably faster.

3.2.3 Message Passing

The *message passing layer* is a crucial component of ERIS and a key aspect in high system performance. As pointed out in [AAP+17], message passing alleviates the problem of synchronizing data access. However, treating a shared memory system as a distributed system comes at the cost of messaging overhead, which has to be paid off. Thus ERIS implements a high throughput and hierarchical designed message passing interface.

The previously described messaging process from Listing 3.1 is further illustrated in Figure 3.7. The shown layout abstracts the messaging infrastructure and is replicated to all sockets. During query processing, LPVs communicate qualifying tuples, i.e. intermediate results via messaging through the `MessageBuilder` class. Every socket has multiple local and remote outgoing and incoming buffers, where one outgoing buffer is allocated for each socket in the system. The outgoing buffers are filled by the local LPVs (1). Messages targeting local LPs are placed in the local outgoing buffer and messages targeting LPs of other sockets are placed in the according remote outgoing buffer. Whenever an LPV becomes the NC (2), it distributes the pending messages to the incoming buffers of the corresponding sockets (3, 4, 5), where the NC of these sockets redistributes them to the appropriate partition queues. Step (3) shows the creation of local LP assignments, i.e. messages are placed in incoming message queues on an LP basis, where LPVs can later pick up the work and process that LP.

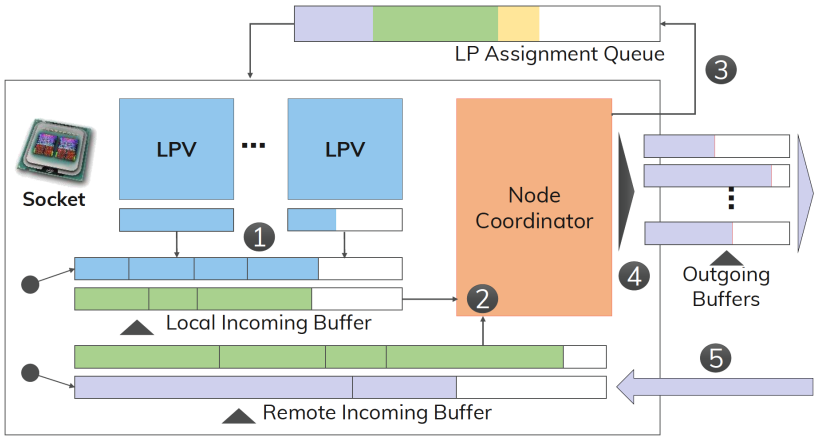


Figure 3.7: Living partition-enabled message passing layer in ERIS (socket-level), cf. Figure 3.25 of [Kis17].

The structure of messages is shown in Figure 3.8. For us, the most important parts are the *Target LP*, the *Micro Operator*, and *Cmd 1* through *Cmd N*. The target LP determines, which socket needs to receive the message. This information is stored by the last and most crucial component of the messaging passing layer, the *routing table*. The routing table is an index and can be considered as metadata of the container data structures. It holds information about the location of all LPs and the data in them. Designing the routing table efficiently has a considerable impact on the performance, since every intermediate result, which ends up in a message, will go through this component.

The lookup of a specific data item in the routing table yields the respective target LP, iff the routing table contains an index entry for that specific data of interest. If no index entry could be found, a null pointer is returned. When the routing table returns a valid pointer to an existing LP, ERIS can create a so-called *unicast* message. Unicasts are single messages, which can be directly forwarded to the respective LP. However, upon returning a null pointer, ERIS forms a *broadcast* message. The NC will place instances of that broadcast message into the message queue of *every* LP of the specified container (cf. Figure 3.8), thus replicating the message according to the number of LPs in a container.

The `MicroOperator` part of a message contains a pointer to the lambda function of the specific `MicroOperator`, which needs to be executed on the receiving partition. As explained in Section 3.2.1, `MicroOperators` will send their messages to successors, and

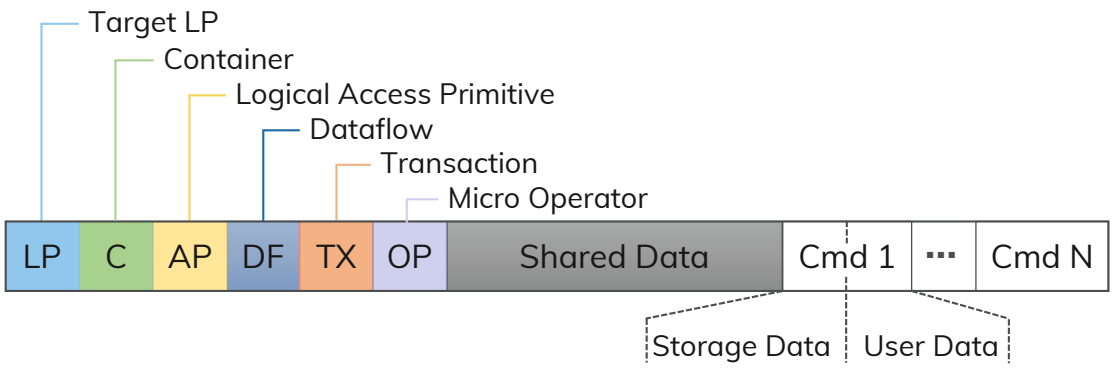


Figure 3.8: ERIS message format, cf. Figure 3.24 of [Kis17].

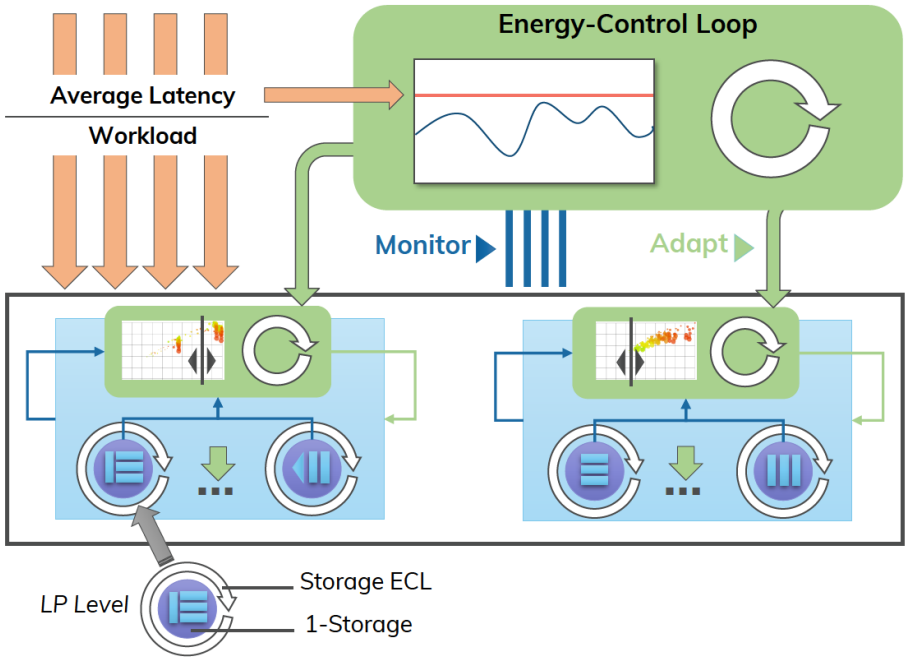


Figure 3.9: ECL hierarchy including the storage ECL per LP, cf. Figure 5.1 of [Kis17].

thus their receiving lambda function pointer will be sent along in the message. Lastly, the *Cmd* parts contain the data, which has been fetched by the storage, together with the user data. User data refers to the actual intermediate results, e.g. if a query builds a result tuple over multiple operator instances, these partial intermediate results are packed in the user data area.

3.2.4 The Energy Control Loop (ECL)

The declared goal of ERIS is to be energy proportional, i.e. if there is a decrease in the system load, the systems energy consumption should decrease proportionally and vice versa. Controlling the energy consumption is again done via a hierarchicaly organized *Energy Control Loop* (ECL), as shown in Figure 3.9. The ECL is responsible to tweak control knobs on each layer and triggers the respective ECL layers to take measures, based on internal monitoring values.

For the *processing layer*, the ECL can apply Dynamic Voltage Frequency Scaling (DVFS) or even turn off complete cores or sockets, to fine tune the performance and energy consumption of every core in the system. The adjustment of the processor clock rates is guided by so-called *Work-Energy-Profiles* (WEP). These show a set of configurations and their energy efficiency in relation to the performance they provide. A configuration in the context of a WEP means the number of active cores, their individual clock speed and which processor they are located on.

The *storage layer* can be triggered to adapt the underlying physical representation. Here, the ECL can select from three available formats. One is the generally employed row store model, the second is a column store and the third format is a hybrid of a column store and a row store, where only some attributes of a data record are organized in columns and the rest is stored in a row wise manner. In addition to switching the data layout itself, the access patterns within a partition are constantly monitored. This allows for triggering a

(re-)build of single- or multidimensional index structures on a per-attribute basis. These operations are steered by the LP-level ECL.

The reduction of the overall energy consumption is rather easily achieved, by simply turning off all cores except one, which does the whole processing. However, this would leave the system with insufficient resources and thus, ERIS is guided by the average query latency with thresholds. Consider a targeted maximum query latency of, e.g. $1\,\text{sec}$. ERIS measures, how many *good* and *bad* queries are answered in a certain time frame, where *good queries* conform the target latency and *bad queries* exceed it. The control loop periodically checks, if the amount of *bad queries* exceeds a predetermined toleration threshold. If more than the toleratable amount of *bad queries* are measured, the ECL provides gradually more resources to satisfy the demand. On the other hand, if the system yields a rate of $100\,\%$ *good queries*, it can gradually drain resources, like turning off cores or reducing their clock or both, while maintaining a sufficient amount of *good queries*.

3.3 NEMESYS - ALLOWING NUMA-AWARE GRAPH PATTERN MATCHING ON ERIS

In this section, we want to unite the basics from Chapter 2 and the ERIS system, as described earlier in Section 3.2. Parts of this section are based on our previous work [KKH+17, KUK+17]. Following the reasoning from Section 3.1, modern servers incorporate an increasing amount of cores and sockets. Since current hardware trends are continuously aiming towards an increasing amount of parallelism and main memory capacities, a non-uniform memory access becomes more and more prevalent to allow hardware resources to scale up to dimensions of the likes of thousands of cores.

To address the issue of increasing NUMA effects and to achieve scalability for GPM algorithms inside of a single machine, we adapt the NORAD architecture and build a graph pattern matching engine on top of ERIS. By combining the relational foundation and graph processing principles, we create our new research prototype called NEMESYS, which is short for **Ne**ar-**Me**mory Graph Processing **Sys**tem, and present its architectural overview in Figure 3.10. The term *near-memory* describes, that the data is processed by a worker from the same socket, where it is actually located and thus differs from plain *in-memory* processing. We will use the term NUMA-aware as a synonym for *near-memory* in this thesis, since our system enforces socket-local computation by design. NEMESYS has been published in our previous work [KKHL19, KUK+19].

Being based on ERIS, NEMESYS shares the same topological structure and thus consists of a *storage layer*, a *processing layer*, a *communication layer*, which we call infrastructure and components related to user I/O. In contrast to ERIS, NEMESYS allows for user defined, i.e. online data ingestion and query generation.

We argue, that a scalable relational engine like ERIS is a perfect starting point for our endeavor to build a NUMA-aware GPM engine, since basic problems have already been addressed. That is, parallel workload distribution, general data distribution and worker allocation is already present and thus we can focus on graph-specific aspects. In the following sections, we will highlight the general design aspects of NEMESYS and how we modify the underlying research prototype, to accommodate a GPM engine.

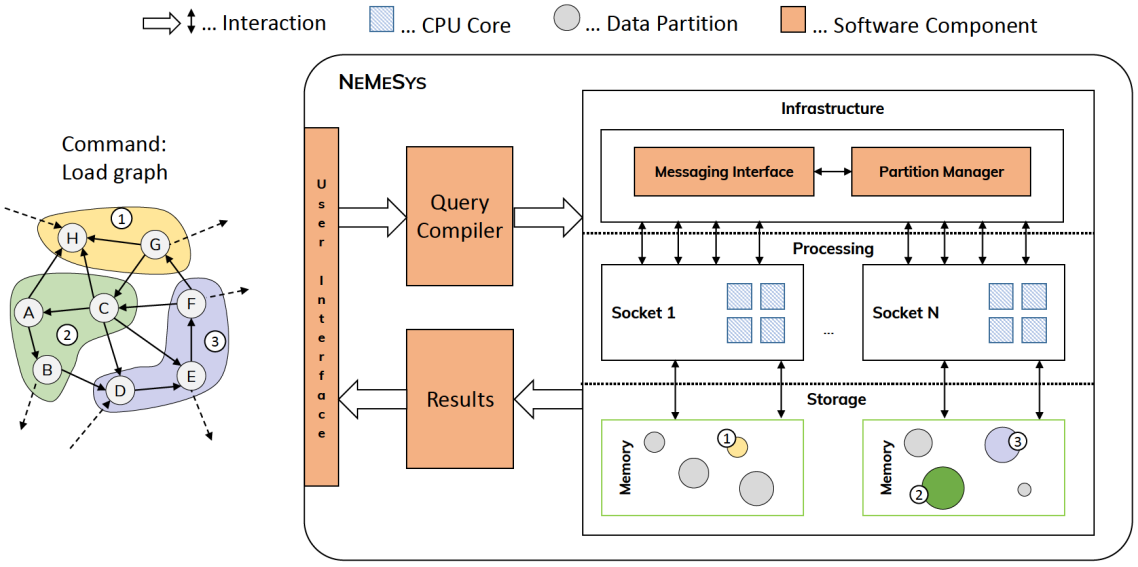


Figure 3.10: Architectural overview of NEMESYS. Here, a part of a graph is shown and divided into three disjunct partitions, which are distributed among all sockets. Adjacent partitions are not necessarily stored on the same socket.

3.3.1 Data Storage

Storing a graph on a NUMA-affected system inherently requires data partitioning, to cope with the NUMA effect. Hence, we need to dissect the graph into a disjunct set of partitions, which can then be stored on the individual sockets, thus leading to the graph partitioning problem (GPP). We consider every edge between two vertices as a potential communication path between two partitions, which ultimately result in messages in NEMESYS. Therefore, it is advisable to reduce the number of edges and consequently messages, that span across multiple partitions in order to be NUMA-aware. According to [HR73], partitioning a graph into a balanced and disjoint set of k partitions while minimizing the edges, that are cut between partitions, is an NP-complete problem. Thus, we need to find appropriate heuristics in order to partition our data in a timely manner, which we present in detail in Section 4.2.1. For now, we assume an appropriate partitioning strategy to be applied, regardless of the actual algorithm.

In Section 2.1.2, we presented our data model, which is the edge-labeled multigraph. This graph format can be easily represented using a triple notation, and thus perfectly fits the relational storage model of ERIS. By applying some partitioning strategy, we can extract Table 3.2 from the graph in Figure 3.10.

Table 3.2: Outgoing edge table for the graph in Figure 3.10.

Source	Target	Label
Partition 1		
G	C	label_{GC}
G	H	label_{GH}
Partition 3		
D	E	label_{DE}
E	F	label_{EF}
F	G	label_{FG}
F	C	label_{FC}

Source	Target	Label
Partition 2		
A	B	label_{AB}
A	H	label_{AH}
B	D	label_{BD}
C	A	label_{CA}
C	D	label_{CD}
C	E	label_{CE}
C	H	label_{CH}

In general, a vertex is defined by its incoming and outgoing edges. However, to represent the topology of a graph, it is sufficient to store only the outgoing edges of a vertex, since we would otherwise duplicate every edge between any source and target vertices. Within NeMeSys, a vertex is stored as a collection of its outgoing edges and therefore consists of multiple records. The 11 edges of the example graph are organized within a dedicated class, the `LabeledGraphContainer` (LGC). Every LGC is responsible for holding exactly one instance of a graph, hence storing multiple graphs results in an equal amount of LGC instances. An LGC in turn holds multiple instances of the previously described LPs, denoted as *Partition* in Table 3.2 and colorful highlighted in Figure 3.10, which are responsible for the actual physical storage of the data records. Considering an edge as a 3-tuple naturally implies storing graphs in a row-wise organization on the physical level. However, due to GPM being a vertex-lookup heavy use case, we believe that a columnar representation is a more performant approach. That is, since the lookup of a vertex id in a specific partition is always the first processing step and applying a columnar storage layout leads to reduced data transfer during this step. Furthermore, a vertex is stored as an edge collection, and thus a partition can also contain a *virtual* vertex, i.e. a vertex, which does not have any outgoing edges, just like `H` in Partition 1. If the intermediate state of a GPM query now binds vertex `H`, since it has an appropriate incoming edge, the system would have to scan Partition 1 for its outgoing edges, but to no avail. Analogous to partitioning the graph, finding a suitable LP-to-socket placement is non-trivial and therefore it can happen, that adjacent partitions are placed on different sockets.

Loading a graph into NeMeSys can be triggered by issuing a command via the *user interface*. Every GPM engine related command is prefixed with the *graph* keyword, and thus a graph can be loaded by issuing `graph load <graphIdent> <options> </path/to/GraphFile/>`. The options part contains switches for building partition internal indexes or setting the initial physical representation to row- or column-wise. Graph data files are expected to be in an N-Triple[1] like format, however we dictionary encode the data with integer values to speed up internal processing. In addition to plain text files, NeMeSys also supports the ingestion of binary formatted files. The content of an example binary file with two edges and 8 bit encoded vertices would look like shown in Figure 3.11.

However, this requires a preceding encoding step, such that an edge consists of 3 consecutive integer values of fixed length. On the other hand, NeMeSys is aimed to be an online processing engine and ingesting binary files is significantly faster than plain text. Therefore the overhead of previously encoding the data is easily amortized, especially when a graph is loaded multiple times over several working sessions.

For the numbers shown in Table 3.3, we generated a synthetic graph with different sizes using gMark [BBC+17]. The table shows the graph sizes in both edge count and approximate data size and the time elapsed for loading graphs of different sizes. The binary

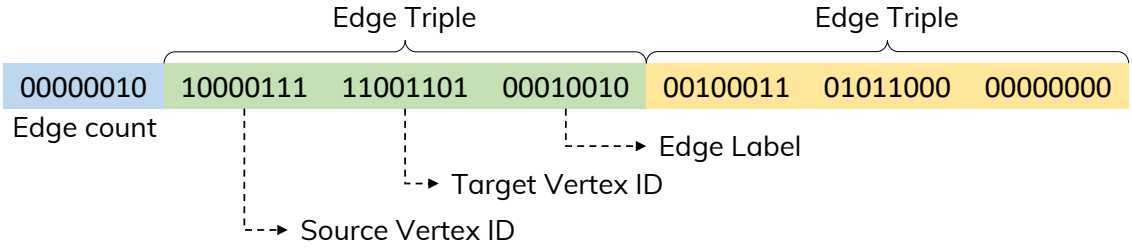


Figure 3.11: Binary graph file format with two stored edges.

[1] https://www.w3.org/TR/n-triples/

Table 3.3: Loading times in milliseconds for different graph sizes and data formats.

		Plain Text		Binary	
Edge count	Size on disk	Row Store	Column Store	Row Store	Column Store
56 k	725 kB	18	16	14	19
562 k	8 MB	178	175	114	123
5 M	93 MB	1759	1752	975	1009
56 M	1.1 GB	17920	18081	9580	9574
569 M	12 GB	187649	184868	94164	94108

graph files are in fact larger, since characters in plain text files require only 1 byte per character, whereas the binary format requires 4 bytes per *integer*. Consequently, every integer, whose decimal representation requires less than 4 digits, requires more space in the binary format than in a plain text file.

The experiment shows three important observations. First, we can observe a linear scaling for ingesting graphs with varying sizes. That is, loading a 10 times bigger graph yields a 10 times longer loading time, meaning NEMESYS does not incur any overhead for loading graphs with more edges. Second, reading from the binary format is expectedly faster, than reading from plain text files with a speedup of approximately 2X. Third, there is virtually no overhead in storing a graph in either row or columnar representation.

3.3.2 Query Generation

NEMESYS supports CQs as described in Section 2.2. Forming CQs can be naturally done following the prevalent triple notation, which is also used in the graph representation itself. To post a query, users write a triple-based query string to the user interface, which will then be further processed by the `GraphQueryManager` class. Queries are posted as a set of white space delimited triples and are of the form [<source>,<target>,<label>], where source and target describe vertex ids and label represents the value for an edge label. NEMESYS accepts fixed values and variables for the vertex part or fixed values and wildcards for the label. Query strings are dissected into a set of edge predicates (triples) and for every edge predicate, the required operator is determined.

Operators are defined via the C++ programmable interface provided by ERIS' underlying infrastructure and we argue, that only three operators are necessary to process GPM on our targeted hardware. Reading a query string triple wise from front to back yields an initial binding order and thus known and unknown bindings at a given processing step. We identified that, based on the order of variables in a query, we can have 0, 1 or 2 bound variables for any given edge predicate and thus the following three operators emerged:

Scan Operator. The Scan operator performs a parallel vertex scan over all partitions in the case that the source as well as the target vertex of a CQ triple are unknown. By specifying a certain edge label predicate, the operator returns only bindings for vertices, where the connecting edge is labeled accordingly. The `Scan` operator is always the first operator in the pattern matching process. As a straight forward optimization step, the Scan operator can be fused with the directly following VB or EB operator to create a processing pipeline.

Table 3.4: Operator assignment based on variable bindings in a query triple. A vertex is considered bound, if it has been matched in a previous edge predicate or if a constant value has been set accordingly.

No. of Bound Vertex Variables	No. of Unbound Vertex Variables	NeMeSys Operator
0	2	Scan
1	1	Vertex-Bound
2	0	Edge-Bound

Vertex-Bound (VB) Operator. The `VB` operator takes an intermediate pattern matching result from either the Scan or the EB operator as input and tries to match new vertices in the query pattern according to the following CQ triple. The operator has to be only applied when either the source vertex or target vertex is known in the current processing step and thus bound.

Edge-Bound (EB) Operator. The `EB` operator ensures the existence of additional edge label predicates between known vertex matching candidates for certain vertices of the CQ. It performs a data lookup with a given source and target vertex as well as a given edge label. If the lookup fails, both vertices are eliminated from the matching candidates. Otherwise the matching state is passed to the next operator or is returned as final result. In this case, both vertex variables are bound.

Considering the GPM query example from Figure 2.5a, we identify five edge predicates, as shown in Figure 3.12. According to Table 3.4, this sequence results in the operator chain: **Scan** → **EB** → **VB** → **EB** → **VB** → **EB** → **Result**. From a relational database point of view, GPM can be considered as a self join on the triple table. Hence, during query generation, we require triples to be ordered such that no two subsequent edge predicates have two unbound vertices. Otherwise the system would perform an unnecessary full join, with a prohibitively large intermediate result set.

After parsing the query, the operators have to be parameterized and instantiated. At this point, we know how many different variables have to be bound and thus we can prepare intermediate state structs and vertex lookup orders. We explored two different approaches, one being skeleton based and the other one being based on online instantiation of C++ class operators. Naturally, we implemented the three operators in C++ with the appropriate hooks for source and target vertex ids as well as to be matched labels. This approach leads to a plethora of branches during the execution, since one operator has to check every possible order and position of a bound vertex in the intermediate matching state. To overcome the extensive branching, we further implemented C++ skeletons as template files. When generating the query, we can then analyze and precompute the positions of the vertices in our intermediate state structs and directly insert them into the

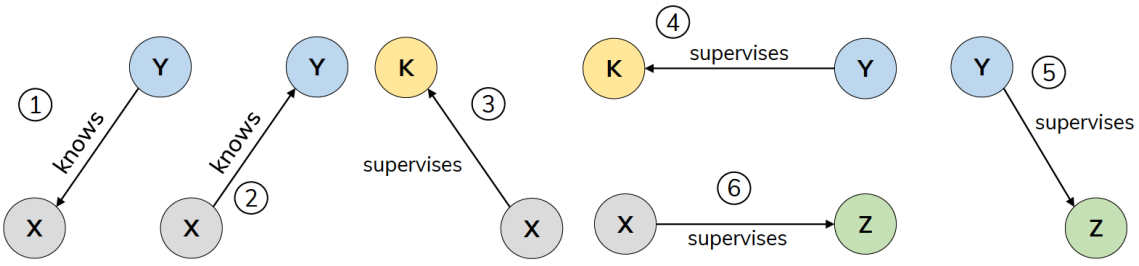


Figure 3.12: Edge predicates for the query from Figure 2.5a.

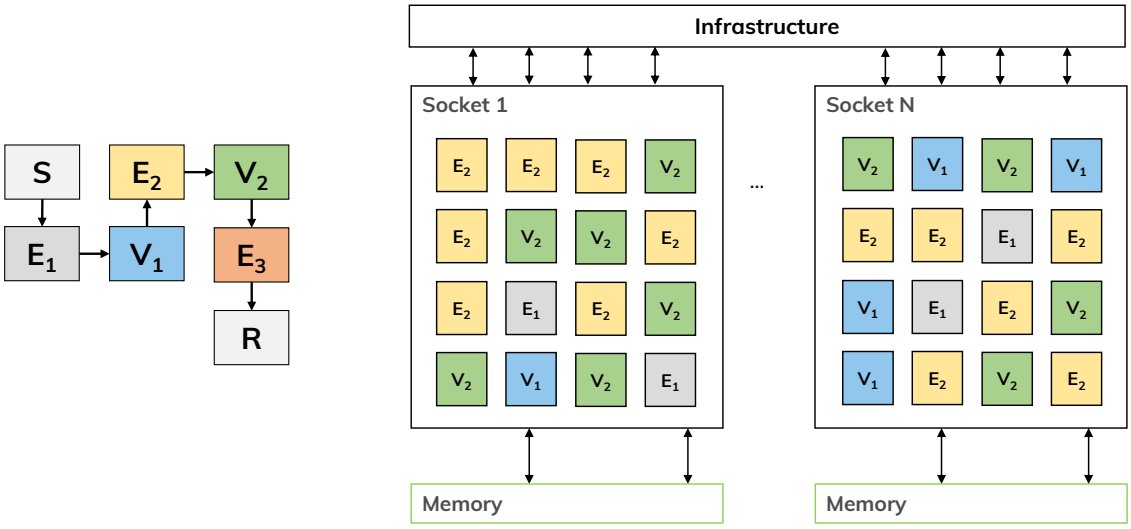


Figure 3.13: Operator placement during a GPM process.

skeleton. However, this code has to be explicitly compiled during runtime by calling a compiler to create a runnable query object, i.e. a shared library, which is then linked into the address space of NEMESYS and finally executed. Obviously, submitting a system call to a standard compiler yields too much overhead to be considered a viable option, and thus Just-in-Time (JIT) compilation could be an option. We tested different variants and found, that the wall clock time for instantiating the operators with branches and subsequently executing the query was consistently faster than reading the operator skeletons from disk, parameterizing and compiling them and finally executing a branch-free version. With the online instantiation of C++ class based operators, we identified further optimization potential by rewriting the operator code as fully templated C++ classes. However, this measure is out of scope of this thesis and left for future work, since we could only optimize parts of the objects, where boolean parameters are used.

3.3.3 Processing Model

As outlined in Section 2.3, we target an asynchronous model to allow for maximum scalability, and thus the Parallel Operator Execution (POE) model emerged. In NEMESYS, we assign a dedicated worker thread to all logical cores of every socket in the system. Just like in ERIS, we restrict worker access to the socket local memory domains and thus communication, i.e. exchanging intermediate results, is forced to be messaging based over the infrastructure layer.

Workers periodically cycle through an event loop, which accounts for sending out pending messages and processing incoming messages, i.e. grabbing a partition and extracting the relevant data, according to the operator code, which is sent accompanying intermediate data from any previous operator. On a physical level, the messages are structured according to Figure 3.8. After building up an operator chain, such as the one consisting of six operator objects for Figure 3.12, processing starts by forking the scan operator code to all workers in the system. Thus, every worker is executing a private copy of the operator code, allowing for independent operations. Usually, data partitions contain varying amounts of relevant data for a given query, which leads to different runtimes for the same operator, based on the worker or partition it is executed on. The eponymous effect of the parallel operator execution is shown in Figure 3.13, where we present a possible operator placement for the operator chain resulting from the edge predicates of Figure 3.12.

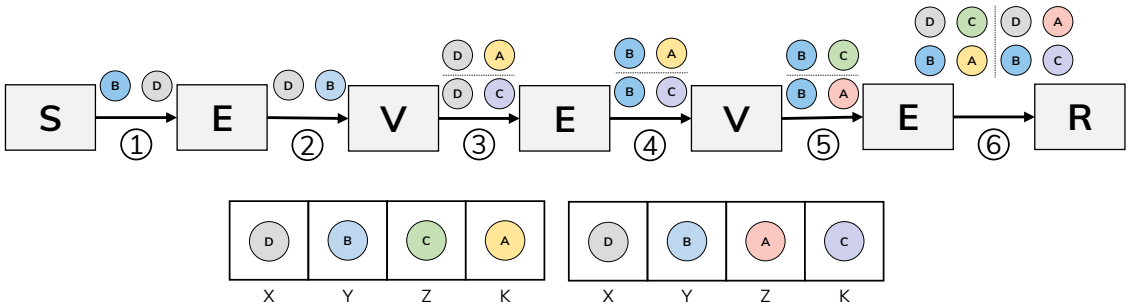
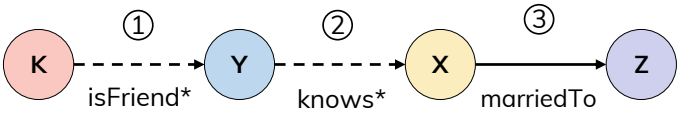


Figure 3.14: Matching sequence for the query and graph from Figure 2.5.

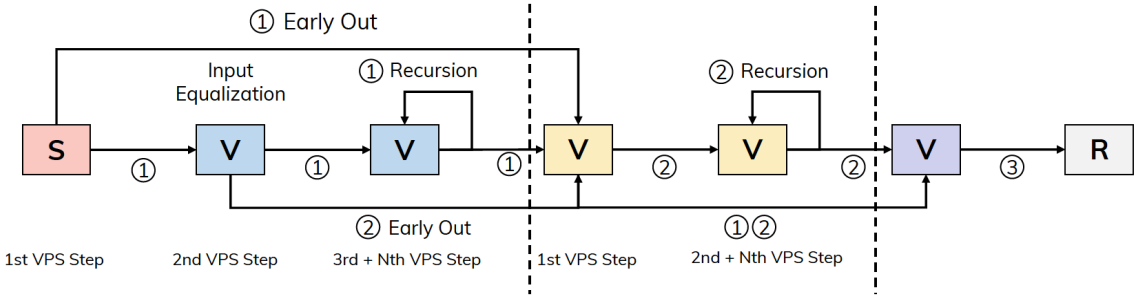
During its execution, an operator matches intermediate results from previous operators according to the edge predicate, which is assigned to it, and sends out new intermediate states to the succeeding operator in the chain, until final matching states are formed to a result. Figure 3.14 illustrates the matching process for the query and graph from Figure 2.5. First, the Scan operator fetches all edges for the first direction of the *knows* relationship and assures the existence of a mutual edge in return during steps (1) and (2). After that, the engine identifies two different matching possibilities for query vertex K. Both data graph vertices A and C qualify for that position and thus, a second intermediate matching state is created during step (3). For both states, A and C are matched to query vertices Z and K respectively, leading to two results during step (6). However, either state can be regarded as a duplicate of the other, since both results contain the same vertices and are therefore a candidate for proper duplication handling at the end of the processing. We exclude the duplication of states for swapping the matchings for query vertices X and Y for brevity. Semantically, the messages are flowing from one operator to its successor during steps (1) through (6) but in reality, operators enqueue messages with partitions, respectively their message buffers. This asynchronicity also allows us, to return first results even before all operators finished, although this approach inhibits the process of duplicate elimination or sorted output.

Besides processing CQs, NEMESYS also supports a deviation of RPQs, i.e. Variable Path Length Queries (VPQs) as a proof of concept implementation for recursive query answering within the NORAD architecture. Figure 3.15a is a VPQ, which asks for a married couple and all people who know one of them, if there are any, and their friends or friends of their friends, if they have any. The resulting operator chain is shown in Figure 3.15b. The arrows in the operator chain indicate message paths between operators and the circled numbers represent the messages, which target the accordingly numbered step in Figure 3.15a. Like in RPQs, our VPQ implementation supports the Kleene star, which requests zero to infinity iterations, as well as the $+$ notation, which requests at least one to infinity occurrences of the predicate to be present. This query is in fact a hybrid of a CQ and a VPQ, since step (3) is a basic edge predicate, but we call any query a VPQ, as soon as it contains a predicate with any recursion indicator. Such statements will be called a variable path statement (VPS) henceforth.

In contrast to CQ parsing, VPQs lead to more operators than actual edge predicates and thus VPQ processing yields three major differences when compared to CQ processing. First, for every VPS, two operators are instantiated. This is a basic runtime optimization for reducing the branching within the operators. The first VPS step has to consider sending messages according to the actual recursion indicator. For example, the Kleene star allows, that the predicate does not even have to be present in the final result and thus, we can skip e.g. from the Scan operator, which is part of the first VPS step of (1), directly to the first VPS step of (2). We call this measure an *early out*, since it allows to leave the operator compound earlier than usual and which is the second difference to CQ operator chains. An early out is added for every first operator of a VPS. The third difference is the



(a) VPQ statements.



(b) VPQ Operator chain.

Figure 3.15: A VPQ query visualization.

messaging behavior of the second VPS operator. In contrast to CQs, these operators send two messages for every matching candidate. That is, since every matched target vertex of a VPS can be either the starting vertex of the next iteration for the VPS or the starting vertex of the following edge predicate. To avoid branching in every VPS iteration, we split the responsibility for sending early out messages and recursion messages into two separate operators. However, this comes with zero overhead, since the total amount of messages in the system stays constant. That is because the communication between the two VPS steps can be regarded as one round of recursion. Subsequently, the forked operator code is called by a worker on a batch of incoming messages and thus nothing changes for the execution model itself, despite a slightly longer operator chain. The main challenge in VPQ processing is an efficient state handling within a VPS. An asynchronous execution model hinders the coordinated access to shared structures. In its current state, we protect the operator-local shared lookup structures with scoped locks to synchronize the access from other operator instances. To become fully NUMA-aware, we need to consider replication strategies to make the information equally available for all sockets, but this is left for future work.

Due to the required data partitioning, we do not always have all the necessary information for one query within a single partition and thus it is inevitable to communicate intermediate results between workers. In general, the message passing is handled by the infrastructure component (cf. Figure 3.10), which hides the latency of the communication network. Locally, within a partition, we employ indices to speed up finding the right vertices. On a global level, NEMESYS relies on a *routing table*, to keep track of locality information for all vertices. This component is part of the LGC and stores information about the corresponding NUMA node and data partition for every vertex in the stored graph. Therefore, the design of the routing table needs to be carefully considered, since graphs tend to exhibit a high number of vertices and fast, parallel lookups are necessary for high performance. Since the *routing table* and any employed *partitioning strategy* depend on each other, we consider the following two design options:

Compute Design. The compute design is a combination of a hash function as *routing table* and a locality-agnostic *partitioning strategy*. Hash partitioning is easy to compute, because it only needs to consider the id of a vertex to assign it to a partition. This implementation calculates the target partition on the fly and does not use any additional data structures. Nevertheless, due to hashing being a trivial approach, partitioning is performed without any topology-based locality information.

Lookup Design. The lookup design is the opposite to the compute design and is a combination of a lookup table – instead of a hash function – as *routing table* and a locality-aware *partitioning strategy*. The routing table is represented as a hash map, that contains a one-to-one mapping of all vertices of the graph to their respective partitions. Thus, we precompute a graph partitioning, which considers the locality of a vertex' neighborhood. This approach leads to a routing table, which is as big as the number of vertices, because we need to store the partition for every single vertex in the graph. We employ this design for any partitioning strategy, that produces a per-vertex partition assignment.

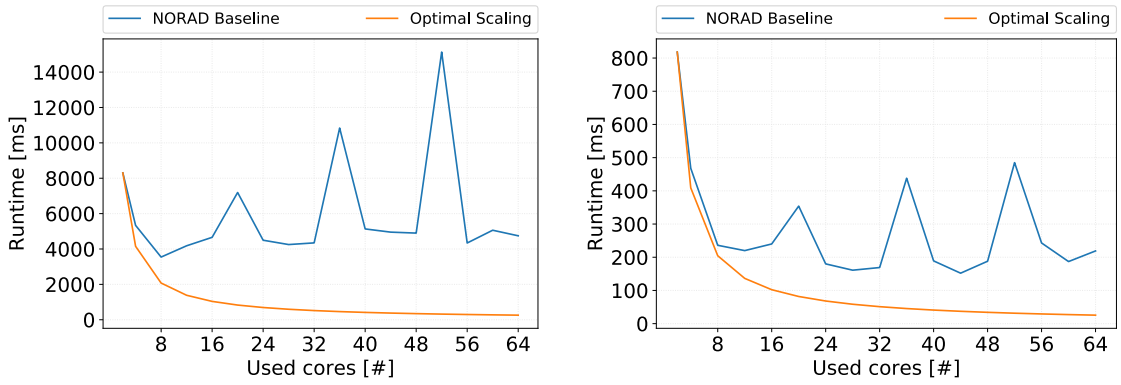
Both, the *compute* and the *lookup design* face advantages and disadvantages. On the one hand, the *compute design* is the fastest implementation for a routing table but lacks the ability to consider graph properties like locality or semantic relationships between vertices, to create well-balanced and locality-aware partitions. On the other hand, the *lookup design* is able to exploit such graph properties, which comes at the price of an additional storage overhead. Due to the size of the routing table being proportional to the number of vertices, it can easily exceed the cache size of the sockets. This leads to an increasing amount of NUMA accesses from many workers, because the routing table itself is stored on a single socket. The major advantage of the *lookup design* is its consideration of locality. As mentioned earlier in Section 3.3.1, edges between partitions create messaging paths between the two. This fact can become a considerable issue, if the neighborhood of such a border vertex is spread among partitions, which are placed on distant sockets, since sending messages between sockets is always slower than keeping the communication locally on the same physical socket.

3.4 CHALLENGES OF GRAPH PATTERN MATCHING ON NORAD

Employing a system like NeMeSys fundamentally allows for highly scalable processing by design. However, although the baseline implementation exhibits high scalability for relational workloads, it does not necessarily perform equally well for graph workloads. We analyze the general properties of NeMeSys and demonstrate, that graph data inherits a set of custom properties, which are not fully covered by relational processing and thus, more sophisticated approaches are necessary. For comparability, we generated three different synthetic datasets, again leveraging gMark [BBC+17]. These represent a bibliographical network, a social network and a protein network, which we call Biblio, Social and Uniprot henceforth for brevity.

3.4.1 Holistic but compact locality metadata for scalable GPM

In this thesis, we refer to scalability as a proportional increase in processing capabilities, according to the amount of invested resources, i.e. using double the amount of workers should result in reducing the query runtime by factor two. To test the natural scaling behavior of a NORAD system for GPM, we used the Biblio and Social graph. We generated a set of different queries out of their respective schemes and ran them against the graphs. In Figure 3.16, we present the results for queries on the Biblio and Social graph. The experiments were run on the *Small* server from Table 3.1, a 4 socket NUMA server with 8 physical cores per socket, which in turn provide 2 logical cores. Our benchmark server thus has 64 cores in total and a sufficient amount of main memory to accommodate the graphs, all indexes and the queries intermediate results all in-memory. We measured the baseline performance with 2 active workers, increased the worker count to 4 and then



(a) Biblio, Compute Routing Table.

(b) Social, Lookup Routing Table.

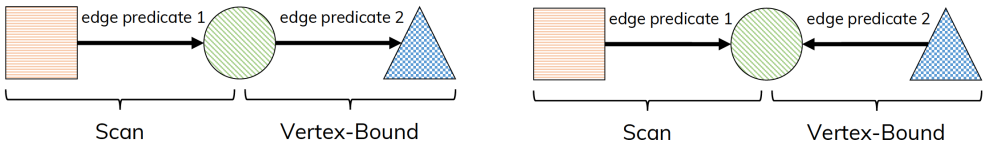
Figure 3.16: Runtime scalings with increasing worker count on different graphs.

subsequently added 4 more workers, i.e. 1 per socket, per measurement. The optimal scaling was calculated based on the baseline performance, divided by the amount of additional cores.

Although exact runtimes differ for distinct queries, we observe the shown general trends and thus report two representative examples. Figure 3.16a shows a completely non-proportional trend, where adding cores first yields a general performance improvement. Increasing the invested resources leads to at least stable performance, but also can cause a general slowdown. In contrast, Figure 3.16b we can observe a trend towards the desired behavior. Whilst the first example should not be considered for scaling factors, we observe a speedup of factor approx. 3.7 x for the second one. In an ideal case, scaling the resources by a factor of 32 should result in a somehow similar speedup. We identify multiple reasons for the hindered scalability, which can be tracked down to both hardware and the NORAD architecture. First, from a systems point of view, we acknowledge the negative impact of hyperthreads (HTs), i.e. scheduling two workers on both logical cores of the same physical core. The general idea of HTs is to provide excess resources when using I/O heavy applications and thus outsourcing these operations with usually long latencies to the HTs instead. However, scheduling two compute intensive tasks on the same physical core leads to worker starvation and often eventually to an overall performance degrade, since both logical cores have to share resources, e.g. local registers. This issue can be easily mitigated by simply not instantiating more workers, than there are physical cores on a socket.

On the NORAD system architecture side, we identify the infrastructure layer as a key component in scalable GPM processing. The messaging procedure heavily relies on exact information about where to send specific requests. If this information is not available, the system defaults to broadcast messages. In contrast to computer networks, where all devices listen on a certain broadcast address and feel responsible, whenever a broadcast is sent to that specific address, we need to perform some additional steps when creating a broadcast. As outlined in Section 3.2.3, messages are queued with their respective target partition. Whenever a broadcast is sent, the `NodeCoordinators` are responsible for duplicating the received broadcast and placing a copy in the queue of every partition on their socket. Thus, sending a broadcast is always as expensive as the number of data partitions in the system.

The broadcast problem has two root causes. First, the baseline prototype implements only basic data partitioning primitives. To optimally balance the data among partitions, the original routing table defaults to distributing incoming tuples in a round robin (RR)



(a) Forward oriented query. (b) Mixed oriented query.

Figure 3.17: Different edge predicate orientations in GPM.

fashion, but without keeping track of the actual data placement. Whenever a lookup request is posted against an RR-based routing table, it can only be answered by returning a broadcast indicator. This is even more worse for graph data. In our data model, a single vertex is composed of a set of edge triples and thus, the adjacency of a single vertex is always distributed among multiple partitions. The issue of missing locality information can be mitigated by using our proposed *Compute* or *Lookup design* routing tables. Considering the storage layout from Table 3.2, we can keep track of the location of every vertex by simply hashing its id or adding it into a lookup table.

Since the topology of a graph is inherently encoded through directed edges, we do only have to store outgoing edges for every vertex to represent the graph (cf. Section 3.3.1). Based on this design decision, we can only efficiently process a specific class of GPM queries, with an example being given in Figure 3.17a. The figures shows an abstract example CQ with two edge predicates, searching for all rectangles, which have an outgoing edge to a circle, which in turn have an edge towards a triangle. The rectangle, circle and triangle are abstract and thus unbound vertices. Answering such queries can be effortlessly done by looking up first all squares in the Scan operator and subsequently looking up all outgoing edges for every previously matched circle. There is no overhead in this process, since per default indexing has been done for source vertices and all outgoing edges of that source vertex within the routing table. However, the queries like the example from Figure 3.17b result in the undesired scaling behavior as previously shown. The query differs from the first example by only inverting the direction of the second edge predicate, which leads to two follow up issues. First, the outgoing edge table only indexes source vertices and thus, requesting the source vertex of an edge with a known target vertex can not be answered. Hence, a broadcast is once again sent out, impairing the systems performance. Second, no single partition can be identified as target for the next operator. Considering vertex C from the original edge table (c.f. Table 3.2), we would receive two different partitions. However, sending a multicast message, i.e. a message targeting a specific subset of data partitions in the system, is not supported by the baseline system and thus, a broadcast is again necessary.

Challenge 1: We identify sufficient data locality information as a key aspect for scalable GPM processing. Being able to directly address the correct target partition drastically reduces the load on the memory subsystem, since only a small number of messages have to be sent. Not knowing which data partition should receive a specific message leads to contention on the memory bus, since broadcasts are multiplied by the number of partitions in the system. For e.g. 64 data partitions, this could already lead to $7.07\,\mathrm{M}$ messages in the system, when a query with three operators creates only three broadcasts per operator per received message. In addition, messages targeting inappropriate partitions lead to local workload overhead. That is, since every message has to be processed, it leads to a local lookup routine in the partition. If there is no local index on the required data, this lookup deteriorates to an even more expensive scan, which in turn means an even more severe performance drop, i.e. unnecessary work.

Challenge 2: The routing table is a key component for performance, since both its content as well as its overall size contribute to efficient query performance. While the content of

the partitions is dominated by the employed partitioning algorithm, the size of the routing table lookup structure itself can be influenced by the concrete implementation. Since this data structure is accessed for each and every message, it should be cache resident and thus rather small. This requirement actually prohibits a direct vertex-to-partition mapping, since a graph with e.g. $1\,\mathrm{M}$ vertices requires at least $8\,\mathrm{MB}$ to store only the $64\,\mathrm{bit}$ vertex ids, let alone data structures to hold both vertex and partition information. Considering current server processor L3 cache sizes of approx. $22\,\mathrm{MB}$, the routing table would be at least partially evicted during query processing. Thus, a performance degrade is inevitable when fetching target partitions during the messaging procedure.

3.4.2 Proper data placement and data allocation

Leveraging NUMA systems requires a thoughtful system design to fully exploit hardware capabilities. As outlined in Figure 3.2, such systems yield decreasing bandwidths and increasing latencies, the farther away a targeted NUMA domain is located. Table 3.5 shows the memory statistics for two of our benchmark servers, with Table 3.5a being a fully connected four socket server and Table 3.5b being only a mesh connected four socket server, i.e. communication between distant hops has to be coordinated by a mediator core. Considering the local bandwidth, we see a slowdown of factor 6 in Table 3.5a, when performing a write access to a remote socket. In NEMESYS, data partitions are placed on a single socket an can solely be processed by workers, which are pinned to cores of the very same socket and thus at least remote access of data is avoided.

Figure 3.18 shows an example of a partitioned graph with the dashed lines illustrating communication paths between the vertices within the partitions. The vertices from the blue partition are rather tightly connected to the vertices from the grey partition, however they are placed on different sockets. Consequently, graph patterns, which target vertices of the blue partition are likely to generate a lot of remote memory accesses. The content of each data partition is determined by the employed graph partitioning strategy. However, the actual placement of the data partitions is governed by the system itself. The NORAD baseline system implements a straight forward linear assignment strategy, i.e. distributing 64 partitions on a four socket server means partitions 1 through 16 are placed on the socket with the physical id 0, partitions 17 through 32 on the socket with physical id 1, etc. This approach is feasible when dealing with relational data, which is range partitioned and thus evenly distributed among all partitions. However, graph data exhibits semantic connections between vertices and thus, their actual placement in relation to each other matters for GPM performance.

To highlight the general weakness of this standard range (SR) partitioning baseline, we ran queries against the Social graph. One instance of the graph was partitioned with

Table 3.5: Server bandwidth matrices for the *Medium* server (4x Intel Xeon Gold 6130) and a comparable platform (4x Intel Xeon Gold 5120) with different connections.

(a) Fully connected bandwidths in GB/s.

	NUMA Nodes			
	0	1	2	3
0	108.2	17.2	17.2	17.2
1	17.2	107.5	17.1	17.2
2	17.2	17.2	108.6	17.2
3	17.2	17.2	17.2	108.5

(b) Mesh connected bandwidths in GB/s.

	NUMA Nodes			
	0	1	2	3
0	65.9	17.1	10.6	16.1
1	17.0	67.8	16.4	10.4
2	10.6	16.1	67.4	17.0
3	16.5	10.4	16.9	67.5

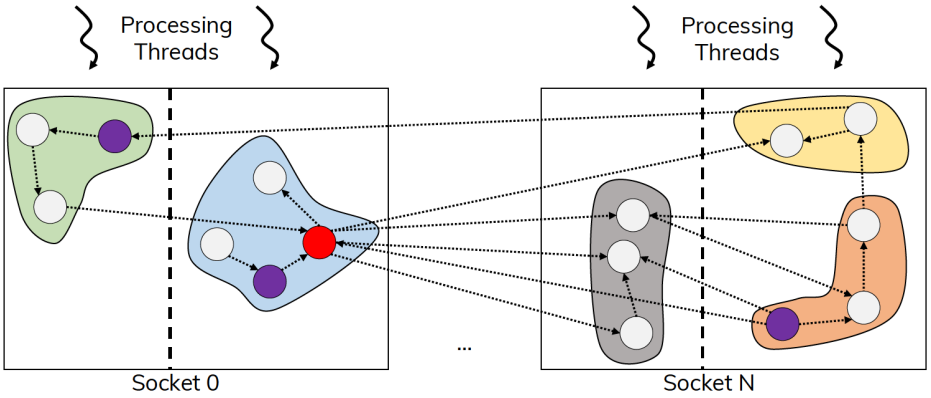


Figure 3.18: Graph partitions with communication paths.

the SR algorithm, and a second instance was partitioned with a more graph oriented approach, which we call Balanced Edges (BE). During the graph ingestion period, the BE algorithm counts the amount of edges stored per socket and places the next vertex on the socket with the currently lowest amount of edges. We ran the same queries against both instances and present the results in Figure 3.19. Because of the deviation of the runtimes between individual queries, the figure shows relative performance, with SR being the baseline of 1 We observe a considerable performance gain in the upper part of the figure, just by applying a partitioning algorithm, which heuristically evens the amount of edges stored per socket. With the SR algorithm, every socket receives exactly the same number of vertices, but the number of edges per socket differs. Balanced Edges may lead to unevenly distributed vertices among all sockets, but the number of edges is approximately the same. This is especially reasoned and mirrored by the amount of messages, which are processed per socket and which is shown for each query in the corresponding lower part of the figure. Such imbalances can happen, when the graph has been encoded on an entity level, e.g. first all *human* vertices are numbered, then all *city* vertices, etc. As presented in Table 3.5, local communication has the highest possible bandwidth and thus, having a maximum of local communication is a desirable optimization goal. However, using only one socket greatly limits the systems parallelism, since in the given example only a maximum of 25 % of all available compute resources could then be utilized.

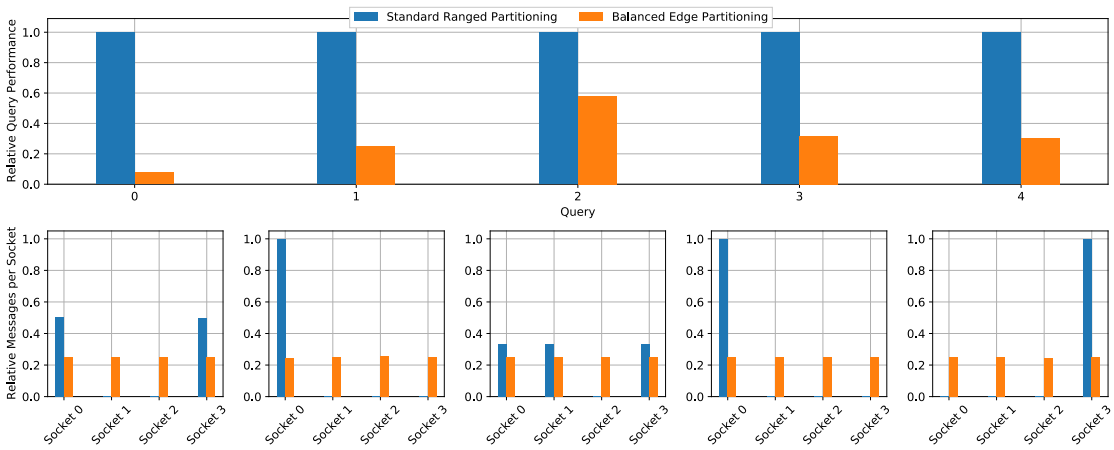


Figure 3.19: Comparing a relational partitioning approach against a graph partitioning algorithm by relative query performance and messages processed per socket.

Challenge 3: Data partitioning as well as data placement are important to consider for efficient GPM processing. Whilst the former is a direct result of the employed partitioning strategy, the latter can also be adjusted by concrete allocation strategies. Furthermore, the partitioning algorithm also indirectly influences the communication paths, depending on the vertex-to-partition assignment. The aforementioned GPP is an NP-complete problem and thus can not be optimally solved for bigger datasets. Distributing the data in a generally well performing manner can thus only be solved heuristically but we also acknowledge that there is no one-size-fits-all strategy. We argue that deciding between a fast and locality agnostic algorithm like hashing versus any graph-oriented partitioning algorithm is non-trivial and should be done on a per-graph basis, if not on a per-workload or even a per-query basis. Furthermore, placing adjacent partitions on the same sockets is also beneficial for parallel GPM, to fully exploit the bandwidth characteristics of the system. Identifying communication paths among a set of partitions for a given workload is thus a considerable problem, which we also tackle in this thesis.

4

NEAR-MEMORY GRAPH PROCESSING ON SYMMETRIC MULTIPROCESSOR SYSTEMS

The previous chapter outlined the aspects of our targeted hardware, as well as the properties and capabilities of the baseline implementation. In this chapter, we will cover our adjustments to said baseline implementation, to allow NEMESYS becoming a near-memory graph pattern matching engine, which is based on the novel NORAD architecture. We will thoroughly explain the missing features, which are required to allow scalable processing with adequate query response times. Furthermore, we will present relevant implementation details on the individual components. Since we focus on SMP server systems, we need to consider several aspects, which we consider to be of fundamental importance to efficient graph pattern matching. First, we improve the general query infrastructure by applying graph-specific query optimizations. Following that, we investigate the influences of workload dependent system configurations, i.e. the effect of data partitioning and the number of effective partitions, paired with varying worker resources. After GPM processing is enabled by well-ordered query triples and adequate partitioning, we thoroughly investigate measures to improve the overall messaging behavior in the system. We then further examine the influence of data locality by not only managing the data partitioning, but applying different data allocation strategies, based on the inter-partition communication paths to conclude this chapter.

4.1 QUERY EXECUTION PLAN OPTIMIZATION

Optimizing incoming queries is a well researched field. From a database perspective, GPM represents a join-ordering problem. The amount of possible intermediate results per step depends on the order of the subsequently matched statements in the query. Thus, we can leverage state-of-the-art approaches from the database world, like heuristically choosing the most promising join tree, which represents the order of the statements in the query. This ordering is also called query execution plan (QEP). To achieve that, we need to define a suitable heuristic, which estimates the cost of a given statement order and thus allows us to rank the QEPs, based on their expected cost.

As outlined in Section 3.3.2, NEMESYS allows querying based on a triple language. Furthermore, within our processing engine we allow a triple sequence only to be in an order, such that no two unbound variables occur together in another triple than the very first. Compared to our previous example, statements (4) and (6) from Figure 3.12 being the

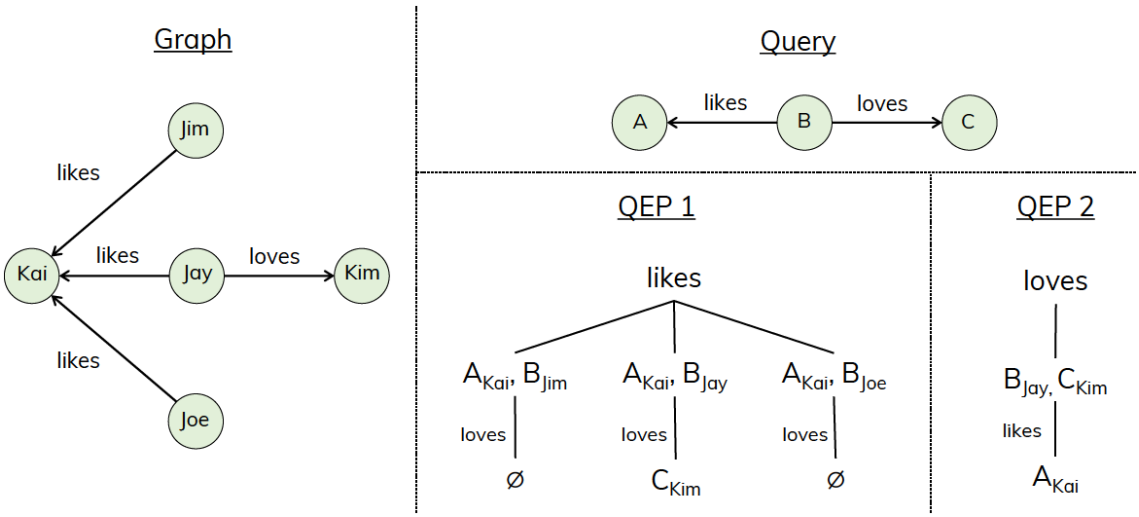


Figure 4.1: Query execution plan optimization on a small graph.

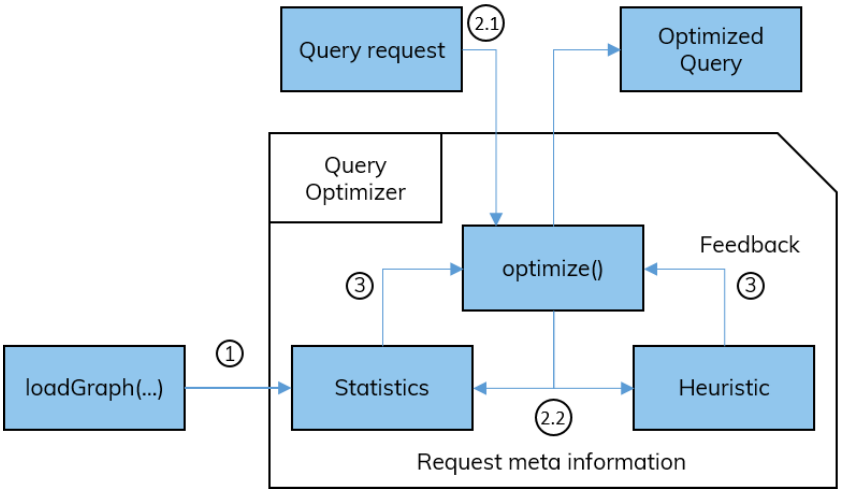


Figure 4.2: Workflow for CQ optimization.

first and second statement in a query would be an invalid query. This measure prevents the matching of any unconnected edge of the graph to an intermediate state of a graph pattern, but it seriously limits the possibility of exploding intermediates.

The general problem is illustrated in Figure 4.1, with a trivial query asking for someone who likes any other person, but in addition loves a distinct person. For the graph given on the left-hand side of the figure, we can create exactly two different QEPs, which are shown on the bottom right part of the figure. In the worst case, the system creates three intermediate results for the first evaluated edge predicate. Thus the system is forced to check all of them, but only one participates in a final result. The best case QEP allows for only one intermediate result after the first evaluation step, which is a third of the worst case plan for this example. Rearranging triples in a way that they produce the fewest amount of intermediates possible is therefore a crucial requirement for scalable GPM. The cost of evaluating an edge predicate can be heuristically determined by the amount of variables in it and whether an edge label or a wildcard is given.

A heuristically optimal triple ordering is produced by our query optimizer, as shown in Figure 4.2. First, during the graph loading procedure (1), NeMeSys collects general graph statistics, like the degrees of the vertices or label frequencies (cf. Section 2.1), for every graph that is stored in the system. Whenever a query is posted (2.1), the query optimizer applies our heuristic to the collected statistics (2.2) and uses the result (3) to reorder the query triples. Our empirically determined heuristic is calculated on a per-edge basis and takes a query triple as input:

$$h(\langle \text{source}, \text{taret}, \text{label} \rangle) = \log(|E|) * S + \theta * T + |E| * L \tag{4.1}$$

First, we need to precalculate the intermediate results for **S**ource, **T**arget and **L**abel accordingly (cf. Section 2.1):

$$\begin{aligned} S &= \begin{cases} \deg_{\text{out}}(\text{source}), & \text{source is constant} \\ |V|, & \text{source is a variable} \end{cases} & \theta &= \begin{cases} \log(|E|), & \text{target is indexed} \\ |E|, & \text{target has no index} \end{cases} \\ T &= \begin{cases} \deg_{\text{in}}(\text{target}), & \text{target is constant} \\ |E|, & \text{target is a variable} \end{cases} & L &= \begin{cases} count(label), & \text{label is constant} \\ \infty, & \text{label is a wildcard} \end{cases} \end{aligned} \tag{4.2}$$

After the appropriate substitutions, the influence of source and target vertices is further multiplied with the general complexity to find the necessary information within the partitions. We always build tree based indexes on at least the source vertex information and

thus, we observe a logarithmic influence of the amount of graph edges for S. For target vertices, we check for available indices and set θ accordingly.

The result of step (3) from Figure 4.2 is a set of weighted edge predicates, with lower scores for fast retrievable edge predicates and higher scores for more expensive ones. The edge predicates are then sorted in an ascending order of their respective scores. This could lead to the aforementioned invalid query state, when e.g. two statements with two unbound vertices each are ordered subsequently. Hence, we apply additional sanity checks to avoid that situation and reorder statements as minimal invasive as possible. As a general rule of thumb, edge predicates with wildcards at the label position are always processed last. That is, since they simply return all edges for a given source or target vertex and thus would unnecessarily inflate the size of the intermediate matching state.

Collecting graph statistics prolongs the graph ingestion procedure as a whole and the amount of collected statistical information has a an increasing impact, the larger the stored graphs grow. Therefore we built our optimizer, such that it can also work without statistics and a reduced rule set. If no statistics are present, our optimizer reorders the edge predicates solely by their number of constants, i.e. edge predicates with constants will always be processed before edge predicates with at least one variable. The same rule is applied to the label part of the edge predicate, with the most weight being put on the edge label portion. A statement with two variables and a constant label is placed before a statement with a known source *or* target vertex but a wildcard for the edge label.

We evaluated the performance of our optimizer with query sizes ranging from 2 to 10 edge predicates. The overall optimization time ranges from $2\,\mu s$ up to $40\,\mu s$ for all query sizes. The variation stems from queries with either more or less complex statements, e.g. when variable names consisted of longer strings like *author* than just the single letter *a*. Considering that GPM queries usually run longer than a couple of milliseconds, we consider this overhead to be negligible. However, we do not yet cache optimization results. Hence, if a user enters a query, which was previously seen and optimized, we would still run a whole optimization pass again. It is yet to be determined, if cached results could also be applied to semantically equivalent queries or if testing two queries for similarity is more expensive than just rerunning the optimization itself. However, further improvements of the optimizer are deferred to future work.

4.2 TOPOLOGY-BASED OPTIMIZATION

From a software point of view, an SMP system with its globally coherent main memory is no different to any other computer. Worker threads are executed by any processor with free capacities and requested main memory is provided through standard `malloc` calls. Thus, the parallel processing of graphs can be done with the same software on either a commodity laptop with a dual core processor or a server with numerous multiprocessors without the need to adapt the internal software architecture. Such hardware oblivious approaches can never achieve their best performance on highly parallel SMP systems, as they completely miss out on NUMA awareness, which we described in Chapter 3.1. Achieving appropriate performance gains on a system with multiple memory domains thus requires intelligent data partitioning. The topology of a graph is defined by the direction of the edges between the graph's vertices. Traversing edges implies a message being sent from the partition of the source vertex to the partition of the target vertex. If the source and target partitions are placed on the same socket, this communication is considerable faster due to higher local bandwidth, as shown in Table 3.5a. Thus considering the topology of the stored data graph is a crucial aspect for efficient graph processing on an SMP systems.

In this section, we elaborate on how to optimize NEMESYS, based on said topology, i.e. the *storage layer* of our system. Figure 3.19 already showed, that graph agnostic partitioning can not achieve the same performance, as a simple graph partitioning algorithm. Chapter 2 introduced different types of graphs, like road networks or protein interaction graphs and each of them exhibits unique topology patterns. Therefore we argue, that considering different strategies for data partitioning is necessary to achieve optimal results for different graphs and even different queries. That is, since every query pattern implies a different set of traversed edges, which may result in different partitions, that contain the relevant data.

Furthermore, storing only outgoing edges may be appropriate to mirror the graph's topology, yet this hinders the communication, since edges can thus only be traversed in their actual direction. This leads to the necessity of broadcasts, whenever a backwards traversal is necessary. The negative impact of broadcasts with an increasing amount of worker threads was already shown in Figure 3.16. We overcome this limitation by introducing redundancy in terms of additionally storing incoming edges. These additional edges increase the locality information for backwards traversals and therefore eliminate broadcasting completely by trading a higher memory consumption for less messages in the system.

4.2.1 Workload Dependent Graph Partitioning

Section 4.1 discussed how the order of query triples could impact the query runtime, based on the resulting messaging scheme. This impact is further influenced by the content of the targeted data partitions and the socket on which a partition is located in the system. In this section, we tackle the required data partitioning by generally classifying graph partitioning algorithms. We will then discuss our approach on graph partitioning for a NORAD based system like NEMESYS. Parts of this section have been considered in our previous work from [KKH+17].

Partitioning a graph is inevitable when processing it in-memory on a NUMA system, if system scalability is a desired goal. However, the GPP is NP complete, as mentioned in Section 3.3.1. Thus, a general default is to rely on hash partitioning and replication, which is also discussed in [Pot17]. Although their work on cluster-based query processing is orthogonal to our in-memory solution, the GPP arises for both systems. Hashing is usually done on either source or target vertices. For RDF representations, which can be also used for Wikidata (cf. Section 2.4), the subject or object part of the RDF triple are usually used as input for the hash algorithm. NEMESYS already allows the implementation of hash based partitioning by providing the *compute design* routing table from Section 3.3.3. The major advantage of hash based partitioning is its low overhead for generating the actual partitioning. The same hash function can be used both for storing the data during the ingestion phase and for the lookup procedure during GPM processing. However, the main drawback is the lack of locality aware partitioning or the amount of cut edges between partitions. Vertex locality, i.e. placing a vertex and its neighborhood on the same partition or socket, leads to more local than remote messages. Considering that the local bandwidth of a NUMA node is always higher than towards a remote node, it is advisable to facilitate more locality aware strategies. The amount of cut edges does not directly contribute to this statement. In NEMESYS, traversing any edge leads to a message. Thus, even inner-partition edges will create forced communication, although these messages are guaranteed to reside on the same socket and leverage maximum bandwidth. Any inter-partition edge is therefore a potential remote communication.

Achieving an appropriate graph partitioning is thus non-trivial and should be carefully considered. The field of graph partitioning is well researched and an ongoing topic. A

well known partitioning algorithm is the multilevel k-Way partitioning, which was originally introduced by [KK98]. It addresses the problem of minimizing the edge cut, thus creating self-contained data partitions. However, for our use case, we do not desire completely self-contained partitions. The fewer edges span between multiple partitions, the more we limit our potential parallel computation. Reasoned by the NORAD architecture, only one worker is allowed to process one specific partition at any point in time, which limits the systems scalability in the case of completely self contained partitions. Thus, we need to define heuristical graph partitioning algorithms, which achieve a balanced partitioning. Generally, *balanced* can mean to achieve either equal vertex or edge count per partition or both, which defaults back to the original GPP. Therefore we first need to classify potential partitioning strategies, which is shown in Figure 4.3. We split our classification in two dimensions:

(1) The *partitioning criterion* represents the basic unit of a graph, that is supplied to the partitioning strategy and on which the actual partitioning is applied.

(2) The *balancing criterion* describing the unit of the graph that is subject to be balanced out between data partitions. In this thesis, balancing is restricted to only one criterion.

Both dimensions can be either fine- or coarse-grained. *Edges (E)* are the smallest achievable granularity, as they are the basic building blocks of a graph's topology. *Vertices (V)*, as a set of edges, are the next larger category, followed by *components (C)* as the coarsest. We define a component as a set of vertices, which are more tightly connected to each other, than to most vertices of the graph. An obvious example for a component in a social network are groups of friends in the university. Usually, people inside of such a group know each other well, but they often do not know much about other students, thus there would exist much more edges between student vertices of the same group, than to students of other friend groups. Hence, a *partitioning strategy* is a combination of a *partitioning criterion* and a *balancing criterion*. Partitioning a graph at a specific granularity limits the balancing to the same level of granularity, or below. For example balancing components is not possible, when the partitioning is based on edges. For this thesis, we designed four different heuristic approaches, following the classification table. However, to the best of our knowledge, there are no known viable representatives for the C/E and C/C strategy.

In the following, we detail on the feasible strategies and describe our heuristic implementations that we use for our evaluation. We restrict our considerations to one representative algorithm per partitioning strategy. Furthermore, we only allow the generation of a disjoint set of graph partitions and consider the replication of individual parts of the graph as out-of-scope for this thesis.

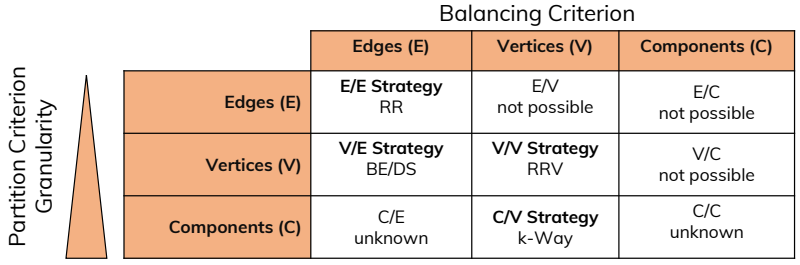


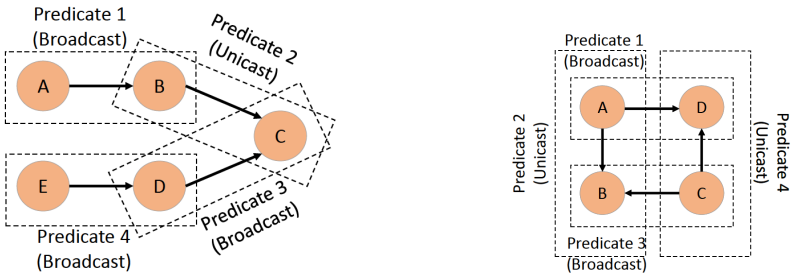
Figure 4.3: Classification of graph partitioning strategies and representative algorithms.

E/E Strategy. This partitioning strategy works on the most fine-grained level and considers each edge triple individually. It is trivially implemented using a *round-robin (RR)* approach, which evenly distributes edges to partitions in a lightweight fashion. This strategy will distribute all edges of one vertex with more than one edge to multiple partitions. Applying the E/E strategy disables the use of any of our previously described routing table designs, since no hash function could project the same source vertex id to a set of target partitions, which are selected during runtime. Thus, processing an E/E partitioned graph can only use broadcasts. This design is only viable, if most of the data partitions hold any relevant data for the processed edge predicate of a GPM query.

V/V Strategy. This strategy partitions a graph by its vertices and balances the amount of vertices per partition. Hence, our *round-robin vertices (RRV)* algorithm is a specific implementation of this strategy, that too leverages an RR approach. When triples are read during the ingestion phase, we keep track of assigned vertices and assign the next not-yet-assigned vertex id to the next partition. If a vertex id has already been assigned, every future edge will be stored in the same partition and therefore we have a guaranteed disjunctive partitions, based on the source vertex.

V/E Strategy. Similar to the *RRV* strategy, the graph is partitioned by its vertices. However, this partitioning strategy balances the number of edges. We consider two specific algorithms as implementation of this strategy: *balanced edges (BE)* and *distributed skew (DS)*. Both algorithms sort the vertices by the number of edges in a descending order. The BE algorithm iterates over this sorted list and assigns each vertex and all of its edges to the currently smallest partition to greedily balance the edges across the partitions. The DS algorithm is a state-of-the-art approximation for handling skewed data in distributed joins [CKWT14] and extends the BE algorithm. To diminish the influence of vertices with high degrees, DS considers the degrees of all vertices and divides them into two sets. One set contains all vertices up to a given threshold degree and distributes them according to the BE strategy. The second set contains all vertices, whose degree surpasses the threshold. Such vertices are more likely to create skewed load on the one partition, which they are assigned to. This is remedied by evenly distributing *all* edges of this vertex among all partitions, i.e. following the E/E strategy for only those vertices. Therefore, performing GPM processing on a DS partitioned graph leads to a mix of unicast and broadcast messages, even for solely forward-oriented queries like shown in Figure 3.17a. Because most real world graphs exhibit a non-uniform edge per vertex distribution, all vertex-oriented partitioning strategies (RRV, BE and DS) lead to different partitioning results.

C/V Strategy. The goal of a component-oriented strategy is to achieve a maximum of local communication, by storing naturally connected groups of vertices together in a partition. Identifying such groups can be either done by well-known graph traversal algorithms to identify strongly or weakly connected components or by leveraging traditional graph partitioning, which minimizes edge-cut between partitions. The previously mentioned *multilevel k-Way* algorithm can be regarded as state-of-the-art to achieve that goal. This algorithm creates partitions, which are generally self-contained and exhibit only few inter-partition edges, compared to the other partitioning strategies. Thus, we select the k-Way algorithm as representative for the C/V category. In this thesis, we use the k-Way implementation from the METIS library 5.1 [KK13]. Similar to the V/* strategies, we store all edges of a vertex in the same partition to avoid broadcasts during the pattern matching process.



(a) *V* Query. (b) *Quad* Query.

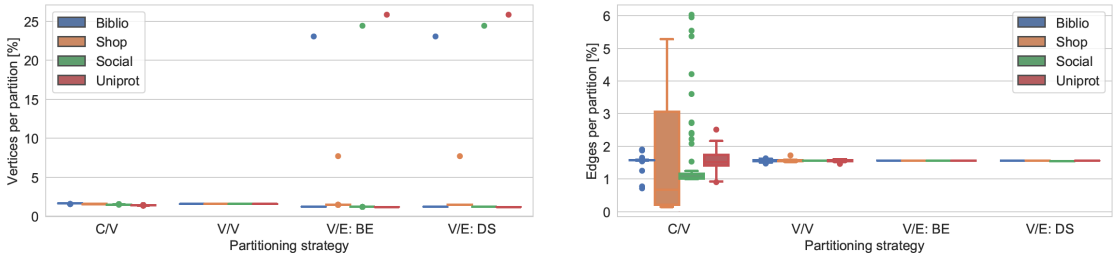
Figure 4.4: Evaluated query patterns.

A straight forward partitioning approach is simply hashing a vertex by its id. Assuming vertex ids to be equally distributed, we could fit hashing into a V/V strategy, however hashing is agnostic to the graph topology and is thus omitted in this section. Besides using a single strategy, we see the potential of hierarchical graph partitioning, i.e. a combination of a C/* strategy together with the others. A possible combination would be to leverage k-Way partitioning to dissect the graph into as many partitions as the server provides sockets and then further process these resulting super-partitions with any other strategy. However, the synthesis of partitioning strategies is out of scope for this thesis and left for future work, as we want to examine the individual properties for each category.

To investigate how a partitioning strategy influences the GPM performance, we conducted an exhaustive evaluation on a small and large-scale multiprocessor system, which are listed as *Small* and *Large* in Table 3.1. We use four test data graphs, each representing an individual application domain, that are again generated with the graph benchmark framework gMark [BBC+17]. Additionally, we defined two CQs as depicted in Figure 4.4: (1) the *V* query shapes a V with five vertices and four edges and (2) the *Quad* query is a rectangle, which consists of four vertices and four edges. Both queries require four *edge predicate* evaluations, however the *Quad* query leads to an EB operator, which the *V* query does not. We generate these two query types individually per graph and only assign a value to the edge label part of the query, such that all query vertices are variables. Since the label value will be different for each graph, we omit its value in the figure. Based on the query semantics, the evaluation of the edge predicates is performed as follows:

V Query. The first edge predicate evaluation is broadcasted to all partitions, because only the edge label is known and not the source vertex. Since edge labels are not unique within any partition, an initial broadcast is inevitable to start off the GPM processing. The target vertices from this intermediate result set are then used as source vertices for the second edge predicate evaluation. Depending on the partitioning strategy, the second edge request is evaluated using either unicast or broadcast messages, e.g. if DS was used. The target vertices of the resulting matching candidates are now utilized as target vertices for edge predicate 3.

Quad Query. The edge predicate evaluation of the *Quad* query is mostly similar to the one of the *V* query. An initial broadcast is required, because only an edge label is given in the pattern. Edge predicate 3 is the only backward oriented edge in the pattern and thus necessitates a broadcast. The fourth edge predicate can again be processed with solely using unicasts, if DS is not used.



(a) Vertex distribution.

(b) Edge distribution.

Figure 4.5: Partitioning results for 64 partitions.

For all of our experiments, we loaded the graph-under-test into the main memory, partitioned it, and evenly distributed the partitions across the sockets and executed both pattern queries for all partitioning strategies and all possible *system configurations (SCs)*. In this thesis, a system configuration denominates a combination of the active workers and the total number of partitions. We repeated each experiment 20 times and calculated the average over all runs.

Figure 4.5 shows an overview of the partitioning results for the different strategies on our test graphs. Since the *Small* server exhibits 64 hardware threads, we split the graphs into 64 partitions. The boxplots show the deviation of vertex and edge distribution among the 64 available partitions. Narrow lines for some strategies represent the median, as no real deviation could be observed. From these plots, we can derive the following observations:

(1) The partitioning and balancing criteria of the respective strategies are fulfilled independently of the graphs. For instance, our RRV algorithm from the V/V strategy partitions the graphs by vertices and ideally balances the vertices among all partitions, i.e., the vertices are evenly distributed over the partitions as depicted in Figure 4.5a. The same applies for BE and DS of the V/E category, which perfectly balance the edges among the partitions.

(2) Depending on the strategy, balancing is done either by vertices or edges. This can lead to an imbalance on the non-balancing criterion depending on the underlying graph. For instance, BE and DS balance edges among partitions. However, there are few partitions with a much higher number of vertices than the others, which lead to the outliers in Figure 4.5a. The V/E strategies create vertex imbalances on all partitions, whereas the V/V strategy leads to slight edge imbalances across the datasets.

(3) The k-Way algorithm partitions graphs by components and balances the vertices. On the one hand, this leads to an even distribution of the vertices over the partitions for our test graphs as shown in Figure 4.5a. However, this algorithm produces highly skewed edge distributions among all partitions. That is due to the reduction of edge cut being an optimization goal.

The E/E strategy defaults to permanent broadcasts for all edge predicate requests and thus marks the worst case, as outliend in Challenge 1 from Section 3.4. Because of its poor performance, we will omit this strategy from further consideration. To summarize, each partitioning strategy is able to successfully maintain its respective balancing criterion while partitioning the graph into the considered number of partitions. However, the quality of the result is different for each case. Depending on the graph, there are partitions that vary greatly from the majority.

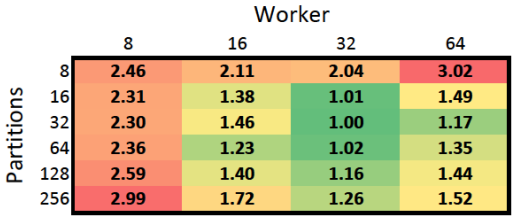
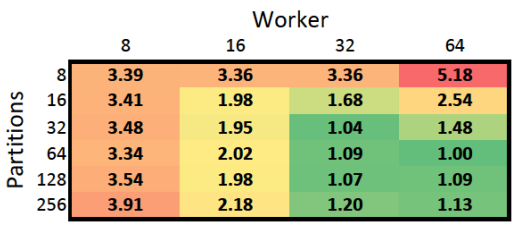


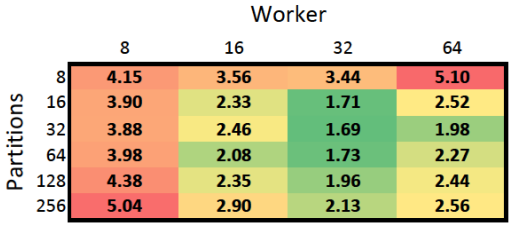
Figure 4.6: System configuration heat map for RRV, *V* query on Biblio graph, *Small* server.

If we compare the partitioning results of Figure 4.5 for the Biblio graph, we find that the RRV algorithm of the V/V strategy achieves the best partitioning result in terms of balanced partitions for both vertices and edges. Generally, such a balaced partitioning is beneficial for GPM, as it limits potential workload skew upfront. In the first set of experiments, we use that setting to investigate the influence of the system configuration on the pattern matching performance for the *V* query. Thus, we varied the number of active workers between 8 and 64 and used 8 to 256 partitions. The heat map from Figure 4.6 shows the slowdown factors compared to the heatmap-local optimal configuration. For the *V* query on an RRV partitioned Biblio graph, we found the optimal SC to be using 32 partitions and 32 workers. Generally, the pattern matching scales well for physical hardware threads, which is indicated by the coloring trend from orange to green between the columns for 8 and 32 workers. In this case, 64 workers are not beneficial, because the *V* query employs two broadcasting requests at the end and the hyper-threads do not provide as much performance as their physical siblings.

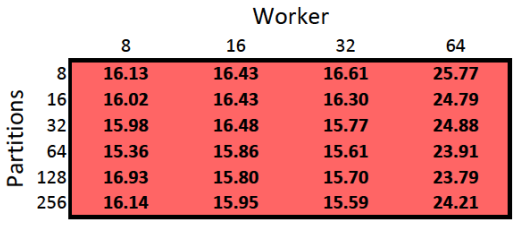
After examining the query performance for a single partitioning strategy, we conducted the same experiments with the remaining strategies on the same graph to show the influence of the different partitioning strategies in detail. The resulting heat maps are depicted in Figure 4.7 and show the relative performance, compared to the global optimum, which is found in Figure 4.7a at 64 workers with 64 partitions. From these heat maps, we derive the following facts:



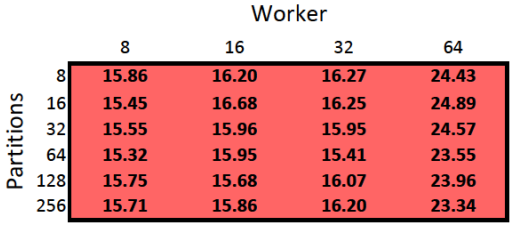
(a) C/V: k-Way



(b) V/V: RRV



(c) V/E: BE



(d) V/E: DS

Figure 4.7: System configuration heat map, *V* query on Biblio graph, color shadings relative to the global optimum (k-Way 64/64), *Small* server.

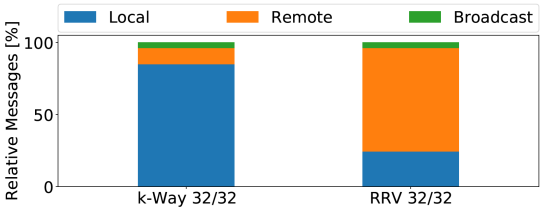


Figure 4.8: Messages per partitioning algorithm, *V* query on Biblio graph. *Small* server.

First, the V/E strategy, represented by the BE and DS algorithms, performs comparatively bad. This happens because the generated query massively hits the vertex outlier partition, which is also visibile in Figure 4.5a. Hence, this partition becomes a bottleneck for the third and fourth edge predicate of the *V* query. A partition can be processed, as soon as it receives messages. However, a worker snapshots the current message queue, when it start processing and thus can not work on messages, which arive after it started the operator logic. Thus, when a single partition receives a lot of messages, which are at least slightly delayed, this can cause noticeable overhead.

Second, the k-Way partitioning results in a better query performance and utilizes the whole system with its optimal system configuration being 64 partitions by 64 workers. For the Biblio graph, this strategy results in evenly distributed vertices and an almost even distribution of edges among the partitions. Furthermore, connected vertices are partitioned together, which is not necessarily the case for RRV. For the k-Way partitioning, the system creates mostly socket local messages and only a few remote messages whereas the V/V strategy results in many remote messages. This effect is illustrated in Figure 4.8.

For the experiments on a single graph, we can conclude, that the C/V strategy is able to utilize all hardware threads. However, we can also deduct limited scalability behavior from the slowdown factors, which are shown within Figure 4.7a. The V/V strategy exhibits comparable performance, but leads to a smaller area with similar performance, which means less possibilites to adapt either worker or partition count, if a budget on either resource would be given.

After thoroughly examining the influences of different partitioning strategies on one graph, we conducted the same experiments for the previously introduced Social and Uniprot graph and added a webshop grpah, called Shop, for more variety. Figure 4.9a presents the best system configurations per partitioning strategy and highlights the overall optimum. We showed that the C/V strategy performs best for the *V* query on the Biblio graph by utilizing the whole system and therefore should be used as the best strategy. However, when querying the Shop graph with a k-Way partitioning, the optimal SC changes to 32/32 and yields a slowdown of factor 2.3, compared to the optimal SC of the V/V strategy. The slowdown can be explained by the massive imbalance of edges within the partitions of k-Way as shown in Figure 4.5b. Yet, the other strategies show

	Biblio		Shop		Social		Uniprot	
Strategy	SC	ms	SC	ms	SC	ms	SC	Ms
V/V: RRV	32/32	65	**32/32**	**11790**	**32/32**	**665**	8/8	884
V/E: BE	32/128	838	32/32	12387	16/16	666	**8/8**	**878**
V/E: DS	8/16	849	32/32	11964	32/32	673	8/8	890
C/V: k-Way	**64/64**	**48**	32/32	27376	32/32	864	8/8	885

(a) *V* query.

	Biblio		Shop		Social		Uniprot	
Strategy	SC	ms	SC	ms	SC	ms	SC	Ms
V/V: RRV	32/32	2663	**32/64**	**5773**	32/32	102	32/32	22
V/E: BE	32/32	2617	32/64	5850	16/16	132	**32/32**	**21**
V/E: DS	32/32	2682	32/64	5982	**32/32**	**94**	32/32	22
C/V: k-Way	**32/32**	**2254**	64/128	15217	32/64	304	32/32	24

(b) *Quad* query.

Figure 4.9: Optimal system configurations per graph and partitioning strategy for both query patterns on four different graph types, *Small* server.

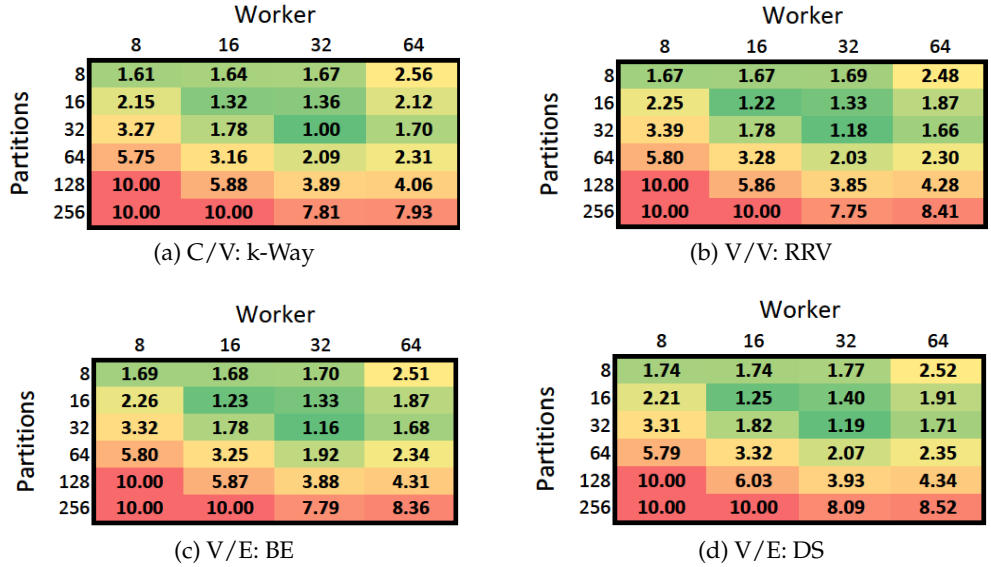
Messages per Edge Request	Biblio	Uniprot
1	299,488	971
2	117	970
3	267	294,932
4	837	10.320

Unicast	Broadcast	Final result

Figure 4.10: Intermediate results for each edge predicate of the *V* query.

well-balanced edges per partition, therefore the merely equal query performance is not surprising. The same holds true for the Social graph. The Uniprot graph is special in terms of the intermediate results, which are shown in Figure 4.10. Compared to the Biblio graph, the *V* query produces almost all messages as broadcasts for the Uniprot graph in the third edge predicate. Broadcasts are known to inhibit scalability and thus, less partitions mean less total messages, which results in less runtime.

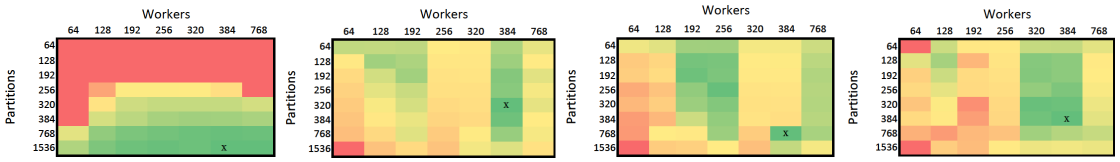
The previous paragraph concluded our test series for the *V* query. Now we want to show the performance implications of all considered influence factors for a second query type, namely the *Quad* query from Figure 4.4b. The results for all system configurations, graphs and partitioning strategies are shown in Figure 4.9b and the respective heat maps are presented in Figure 4.11. The optimal configurations are now always tied to 32 Workers with a varying number of partitions. We see the same runtime behavior as for the *V* query, except for the V/E strategy. The *Quad* query does not hit the vertex outlier partitions (c.f. Figure 4.5a), which enables the BE and DS partitionings to compete with RRV and k-Way. The Shop and Social graphs show an equal slowdown for C/V, compared to the other strategies. However, the Uniprot graph now scales well with the hardware threads, since there are more intermediate results in the Unicast edge predicate. From these experiments we can already conclude, that there is no single best partitioning strategy. To achieve the best performance results for GPM, we need to consider the underlying



(a) C/V: k-Way (b) V/V: RRV

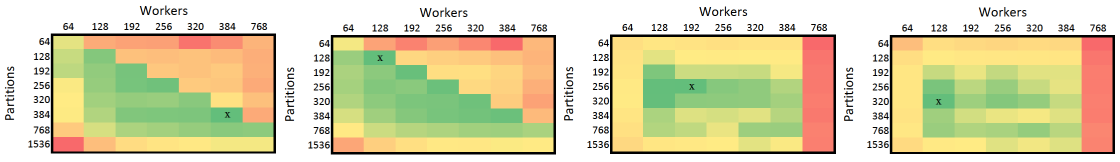
(c) V/E: BE (d) V/E: DS

Figure 4.11: System configuration heat map, *Quad* query on Biblio graph, color shadings relative to the global optimum (k-Way 32/32), *Small* server.



(a) C/V: k-Way (b) V/V: RRV (c) V/E: BE (d) V/E: DS

Figure 4.12: System configuration heat maps, *Quad* query on Social graph, color shadings relative to the local optimum, *Large* Server.



(a) C/V: k-Way (b) V/V: RRV (c) V/E: BE (d) V/E: DS

Figure 4.13: System configuration heat maps, *V* query on Biblio graph, color shadings relative to the local optimum, *Large* server.

data graph, the partitioning strategy and the currently selected SC. Furthermore, results vary on a per-query basis.

First, we have shown the effect of partitioning and SCs on the *Small* NUMA server. Now we want to study their impact on a large-scale NUMA system, which is an SGI UV 3000 and is called the *Large* server in Table 3.1. Therefore we repeated the previous experiments and used gMark to scale up all graphs accordingly, while preserving all other graph properties. Generally, we found that the entirety of our experiments on the large-scale system confirmed our observations from the experiments on the *Small* server. Figure 4.12 illustrates the heat maps for the *Quad* query on the Social graph and Figure 4.13 shows the results for the *V* query on the Biblio graph for the *Large* server. We observe, that the HT column (768 workers) never holds the highest performing configuration, with the general performance being even worse for the *V* query. For the *Quad* query, utilizing all physical cores leads to optimal performance in many cases, which underlines that our processing model scales well with the employed hardware. In contrast to the *Small* server, we see more variations in the heat maps, which is explained by the bigger number of sockets and the increasing influence of the NUMA effect on query performance. Furthermore, due to the larger well-performing areas, i.e. regions in the heat map with green shaded colors, we could better follow resource budgets, like limited workers or partitions. The *V* query heat maps show comparable results for both C/V compared to V/V and between the both V/E strategies. Contrary to the previous results, employing the maximum thread count generally inhibits the performance, which is again explained by the increased amount of broadcasts during query processing. Performance wise, V/V and C/V strategies yield similar query runtimes, even with greatly varying SCs. The V/E strategies may exhibit local optima, however their overall query performance is on average almost ten times slower, compared to the C/V and V/V strategies. This further underlines our claim, that query performance is greatly dependent on the graph, the employed partitioning strategy and the available resources.

Employing an optimal partitioning strategy is crucial for query performance. We argue, that weighing the amount of broadcasts against unicasts, that result from the query pattern, is important to find the best SC and partitioning strategy. For dominant unicasts, it is desirable to partition the graph using a strategy, that balances both edges and vertices.

We found, that employing the C/V strategy is beneficial for the selected experiments, even if there is a minor edge imbalance, since the unicast part of the query will benefit from the locality property of adjacent graph partitions. However, if the edge imbalance exceeds a certain limit, we suggest switching to the V/V strategy. When the broadcasts become dominant, each partitioning strategy performs reasonably even, whereas it is desirable to achieve a balanced amount of edges between the partitions, as edges represent the amount of data records per partition. Balancing them will thus result in an apriori avoidance of workload skew and thus more evenly distributed work in the system. The challenge is to adequately estimate the influences of broadcasts and unicasts due to their dependency on the underlying graph. Our experiments showed, that the optimal system configuration varies among the different workloads. As a rule of thumb, we conclude that it is mostly beneficial to not use hyper threads in most cases and directly map the number of graph partitions to the number of workers. To conclude, the employment of suitable partitioning strategeis can therefore be considered as an answer to Challenge 3 from Section 3.4.

4.2.2 Graph-Aware Infrastructure

After identifying optimal SCs for a given graph, we can now focus on the *infrastructure layer* and reduce unnecessary messages in the system. The main antagonist for scalable performance is represented by broadcasts as a result of one-directional edge storage. Thus, we identify sufficient locality information as a crucial component, to eliminate the need for broadcasts, since efficient messaging is a key component for *Delegation*-based systems like NEMESYS. Answering mixed oriented queries like shown in Figure 3.17b is often required. The inherent broadcasting hinders the systems scalability and thus we investigate, how the amount of messages can be reduced. Parts of this section are based on our previous work from [KUK+17].

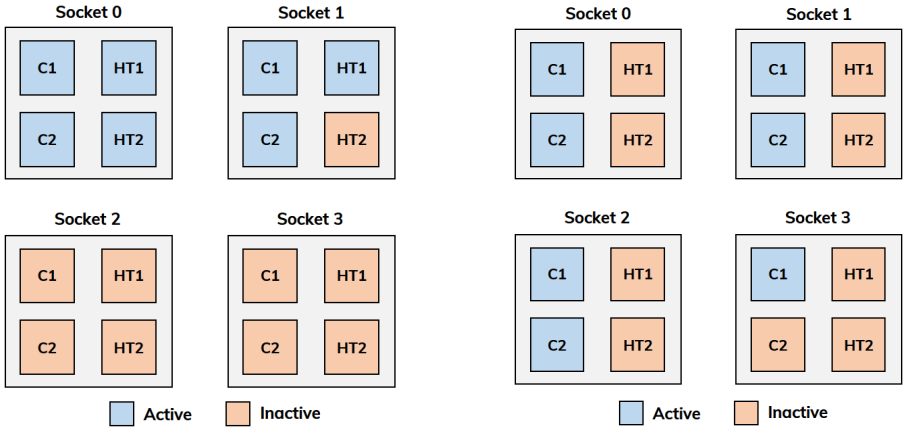
Implementing redundancy

The target partitions of a message are always identified by the source vertex of an edge and as described, we require disjunct partitions on our graphs. However, when storing triples, the disjunction can only be ensured for either the source, or the target column. Thus, indexing both columns to lookup target vertex ids inside a partition will not alleviate the broadcast problem. The straight forward solution to this problem is to not only store outgoing edges, as the prototype requires, but also all incoming edges for every vertex, separately. This can be done simultaneously, while loading the graph and creating

Table 4.1: NEMESYS incoming edge table for the graph in Figure 3.10.

Source	Target	Label
Partition 1		
G	F	label_{GF}
H	A	label_{HA}
H	C	label_{HC}
Partition 2		
A	C	label_{AC}
B	A	label_{BA}
C	F	label_{CF}
C	G	label_{CG}

Source	Target	Label
Partition 3		
D	B	label_{DB}
D	C	label_{DC}
E	C	label_{EC}
E	D	label_{ED}
F	E	label_{FE}



(a) Cores natural, sockets natural

(b) Cores natural, sockets interleaved.

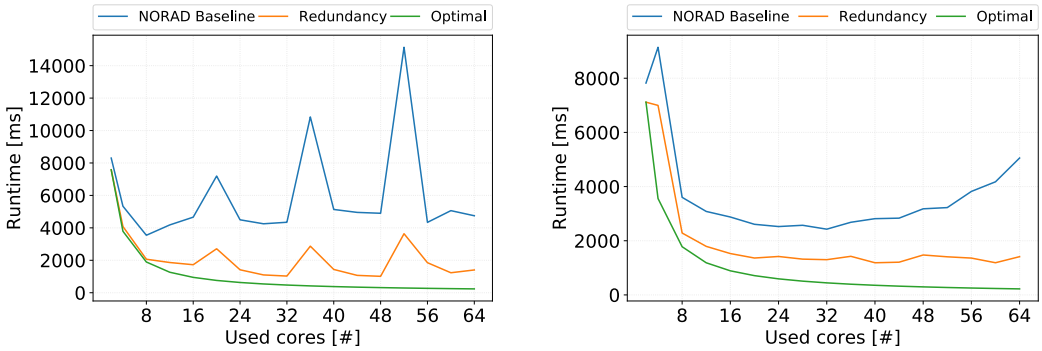
Figure 4.14: Different allocation strategies for 7 workers on a 4 socket system with two logical cores per physical core.

the outgoing edge table. Naturally, every parsed edge [<source>,<target>,<label>] is reversed, such that it becomes [<target>,<source>,<label>]. This triple is then inserted into a second table, the *incoming edge table*, which is shown in Table 4.1. As for the outgoing edge storage, we again create three partitions, which have to be distributed among all sockets as well.

Most importantly, the partitions of both tables do not necessarily exhibit the same cardinalities or content, which already happened in this simple example for Partition 1. Here, vertex H is now also stored in this partition, which it was not for the outgoing edge table. This leads to the effect, which we call *vertex schizophrenia*. Performing a lookup operation on either the outgoing or incoming edge table for a specific vertex leads to answers with different semantics. Topology-wise, we do not add any further information. However, the employed redundancy consolidates all incoming edges of a vertex into one partition, which again creates a disjunct set of partitions for all target vertices of any edge.

To check the effect of redundancy, we repeated the experiments from Figure 3.16a on the *Small* server (cf. Table 3.1). When scaling up the available compute resources, we have multiple schemes to choose from. As each physical core of the underlying multiprocessor usually provides two or more logical cores, we can decide if we want to first allocate all workers for one physical core (*siblings first*), or if we first allocate one worker per physical core and fill up the remaining logical cores afterwards (*naturals first*). Furthermore, worker allocation can choose to place consecutive resources on one socket, until all cores of that socket are used and then move on to the next (*sockets natural*) or interleaving worker allocation on all sockets (*sockets interleaved*). The individual effects are illustrated in Figure 4.14. We define the short handles for core allocation as *nat*/*sib* and for socket allocation as *nat*/*int*. Obviously, these two versions of each core and socket allocation can be combined to get a total of four worker allocation strategies being *nat/nat*, *nat/int*, *sib/nat* and *sib/int*.

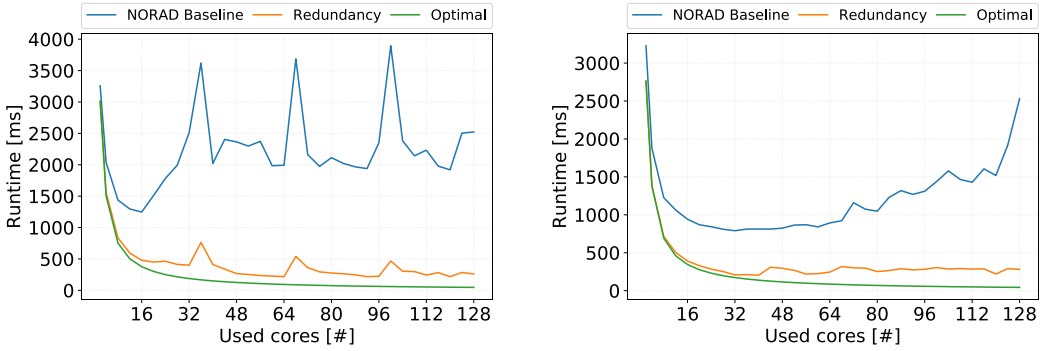
The results for Figure 4.15a are tested using *nat/nat* and for Figure 4.15b we changed allocation scheme to *nat/int*. Figure 4.15a shows the direct impact of redundancy on the systems scalability, compared to Figure 3.16a. The NORAD baseline achieves a speedup of factor $1.7\,\mathrm{x}$ using 64 cores with no redundancy, which increases to a speedup of $5.4\,\mathrm{x}$, when redundancy is employed. The dents in the runtime curves are explained by the worker allocation strategy. Every socket has 8 physical cores and 8 HTs, thus the performance drops with the first cores being allocated on the next socket. In this experiment,



(a) Biblio, Compute Routing Table, *nat/nat*.

(b) Biblio, Compute Routing Table, *nat/int*.

Figure 4.15: Runtime scalings for increasing worker count on the Biblio graph, *Quad* query, *Small* server.



(a) Biblio, Compute Routing Table, *nat/nat*.

(b) Biblio, Compute Routing Table, *nat/int*.

Figure 4.16: Runtime scalings for increasing worker count on the Biblio graph, *Quad* query, *Medium* server.

the next socket is utilized at 17, 33 and 49 used cores respectively. Furthermore, the sockets are not fully but mesh connected, therefore the communication between the two farthest sockets can reach two NUMA hops. Because the baseline implementation keeps a direct mapping of partition to worker count, increasing the workers also increases the number of partitions in the systems, which in turn leads to more broadcasts, that have to be replicated to those partitions. In addition, one worker per socket has to become the `NodeCoordinator` once in a while (cf. Section 3.2.1) and perform the actual communication and coordination tasks, which further decreases the performance of a socket with only a few active workers. If only one worker is active on a socket, it is the only eligible worker for this task, which imposes additional overhead. If redundancy is added, we can still see the overhead of the remote communication. However, because we only target the partitions with relevant data, the severity of this effect is greatly reduced, resulting in a smoother curve. In Figure 4.15b, the first four workers are each allocated on a new socket, which leads to an initial performance decrease, due to inter-socket communication and coordinator duties. As soon as all sockets are active and more compute resources are added, we see an overall and smooth performance increase until the number of physical cores is reached. At this point, adding workers on the HTs of every core introduces slight overhead, since an HT does not provide the same compute power as its physical sibling core. Since the workers are already distributed on all sockets, this effect is less drastic than in Figure 4.15a.

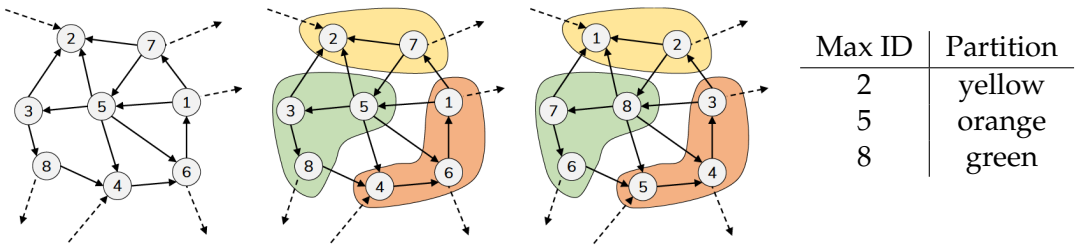
To confirm the general scalability behavior, we repeated the above experiments on the *Medium* server (cf. Table 3.1). In contrast to the *Small* server, the sockets are fully connected, which reduces the maximum NUMA hops to one. As anticipated, the increasing NUMA distance explains the increasing dents in the performance for the baseline implementation with regards to Figures 4.15a and 4.16a. For the *nat/int* configuration, the same behavior as for Figure 4.15b. The generally higher performance of Figure 4.16 is easily explained by the newer hardware of the *Medium* server. Most importantly, employing redundancy helps to alleviate both the severity broadcasts, by completely eliminating them and in addition it improves the overall system performance. The introduction of redundancy can therefore be considered as an answer to Challenge 1 from Section 3.4.

Improving the routing table design

Implementing redundancy effectively doubles the infrastructure cost and storing additional locality information applies pressure on the partition manager (cf. Figure 3.10). Keeping this information in the processor cache can be substantial for scalable processing, since it is required for every single message, that is sent during the GPM process. With the information being stored directly within two routing tables, we reviewed their designs to improve their memory footprint. We found, that both the *compute design* and the *lookup design* have advantages, that can be combined to remedy their individual disadvantages. Thus, we propose a *hybrid design* that combines the low memory footprint of the *compute design* and the locality awareness of the *lookup design*.

To enable this design, we borrow the idea from a standard range-based routing table, which only contains as much entries, as there are partitions in the system. The *hybrid design* is therefore combination of a range-based routing table and any locality-aware *partitioning strategy*, which were described in Section 3.3.3. Creating this design is performed in three steps, that are illustrated in Figure 4.17. First, we partition the data using any suitable partitioning strategy. The outcome of this step is a vertex-partition assignment table. Storing this assignment in a *lookup design* would need 8 entries, since we would have to store the partition locality information per vertex. We now apply a technique which is commonly known as vertex reordering, to ultimately reduce the amount of entries in the routing table. As shown in Figure 4.17c, every vertex in every partition will thus be assigned a new id in an ascending order, such that every partition contains only dense id ranges without gaps.

The concept of a range-based routing table requires to only store upper bounds for the stored data. Figure 4.17d shows the resulting *hybrid routing table* for the graph from Figure 4.17a. In contrast to the lookup design, we only need to store 3 entries, i.e. the individually highest vertex ids, that are stored in the partition. Hence we can state, that

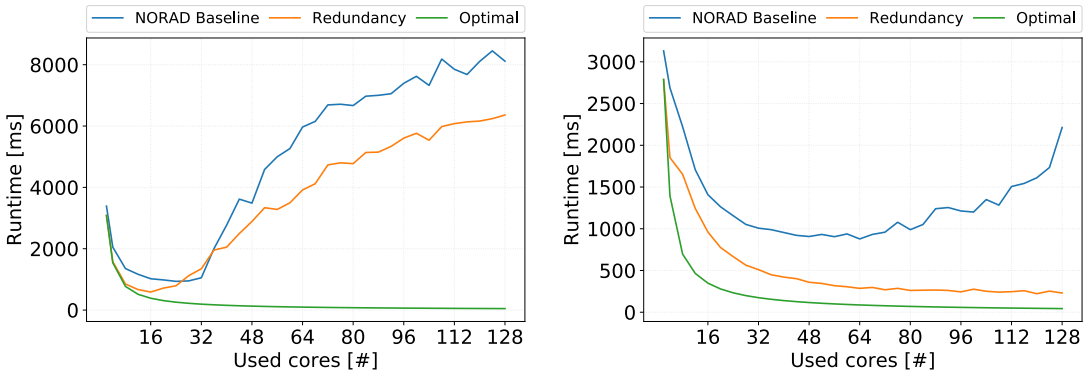


(a) Data graph. (b) Initial partitioning. (c) Vertex reordered.

Max ID	Partition
2	yellow
5	orange
8	green

(d) Hybrid table.

Figure 4.17: Generating a hybrid table.



(a) Biblio, Lookup Routing Table (RRV), Order nat/int.

(b) Biblio, Hybrid Routing Table (RRV), Order nat/int

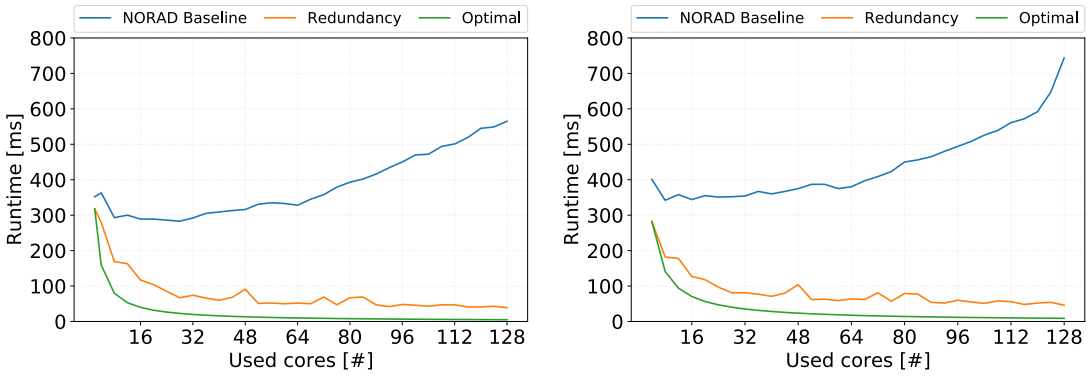
Figure 4.18: Comparing the scalability behavior of the *lookup design* and the *hybrid design* routing table for the Biblio graph, *Quad* query, *Medium* server.

the *hybrid design* yields a very low memory footprint, which in turn allows the routing table to easily fit or stay in the cache of the multiprocessors. A downside of this approach lies in the generation process. Vertex reordering can only be done after ingesting and partitioning the graph completely and thus, we create some additional overhead during the ingestion phase. Furthermore, we need to decide whether we keep the virtual-to-physical id mapping in-memory or if we materialize the data graph with the new ids, together with a partitioning index. The mapping requires twice the amount of memory of the original data graph for the original and the mapped id, thus further increasing the memory footprint. However, this data is not needed during the GPM processing and if enough main memory is available, the additional memory could be spared. If a memory budget exists, the mapping can still be materialized as described before.

We tested the impact of the *hybrid design* routing table on the scalability behavior in Figure 4.18. For this experiment, we partitioned the graph using the RRV strategy from Section 4.2.1. Figure 4.18a shows the scaling of the original lookup table. Here, the *lookup design* performs poorly. The routing table itself is not explicitly replicated among sockets and is thus only present in their caches, but not as a separate copy in their memory domains. If we transform the lookup table into our hybrid table in Figure 4.18b, the routing information gets small enough to stay cached during the whole GPM processing.

Figure 4.19 shows a similar experiment with the V query for a k-Way partitioned Social graph. The NORAD baseline without redundancy shows a performance degrade, but our optimized version with redundancy can achieve a small performance gain. We identified the general partitioning *quality* as the reason for this behavior. The RRV strategy balances vertices among sockets, but it does not consider vertex neighborhoods. The smaller routing table can thus add a considerable locality boost for Figure 4.18, but not for Figure 4.19, because the k-Way strategy already creates more self-contained partitions and thus enforces more socket-local communication. We could observe the same effect for the V on the Biblio graph query earlier in Figure 4.8.

As a conclusion, implementing a *hybrid design* does not only reduce the overall size of the routing table, but also enables the system to achieve scalable performance, when the stored locality information would exceed the processor cache. On the other hand, creating a *hybrid design* routing table imposes more overhead, the larger an ingested graph becomes. Depending on the graphs size, it may be advisable to perform the conversion offline and import the only the converted version, to save on ingestion time.



(a) Social, Lookup Routing Table (k-Way), Order nat/int.

(b) Social, Hybrid Routing Table (k-Way), Order nat/int.

Figure 4.19: Comparing the scalability behavior of the *lookup design* and the *hybrid design* routing table for the Social graph, *V* query, *Medium* server.

4.2.3 Lessons learned

We tested different partitioning strategies for three different graphs on both a small and a large SMP system. The highest performance was not always achieved, when the maximum of physical cores was active, but varied between experiments. We found, that selecting a different partitioning strategy for any combination of graphs, queries and SCs is thus necessary for achieving an optimal runtime behavior. However, swapping between partitionings is not feasible between individual query runs. As a rule of thumb, we argue that a partitioning strategy should be selected based on the biggest performance islands. Whilst this approach is not achieving the maximum possible performance, it would allow for the most flexibility, in terms of system resource usage adaptivity towards changing workloads. However, if a sufficient amount of main memory is available, we could also load the graph with different partitionings and dispatch any incoming query according to its best performing configuration.

Leveraging redundancy to improve the available locality information does completely eliminate broadcasts during the GPM process. Yet it introduces a considerable storage overhead, both on the *storage* and *infrastructure layer*. The additional book keeping for the routing table is remedied by our *hybrid design*, but the storage overhead remains. We consider this a potentially viable trade off, however it still yields further optimization potential.

4.3 INFRASTRUCTURE-BASED OPTIMIZATION

Section 4.2 covered the structural aspects of the data and mapped their relevance to the underlying hardware. The adjustments to the *storage layer* allow us to determine suitable SCs for any combination of a given graph, queries to perform and partitioning schemes. The previous section introduced redundancy as a countermeasure for broadcasts. However, combining redundancy with the *edge-labeled multigraph* (ELMG) data model from Section 2.1.2 does introduce a considerable overhead. An ELMG naturally adds one edge per piece of information, to incorporate additional knowledge into the graph. The amount of stored triples can thus be heavily inflated, depending on the number of additional properties in the graph. Blindly applying redundancy doubles all edges by adding

a reversed version between the respective source and target vertices. Despite easing the traversal in both edge directions, this also adds a significant storage or memory overhead. Storing all incoming edges furthermore implies the necessity of a second routing table, which was defined in Section 3.3.3, i.e. in addition to the raw data, we also need twice the amount of metadata.

This section therefore aims to alleviate the storage overhead of the changes made by Section 4.2.2 while still maintaining the achieved performance gains. We investigate measures on how to provide sufficient locality information without requiring as much main memory as fully redundant storage. We will show, why HashSets are not applicable and introduce Bloom filters as an alternative. Reducing the number of messages, that are exchanged between a source and a target partition does increase the performance, as previously shown in Section 4.2.2. The size of Bloom filters can be tuned, according to the desired false positive probability. This allows us to trade memory for accuracy, but much more fine grained, than full redundancy.

Besides reducing the number of exchanged messages, it is also important on which physical sockets the communicating partitions reside. Exchanging messages with one remote partition can have a significant runtime impact, as shown with the peaks in Figures 4.15a and 4.16a. In addition to the data placement, which is predetermined by the partitioning, we want to further increase the data locality and change the data *placement*, i.e. placing partitions on the same socket, if they have a high message flow between them.

4.3.1 Adaptive Message Filtering Mechanisms

In the previous sections, we examined the interplay of partitioning strategies and system configurations. Ultimately, avoiding broadcasts was found to be of fundamental importance for scalable GPM processing on a NORAD system and thus our main goal is to avoid said costly broadcasts, when searching for a target vertex with unknown source. While this can easily be addressed by just redundantly storing the inverse direction as shown with Tables 3.2 and 4.1, this doubles the overall memory requirement of the *storage layer* and the *infrastructure layer*. Since we work on a system with $64\,\mathrm{bit}$ vertex and label ids, that means adding $192\,\mathrm{bit}$ per edge: $(\{\text{target}, \text{label}\} \rightarrow \text{source})$.

Fully redundant storage, as proposed in Section 4.2.2, creates the necessity to double not only the base data, but all secondary data structures as well. This includes indexes or routing tables. We have already explained, that fully redundant storage can be unfavorable, e.g. if the routing table exceeds the processor cache. Thus, we want to avoid broadcasts, but still keep the memory footprint as low as possible. To achieve that, we envision a secondary data structure for tracking all target vertices within a partition. This index-like structure is then used to quickly exclude a whole partition, if a requested target vertex does not exist in it.

HashSet principles

One of the most common data structures to quickly answer such requests is the HashSet. A typical architecture for this data structure is depicted in Figure 4.20. The name is derived from a hash function $H(\boldsymbol{x})$, which converts all bits $x_i \in \{0, 1\}$ of a to-be-stored element to a fixed size integer:

$$\begin{gathered} H(\boldsymbol{x}) = y, \;\; \boldsymbol{x} = (x_1, \ldots, x_n), \;\; x_i \in \{0, 1\}, \\ y \in \mathbb{N}, \;\; y \sim \mathcal{U}(0, \ldots) \;\; . \end{gathered} \tag{4.3}$$

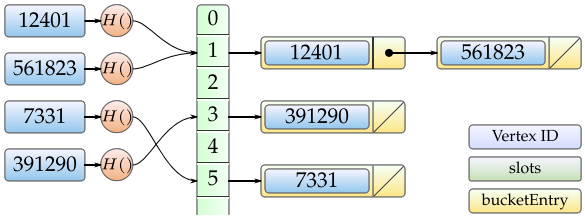


Figure 4.20: Internal architecture of a HashSet. Each vertex id is hashed to derive an index to a slot, where it is added to a Bucket using a list of entries.

As our input to the hash function, the dictionary encoded vertex id, is an integer as well, (4.3) can be rewritten as:

$$H'(x) = y, \quad x \in \mathbb{N}, \quad y \in \mathbb{N}, \quad y \sim \mathcal{U}(0, \ldots) \ . \tag{4.4}$$

The calculated y serves as an address of where to store this element, assigning it to a slot within the set. Due to potential collisions $H'(x) = H'(x')$ for $x \neq x'$, two different vertices might share the same address, or slot, as depicted in Figure 4.20. Thus, each slot usually carries a list of all elements that belong to it, called bucket. To check, whether a vertex is present within the set, the hash is calculated to determine its potential bucket, and all corresponding bucket entries are checked for equality with the vertex in question.

Ideally, each slot contains no more than one element, yielding a complexity of $O(1)$ for both, insert and retrieval. This can be controlled by using a uniformly distributed hash function and many slots, at the expense of memory, to prevent hash collisions. For well-balanced sets, the main limitations are given by the speed of the used hash function, memory, and cache for performing lookups, traversing the entries and conducting the equality check.

For a collision free storage of N elements, with one entry per slot, the set needs, at least, N slots. If a slot is occupied, it points to a bucket, which is a linked list of entries, storing the vertex id and pointer to the next entry, which is unset when being collision free. For $64\,\text{bit}$ systems, storing $64\,\text{bit}$ vertex ids, this yields the following storage requirements:

$$\underbrace{N}_{\text{slots}} \Big(\underbrace{64\,\text{bits}}_{\text{ptr: bucket}} + 1 \big(\underbrace{64\,\text{bits}}_{\text{vertex id}} + \underbrace{64\,\text{bits}}_{\text{ptr: next}} \big) \Big) = N \cdot 192\,\text{bits} \tag{4.5}$$

In this case, the fully redundant storage of the whole graph requires exactly the same amount of space. Thus, the set only prevents questioning the central *partition manager* but allows a worker to check locally, if it is worth to send a message to another worker. The main purpose of both, the HashSet or the usage of redundancy, is the reduction of broadcasts, to increase overall system performance. Therefore, the question arises, whether it is possible to provide the same information with reduced storage requirements.

Bloom Filter Based Messaging Optimization

As each prevented broadcast already yields a speed-up, no exact solution for existence checking, as provided by the HashSet, is required. This yields room for space efficient, probabilistic data structures, answering existence checks with some degree of uncertainty. Within NEMESYS, we require any potential data structure to not provide false negatives. That is, if an element is present, that data structure must be exact. However, false positives, where the element is indicated as present even if it is actually missing, are allowed. In such cases, unnecessary messages would be sent out.

One candidate of a probabilistic data structure, suited for those requirements, is the well-researched Bloom filter. Compared to HashSets it is much more space and time efficient, as both, insert and retrieval are always $O(1)$, at the cost of correctness. Here, the vertex id itself is not stored within the data structure, obsoleting the equality check during lookups and thus reducing the memory footprint. Instead of linked lists, only a single bit is stored within the slot, that is indicated by a hash function, which is applied to the vertex id. Slots thus are implemented using a bitfield. To check whether a vertex id is present, its hash is calculated and the bitfield is checked, whether it contains a 1 – item is present – or 0 – item is not present – at the index given by the hash. Due to hash collisions, using just one bit per vertex usually yields false positive rates above an acceptable level. State-of-the-art implementations thus use more than one hash function per element, where $H'_i(x) \neq^! H'_j(x)\ \forall x$ and $i \neq j$.

For each hash, a 1 is stored within the corresponding bit. When querying the Bloom filter, an element *might* exist, if *all* bitfield indices, denoted by the results of the hash functions, contain a 1. The false positive rate p is thus directly affected by the number K of hash functions used per vertex, and the total number of bits M within the bitfield. Previous work about Bloom filters [Blo70, BM03] has shown, that the approximate number of bits required for storing N vertices at a desired false positive rate, can be estimated by:

$$M \approx -1.44N \log_2(p)\ . \tag{4.6}$$

(4.6) yields a space requirement of only $\approx 10\,\text{bits}$ per vertex for a false positive rate of $1\,\%$. With an overall memory overhead of roughly $15\,\%$, this is much less than a HashSet or fully redundant storage would need. As shown in [BM03], similar rules hold true for the optimal number of hashes, and thus bits, that should be calculated for each single vertex:

$$K = \frac{M}{N} \ln 2 \approx -1.44 \ln(2) \log_2(p)\ . \tag{4.7}$$

We can trivially deduce, that the computational cost of the Bloom filter is mainly determined by the performance of the chosen hash algorithms. Applicable and fast candidates for these algorithms are chosen based on the type of to-be-stored elements. Key task is the conversion of some arbitrary data into a numeric value, uniformly distributed throughout the whole range, i.e. the size of the bitfield. While $64\,\text{bit}$ unsigned integer vertex ids already are numeric values, these ids can not be used as-are, since the Bloom filters bitfield will be much smaller than all potential 2×10^{64} candidates.

Residual class ring benefits

Yet, a hash function for this kind of input data is much simpler, as it just needs to distribute all potential vertex ids uniformly among the available bitfield slots. This can be achieved by e.g. multiplying the vertex id x with some number a_i, and limiting the result to the number of slots using the modulo operator:

$$H'_i(x) = y = a_i \cdot x \pmod M,\ \ x \in \mathbb{N},\ \ a_i \in \mathbb{N} \setminus \{0\}\ . \tag{4.8}$$

While omitting a_i via $a_i = 1$ is valid, the resulting y is distributed similarly to the original input x, which could potentially lead to an increased number of collisions. Best results can usually be expected when both, a_i and M are prime numbers [HD62]. Compared to the multiplication of a_i, the modulo operator is mandatory and very costly [Gra17]. However, for all M that satisfy $M = 2^k$, the modulo operator can be replaced by a *bitwise*

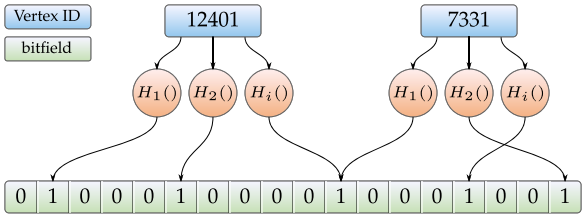


Figure 4.21: Internal architecture of a Bloom filter. Every vertex id is hashed by K algorithms, each providing an index to a bitfield, where a 1 is stored accordingly.

and operation. This is due to the fact, that the registers of a processor represent a residue class ring where:

$$(x)_{\text{base}} \pmod{\text{base}^k} = \text{last } k \text{ digits of } (x)_{\text{base}} \ . \tag{4.9}$$

For $64\,\text{bit}$ unsigned integer vertex ids, which are of base 2, the last k digits of $(x)_2$ are given by applying a bitmask, or *bitwise and*, of $(2^k - 1)$ to $(x)_2$, as

$$2^k - 1 = 2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} \ . \tag{4.10}$$

The modulo operator in (4.8) can therefore be replaced by:

$$(x)_2 \pmod{M} = (x)_2 \wedge (M - 1), \quad \text{for } M \in 2^k, \ \ k \in \mathbb{N} \ . \tag{4.11}$$

While applying *bitwise and* using bitmasks instead of using the modulo operator dramatically reduces the computation time [Gra17], it limits the potential sizes M of the bitfield to values $\in 2^k$. The performance impact, however, is remarkable. We thus use (4.12) as final hash algorithms for our Bloom filter implementation, where each a_i is a prime number and is thus called *PrimeHash* henceforth:

$$H'_i(x) = y = (a_i \cdot x) \wedge (M - 1) \ . \tag{4.12}$$

Despite returning false positives, the Bloom filter suffers from additional drawbacks that need to be considered for real world scenarios: As only single bits are stored, collisions can not be tracked (see Figure 4.21), and vertices can not be deleted from the filter. This implies, that vertices deleted from the graph can not be removed from the Bloom filter. For dynamic graphs, where many deletions may occur, the false positive rate might reach unacceptable levels over time.

Scalable Bloom filters

To overcome the ambiguity due to collisions, the *counting Bloom filter* stores the number of elements contributing to each slot. This enables deletion by simply decreasing the corresponding counter, but requires multiples of the memory of the simple Bloom filter to store the counters. Using $8\,\text{bits}$ per slot, 8 times the size of the simple filter, allows for 255 collisions per slot. However, the number of contributions to each slot can not be predicted for real world scenarios and even more bits could be required. A potential compromise, requiring less amounts of memory, might store $2\,\text{bits}$ per slot: One to denote whether the slot is taken, and another one to track collisions. Deletion is allowed for all slots, where no collision occurred, i.e. the collision bit is not set. Depending on the fill-rate of the Bloom filter, uniformity of the hash functions, and the number of deletions from the graph, this might provide an acceptable trade-off between required space and false positive rate over time.

Similar to the missing support for deletion, is the lack of scalability in terms of size during runtime, since the size of a traditional Bloom filter is constant. The more vertices are added to the filter, the more bits are set to 1. Eventually, all bits will be set and the filter is rendered useless. The HashSet addresses this issue by rehashing. That is, the underlying data structures are completely rebuilt, the number of slots is increased and a new hash/slot for every stored vertex is then calculated. As the Bloom filter does not store the vertices themselves, growing and rehashing is not supported by the data structure on its own. A complete rebuild would require re-scanning the whole graph and recalculating the hashes for all to-be-added vertices. To reach an acceptable false positive rate for a given graph, the Bloom filter's final size, and thus the number of to-be-added vertices, must be known beforehand at the time of its instantiation. This is trivially achievable by collecting the appropriate statistics during the graph ingestion or partitioning phase.

Our vision of a so-called scalable Bloom filter (SBF), which is illustrated in Figure 4.22 and similar to [ABPH07], address this drawback to some degree. It starts with a reasonably sized bitset, that matches the number of to-be-expected vertices. If the actual number exceeds this threshold, the current bitset is marked read-only and a second bitset is added, which is usually larger than the first one, to prevent the problem from getting worse. For checking the existence of some vertex, all created bitsets have to be examined, increasing the overall computational complexity. Furthermore, continuously adding more bitsets will not decrease the false positive rate, but just slow down its degrade. Figure 4.22 clearly depicts this issue: The first bitset is small and filled by 75 %. Querying an SBF is always performed against all available bitsets. Thus, if the first set yields many false positive results, this can not be mitigated by the following bitsets. The SBF can be considered as an intermediate solution to the problem of an increasing false positive rate by additional insertions, but it is unable to fix it. Since static and dynamic graphs share the same messaging procedure, we will concentrate on static graphs and static Bloom filters in this thesis. Scalable Bloom filters with dynamic graphs would only add to computational complexity, but not to the messaging issue and are thus widely excluded from this thesis and left for future work.

Integrating a Bloom Filter into the system

Based on the internal architecture of NEMESYS, we have identified the actual *data partitions* and the *partition manager* as well suited to leverage a Bloom filter to discard messages. As explained in Section 3.2.2, every partition has its own, dedicated message queue. When workers are communicating throughout the pattern matching process, any applicable partition gets its queue filled. In case of a broadcast, every worker will send the same message to all data partitions. However, since always multiple data partitions

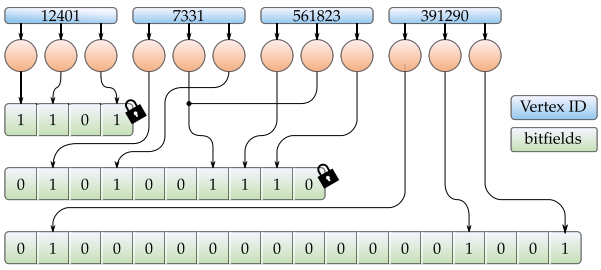


Figure 4.22: Internal architecture of a scalable Bloom filter. As soon as the current bitfield reaches a certain fill level, it is locked and new entries are added to a new, often larger bitfield.

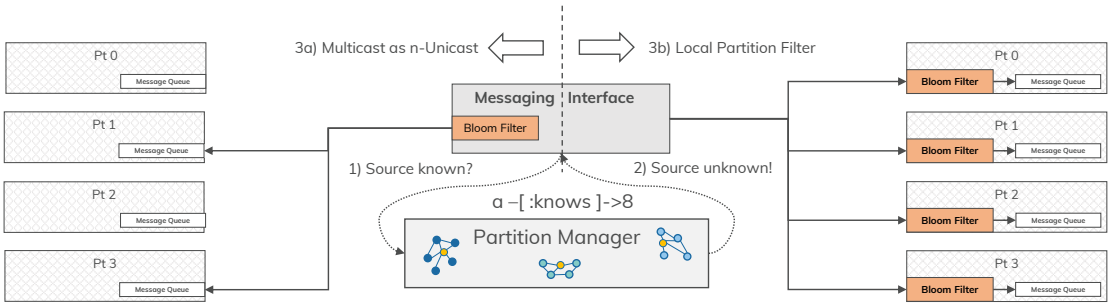


Figure 4.23: System integration opportunities for our Bloom filter approach.

are processed at the same point in time, we envision to exploit the inherent parallelism and filter out unnecessary messages at partition level, which allows us to exploit the full parallelism of the hardware.

The first approach is called *partition based workload reduction* with its process being illustrated on the right hand side of Figure 4.23. In this scenario, the edge $a \rightarrow_{knows} 8$, with 8 being an encoded vertex id, is evaluated. The source vertex of the edge is unknown, but both the edge label *knows* and the target vertex with id 8 are known. Within NEMESYS, the workers will use the *messaging interface* to fetch the partition information from the *partition manager* (cf. Figure 3.10). As previously explained, only the source vertex is indexed and thus we have to send a broadcast. This means, the global coordinator will insert a new message into the message queues of all data partitions within the *storage layer*.

When any worker finishes its current task and selects a new data partition to process all of its messages, we can now employ the Bloom filter to check, whether there is any edge with this known target vertex stored inside the partition. Instead of directly scanning the data partition, the Bloom filter will be probed for the known target vertex. If the result is negative, we can safely discard the message, since our Bloom filter guarantees the absence of false negatives. On the other hand, a false positive can always occur and still force an unnecessary access to the data partition. The advantage of this method is, that we can concurrently filter messages, within all data partitions at once. A major drawback of this method is however, 1that the system will still produce too many messages.

The second approach is called *messaging based workload reduction* and its essence is depicted on the left hand side of Figure 4.23. Instead of implementing one Bloom filter per data partition, we embed a set of Bloom filters inside the *infrastructure layer*, i.e. we generate one Bloom filter for every data partition and store them inside the *partition manager*. These Bloom filters will also be filled during the graph loading phase.

NEMESYS will then load a graph edgewise using multiple steps. First, for every edge, the source vertex will be inserted into the *partition manager*, according to the partitioning strategy. This information is later used to fetch the data partition for the adjacency of a vertex. Second, the target vertex of an edge is added to the Bloom filter of the partition, where the source vertex is stored. Instead of creating a memory intensive and slow index, we get a tunable but blurry picture of the adjacency of all source vertices inside every partition. For every triggered broadcast, the *messaging interface* linearly probes all Bloom filters for the target vertex in question. Instead of sending the original broadcast to all partitions, the message will only be forwarded to those partitions, where the Bloom filter returned a positive result, thus resulting in a *multicast* message.

We envision a performance boost from this approach, since it will dramatically reduce the number of messages in the system, as well as the unnecessary data partition accesses.

Since the amount of false positives is directly dependent on the employed hash function and the used memory, we can fine-tune the Bloom filter to reduce the amount of undesired work, based on a given workload or graph. On the other hand, the probing will be handled inside the *infrastructure layer*. Since sending a message is inherently serial, we are possibly generating a new performance bottleneck, as the amount of to be probed Bloom filters scales linearly with the number of data partitions.

Evaluating the Bloom Filter quality

We will now provide experiments to prove the applicability and the suitability of our Bloom filter approach. After that, we discuss the benefits and drawbacks of implementing a Bloom filter either within the data partitions or in the partition manager, as shown in Figure 4.23. We have conducted all of our experiments on the *Medium* server (cf. Table 3.1) and all experiments were performed using 64 cores with the *nat/int* allocation policy, if not stated otherwise. Our experiments were run on the four different and already introduced graph datasets: a bibliographical network, a social network, a webshop and a protein network. We have selected these graphs because of their individual topologies, as they impose different characteristics, e.g. such as average vertex degree or connected components, which will most likely lead to different runtime behavior of NEMESYS in terms of necessary messages. To evaluate the impact on the runtime characteristics, we will again use the two queries *Quad* and *V* as depicted in Figure 4.4. If the introduced full redundancy from Section 4.2.2 is not used, both queries produce broadcast messages, according to their edge predicates. We disabled query optimization measures to ensure the edge directions and triple ordering within the queries. In case of fully redundant storage, both queries generate only unicasts.

In Section 4.2.1, we have shown the influence of the graph partitioning algorithm on the query performance. Since the V/V and C/V strategies performed well, we will now use the RRV and k-Way partitioning algorithms and extend this strategy set by a simple hash partitioning, such that we can use both a compute and lookup design routing table for these experiments. We execute all queries on all graphs, after they have been partitioned according to the respective strategy and only present the figures containing the best overall query runtime, given a certain partitioning. This will lead to different baselines within the figures, where applicable.

As a basis for our experiments, we have chosen PrimeHash (cf. Equation (4.12)) for our Bloom filter, called PrimeBloom, since it provides sufficient randomness as known algorithms like Googles HighwayHash [ACW16] or MurMur3. Additionally, because of the limited input and output data, PrimeHash is also faster, than the contestants, shown in Figure 4.24. We compare the two aforementioned algorithms against two versions of our PrimeHash: (1) as the original version with setting a_i to a prime number and (2) with setting a_i to a random integer for comparison of the applicability of using prime numbers in general. Figure 4.24 shows, that PrimeHash yields a reasonable false positive probability per element for all datasets as the competitors and is also faster. The performance advantage of PrimeBloom originates from the residue class ring optimization using a bitwise AND instead of the modulo operator, together with an appropriate storage of the bits inside the Bloom filter.

After distributing all bits uniformly among the number of available slots, they need to be stored in memory. Depending on the underlying architecture and available assembly instructions, two major options exist. The first one stores each entry as actual bit within a larger data type, like an 8, 32 or $64\,\mathrm{bit}$ wide integer, packing the data as densely as possible. Therefore, this approach requires efficient bit-masking instructions. The other one

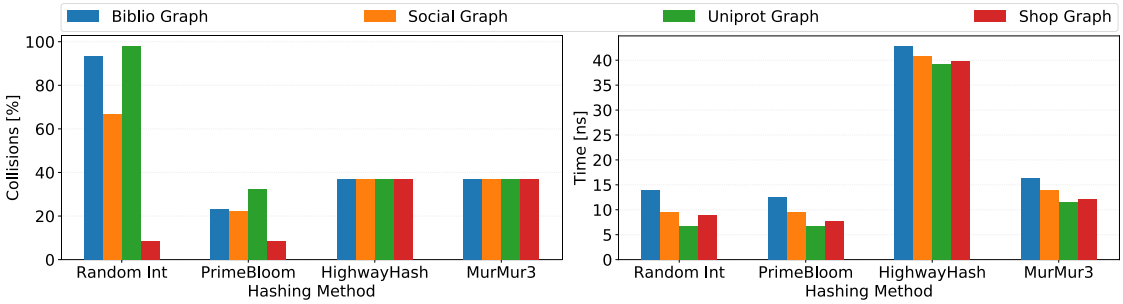


Figure 4.24: Hashing quality (left) and average time spent per item while hashing the corresponding dataset (right) using a random integer, HighwayHash, MurMur3 and our prime based approach, single threaded execution.

stores each bit on its own within a larger data structure, like an 8, 32 or $64\,\mathrm{bit}$ wide integer, requiring more memory but obsoleting any bit-masking. Due to increased storage requirements, caches integrated within the CPU might be less efficient. If the latter runs faster on the actual hardware, using a counting bloom filter will usually provide similar performance but additionally allows for element deletion.

Figure 4.25 compares our PrimeBloom implemented with a counting bitset and both single and packed storage approaches with different bitwidths on the *Medium* server. The actual results may vary among different test servers, since bit-masking instructions are processor dependent in terms of their performance. The counting variant uses an *add* operation on the assembler level and is thus the least performant, just like a bitfield, that stores single bits unpacked within larger data structures and uses *mov* operations on the assembler level. The packed data storage, where each bit is stored using an actual bit within a larger scalar value, requires bitwise operations to set the correct bit. However, this leaves the remaining bits intact and available for further use. Despite additional complexity compared to storing bits unpacked, Figure 4.25 clearly depicts, that all bitwidths of the packed storage clearly outperform the other solutions. This is most likely due to reduced storage requirements, fitting the internal caches of the CPU, which can also be deduced from the figure. Up to a certain size of the bitfield, packed and unpacked storage, using an $8\,\mathrm{bit}$ unsigned integer for each bit, are almost equal in speed. Hereafter, they

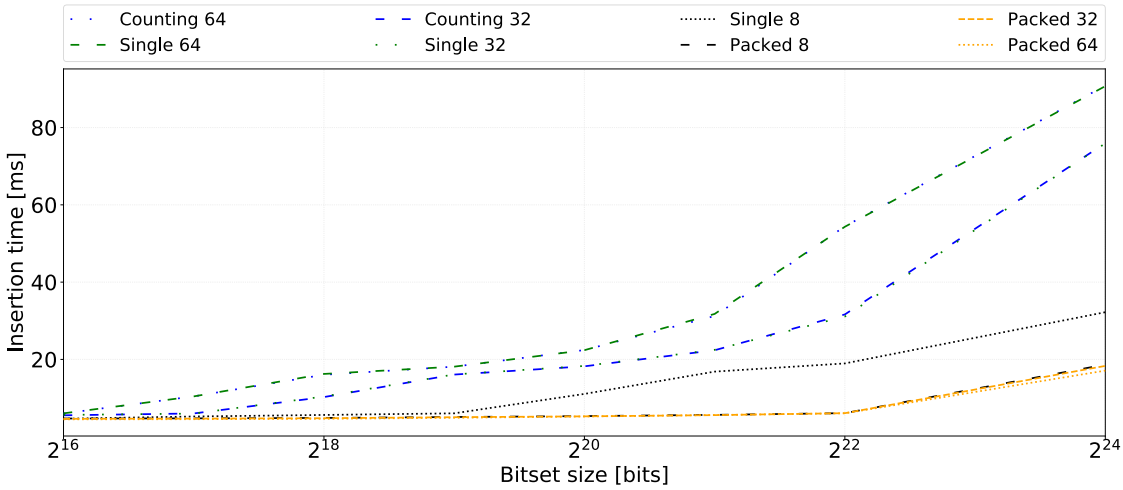


Figure 4.25: Time to insert 5 million entries into a bitset, less is better, depending on its size and the used storing strategy, single threaded execution.

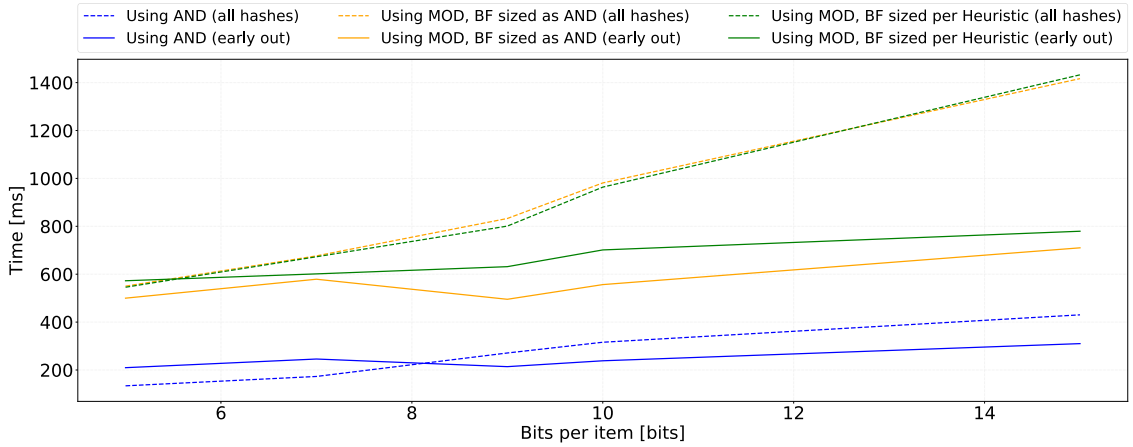


Figure 4.26: Time to query 1 million items, Bloom filter using AND or MOD operator, single threaded execution.

drift apart. Additionally, with growing space requirements to store the whole bitfield, the time needed to insert elements is growing as well. This further indicates cache-related effects.

Figure 4.26 shows the performance advantage of bitwise AND over mod. For this experiment, we varied the number of employed hash functions by explicitly forcing a certain number of bits per stored element, which is normally provided by (4.6). We implemented our PrimeBloom with both the bitwise AND as well as the MOD operator and compared the time needed to probe the Bloom filter for 1 million items. According to the size heuristic, the bitwise AND version will always yield a lower false positive rate for the same amount of hash functions compared to the MOD implementation, since it must increase the required bitset size to the next power of two, thus effectively providing even more available bits per element. When querying a Bloom filter, the evaluation can stop as soon as the first hashed index contains a 0, which we call an early out. The lower false positive rate for the AND implementation has therefore an increased probability for early outs and thus we compare the query time for using all hashes and early outs. Furthermore, we also compare the MOD implementation with both the actual size as returned by (4.6) and for the size used by the AND implementation. Unsurprisingly, computing all hashes is more time intensive than returning upon the first *false*. Yet using bitfields sized to the next power of two, together with a bitwise AND yields a considerable performance increase, even to smaller sized bitsets using a modulo operator for calculating the Bloom filter index.

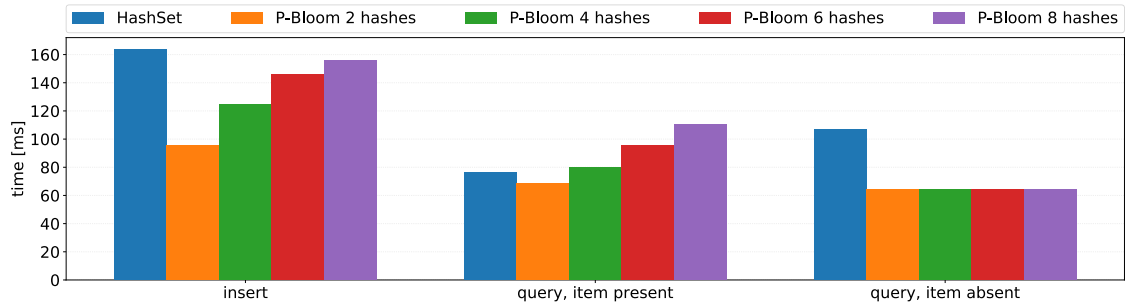


Figure 4.27: Comparison of insert and retrieval times for 1 million values between a HashSet and a prime-based Bloom filter using different numbers of hash functions K, single threaded execution.

In another experiment, we compared the performance of inserting and querying vertex ids from a Bloom filter using several prime-based hash functions and a bitfield against a standard HashSet, using the *std::unordered_set<uint64_t>* from C++ where vertex ids were inserted directly. The corresponding results are depicted in Figure 4.27. One million entries were added to each data structure and hereafter queried again. The Bloom filter was initialized with $10\,\mathrm{bits}$ per entry, which were stored packed within *uint8_t*.

The HashSet's query time is slightly increased, whenever elements are not within the HashSet. This is due to the requirement of performing equality checks on all elements that belong to the slot's bucket, as described earlier in this section. For PrimeBloom, the opposite holds true. For an element to be possibly contained, all of the used K hash functions have to be true. This increases the time for both, insert and retrieval, whenever K is increased. Checking for non-contained elements, however, is faster, as it can be stopped, as soon as the first 0 is detected at a requested index. We also examined results for a pre-allocated HashSet, which were slightly faster than for the dynamic one, but at the expense of even further increased memory requirements.

Partition based workload reduction

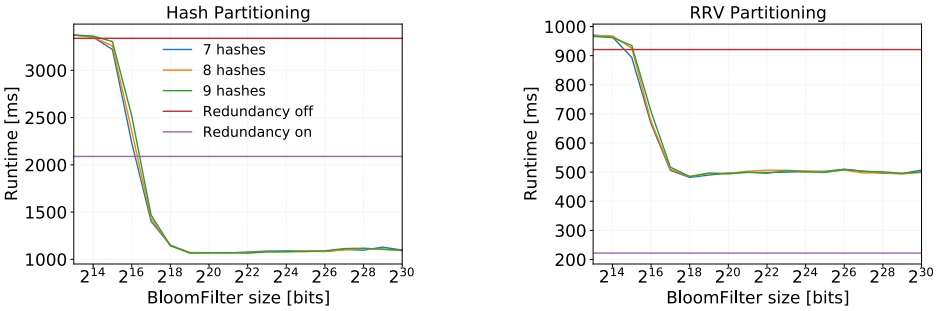
For the experiments in Figures 4.28 and 4.29, we examined both testing queries on all mentioned datasets. We have tested the impact of a different number of hash functions against the baseline, where no Bloom filter was used and thus broadcasts have to be processed by all partitions. Figures 4.28a to 4.28d show the results for using a Bloom filter on baseline NORAD, i.e. broadcasts are used and Figures 4.29a to 4.29d apply PrimeBloom on top of full redundancy.

All experiments from Figures 4.28 and 4.29, except Figure 4.29a, clearly show the desired results. That is, the more bits we spend on the Bloom filter, the bigger is the performance benefit within NEMESYS. Increasing the Bloom filter size is indirectly proportional to the expected false positive rate, thus leading to less unnecessary work.

In detail, Figures 4.28b and 4.28d behave like anticipated. Adding a Bloom filter will increase the performance to allow query runtimes somewhere between standard NEMESYS and the fully redundant storage. On top of that, Figures 4.28a and 4.28c show an improved performance even beyond the fully redundant storage. The reason is, that the Bloom filter can speed up the EB operator by eliminating edge requests for target vertices, that are not stored in the partition. This is not covered by fully redundant storage.

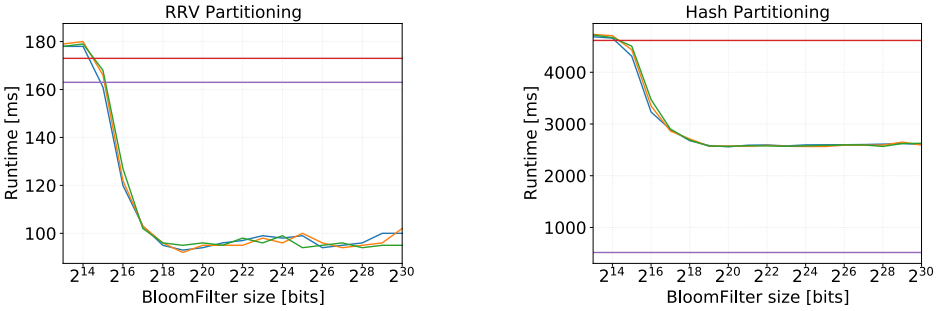
The same holds true for Figures 4.29b to 4.29d. Applying our Bloom filter technique on top of the fully redundant storage leads to an additional performance boost, since the Bloom filter acts as a target vertex index structure.

Figure 4.29a is a special case. For smaller Bloom filters, the systems' performance is slower than standard NEMESYS. Although the query runtime is consistently decreasing, it barely reaches the baseline performance, where no Bloom filter is used. This behavior can be explained with the amount of broadcasts in the system, when only a small number of partitions contains actual data. Since the Bloom filter is only active, *after* the messages have been sent, we still see the same amount of messages in the system and the overhead of checking the Bloom filter is added on top of it.



(a) Biblio, *Quad*, Broadcasts

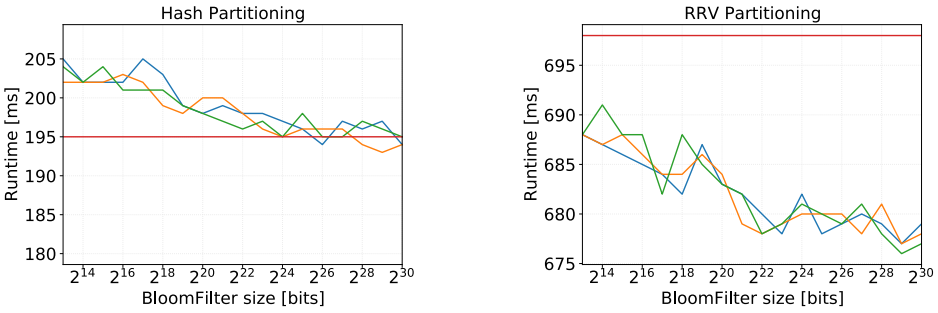
(b) Biblio, *V*, Broadcasts



(c) Social, *Quad*, Broadcasts

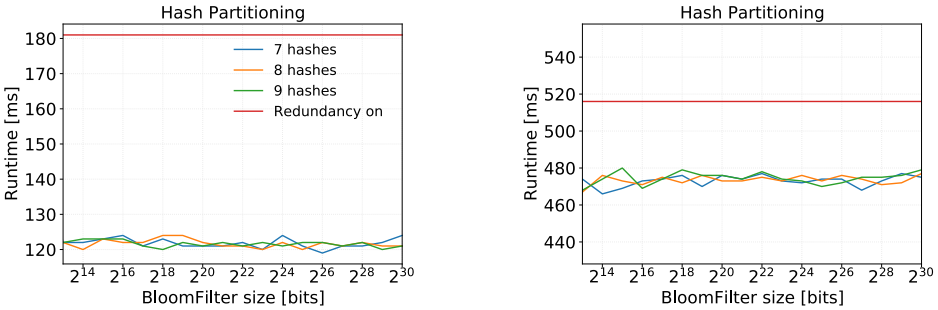
(d) Social *V*, Broadcasts

Figure 4.28: Query Performance with varying Bloom filter parameters, Bloom filter implemented in the partition message queue. Redundancy disabled.



(a) Biblio, *V*, Unicasts

(b) Uniprot, *V*, Unicasts



(c) Social, *Quad*, Unicasts

(d) Social, *V*, Unicasts

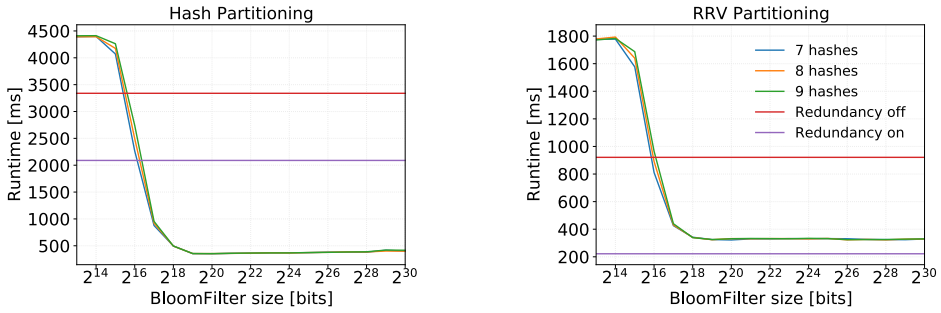
Figure 4.29: Query Performance with varying Bloom filter parameters, Bloom filter implemented in the partition message queue. Redundancy enabled.

Messaging based workload reduction

According to Figure 4.23, we have now implemented the Bloom filter inside the *partition manager*. Instead of creating every message and filtering it later, we can now probe the Bloom filter right before creating the message and thus prevent unnecessary load on the *infrastructure layer*. For better comparability, Figures 4.30 and 4.31 show the same experiments as Figures 4.28 and 4.29, but with the adjusted Bloom filter location.

Most surprising is the observation, that avoiding messages is not generally better. For example, the experiments of Figures 4.29b and 4.31b show a better performance, when messages are filtered out in parallel, instead of avoiding them; the same holds for Figures 4.29a and 4.31a and Figures 4.29d and 4.31d respectively. This effect can be explained with the large serial code segment, which is executed, whenever the Bloom filter is probed. On the other hand, the experiments in Figures 4.30a to 4.30d greatly benefit from the reduced message load. We observe that using the *partition manager* based solution generally improves the GPM performance and yields a speedup of roughly $2\,\mathrm{x}$, compared to the partition based workload reduction.

As a side note, Figure 4.32 shows the amount of messages from *Quad*, which yield no result in their target partitions, using log scale. At first, the effect of the Bloom filter is almost negligible, but starting at a size of 2^{13} bits, we can see a drop in the amount of messages without a result. The drop of unnecessary messages can be perfectly matched to the query runtime, e.g. as in Figures 4.30a and 4.30c. The remaining experiments yield almost equivalent runtime results, which shows, that a GPM query can benefit from both approaches. Based on the results from both the partition and messaging based workload reduction, we can conclude that NEMESYS benefits from using Bloom filters to reduce



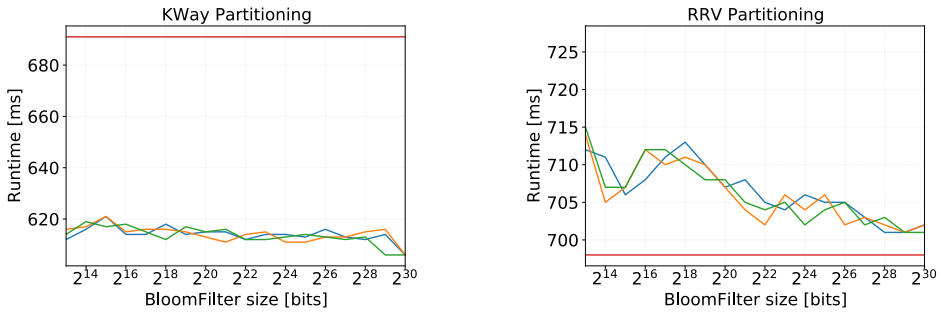
(a) Biblio, *Quad*, Broadcasts

(b) Biblio, *V*, Broadcasts

(c) Social, *Quad*, Broadcasts

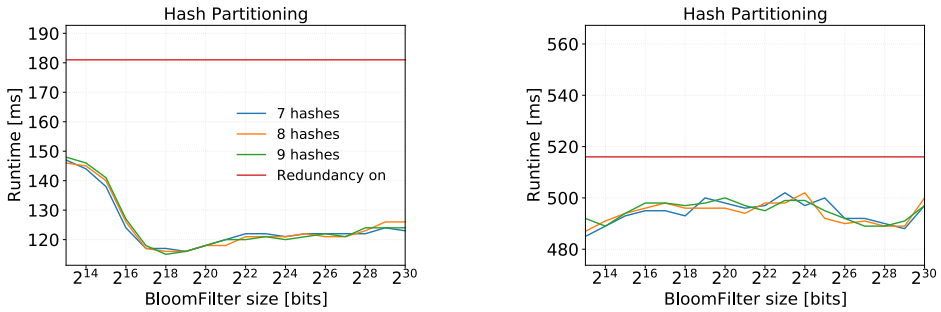
(d) Social *V*, Broadcasts

Figure 4.30: Query Performance with varying Bloom filter parameters, Bloom filter implemented in the partition manager. Redundancy disabled.



(a) Biblio, V, Unicasts

(b) Uniprot, V, Unicasts



(c) Social, *Quad*, Unicasts

(d) Social, V, Unicasts

Figure 4.31: Query Performance with varying Bloom filter parameters, Bloom filter implemented in the partition manager. Redundancy enabled.

broadcasts. Not only can the number of unnecessary messages be effectively reduced, we can also scale the size of the Bloom filter and thus the false positive rate, to perfectly match the current scenario. However, increasing the Bloom filters complexity by increasing its size or hash function count does also introduce more overhead to probe it. Therefore, we should not rely on static values for both M and K and adapt them, as a given graph or workload demands for it. Our implementation of PrimeBloom is thus suitable to answer Challenges 1 and 2 from Section 3.4.

4.3.2 Communication Driven Data Placement

The previous sections provided answers to Challenges 1, 2 and 3 from Section 3.4. As we have shown in Section 4.3.1, combining our PrimeBloom approach with a suitable partitioning strategy can keep up with full redundancy or even outperform it. Thus, our setting allows to efficiently partition data and to avoid unnecessary messages. However, Challenge 3 does not only cover data partitioning, but also where the actual partitions are placed in the system. Since socket-local communication is always more performant than remote communication, we desire to place more tightly connected partitions on the same socket. As a general rule of thumb, the processing of a GPM query should be as distributed as necessary, but also as local as possible. Thus, reducing messages is not the only way to optimize the GPM processing. We can also alter the partition placement, to enforce utmost local computation, while preserving maximum parallelism.

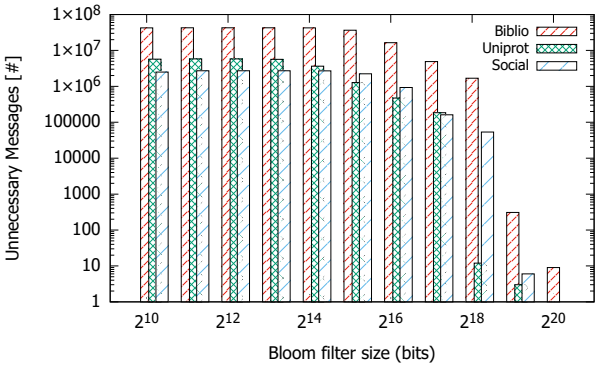
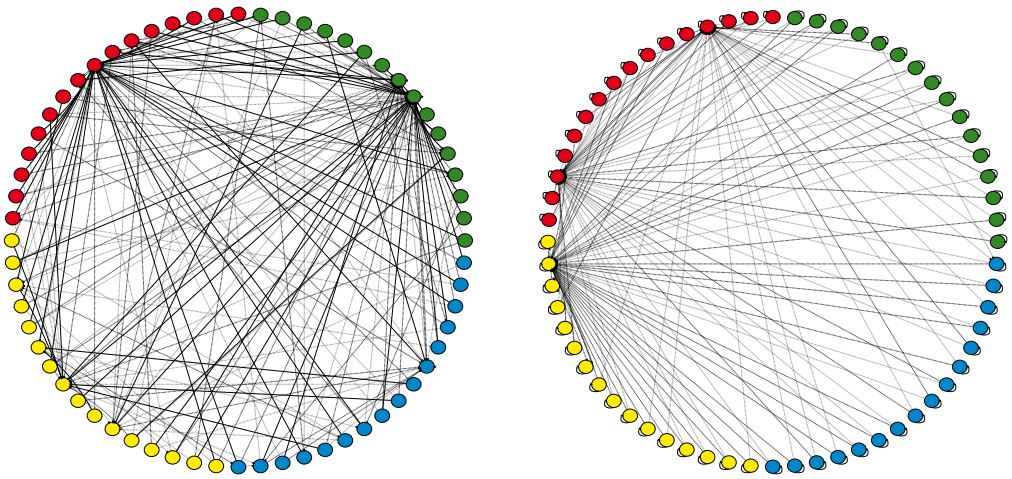


Figure 4.32: Reduction of unnecessary messages for *Quad* with Broadcasts, based on Bloom filter size.

Identifying Locality Hotspots

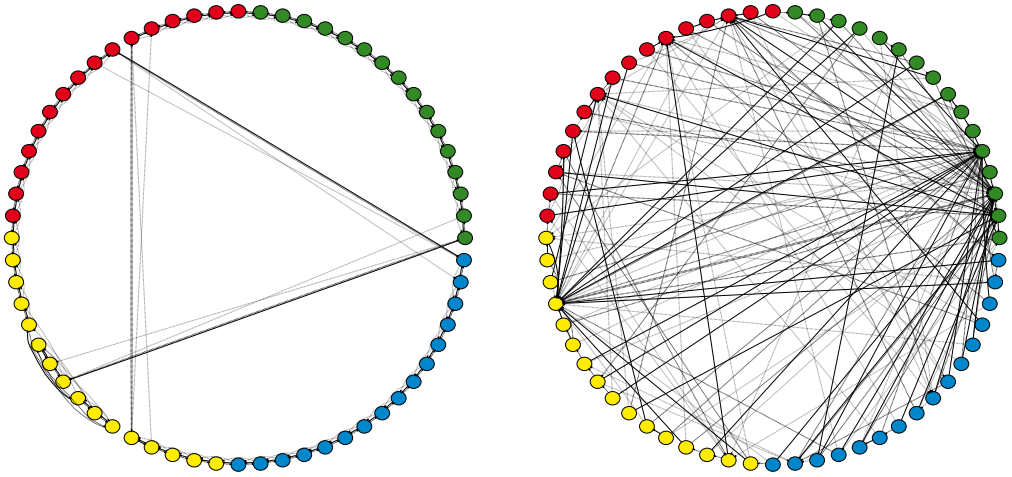
Before altering their placement, we need to identify the most tightly connected partitions. This can be done either offline, based on the partitioning results or online, on a per-query or per-workload basis. For the offline approach, we iterate over all partitions and count all edges towards all other partitions. The result is a set containing the number of inter-partition edges and their target partition. This set can then be sorted descendingly by the interpartition-edge count for every partition, to allow for fetching any Top-n locality demands per partition. The Top-3 connected partitions for the Hash and k-Way partitioning on the Social graph are depicted in Figure 4.33. Nodes represent partitions, same color means they are placed on the same socket and the lines between them visualize inter-partition edges. The linewidths indicate stronger bonds, i.e. more edges between these partitions and nodes with the same color indicate, that these partitions are placed on the same socket. The figure underlines our assumption, that a hash based partitioning is unable to preserve the locality, which stems from the graph's topology. That is, since more arrows are connecting partitions across all sockets, than locally. In contrast, the k-Way partitioning shows the anticipated self-contained partitions, indicated by the



(a) Social, Hash partitioning, 64 Partitions

(b) Social, k-Way partitioning, 64 Partitions

Figure 4.33: Partitioning strategy based Top-3 locality needs per partition, Social graph, 64 partitions.

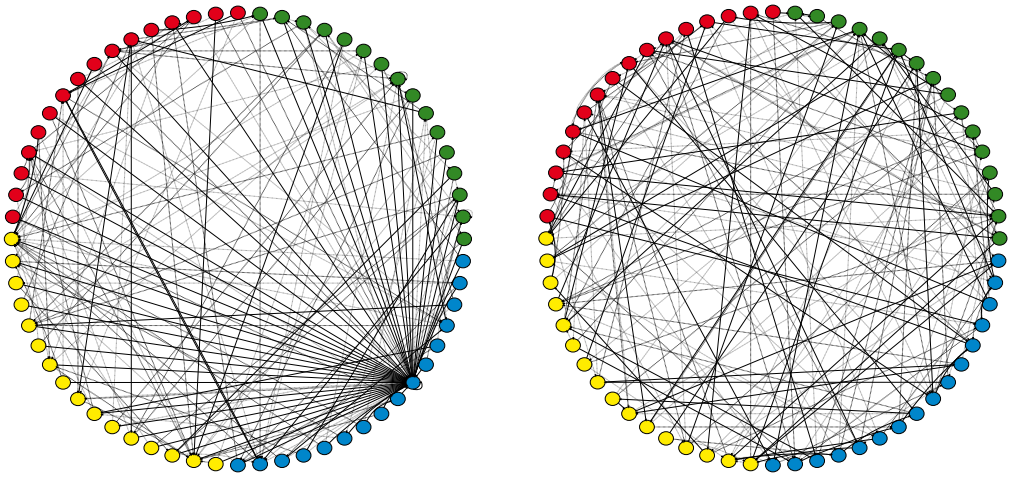


(a) Biblio, BE partitioning, 64 Partitions (b) Social, BE partitioning, 64 Partitions

Figure 4.34: Comparing Top-3 locality needs per partition for BE, 64 partitions.

looping arrows on the sides of the partitions. From these schemas, we can draw several conclusions. First, the hash partitioning distributes the data fairly even among sockets and partitions, as almost all of the partitions are targeted by arrows of any width. Such a schema generally allows for higher degrees of parallelism, due to the broad data distribution. Second, both strategies create local hotspots, which are targeted by a vast majority of other partitions. Especially the single partitions on the red and green socket being potential bottlenecks for the hash partitioning and on the yellow and red socket for the k-Way partitioning. However, how well data is distributed is always dependent on the underlying graph, as shown in Figure 4.34, which shows locality demands for the BE algorithm on the Biblio and the Social graph. This observation further underlines the importance of graph-dependent partitioning strategy selection to provide the best possible baseline, as we have outlined in Section 4.2.1.

The online approach is to perform message tracing during the GPM processing and ana-



(a) No redundancy, no Bloom filter (b) With redundancy, with Bloom filter

Figure 4.35: Comparing actual messaging behavior for the *Quad* query on the Hash partitioned Biblio graph.

lyze the actual dataflow between the partitions. Figure 4.35 shows the Top-3 actual message paths for the *Quad* query on the hash partitioned Biblio graph both with and without our optimizations from Sections 4.2.2 and 4.3.1. Most interestingly, the connectedness of the partitions is different from the partitioning and thus, the online approach is more promising for performing actual data placement optimizations. Comparing Figure 4.35a with Figure 4.35b, we can clearly see the influence of the redundancy and Bloom filter optimizations, as the messaging becomes more balanced and therefore allows for easier data placement optimization.

Both the offline and online approach have applicable scenarios. The offline optimization can always be applied to create a more socket-local communication, without the need to process any GPM query or workload beforehand. The online approach yields an actual image of the systems messaging behavior and can thus create a well tailored data placement optimization. However, the online approach is only applicable for the specific query or workload, which was monitored to create it and thus offers most likely less global value.

Gradient descent based optimization

Optimizing the data placement can only be solved heuristically and not fully enumerative. Calculating all partition placement combinations is prohibitively expensive. The amount of possible, equally distributed combinations ζ for placing n partitions on an s socket system, assuming n is integer divisible by s, can be calculated with the binomial coefficient:

$$\zeta = \prod_{i=0}^{s-1} \binom{n - \tau * i}{\tau}, \qquad \tau = \frac{n}{s} \tag{4.13}$$

For 64 partitions on a 4 socket system, this yields:

$$\begin{aligned} \zeta &= \binom{64-0}{16} \cdot \binom{64-16}{16} \cdot \binom{64-32}{16} \cdot \binom{64-48}{16} \\ &= \binom{64}{16} \cdot \binom{48}{16} \cdot \binom{32}{16} \cdot \binom{16}{16} \\ &\approx 6.6 \times 10^{35} \end{aligned}$$

different combinations. Thus, to reduce the search space, we leverage the well known gradient descent optimization technique (GDO) [Bis06]. The GDO is an iterative algorithm, which requires us to assess all trivial neighborhoods of a given allocation scheme and use the *best scoring* allocation schema as baseline for the next iteration, until no better scoring solution can be found. Within this thesis, *best scoring* refers to the lowest communication cost. In NEMESYS, we can represent an allocation schema using a binary $s \times n$ matrix A, where a 1 denotes, that the partition with id i at column $n_i, 0 \leq i < n$ is located on socket $s_j, 0 \leq j < s$. By default, partitions are allocated to iteratively fill the available sockets and match the worker count on this socket. For $s = 4$ sockets and $n = 8$ partitions, the default allocation for A looks like:

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{4.14}$$

A neighbor is called trivial, if it is created through an atomar change in the original matrix. For our partition allocation scenario, this would mean to move exactly one partition to another socket. However, the goal of the optimizer is to reduce communication cost and thus, all partitions would be moved to one single socket eventually. Therefore we require two partitions to be swapped between sockets, to enforce parity and to keep all sockets utilized. Since a partition can only be placed on a single socket at a time, the trivial neighborhoods of A can be easily generated. That is, by identifying any 1 entry in two different rows, i.e. sockets, and moving them to the row of the other entry, respectively. We extend the optimization process by a set of targeted restarts to overcome local optima. That is, we calculate promising configurations A' and use them as baseline for a new optimization pass. Swapping entire rows would exchange the whole partition set of two sockets, but this has no effect on a fully connected SMP system, since the amount NUMA hops would not be affected. If the processors are e.g. mesh connected, the swapped partitions could change their NUMA distance and thus different communication costs could arise. However, we found that heuristically co-locating Top-n locality demanding partitions (cf. Figure 4.33) serves as fast and reliable option for A' configurations.

For the communication cost we extract an $n \times n$ matrix K, which holds either the amount of edges spanning between two partitions, or the actual messages being exchanged during GPM processing. For $n = 8$ partitions, K could be set as follows:

$$K = \begin{bmatrix} 0 & 3 & 9 & 6 & 5 & 7 & 0 & 0 \\ 0 & 0 & 2 & 1 & 9 & 4 & 6 & 9 \\ 9 & 5 & 0 & 7 & 0 & 2 & 6 & 0 \\ 1 & 6 & 0 & 0 & 1 & 8 & 4 & 4 \\ 5 & 9 & 3 & 3 & 0 & 0 & 5 & 1 \\ 3 & 9 & 0 & 0 & 2 & 0 & 3 & 2 \\ 9 & 5 & 4 & 8 & 0 & 5 & 0 & 0 \\ 1 & 8 & 0 & 9 & 7 & 2 & 6 & 0 \end{bmatrix} \tag{4.15}$$

Every row vector k_i of K represents the respective traffic, originating from the i^{th} partition towards all other partitions, including itself. We model only outgoing messages, since sending a message is performed by a source socket, where the `NodeCoordinator` actively copies messages *towards* a target socket. Thus, the cost is only determined by the target sockets. To calculate the communication cost, we need to estimate the actual cost for exchanging messages between sockets. In this thesis, we leverage the ratio of the actual memory bandwidths, as shown in Table 3.5. For the bandwidths from Table 3.5a, we can derive the $s \times s$ cost matrix C:

$$C = \begin{bmatrix} 1.00 & 6.29 & 6.29 & 6.29 \\ 6.25 & 1.00 & 6.29 & 6.25 \\ 6.31 & 6.31 & 1.00 & 6.31 \\ 6.31 & 6.31 & 6.31 & 1.00 \end{bmatrix} \tag{4.16}$$

With the three matrices A, K and C, we can now calculate the communication cost per socket i. First, we calculate the communication, that originates from socket c_i with the i^{th} row vector a_i of A:

$$a_i K \tag{4.17}$$

This resets to cost of all partitions, which are not located on socket i, to 0. Then we weight the communication of all partitions, with the cost vector of socket c_i:

$$c_i A \tag{4.18}$$

By multiplying (4.17) with the transposed result of (4.18) we receive the total communication cost of socket c_i:

$$(a_i K)(c_i A)^T \tag{4.19}$$

Thus, the total communication cost γ can be expressed as the sum of the individual socket cost scores:

$$\gamma = \sum_{i=0}^{s-1} (a_i K)(c_i A)^T \tag{4.20}$$

With (4.15), we can determine the Top-3 locality demands for (4.14) and create the targeted restart candidates A' accordingly:

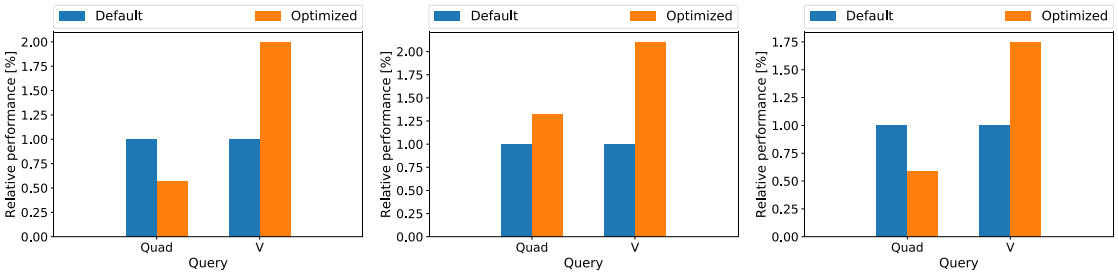
$$A'_1 = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \qquad A'_2 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{4.21}$$

Using (4.20), we get $\gamma_A = 1308.7$, $\gamma_{A'_1} = 1138.79$, $\gamma_{A'_2} = 1202.15$. Thus, one of our targeted restart points already yields a theoretically better communication cost, than the original allocation. For the given matrices, our GDO implementation can achieve $\gamma_{\text{optimized}} = 1101.68$, i.e. a $16\,\%$ cost reduction, for the optimized allocation:

$$A_{\text{optimized}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.22}$$

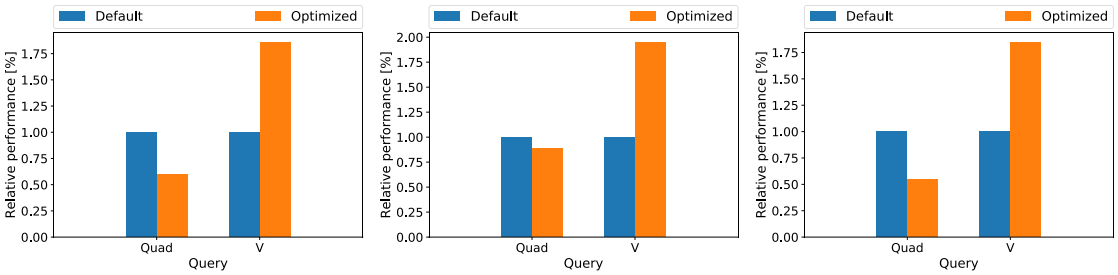
Evaluation

We evaluated the effect of our online partition placement on the *Medium* server with different partitioning strategies and use cases. Figure 4.36 shows the results for the *Quad* and *V* query on the Biblio graph for the NORAD baseline, i.e. when no Bloom filter is active and no redundancy is available. The relative performances of the optimized executions are only to be considered against the respective baseline, i.e. the *Quad* default execution has another query runtime than the *V* query. We observe a speedup of $2\,\text{x}$ for the *Quad* query, when its executed against the BE or RRV partitioning. For $Quad_{\text{BE}}$, we calculate $\gamma_{\text{default-BE-Q}} = 639\,113\,917$ and the optimized communication score as $\gamma_{\text{opt-BE-Q}} = 511\,228\,551$ which is $79.9\,\%$ of the score for the default allocation. For the RRV and HV partitioning, our optimizer yields similar results of a $20\,\%$ lower communication score, yet only RRV partitioning also benefits from adapting the partition placement. After the



(a) BE partitioning (b) HV partitioning (c) RRV partitioning

Figure 4.36: Relative query runtimes for *Quad* and *V* with adapted partition placement. Biblio graph, 64 partitions, Bloom Filter disabled, redundancy disabled, *Medium* server.



(a) BE partitioning (b) HV partitioning (c) RRV partitioning

Figure 4.37: Relative query runtimes for *Quad* and *V* with adapted partition placement. Biblio graph, 64 partitions, Bloom filter enabled, redundancy disabled, *Medium* server.

movement, we can trace more local communication than for the original allocation, yet the query performance decreases. The same effect occurs for all *V* query runs, where we face a slowdown, even with a theoretically lower communication score. We see the main issues for this effect to be based on the routing table designs. HV uses a *compute* routing table design, BE and RRV employ *lookup* routing tables. Moving partitions among sockets creates a thin layer of indirection inside the `NodeCoordinators`, which handle them. The main benefit of the compute design roots in the absence of a lookup structure, which eliminates memory accesses to retrieve the actual target partition for a given message. Creating more indirections will thus only add overhead, which could not be amortized in this experiment.

Figure 4.37 shows the same experiment for NeMeSys with enabled Bloom filters but without redundancy, since Bloom filters are intended to be a substitute for it. For this experiment, we used the messaging based workload reduction, i.e. the Bloom filter was placed in the *infrastructure layer*. We can now observe a speedup for the *Quad* query on all partitioning strategies, yet the *V* query still suffers from slowdowns. The newly occurring speedup for HV can be easily explained by the reduced amount of messages, that have to be processed. Less messages lead to less overhead through the partition movement indirection layer, which manifests as a small, yet observable speedup. The general slowdown for the *V* query might be an artifact of the results of our optimizer. Since all partitions are potentially communicating with each other, we build a 128×128 communication matrix, to represent all possible communication paths. However, not all partitions hold data, which is relevant for every query. Thus, some partitions of both the outgoing and incoming edge containers do not produce or receive any messages. This leads to the effect, where we see half of the sockets being filled with all outgoing edge partitions and the other half of the sockets holds only incoming edge partitions. Despite locally minimized communication scores, this effect can introduce more overhead. The system is unable to perform messaging optimizations like batching, when only one partition on a remote socket is touched. On the machine level, sending a message is a `memcpy` operation. Performing many `memcpy` operations on small chunks of data is always slower than using `memcpy` on one large chunk of data. Thus, if only very few messages are sent to a remote socket, we can face runtime slowdowns.

Both the *Quad* and *V* query represent completely unbound analytical queries, i.e. all query vertices are variables. To investigate the effect of our online adaption method on more specific queries, we took the graph schema of the Biblio graph and generated 100 distinct GPM queries. These queries form differently shaped patterns and exhibit at least one bound variable, i.e. a given author or a fixed conference. The query set was then executed both as single queries and as a workload with batch sizes of 10 queries each. For

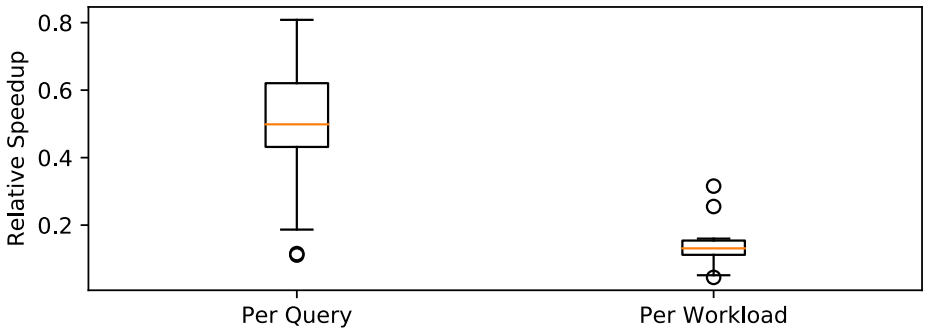


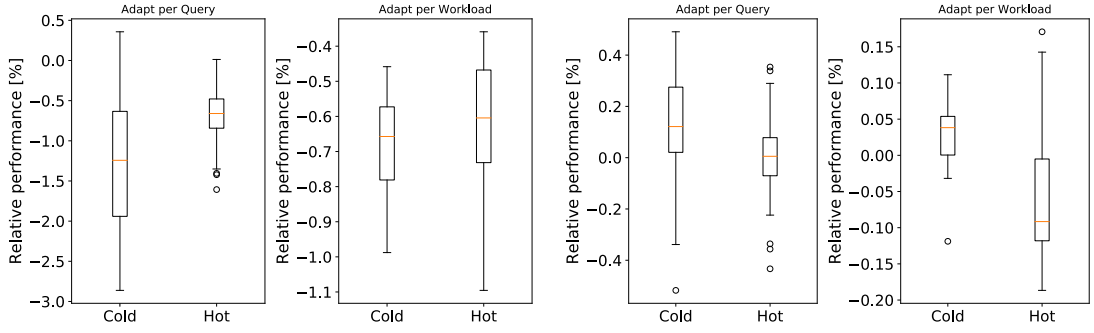
Figure 4.38: Speedup variances for scenario dependent partition movement. BE partitioning, Biblio graph, 64 partitions. Bloom filter active, redundancy used, *Medium* server.

both scenarios, we monitored the actual messaging and the resulting traces were then fed into (4.20) for optimization. Figure 4.38 shows the achieved speedups for all queries and workloads by co-locating partitions with higher message traffic using boxplots.

The optimized partition placement for single queries can lead to a considerable performance gain of up to $80\,\%$, compared to the default allocation schema. However, such fine granular movement is rarely efficient, because moving the partitions within the NUMA system can impose a significant cost. This depends on how much partitions have to be moved and if the NUMA system is fully connected or not. The workload-based optimization still yields performance gains ranging from $5\,\%$ to $40\,\%$, which can amortize the partition movement cost. The reduced performance gains, when compared to individual query optimization, are explained with the more differentiated communication paths, which are now packed into one workload. Individual queries benefit less from the global optimization or could even be slowed down. Yet, we do not need to move many partitions as often, which also adds to the actual performance gain.

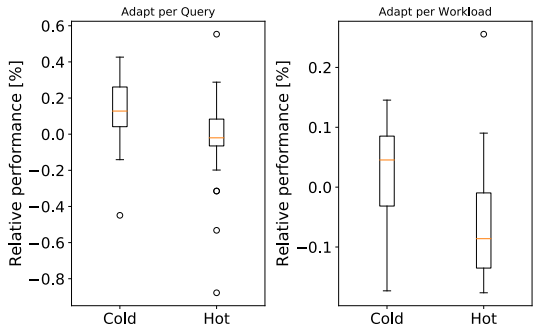
During our experiments, we observed partly strong variations of the query runtimes for all queries and workloads. To isolate the root cause, we repeated the experiments from Figure 4.38 with the same queries and workload configurations using all combinations of disabled and enabled Bloom filters and redundancy. The results are shown in Figure 4.39. In this experiments, we also investigated the effect of *hot* and *cold* adaption, where *cold* means, we restart NEMESYS for every measurement, reload the graph, perform the individual movement and run a query or workload. For the *hot* experiments, we load the graph once, move the partitions and measure the query performance for all queries and workloads subsequently. All measurements were repeated 20 times and we used the average runtime as value per query and workload to create the individual *hot* and *cold* bars. Most surprising are the individually huge slowdowns of almost $-3\,\mathrm{x}$ for a single query in Figure 4.39a. We found, that mainly short running queries, i.e. around $50\,\mathrm{ms}$, are affected by larger slowdowns. The same observation holds true for Figures Figures 4.39b to 4.39d. Figure 4.39c has the same configuration as Figure 4.38, yet it also exhibits individual slowdowns for some queries. We believe the problem is rooted in the general NORAD architecture and the prototype implementation, on which we built NEMESYS.

One potential issue is the degree of indirection, which is intentionally built into the system. The ERIS baseline implementation was designed with a maximum of flexibility and adaptivity in mind. This requires abstraction layers within and between all major system components, such as *storage* or *processing*. These indirections, paired with the inherent asynchronous processing, can result in unpredictable runtime behavior. For example,

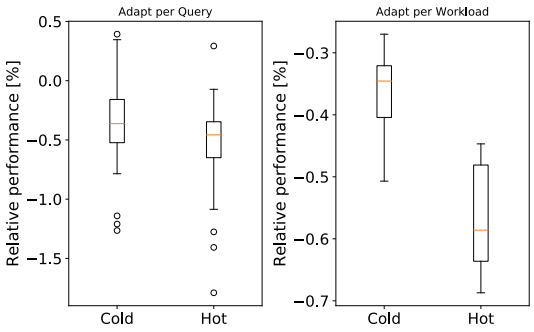


(a) BE, no Bloom, no Red.

(b) BE, no Bloom, Red.



(c) BE, Bloom, Red.



(d) RRV, Bloom, no Red.

Figure 4.39: Speedup variances for different NEMESYS configurations. Biblio graph, 64 partitions. *Medium* server.

the workers monitor themselves, if any work has been performed. If not, a worker will put itself to sleep for a grace period, to preserve energy and keep the CPU utilization low. If such a sleep cycle is initiated, right before a message arrives, the processing of this message is delayed until the worker awakes again.

We tested the influence of the sleep cycles with varying sleep times, ranging from completely disabled sleeps to 100 ms. We found, that completely disabling sleeps leads to a permanent polling of the workers, which in turn produces more pressure on the local message buffers, which are more often locked to check for processable messages. The permanent checks for available messages will then lead to further decreased performance, since enqueuing messages can only be done if the buffers are not locked. Thus, on the one hand, sleep cycles can help to increase the systems' throughput. On the other hand, ill-timed sleep cycles have varying impacts on the performance of a single query.

Asynchronous execution is considered to excel at parallel processing and to allow for maximum parallelism. However, the parallelism enabling asynchronicity can lead to unforeseeable effects like the one we just described. The high potential for more performance is thus traded for less robustness and weaker runtime guarantees.

4.3.3 Lessons learned

This section covered our infrastructural optimization techniques, which are aimed to reduce or streamline the messaging inside NEMESYS. Efficient communication plays a

key role for good performance, as outlined in Section 2.3.2. The baseline NORAD implementation relied on broadcasts, whenever the required locality information was not available. This is not feasible during GPM processing, since the potentially large intermediate result space floods the *infrastructure layer* with broadcasts and thus slows the whole system down. Our first approach was to add full redundancy, to completely eliminate broadcasts, at the expense of higher main memory consumption. However, this overhead could become unbearable, since everything, including metadata and indexes, has to be held twice. With the Bloom filter implementation, we enabled GPM processing to achieve competitive performance to redundancy, while maintaining a low memory footprint. The scalable nature of the Bloom filter further allows us, to set its size according to any memory budget, if necessary. The location of the Bloom filter inside the system can have varying impacts on the performance. Both presented implementation variants can lead to a speedup, where the other does not. Thus, the employed variant should be decided on a per-workload basis, if possible.

The locality of the data is originally only determined by the employed partitioning algorithm and the default partition allocation scheme in the system. This can lead to the problem, where highly communicating partitions are placed on two distinct sockets and thus inter-socket communication occurs. We introduced a method to monitor and update the partition placement. Depending on the desired output quality, our optimizer can work on either a given partitioning and its edge cut or on a full message trace of a query or workload. We showed, that adapting the partition placement to achieve more local or more clustered communication can have a positive impact on the GPM performance. For now, our optimization model does only consider parity for partition distribution. We envision better placement results, when the optimizer considers unevenly distributed partitions, but with penalties for the resulting lack of available parallelism.

Combining the optimal SC with Bloom filter support and a slim routing table design does significantly improve the NORAD baseline. However, the nature of an asynchronous system in combination with indirection layers can lead to unforeseeable runtime behavior. We consider the varying query runtime to be an artifact of the interplay of asynchronicity, indirections and adaptivity knobs, since we observe these variations for every experiment and every distinct start of the system. However, this problem is rooted deep inside the kernel of the baseline implementation and does not hinder the execution of GPM itself. Our optimizations could improve the systems' performance, even under the presence of these phenomena, which underlines their applicability. Identifying the actual reason for varying query latencies is indeed an important part of a final software product, yet it does not prevent the system from functioning. Therefore we leave further tracing of the root cause within the systems internals for future work.

EVALUATING NEMESYS AGAINST WIKIDATAS REAL WORLD DATA

Processing real world graphs is a different challenge than processing synthetic datasets, as their unique characteristics are often not reflected in data generators. We declared Wikidata as the target use case for this thesis in Section 2.4 and thus, we evaluate the capabilites of NeMeSys for processing a real world data graph in this chapter. Therefore, we divide this chapter into two parts. First, we analyze NeMeSys general behavior when processing Wikidata query logs. After the workload analysis, we enable the Energy Control Loop (ECL) features of the underlying ERIS implementation and investigate, if the energy saving principles can be seamlessly integrated for graph processing.

5.1 WIKIDATA AS IN-MEMORY SCENARIO

The knowledge graph of Wikidata is freely available as regularly appearing database dumps[1]. These dumps are provided in different formats, e.g. JSON or RDF. Furthermore, differently sliced versions are available, such as the full graph with all entities and labels in all languages or reduced versions, which only contain truthy statements and no qualifiers or other metadata. These do only represent direct statements, which are true themselves and not inferred from other statements of the knowledge base.

As of April 2020, the most recent compressed archive of the full dump exhibits a file size of approximately 127 GB. Thus, the whole, uncompressed dump could hardly be stored completely in-memory and NeMeSys could not process the graph on a single SMP system, which is currently available to us. Therefore, we limit the usage of Wikidata to a truthy statement dump in this thesis. The dataset is already provided in the N-Triples format, which we introduced in Section 3.3.1. We prepared the dataset according to the steps explained in the same section, i.e. the triples have been dictionary encoded and we created a re-encoded version of the dataset, to speed up the graph ingestion time between the experiments. The final data graph contains a total of 223 M edges, which corresponds to a 3.8 GB data file on disk.

The experiments of this section were performed on the *Medium* server from Table 3.1. Since we are working on a 64 bit platform, we also use 64 bit integers to represent our data. Thus, a triple requires 24 Byte of raw memory to be stored. The raw in-memory size of the whole data graph is therefore expected to be at least 5.3 GB, when stored without redundancy and twice the size, if stored redundantly, without the containers holding the data. However, after ingesting the graph, our system reports a total of 21.24 GB for outgoing edge storage and 44.5 GB for fully redundant storage with a *compute design* routing table, including data containers but without any meta or index data. If we use a lookup routing table without Bloom filters, the memory consumption climbs to 57.6 GB for fully redundant storage (cf. Table 5.1). Due to this overhead, we are unable to build partition-internal indexes to further accelerate GPM for the Wikidata use case, as this ultimately leads to an out-of-memory error. We found the reason to be hidden deep inside the *storage layer* of the ERIS prototype code. The system was intended for a maximum of flexibility and to enable adaptivity, like data format changes. Thus, a single record is stored in a wide table approach, where every record requires additional metadata to

Table 5.1: Wikidata in-memory size for a *lookup* routing table with 256 Partitions.

	Redundancy Off	Redundancy On
BloomFilter Off	29.30 GB	57.61 GB
BloomFilter On	33.62 GB	66.89 GB

[1] https://www.wikidata.org/wiki/Wikidata:Database_download/en [Last Accessed: 02.04.2020]

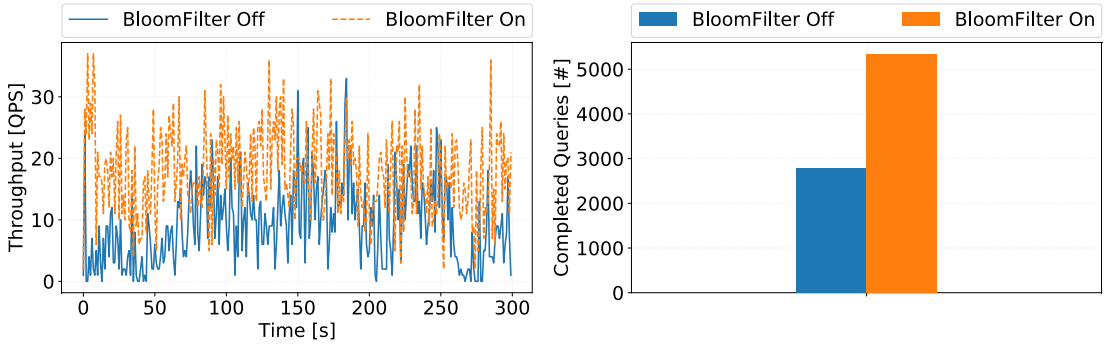


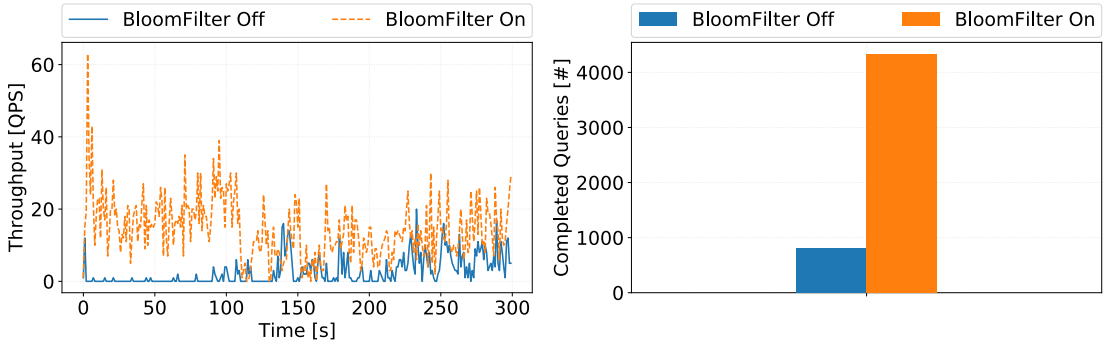
Figure 5.1: Wikidata workload, VPQ mix, redundancy enabled, Hash Partitioning, Medium server.

identify its contents. Furthermore, the stored data is enriched with linkage information, to reconstruct tuples, when they are transformed from row to columnar storage and vice versa.

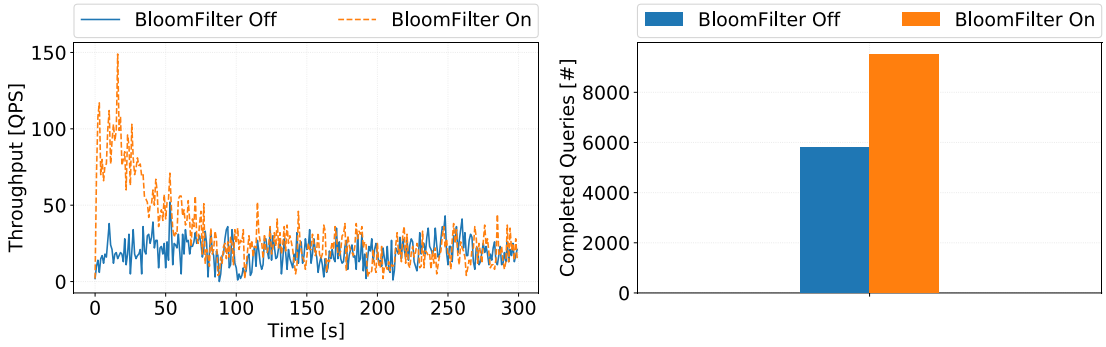
The observed static overhead per tuple of the underlying storage module discourages the processing of larger graphs in its current state. This increased memory consumption does not only affect the data storage, but also online data processing in general, as more data has to be touched to retrieve any triple. However, the current storage backend does only limit the maximal number of edges for a given graph, until our system would run out of memory. Yet, it does not prohibit GPM on NORAD systems in general. We see the development of a dedicated graph storage module as a general improvement for GPM processing on NORAD systems, but defer this endeavor to future work.

After preparing the graph for proper ingestion, we created the query set for our experiments. As part of an earlier cooperation with the authors from [MKG+18], we had access to earlier SPARQL query logs, which were not made available, due to privacy constraints. However, we were allowed to anonymize the queries and use them for benchmark purposes. The raw query set contains approximately $3.8\,\mathrm{M}$ queries, which we classified into single- and multi-statement queries, as well as star-queries, i.e. queries containing a Kleene star. As outlined in Section 3.3.3, NEMESYS supports the RPQ subclass of VPQs, where we limit the Kleene star to one predicate and thus we limit star queries to single-statements with a Kleene star. After removing query duplicates, our three query sets contain $858.0\,\mathrm{k}$ unique single-statement, $54.3\,\mathrm{k}$ unique multi-statement and $309.5\,\mathrm{k}$ single-star queries (VPQs). For the query mix, we used a ratio of $80\,\%$ single-statement queries, $15\,\%$ multi-statement queries and $5\,\%$ VPQs, which approximates their ratio from the original query log. We call this workload the VPQ mix. However, we allow VPQs only for experiments with redundancy, because the inherent path recursion would create an indefinite amount of broadcasts, which would ultimately break the system. For experiments without VPQs, we set the ratios to $80\,\%$ single-statement and $20\,\%$ multi-statement queries and call this workload the standard mix.

For the experiments in this section, we increased the partition count above the number of physical cores to 256 partitions. That is due to the lack of partition internal indexes, to reduce the individual partition sizes and thus allow for faster partition scans. Furthermore, our benchmark setting creates a randomly selected workload mixture of the three query sets, which is seeded with the same value for all experiments to ensure equal conditions. After ingesting the graph, we spawn 100 parallel clients, which each post one query of the workload mix per second against NEMESYS for a benchmark duration of 300 seconds. Figure 5.1 shows the results for the VPQ mix using a *compute design* routing



(a) Round-Robin Vertices / Lookup design.



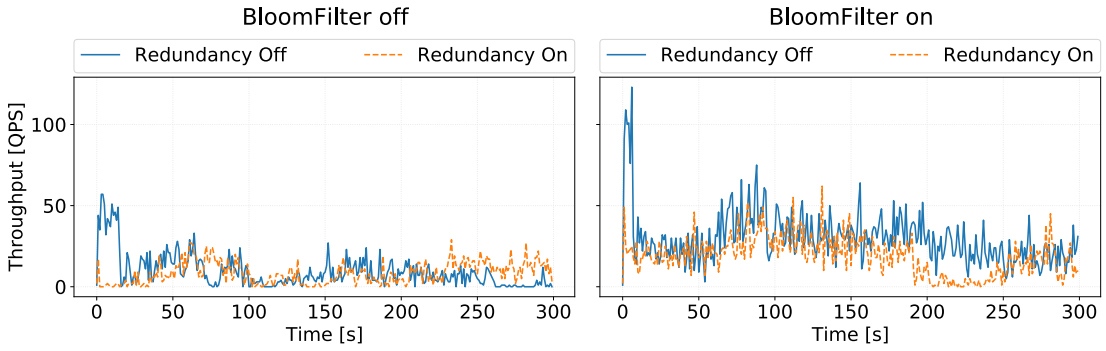
(b) Ranged Vertices / Hybrid design.

Figure 5.2: Wikidata workload, VPQ mix, redundancy enabled, *lookup* vs. *hybrid design* routing table, Medium server.

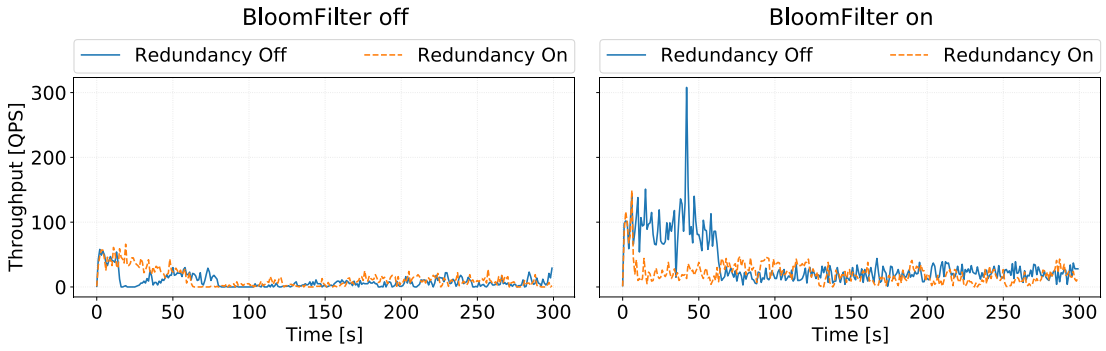
table on the left and the total completed queries over time on the right for runs with and without Bloom filters. When the Bloom filter is disabled, NeMeSys is able to complete 2787 queries in 5 minutes, which corresponds to an average of 9.29 queries per second (Qps). Enabling Bloom filter support increases the systems throughput to 5333 completed queries or 17.77 Qps on average. Clearly, the reduction of unnecessary messages in the system increases the overall throughput.

Figure 5.2 compares the *lookup design* routing table against the improved *hybrid design* routing table for the RRV algorithm from Section 4.2.1. The hybrid version of the RRV algorithm is denoted as *Ranged Vertices (RV)*. Obviously all partitioning strategies, except the hash partitioning, qualify for a conversion into the *hybrid design*. We have chosen the RRV strategy as representative, since all strategies benefit from the performance gain of the *hybrid design*, yet the RangedVertices yields the best overall results.

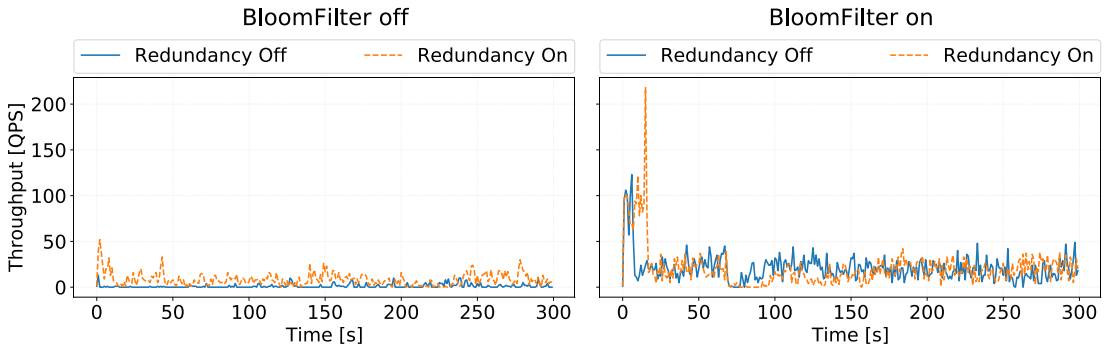
Figure 5.2a shows the major disadvantage of the *lookup design* routing table, with a significant performance drop compared to the *compute design* from Figure 5.1. The *lookup design* routing table holds a copy of all vertices in the graph, to map them directly to their respective target partitions. We can compute the approximate size of the routing table by subtracting the raw memory consumption without metadata of approximately 21 GB from the 29 GB for a a graph with *lookup design* routing table (cf. Table 5.1) without redundancy. This yields a routing table size of circa 8 GB. Since the routing table is stored solely on the first socket of the system, every `NodeCoordinator` has to access that socket to fetch the routing information. One explanation for the very low throughput in the first



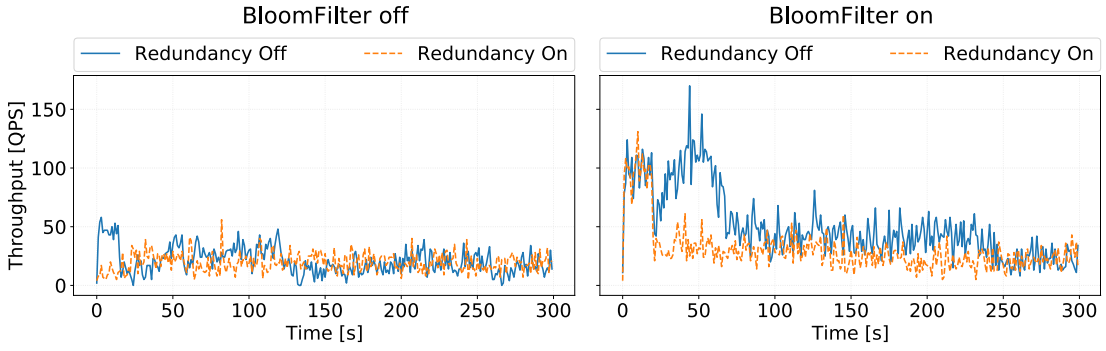
(a) Hash Partitioning / Compute design.



(b) Balanced Edge Partitioning / Lookup design.



(c) Round-Robin Vertices Partitioning / Lookup design.



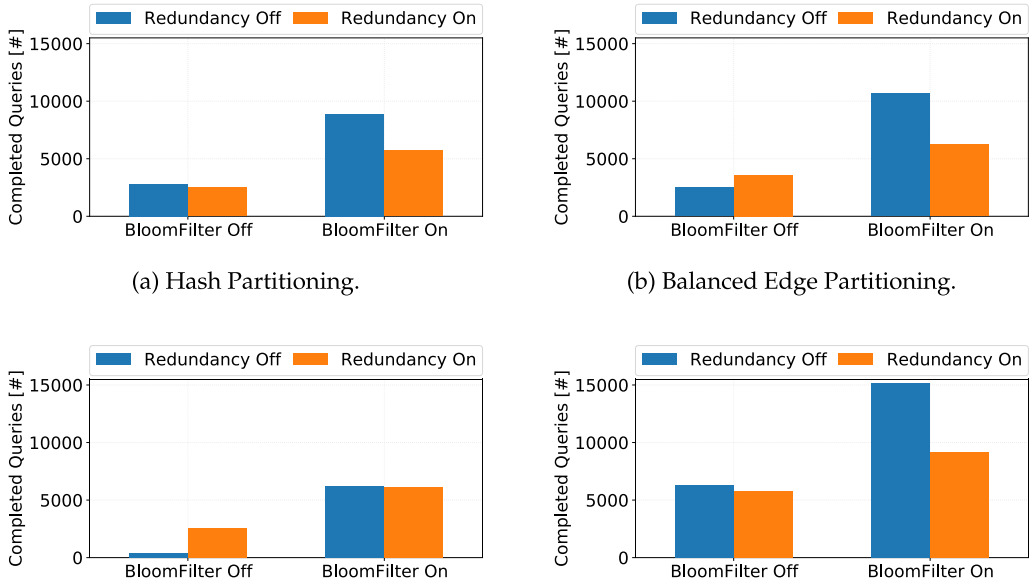
(d) Ranged Vertices Partitioning / Hybrid design.

Figure 5.3: Wikidata workload over 300 s, different partitioning strategies, standard mix, Medium server.

minute is, that the processors need to employ proper caching of relevant parts of the routing table. Obviously, the partitioning itself can lead to a high inter socket communication, which further reduces the throughput.

In Figure 5.2b, we kept the data partitioning but applied our techniques to convert the data and routing table to conform the *hybrid design*, where the routing table only contains as much entries, as there are partitions in the system. For this experiment, the routing table contains only 256 entries, which corresponds to a total size of approximately $4\,\mathrm{kB}$, i.e. $2\,\mathrm{kB}$ for each 256 border vertices and pointers to their partitions. The considerable smaller routing table size leads to a substantial increase of the throughput for both experiments with and without Bloom filter usage. The RV strategy yields a total of 5807 completed queries without Bloom filters, which surpasses the performance of the *compute design* table with Bloom filters from Figure 5.1. This observation underlines our claim from Section 4.2.1, that more graph oriented partitioning algorithms result in a better system performance. Furthermore, the processing of VPQs can also greatly benefit from the routing table size. Due to the small size of $4\,\mathrm{kB}$ and its frequent utilization, the routing table should be cache resident for all processors and thus, frequent partition requests during the path recursion can be answered considerably faster.

The experiments in Figure 5.3 compare the behavior of three different partitioning strategies and all three routing table designs. To include experiments without redundancy, we switch the workload to the aforementioned standard mix, without VPQs. The y-axes in the plots have been adjusted for the different partitioning algorithms, yet the two plots for one partitioning algorithm share the same y-scale to allow for better readability and visual comparison. As a first observation, we see a generally higher throughput for all experiments, where Bloom filters are enabled. In the first seconds of the benchmark, all variants exhibit a plateau or moderate peak in finished queries, followed by a decline of the throughput curve. We see the occurrence of several multi-statement queries as the reason for this behavior. Several partitions are busy scanning for the data, which is related to these queries and thus can not answer other incoming queries quite as fast.



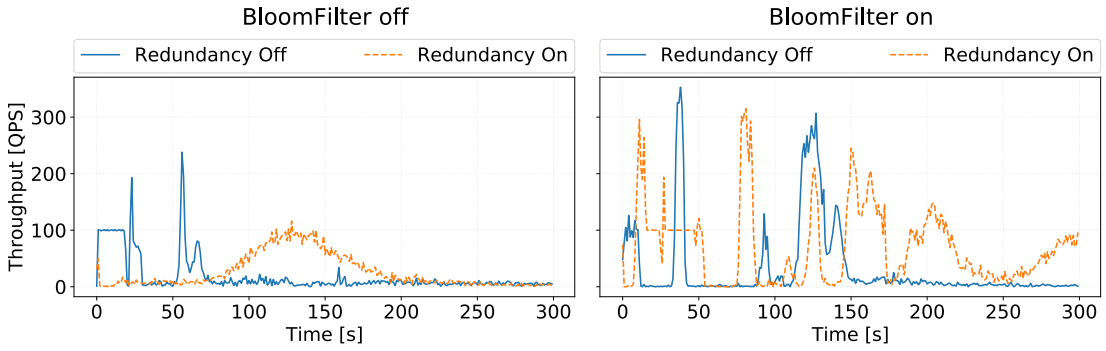
(a) Hash Partitioning.

(b) Balanced Edge Partitioning.

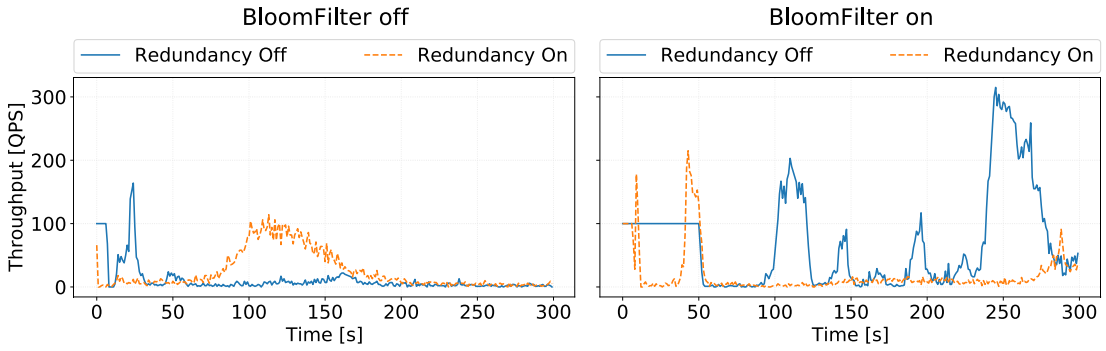
(c) Round-Robin Vertices Partitioning.

(d) Ranged Vertices Partitioning.

Figure 5.4: Total completed queries over $300\,\mathrm{s}$ for Figures 5.3a to 5.3d, standard mix, Medium server.



(a) Hash Partitioning / Compute design.



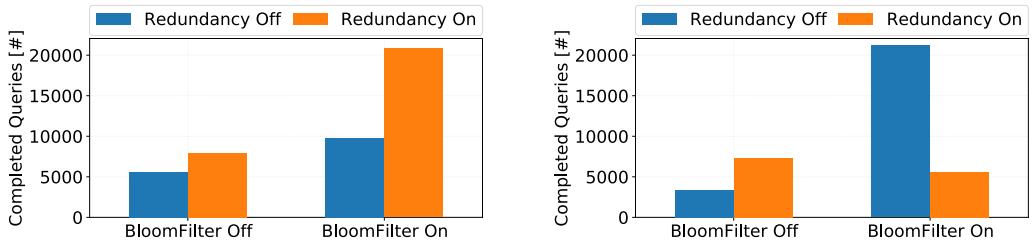
(b) Balanced Edge Partitioning / Lookup design.

Figure 5.5: Wikidata workload over 300 s, different partitioning strategies, standard mix, HPE Superdome Flex.

Second, employing redundancy does not universally lead to a higher throughput, as observed in Figures 5.3a, 5.3b and 5.3d. This holds especially true for all Bloom filter supported experiments. The same effect was already observed earlier, in Figure 4.31b, where the combination of redundancy and Bloom filters lead to a slowdown. However, we see an overall performance increase compared to the experiments without Bloom filters, yet the variants without redundancy can create more individual benefit.

Figure 5.4 compares the overall completed queries over 300 seconds for all experiments from Figure 5.3. For the experiments without any of our optimization techniques, i.e. without redundancy, no Bloom filters and disabled *hybrid design* routing table, the hash partitioning yields the highest completed queries. The Ranged Vertices partitioning, which uses the *hybrid design* routing table, optimizes the lookup metadata and can already outperform the hash-based *compute design* routing table with a factor of 2.24 x. If we further optimize the messaging with Bloom filters, the *hybrid design* routing table can increase the throughput even further to almost 50.4 Qps, compared to 29.6 Qps for the hash partitioning with the same Bloom filter and redundancy settings.

The TU Dresden just installed a new high performance compute server with the name HPE Superdome Flex, shortly before completing this thesis. This server features a total of 1792 logical cores, which are distributed among 32 sockets in total. Every socket provides approximately 1500 GB of main memory, totaling in almost 48 TB of shared main memory inside one single SMP system. We could get access to the bare machine for a brief period of time and were allowed to perform our Wikidata benchmarks. Figure 5.5 depicts the workload processing behavior with exactly the same settings as for Figure 5.3,



(a) Hash Partitioning / Compute design.

(b) Balanced Edge Partitioning / Lookup design.

Figure 5.6: Total completed queries over 300 s for Figure 5.5, standard mix, HPE Superdome Flex.

i.e. the standard mix with 100 parallel clients and the same workload seed. According to our findings from Section 4.2.1, we observed a performance degrade, when all hyperthreads are used and thus we limited the execution to only physical cores, i.e. 896 with the *nat/int* allocation strategy (cf. Section 4.2.2). The figure shows the runtime behavior for both a *compute* and *lookup* design routing table.

Most interesting is the similar, yet a little bit increased, throughput, when we compare this experiment to Figures 5.3a and 5.3b with disabled Bloom filters. However, enabling Bloom filters leads to a drastic change in the runtime behavior, i.e. the system is able to complete more queries, which is reflected in the total completed queries from Figure 5.6. The reason for the higher throughput is simply reasoned by the higher parallelism, that is enabled through more cores. However, we do not see a linear scaling, i.e. 8 x more cores does not lead to the same speedup.

One reason for the slightly higher performance without Bloom filters, compared to the *Medium* server, may simply be reasoned by the slightly stronger hardware specifications of the installed processors. The HPE Superdome Flex uses Intel® Xeon® Platinum 8276 processors, whose base and turbo clocks are 2.2 GHz and 4.0 GHz respectively, whereas our *Medium* server uses Intel® Xeon® Gold 6130 processors with 2.1 GHz and 3.7 GHz as base and turbo clocks. The Platinum processor does also exhibit 38.5 MB of last level cache, which is almost double the size of the Gold's 22.0 MB, which allows to cache more intermediate results or slightly larger parts of the routing table. Thus, we conclude that NeMeSys can also directly benefit from stronger hardware specifications.

Another reason is the size of the data partitions. The amount of partitions was increased from 256 to 896, which lead to less data within the individual partitions. Thus, every worker was responsible for fewer data, which in turn lead to faster scanning of a partition. However, the experiment with enabled Bloom filters shows only a small increase in completed queries for disabled redundancy in Figure 5.6 but a considerable speedup for enabled redundancy when the *compute design* routing table is used. For the *lookup design* routing table, this behavior is swapped. Enabling redundancy even results in a performance penalty, whereas disabled redundancy with Bloom filter support achieves more completed queries, than the *compute design* routing table.

Clearly, the NUMA architecture of the server does play a key role in this experiment. The *Medium* server is a fully connected system, whereas the HPE Superdome Flex exhibits up to 5 hops (cf. Figure 3.1b) to reach the furthest socket. The *compute design* does not require any additional data structure to determine the target partitions, which means, that no processor needs to cache any routing information. Therefore, redundant data storage does only lead to an increase of locally stored data, but no additional network traffic.

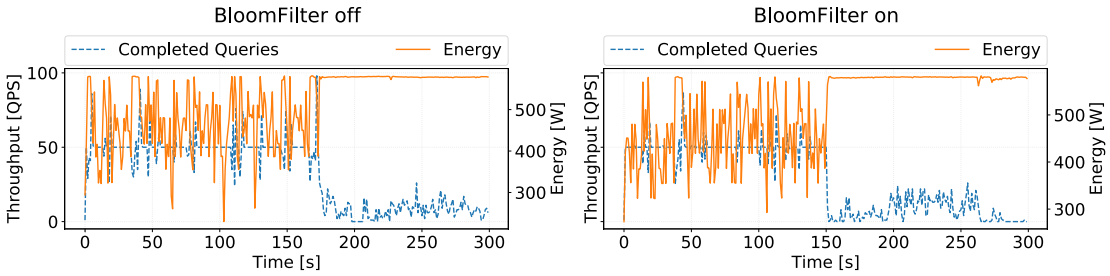
As mentioned in Section 4.2.2, the routing table is not explicitly replicated among the sockets. Thus, accessing uncached parts of the routing table requires a remote memory access, which suffers the NUMA penalty, that depends on the amount of required hops. We can therefore conclude, that the additional routing table for the redundant storage becomes a major bottleneck on servers with more than just one NUMA hop. Furthermore, we see that NEMESYS is able to perform on SMP systems of any size. However it is not trivial to deploy the system on a large scale SMP system and expect linear speedups. We believe, that adjustments in the caching of the routing table could lead to better performance, but our access was very limited in its duration and thus more experiments could not be realized.

5.2 APPLYING ERIS ECL FEATURES ON NEMESYS

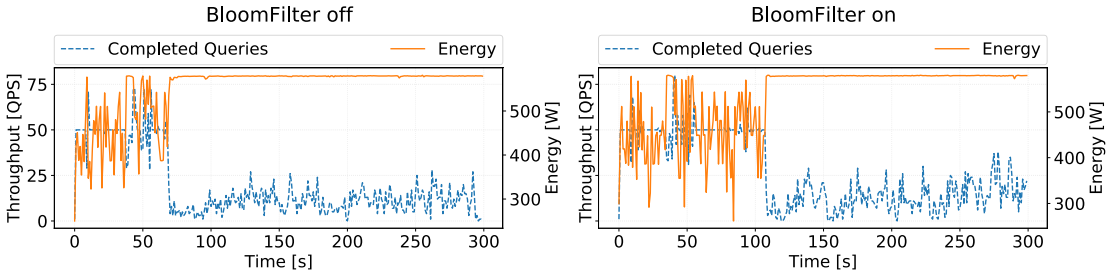
Section 3.2.4 introduced the basic concept of the ECL. This self-optimizing loop performs system internal monitoring and manages the platform resources, such as CPU or memory clocks or storage formats. In this section, we analyze the possibilities of applying the energy controlling facilities of the underlying ERIS implementation to NEMESYS. The experiments could not be extended to the previously introduced HPE Superdome Flex, since our energy controlling facilities require root privileges. These could, however, not be granted, due to security constraints.

One of the possible measures of the ECL is the adaption of the data representation. From a relational point of view, we could switch between the row- and column-store representations for our triple store. However, dissecting the row-wise stored edge triples would introduce even more indirections and metadata, as there are already in the system. Storing edges column wise, with three columns each for source vertex, target vertex and edge labels would require further linkage information, how the values have to be recombined to reconstruct the original edge tuple. Another possibility would be the switch between different graph data formats, e.g. from a triple store to another graph format, such as an adjacency list or the compressed sparse row (CSR) format. However, these variants would require the implementation of a dedicated graph storage module. Changing the underlying storage representation can have both a positive or negative impact on the query performance, since every format is always more tailored towards a specific use case. In addition, changing the format for larger partitions also requires additional work, energy and locking of that partition over the reconstruction period. Thus, this optimization is more a long term investment, which also has to be amortized over a longer time span. How the transition between graph data formats for larger partitions can be efficiently achieved is a research topic of its own and thus we defer these experiments to future work.

With storage adaption being discarded, we will now focus on energy savings through adapting the CPU and memory clocks, depending on the current workload situation. Parts of this section have been demonstrated in our earlier work [KKHL19, KUK+19]. The driving idea is, that depending on a given workload, the system should not permanently require all of its resources. Considering the possible fluctuations of a workload, like shown earlier in Figure 2.9, we need to provide sufficient compute resources, to overcome high workload peaks. However, most of the time these resources would be idle due to underutilization. For cluster based solutions, we can simply turn additional machines on and off. However, NEMESYS works on single box SMP systems and thus needs sufficient resources inside one machine. When enough processors are available, we can always reduce the frequency of currently unused CPUs. This is especially the case, when many simple queries arrive, such as single-statement lookups. Aside the core frequency, we can also manipulate the *uncore* frequency, e.g. the clock for memory buses. When two



(a) Workload Profile, Redundancy disabled.



(b) Workload Profile, Redundancy enabled.

Figure 5.7: Completed queries over 300 s vs. energy consumed, standard mix, Balanced Edges, Medium server.

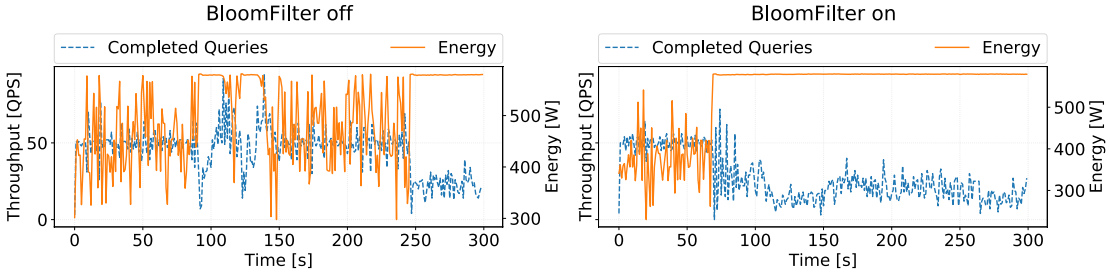
sockets exchange only a small amount of messages, the bus between these two can be slowed down to reduce the energy consumption.

Our *Medium* server consists of several Intel Xeon processors and thus, we measured the energy consumption using their internal RAPL[2] counters. RAPL is an acronym for *running average power limit* and provides different *energy domains*, such as the whole package, i.e. the whole processor chip without the memory controller, or the DRAM. Since NeMeSys is based on ERIS, we use the energy measurement facilities as described in [Kis17].

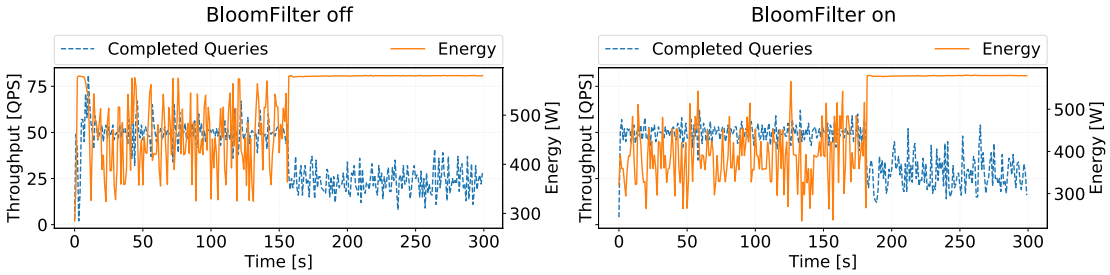
For the experiments shown in Figures 5.7 and 5.8, we executed the standard mix, i.e. without VPQs, and enabled the energy controlling facilities. The number of parallel clients was reduced to 50, to reduce the overall system load and thus allow for more adaptivity. The desired task latency for good queries (cf. Section 3.2.4) was set to 100 ms. During a monitoring period of 1000 ms, we count all completed queries and their total computation time. If the average query runtime exceeds the threshold of 100 ms, we need to provide more compute resources, e.g. by increasing the core frequencies.

Figure 5.7 visualizes the throughput and consumed energy during a given second for the Balanced Edges partitioning, i.e. a *lookup design* routing table. Generally, we observe higher adaptivity possibilities for our experiments without redundancy. During the first 150 s to 170 s, the energy curve is oscillating between higher and lower consumption. During this time period, the amount of arriving trivial queries is significantly higher than multi-statement queries, which leads to underutilization of some cores. The earlier performance drop for the enabled Bloom filter on the right-hand side of Figure 5.7a is most likely explained by the increased inter-socket communication, which is required to

[2] https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf [Last Accessed: 26.04.2020]



(a) Workload Profile, Redundancy disabled.



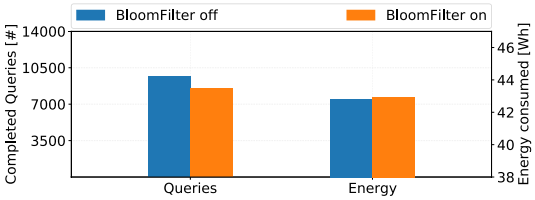
(b) Workload Profile, Redundancy enabled.

Figure 5.8: Completed queries over 300 s vs. energy consumed, standard mix, Ranged Vertices, Medium server.

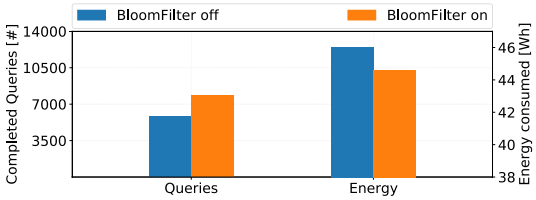
fetch the required information during messaging. This assumption is further underlined by the even smaller throughput for enabled redundancy in Figure 5.7b, when the second routing table has to be utilized additionally. Yet, combining Bloom filters and redundancy can improve the overall throughput for this experiment, because fewer messages are sent in total, and thus processors and memory do not need to run at maximum capacities. All four experiments show a throughput drop at varying time points, i.e. when more complex queries arrive and require more compute resources. Since NEMESYS can not hold the required 100 ms per query anymore, the internal loop increases the respective frequencies to the maximum, to catch up with struggling queries.

Figure 5.8 shows the results for a *hybrid design* routing table, which is based on the RV partitioning. Clearly, the smaller routing table leads to less inter-socket communication for fetching the necessary routing information and thus leads to more possibilities to adapt the core and memory frequencies. This benefit is visible in three out of four combinations of enabling and disabling Bloom filters and redundancy, except for redundancy disabled and Bloom filters enabled, i.e. the right-hand side of Figure 5.8a. The performance reduction compared to the non-Bloom filtered version is more severe than for Figure 5.7a. We assume unfortunate caching effects of the routing table against the Bloom filters to be the root cause, where checking the Bloom filters evicts the routing table from the processors caches. However, this is not measurable in our fully asynchronous system and thus needs further investigation. Since this effect does not hinder the adaptivity mechanism itself, we defer more thorough investigations to future work.

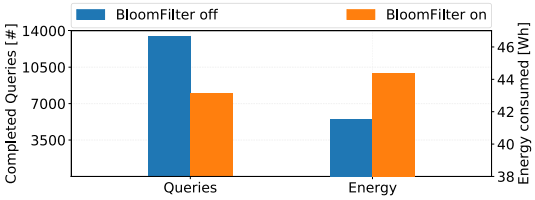
Figure 5.9 compares the total amount of completed queries and total energy consumed for all variants of Figures 5.7 and 5.8. Overall, the experiments with the *hybrid design* routing table achieve the highest throughput, paired with the lowest energy consumption, except for the questionable performance drop with the combination redundancy disabled and Bloom filter enabled.



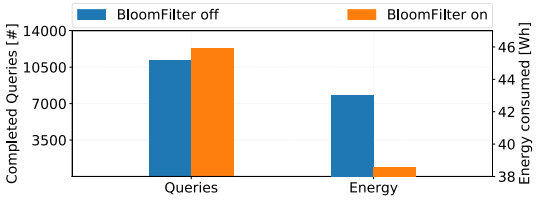
(a) Balanced Edges, Redundancy disabled.



(b) Balanced Edges, Redundancy enabled.



(c) Ranged Vertices, Redundancy disabled.



(d) Ranged Vertices, Redundancy enabled.

Figure 5.9: Total completed queries and total energy consumed, standard mix, Medium server.

5.3 LESSONS LEARNED

In this chapter, we assessed NEMESYS' capabilities of processing real world data, in the form of a knowledge graph. Our experiments showed, that all findings for the synthetic graphs from Chapter 4 can be confirmed for real world data.

We observed, that hash partitioning exhibits the benefit of being independent of any metadata, which can have a performance advantage against lookup tables. This was also confirmed by the experiment on the HPE Superdome Flex. However, even trivial heuristic graph partitioning strategies can have performance advantages against plain hashing, when paired with our *hybrid design* routing table approach or Bloom filter support.

Furthermore, we can integrate energy preserving methods like frequency scaling, to allow for energy adaptive GPM processing. Most of the queries from the log could be expressed with our query interface, except for more complex operators, like the matching of `OPTIONAL` clauses or Wikidatas internal language label service.

We conclude that the processing of GPM on real world graphs can be performed with reasonable throughput using the asynchronous processing model of NEMESYS. Summing up the incoming queries from Figure 2.9 gives $15\,819\,002$ incoming queries in $336\,\mathrm{h}$ or $1\,209\,600\,\mathrm{s}$, which is approximately $13.07\,\mathrm{Qps}$ on average. Our *hybrid design* routing table from Figure 5.4d achieved an average of up to $50.4\,\mathrm{Qps}$ on the *Medium* server, which is $3.8\,\mathrm{x}$ of the workload log of Wikidata.

The main antagonist for higher performance is rooted in the many adaptivity knobs of the baseline prototype and their implications for memory overhead that come with them. Furthermore, our experiments on the HPE Superdome Flex showed the importance of an optimized messaging infrastructure for larger systems and thus underlined the effectiveness of our Bloom filter optimization. However, compared to a real workload log, our proof-of-concept system NEMESYS achieves acceptable query throughput and thus confirms the applicability of NORAD as a suitable architecture for scalable GPM on SMP systems.

CONCLUSION

In our increasingly digitalized world, data is generated in large amounts and numerous data formats. One of these formats is the graph data format, which is a common data model to express relationships between entities. Due to its expressiveness, processing of graphs became a prominent use case in modern data analytics and is widely used among researches and practitioners alike. Real world data graphs can easily grow into the size of millions of edges and beyond, like the Wikidata knowledge graph. Processing larger graphs is usually performed on a cluster of compute nodes. However, we also observe a hardware trend towards more parallel compute resources, i.e. more processors and more main memory, within one single machine. The research of how to utilize such highly parallel machines is plentiful and proposes best practices in terms of system architecture or communication interfaces.

6.1 SUMMARY

This thesis investigated the opportunities of our newly proposed NORAD architecture and its implications for graph processing, based on an in-house developed data processing engine prototype. Therefore, we provided the required foundations for graph processing in Chapter 2. We outlined the applicability of the *edge-labeled multigraph* data model and presented our evaluated example, the graph pattern matching, as a prominent use case. The most used processing models were described and we concluded, that asynchronous processing allows for the maximum possible utilization of parallel compute resources. We motivated our demand for highly parallel, yet adaptive graph pattern matching processing with the generally increasing size of graphs and proved it with the publicly available knowledge graph of Wikidata and its query logs.

In Chapter 3, we thoroughly reviewed and presented the related work for our targeted hardware, which is a NUMA affected symmetric multiprocessor system. Based on the four most popular architectural design principles, we coined the name for a synthesis of these, which we call NORAD, with stands for NUMA-aware DORA with Delegation. The first system to unknowingly use this architecture was ERIS, a data processing engine prototype, which was developed with the chair of databases at TU Dresden. ERIS serves as the foundation for our graph processing engine NEMESYS and thus, we presented its internal structure and how ERIS has to be extended to allow graph processing on SMP systems. As a result, we formulated three main challenges, which explicitly focus on data exchange during query processing, efficient routing information provision and data placement and allocation.

Chapter 4 tackled these challenges by systematically elaborating optimization measures from a data and a systems point-of-view, which result in graph topology-based and system infrastructure-based optimization techniques. We presented our concepts for graph-agnostic and graph-aware query optimization, followed by an analysis of the influence of the data partitioning strategies, based on available system resources. Aside from finding suitable system configurations, we improved the messaging architecture to be more graph-friendly and enable the efficient processing of more diverse query patterns by supporting not only outgoing but also incoming edge traversal through more redundancy. The additional memory usage of redundant data storage can have a significant memory overhead, depending on the size of the stored graph. Thus, we investigated further messaging optimization techniques by applying the well known Bloom filter approach. To achieve maximum throughput, our hand-crafted hashing method PrimeHash was developed and has proven to be of higher performance than well-known hashing methods like MurMur3 or Googles HighwayHash. The chapter was then concluded with a thorough analysis of inter-partition communication and how co-locating partitions with higher message exchange can influence the query performance. The result is a gradient

descent-based optimizer, that leverages a hardware-based cost model. This optimizer provides a partition placement, that tries to minimize remote communication and maximize socket-local messaging by using either anticipated or measured communication paths.

To this point, we evaluated NEMESYS against individually generated synthetic datasets. The general applicability of our GPM engine had thus to be evaluated against real world graphs. Its prominence and public availability lead to Wikidata to serve as a proof-of-concept for NEMESYS and thus we have selected the knowledge graph and its anonymized query logs for this purpose. Chapter 5 presents a detailed analysis, how the *infrastructure* and *storage layers* influence the query throughput. We could show, that both of our earlier introduced topology and infrastructure based optimization techniques improve the performance of the NORAD baseline, not only for synthetic but also real world scenarios. Finally, we could demonstrate, that graph pattern matching does also allow for energy-adaptivity within the bounds of a query latency budget.

6.2 FUTURE RESEARCH DIRECTIONS

Within this thesis, we developed several optimization measures to allow graph pattern matching on NUMA-affected SMP systems. However, our scope was limited to enabling GPM on a NORAD based system and on investigating the possibilities of this architecture. In the following, we present the four most promising research topics, that arose during the preparation of this thesis. The named features would not only further improve the power of NEMESYS, but also allow for higher query throughput or lower energy consumption.

RPQ support

Section 3.3.3 introduced a subclass of Regular Path Queries, which we called VPQs. This general proof-of-concept has proven, that recursive query answering is possible on NEMESYS' asynchronous processing model. The processing of RPQs however is a research field of its own. Performant processing of RPQs is dependent on numerous factors, e.g. like an efficient representation of visited vertices among path traversal or the actual traversing algorithm. Yet RPQs are in fact a relevant problem, since they also occur in the Wikidata SPARQL query logs.

Metadata replication

Our experiment with the Wikidata graph have shown, that *lookup design* routing tables can grow to dramatic sizes, which eliminates the benefit of suitable graph partitioning algorithms. Even our *hybrid design* routing table had to be retrieved from the socket, on which NEMESYS was originally launched. Efficiently sharing metadata for the infrastructure layer touches multiple research topics, like cache coherency protocols and distributed data updates, e.g. for dynamic graphs.

Graph Storage

Chapter 5 emphasized the structural overhead of a system, which was designed for high adaptivity. A raw storage overhead of $4\,\mathrm{x}$ simply does not scale to larger datasets. Triple stores are commonly used, e.g. within RDF databases and thus served as foundation for this thesis. However, we anticipate substantial performance gains from enriching the *storage layer* with a dedicated graph storage backend. To retain the adaptivity thought, it is possible to incorporate multiple graph data formats, such as adjacency or incidence lists, compressed sparse row or column (CSR, CSC) representations or other state-of-the-art formats. Developing a dedicated cost model, that decides when to switch between any of these formats is a research topic on its own. Such a project needs to weigh numerous factors, such as transformation and amortization times, invested memory and potential speedup gains. Even holding multiple representations at once to mediate an incoming query to the most beneficial storage backend could be an option, yet these questions are to be evaluated separately.

Partitioning transitions

Aside from the data representation, we evaluated the impact of altering the data placement in the system in Section 4.3.2. We showed, that not only the content of data partitions influences the query processing performance, but also where they are placed within the system. These results are the first step towards a new research topic, which is concerned with transforming a given data partitioning to another. It is yet to be determined, if the overhead of converting the content even of a subset of the partitions can lead to performance gains or if the total overhead could not be amortized anyhow.

BIBLIOGRAPHY

[AAB+18] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1421–1432, 2018.

[AAP+17] Raja Appuswamy, Angelos Anadiotis, Danica Porobic, Mustafa Iman, and Anastasia Ailamaki. Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads. *PVLDB*, 11(2):121–134, 2017.

[ABPH07] Paulo Sérgio Almeida, Carlos Baquero, Nuno M. Preguiça, and David Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, 2007.

[ACL+07] Mustafa Atay, Artem Chebotko, Dapeng Liu, Shiyong Lu, and Farshad Fotouhi. Efficient schema-based xml-to-relational data mapping. *Inf. Syst.*, 32(3):458–476, 2007.

[ACW16] Jyrki Alakuijala, Bill Cox, and Jan Wassenberg. Fast keyed hash/pseudo-random function using SIMD multiply and permute. *CoRR*, abs/1612.06257, 2016.

[AG08] Renzo Angles and Claudio Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008.

[AKPA17] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. The case for heterogeneous HTAP. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[Ang12] Renzo Angles. A comparison of current graph database models. In *Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012*, pages 171–177, 2012.

[BBC+17] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. gmark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.*, 29(4):856–869, 2017.

[BC11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.

[BGK18] Adrian Bielefeldt, Julius Gonsior, and Markus Krötzsch. Practical linked data access via SPARQL: the case of wikidata. In *Workshop on Linked Data on the Web co-located with The Web Conference 2018, LDOW@WWW 2018, Lyon, France April 23rd, 2018*, 2018.

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[BKM+00] Andrei Z. Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet L. Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309–320, 2000.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[BM03] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

[BWM+16] Sebastian Burgstaller-Muehlbacher, Andra Waagmeester, Elvira Mitraka, Julia Turner, Tim E. Putman, Justin Leong, Chinmay Naik, Paul Pavlidis, Lynn M. Schriml, Benjamin M. Good, and Andrew I. Su. Wikidata as a semantic framework for the gene wiki initiative. *Database*, 2016, 2016.

[CEK+15] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[CKWT14] Long Cheng, Spyros Kotoulas, Tomas E. Ward, and Georgios Theodoropoulos. Efficiently handling skew in outer joins on distributed systems. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014*, pages 295–304, 2014.

[CLFF10] Artem Chebotko, Shiyong Lu, Xubo Fei, and Farshad Fotouhi. Rdfprov: A relational RDF store for querying and managing scientific workflow provenance. *Data Knowl. Eng.*, 69(8):836–865, 2010.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[DDL02] C. J. Date, Hugh Darwen, and Nikos A. Lorentzos. *Temporal data and the relational model*. Elsevier, 2002.

[DFI+13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254, 2013.

[DJL+16] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. Graphframes: An integrated api for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 2:1–2:8, New York, NY, USA, 2016. ACM.

[DMvH+00] Stefan Decker, Sergey Melnik, Frank van Harmelen, Dieter Fensel, Michel C. A. Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks. The Semantic Web: The Roles of XML and RDF. *IEEE Internet Computing*, 4(5), 2000.

[Erl12] Orri Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[FCP+11] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[FHL18] George H. L. Fletcher, Jan Hidders, and Josep-Lluís Larriba-Pey, editors. *Graph Data Management, Fundamental Issues and Recent Developments*. Data-Centric Systems and Applications. Springer, 2018.

[FNR+13] Arash Fard, M. Usman Nisar, Lakshmish Ramaswamy, John A. Miller, and Matthew Saltz. A distributed vertex-centric approach for pattern matching in massive graphs. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 403–411, 2013.

[GGL+19] Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. Updating graph databases with cypher. *PVLDB*, 12(12):2242–2253, 2019.

[Gra17] Torbjörn Granlund. Instruction latencies and throughput for AMD and Intel x86 processors, 2017. https://gmplib.org/~tege/x86-timing.pdf.

[HCSO12] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-marl: a DSL for easy and efficient graph analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 349–362, 2012.

[HD62] T. E. Hull and A. R. Dobell. Random Number Generators. *SIAM Review*, 4:230–254, 1962.

[HD15] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, 2015.

[HDM+15] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. PGX.D: a fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 58:1–58:12, 2015.

[HR73] Laurent Hyafil and Ronald L Rivest. *Graph partitioning and constructing optimal decision trees are polynomial complete problems*. IRIA. Laboratoire de Recherche en Informatique et Automatique, 1973.

[HTT09] Tony Hey, Stewart Tansley, and Kristin M. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[ITDK16] Kengo Ito, Yu Tsutsumi, Yasuhiro Date, and Jun Kikuchi. Fragment Assembly Approach Based on Graph/Network Theory with Quantum Chemistry Verifications for Assigning Multidimensional NMR Signals in Metabolite Mixtures. *ACS chemical biology*, 11(4):1030–1038, 2016.

[JKA+17] Martin Junghanns, Max Kießling, Alex Averbuch, André Petermann, and Erhard Rahm. Cypher-based graph pattern matching in gradoop. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, pages 3:1–3:8, 2017.

[JRDY12] Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Xu Yu. SCARAB: scaling reachability computation on large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 169–180, 2012.

[KH18] Seongyun Ko and Wook-Shin Han. Turbograph++: A scalable and fast graph analytics system. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 395–410, 2018.

[Kin08] Shiva Kintali. Betweenness centrality : Algorithms and lower bounds. *CoRR*, abs/0809.1906, 2008.

[Kis17] Thomas Kissinger. *Energy-Aware Data Management on NUMA Architectures*. PhD thesis, Dresden University of Technology, Germany, 2017.

[KK98] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distributed Comput.*, 48(1):96–129, 1998.

[KK13] George Karypis and Vipin Kumar. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 5.1. http://www.cs.umn.edu/∼metis [last accessed 28-02-2020], 2013.

[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 (Euro-Par)*, pages 149–163, 2017.

[KKHL19] Alexander Krause, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Nemesys - A showcase of data oriented near memory graph processing. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1945–1948, 2019.

[KKN+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[KKS+14] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. ERIS: A numa-aware in-memory storage engine for analytical workload. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2014, Hangzhou, China, September 1, 2014*, pages 74–85, 2014.

[KN11] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206, 2011.

[KUK+17] Alexander Krause, Annett Ungethüm, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Asynchronous graph pattern matching on multiprocessor systems. In *New Trends in Databases and Information Systems - ADBIS 2017 Short Papers and Workshops, AMSD, BigNovelTI, DAS, SW4CH, DC, Nicosia, Cyprus, September 24-27, 2017, Proceedings*, pages 45–53, 2017.

[KUK+19] Alexander Krause, Annett Ungethüm, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Nemesys - energy adaptive graph pattern matching on numa-based multiprocessor systems. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*, pages 537–541, 2019.

[LBKN14] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754, 2014.

[Les] Jure Leskovec. Snap – stanford network analysis platform. http://snap.stanford.edu/snap/ [Online, last acessed 10-12-2019].

[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 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146, 2010.

[MKG+18] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. Getting the most out of wikidata: Semantic technology usage in wikipedia's knowledge graph. In *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, pages 376–394, 2018.

[MVLB15] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. The graph structure in the web - analyzed on different aggregation levels. *J. Web Science*, 1(1):33–47, 2015.

[MWM15] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, 2015.

[NLP13] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 456–471, 2013.

[OFGK00] Hiroyuki Ogata, Wataru Fujibuchi, Susumu Goto, and Minoru Kanehisa. A heuristic graph comparison algorithm and its application to detect functionally related enzyme clusters. *Nucleic acids research*, 28(20):4021–4028, 2000.

[OR02] Evelien Otte and Ronald Rousseau. Social Network Analysis: a powerful strategy, also for the information sciences. *J. Information Science*, 28(6):441–453, 2002.

[PCWF07] Shashank Pandit, Duen Horng Chau, Samuel Wang, and Christos Faloutsos. NetProbe: A Fast and Scalable System for Fraud Detection in Online Auction Networks. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 201–210, 2007.

[PJHA10] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[Pot17] Anthony Potter. *Query answering in distributed RDF databases*. PhD thesis, University of Oxford, UK, 2017.

[PTB+11] Ippokratis Pandis, Pinar Tözün, Miguel Branco, Dimitris Karampinas, Danica Porobic, Ryan Johnson, and Anastasia Ailamaki. A data-oriented transaction execution engine and supporting tools. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1237–1240, 2011.

[PTJA11] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. PLP: page latch-free shared-everything OLTP. *PVLDB*, 4(10):610–621, 2011.

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

[RFA16] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. Design principles for scaling multi-core OLTP under high contention. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1583–1598, 2016.

[RKB04] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. "grabcut": interactive foreground extraction using iterated graph cuts. *ACM Trans. Graph.*, 23(3):309–314, 2004.

[RN10] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010.

[SB13] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 135–146, 2013.

[SC15] Elliot Saltzman and David Caplan. A Graph-Dynamic Perspective on Coordinative Structures, the Role of Affordance-Effectivity Relations in Action Selection, and the Self-Organization of Complex Activities. *Ecological Psychology*, 27(4):300–309, 2015.

[SMS+17] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB*, 11(4):420–431, 2017.

[SPSL13] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013.

[SSV+17] Zsuzsika Sjoerds, Steven M Stufflebeam, Dick J Veltman, Wim Van den Brink, Brenda WJH Penninx, and Linda Douw. Loss of brain graph network efficiency in alcohol dependence. *Addiction biology*, 22(2):523–534, 2017.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[Syl78] J. J. Sylvester. Chemistry and algebra. *Nature*, 17(432):284, Feb 1878.

[TBC+13] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.

[TF76] Robert W. Taylor and Randall L. Frank. CODASYL data-base management systems. *ACM Comput. Surv.*, 8(1):67–103, 1976.

[TKS17] Mustafa Kemal Tas, Kamer Kaya, and Erik Saule. Greed is good: Optimistic algorithms for bipartite-graph partial coloring on multicore architectures. *CoRR*, abs/1701.02628, 2017.

[TU10] Mirco S Till and G Matthias Ullmann. Mcvol-a program for calculating protein volumes and identifying cavities by a monte carlo algorithm. *Journal of molecular modeling*, 16(3):419–429, 2010.

[TZK+13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 18–32, 2013.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[VK14] Denny Vrandecic and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.

[WGGM16] Claudia Wagner, Eduardo Graells-Garrido, David García, and Filippo Menczer. Women through the glass ceiling: gender asymmetries in wikipedia. *EPJ Data Sci.*, 5(1):5, 2016.

[Woo12] Peter T. Wood. Query Languages for Graph Databases. *SIGMOD Record*, 41(1):50–60, 2012.

[YCLN15] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 1307–1317, 2015.

[ZAL18] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. Modeling polypharmacy side effects with graph convolutional networks. *Bioinform.*, 34(13):i457–i466, 2018.

[ZCC15] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 183–193, 2015.

LIST OF FIGURES

LIST OF TABLES

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, July 1, 2020