Technische Universität Berlin

Master Thesis

Evaluation of Meta Path Based Graph Analysis in the Context of an Enterprise Recommender System

Jan Kalkan

Matriculation No.: 369149

Supervisors: Prof. Dr. rer. nat. Volker Markl
Advisors: Dr. Holmer Hemsen
Industry Advisors: M.Sc. Lukas Masuch & M.Sc. Benjamin Raethlein

17-12-2018

Erklärung (Declaration of Academic Honesty)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare to have written this thesis on my own and without forbidden help of others, using only the listed resources.

Datum	Jan Kalkan

Abstract

Graph analysis techniques have become increasingly popular in recent years because most real world scenarios can be naturally modeled as graphs. Especially heterogeneous information networks are well suited for modeling and emphasizing various entities with its diverse relationships [51]. Previously, most network analysis research targeted homogeneous networks without distinguishing between different types of entities exhibiting different types of relationships. Meta path evolved as a state-of-the-art key concept for mining heterogeneous graphs and forms the basis for novel graph analysis algorithms as well as similarity and relevance measures for graph components [57].

In this thesis, we consider the SAP Community which is a platform with more than 2.8 million users, where it is possible to write blog entries, ask questions or comment contributions. SAP is working on a project for extracting information from the platform to be able to generate meaningful content recommendations for its users. The related dataset is stored as a heterogeneous information network in a Neo4j graph database instance. Hence, meta path based recommendation generation is of special interest in this context. The efficient computation of meta paths is a key challenge, which we address in this thesis. Therefore, we define different meta paths for the SAP Community network first. Next, we evaluate the applicability of Neo4j's native query capabilities for computing these paths. Additionally, we implement the parallel meta-graph extraction framework recently proposed by Shao et al. [51] on top of Apache Flink. As a state-of-the art approach for the meta path computation, we use it to compare the results to the graph database approach.

The evaluation results show that Neo4j is indeed capable to compute semantically interesting meta paths. However, the generic applicability is limited by its available memory. Consequently, it is not possible to compute results for high selective meta paths. In contrast, Neo4j is especially suitable for efficiently computing partial paths. The framework implementation on top of Flink outperformed the Neo4j approach significantly, since we were able to demonstrate linear scalability in the observed setting. The computation for the meta path BUTUB took only 3m8s compared to 2h38m with Neo4j and 3m30s for the path BUTB in contrast to 19m with the graph database. Surprisingly, it was not possible to compute high selective paths either, since the approach is strongly dependent on the dataset and the respective data distribution leading to garbage collection stalls and *OutOfMemory* errors. In order to address these problems, we propose a *hybrid approach* leveraging Neo4j for the efficient partial path computation on the one hand and generating the final paths using Flink's dataset transformation operators on the other hand. The results of our proof of concept implementation showed that we outperformed the framework implementation by a factor of ten. Moreover, it was finally possible to compute the high selective meta path with over 130 billion path instances in this way.

Zusammenfassung

Graphenanalyseverfahren haben in den letzten Jahren an Popularität gewonnen. Ein entscheidender Grund dafür ist, dass viele Szenarien natürlich als Graph abgebildet werden können. Heterogene Informationsnetzwerke sind hierfür besonders geeignet, da besonderer Fokus auf der Heterogenität der Entitäten und ihren verschiedenen Beziehungen zueinander liegt [51]. Früher beschäftigte sich die Forschung in erster Linie mit der Analyse von homogenen Informationsnetzwerken. Mittlerweile haben sich Metapfade zu einem modernen Schlüsselkonzept für die Analyse von heterogenen Graphen entwickelt und bilden u.a. die Basis für neue Ähnlichkeits- und Relevanzmaße für Graphkomponenten[57].

In Rahmen dieser Thesis betrachten wir SAP Community, eine online Plattform mit mehr als 2,8 Millionen Nutzern, auf welcher es u.a. möglich ist Blogeinträge zu schreiben oder Fragen zu stellen. SAP arbeitet an einem Projekt zur Informationsextraktion aus dem Netzwerk, um sinnvolle Beitragsempfehlungen für die Nutzer zu generieren. Die relevanten Daten sind als heterogenes Informationsnetzwerk in der Graphdatenbank Neo4j gespeichert. Die gegebene Graphstruktur der Daten machen eine auf Metapfaden basierende Empfehlungen daher besonders interessant. Die effiziente Extraktion von Metapfaden ist ein grundlegendes Problem, welches wir in der vorliegenden Arbeit adressieren. Um verschiedene Ansätze für die Extraktion testen zu können, definieren wir zunächst verschiedene Metapfade für das betrachtete Netzwerk. Auf dieser Basis evaluieren wir die Eignung von Neo4j für die Berechnung dieser Pfade. In einem weiteren Schritt implementieren wir das *Meta-Graph Extraction Framework* [51] in Apache Flink [61] und Vergleichen diesen modernen Ansatz der Metapfadextraktion mit den Laufzeiten von Neo4j.

Die Ergebnisse zeigen, dass Neo4j tatsächlich in der Lage ist semantisch interessante Metapfade zu extrahieren. Dennoch ist die generische Anwendbarkeit durch den zur Verfügung stehenden Arbeitsspeicher limitiert. Folglich war es nicht möglich Metapfade mit einer hohen Selektivität zu berechnen. Im Gegensatz dazu ist Neo4j gut geeignet um Teilpfade effizient zu extrahieren. Die Framework Implementierung in Flink übertraf den Neo4j Ansatz signifikant, da wir lineare Skalierbarkeit für die gleichen Pfade feststellen konnten. Die Berechnung des BUTUB Pfads benötigte 3m8s und des BUTB Pfads 3m30s. Im Gegensatz dazu benötige Neo4j 2h38m und 19m. Die Extraktion der hochselektiven Metapfade ist überraschenderweise nicht möglich, da der Ansatz negativ von der Verteilung der Pfade im Graphen beeinflusst ist, was zu sehr hohen Garbage Collection Aufwänden und *OutOfMemory* Fehlern führt. Wir empfehlen folglich einen hybriden Ansatz zur Berechnung der Teilpfade in Neo4j und der Generierung der finalen Pfade mithilfe von Flinks Datentransformationsoperatoren. Wir beobachten eine Verbesserung der Laufzeiten um den Faktor zehn im Vergleich zur Framework Implementierung und konnten schließlich auch die hochselektiven Pfade mit über 130 Milliarden Instanzen berechnen.

Table of Content

List of Figures

List of Tables

1 Introduction

We are living in an interconnected world with many different entities exhibiting complex relationships of various types among each other. Social networks like Facebook are popular use cases exposing a graph structure. The Facebook network for example comprises many different entities, such as users, products, events, various types of content as well as diverse types of relationships like friendship between users, interest in topics or events. In general, nearly every scenario can be expressed as a network, whether it is the organization of a company, the relationships between actors and movies or even the interaction between proteins in the human body can be modeled as a graph [24].

Previously, most of the research focused on the analysis of homogeneous information networks without distinguishing between the different object and relation types [56]. One of the most well known applications was *PageRank* [9], an algorithm invented by Google for computing the rank of web pages as a measure of relevance based on the *Webgraph*, which models web pages and the hyperlinks between them. In contrast, heterogeneous graphs, so called *heterogeneous information networks* (HIN), are the state-of-the-art approach to model real world scenarios, by incorporating rich semantics into multi-typed objects and multi-typed connections among them [51]. Consequently, the field of HIN analysis emerged recently and enabled new application possibilities such as *Link Prediction*, which can be applied for predicting friendship relationships on Facebook for example. Another application is *Entity Profiling*, which aims to get a deeper understanding of the behavior of entities like users or organizations in a network and can be used in the context of item recommendations for example [52].

1.1 Motivation

Recommender systems (RS) are omnipresent nowadays and a business critical part of many systems. Especially Internet companies like Google, Amazon and Facebook heavily use such systems for content recommendations. RS can be categorized as *content-based systems*, *collaborative-filtering systems* or *hybrid systems* [1]. Content-based systems recommend items by looking how similar two objects are, based on their content. For example, Netflix could recommend movies to a user based on the genre or favorite actors a user often watches. A collaborative-filtering system, on the other hand, recommends items by looking at users with similar preferences or behavior to recommend items those users liked, watched or generally consumed. Hybrid systems combine techniques from both concepts for leveraging the advantages of the respective approaches while diminishing the disadvantages. In order to generate meaningful recommendations in general, the similarity or the relevance between entities needs to be determined. Classical similarity measures

like TF-IDF only focus on the actual content of documents or blog posts in this context, while completely ignoring the structure of the graph as well as the heterogeneity property of the graph's objects. Although traditional graph algorithms like PageRank incorporate the structure of the graph, the heterogeneity property of a HIN is still disregarded [51]. This means a loss of information, because the inherent semantics of these graph objects are not considered. As a consequence, the quality of the recommendation results will be less precise. A key concept for mining HINs is called *meta path* [57], which is a meta-level description of a path between two objects in a network. Consequently, different meta path based similarity measures like PathSim have been invented [56]. The important benefit of using meta path based similarity measures is the fact that the inherent semantic of the graph structure is leveraged.

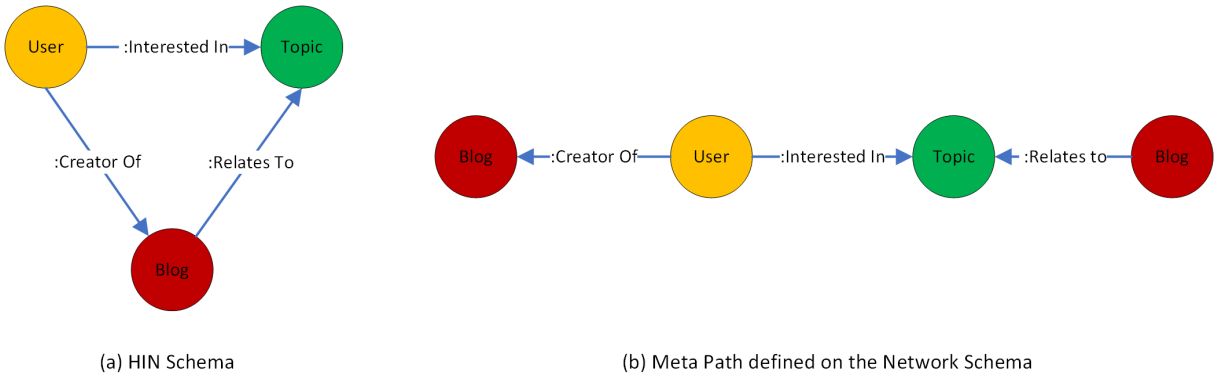


Figure 1: Meta Path Example

Figure 1 (a) shows the schema of a HIN and figure 1 (b) visualizes a meta path defined on this network schema. The path can be interpreted as two blogs are similar if the 2nd blog is flagged with the same term that the user, who created the 1st blog, follows. In other words, a blog $blog_B$ is considered similar to another one called $blog_A$, if the author (user) of $blog_A$ follows the same topic as $blog_B$ is connected to.

The SAP Community is the official user community platform of SAP SE with over 2.8 million users. It is intended for everybody who is working or is interested in the SAP ecosystem like users, developers or consultants. Developers, for example, use the platform for Q&A like Stack Overflow to get community support but mainly for SAP technologies. Users write blog posts about diverse topics related to SAP and SAP technologies for sharing ideas. Sales agents may blog about SAP marketing events, consultants about new technology trends and so on. Users can connect with other users, follow specific topics, create content, comment on blog posts or like content for example. The SAP Community dataset is a knowledge graph created from a variety of source systems. The resulting graph is stored in Neo4j, which is a popular, open source graph database [48].

SAP is in the need of a state-of-the-art, graph-based RS for the SAP Community platform. The recommendation of relevant *blogs* for SAP Community platform users is of particular interest for SAP. The efficient execution of the respective algorithms is a fundamental problem. Recommendations must be instantly available to the user. In other words, the response times of a RS need to be very short, while the actual computation of meta paths is usually costly [51]. For instance, blog recommendations need to be instantly available when a user requests the website displaying a specific blog post.

1.2 Objective of the Thesis

The goal of the thesis is to evaluate an approach for efficiently extracting meta paths from the SAP Community knowledge graph as the basis for meta path based similarity measure calculation. One approach for optimizing the execution of meta path based algorithms is to pre-analyze the HIN. Traditional systems are usually not suitable to process such large amounts of data. Therefore, different parallel processing systems have been invented. In this context, parallel graph processing systems are of great importance and can be leveraged to execute graph algorithms in a distributed manner [35]. Another possible approach is to use the native query capabilities of a graph database. According to IBM, one reason why people are using graph databases is that they want to get rid of the hassle to pre-compute query results, i.e. they aim to perform real time queries instead [27].

The problem of generating meta path based recommendations can be subdivided into four parts:

1. **Meta path selection**

   The meta path selection is the foundation of the RS. Mainly, two different approaches are possible here. One approach is to use domain knowledge to define such meta paths, another approach would be to apply machine learning for selecting useful paths. In this thesis, we define meta paths based on domain knowledge instead of using machine learning to focus on the actual research question. It could be possible that a specific meta path could induce a poor coverage in the data set. The other possible extreme case would be that the number of matched path instances is extraordinarily high. Nevertheless, the selection of semantically meaningful paths is crucial.

2. **Analysis of the graph with respect to meta paths**

   The thesis focuses on this part within the recommendation system, i.e. the actual analysis of the graph with regards to the meta paths retrieved in *step 1*, which also

constitutes the core of a meta path based RS. The analysis of the graph refers to pattern matching in this context [5]. For example, we could analyze a given graph with respect to the meta path given in figure 1 (b). This means, we are trying to find all occurrences of this specific meta path. The problem is visualized in Figure 12 and formalized in section 4.1. At this stage, we are neither computing similarities nor are we generating the final recommendations. One state-of-the-art approach for the graph analysis is to execute the main workload in advance, i.e. before the actual recommendation will be made available to the user of the system and persist the results. Hereby, the processing workload in the real online scenario can be reduced [51]. The main workload refers to the computation of meta path instances within the considered dataset. Furthermore, a distributed implementation can be used to analyze the graph and to parallelize the workload. Shao et al. recently proposed a framework for *parallel meta-graph extraction* [51]. The framework basically presents an approach for analyzing a HIN with respect to meta paths in a distributed manner, which was implemented on top of Apache Giraph. The meta graph extraction framework is presented in section 3.6. In the context of the thesis, the framework will be implemented on top of Apache Flink and the performance and scalability will be evaluated on the SAP Community graph.

Since graph database management systems are specifically designed to store and process large graphs, it is interesting to see how the performance for the meta path extraction using a graph database differs from analyzing the graph using a dedicated distributed processing engine. Query response times tend to be relatively constant in graph databases compared to relational databases, because queries are usually localized to a portion of the graph [48]. This will probably not be the case when analyzing the graph in advance. In this case we are trying to find all occurrences of a specific meta path instead of only looking for all occurrences starting from one specific vertex in the graph, which implies that we need to scan a considerable bigger part of the graph. The question is whether the advertised performance benefits tied to the graph database design outweigh the assumed benefits of using a dedicated and distributed graph processing engine.

3. **Computation of meta path based similarity measures**

   The computation of meta path based similarity measures constitutes *step 3* within the recommendation generation pipeline. Here, the results of the second step will be used to finally calculate different similarity measures like PathSim for example. The actual similarity calculation is strongly simplified, because the bulk of the workload was already done in step two. This fact relates to the trade-off between shorter runtimes and the currency of the recommendation results as highlighted before.

4. **Recommendation generation by evaluating the similarities**

It is trivial to generate the recommendation results, if we have the similarities from *step 3* at hand. So, the last step constitutes the evaluation of similarities of different graph entities and additionally considering similarity thresholds to generate the final recommendation results.

The last two steps within the RS pipeline, i.e. the computation of the meta path based similarity measures and the generation of the actual results, won't be analyzed in detail in the context of the thesis and are only briefly discussed so that the reader is able to understand the whole process. Instead, we will focus on meta path based similarity measures. Especially on evaluating approaches to efficiently analyze the SAP Community graph with respect to meta paths. This means, we consider the actual technical implementation details for analyzing the graph as compared to evaluating the quality of the recommendation results based on different similarity measures like in a precedent thesis conducted at SAP.

The following hypotheses are raised and will be validated in the context of the thesis:

Hypothesis 1 *The distributed implementation for analyzing the SAP Community graph with respect to meta paths proposed by Shao et al. outperforms an implementation using the native query capabilities of a graph database, because the runtimes can be considerably reduced by scaling the execution to multiple nodes within a cluster.*

Hypothesis 2 *Due to its non-scalable architecture Neo4j will exhibit limitations for the computation of meta paths with a very high selectivity.*

In a first step, we will select meta paths based on the SAP Community graph by using domain knowledge and discuss them. Therefore, we will give a semantic interpretation of these paths and approximate the selectivities. With selectivity we refer to the specificity of the graph elements within a meta paths. For example, we could choose the meta path in figure 1 and additionally the same meta path but without edge types. The latter one has most probably a higher selectivity since the specificity of the edges is lower as compared to the first one. Moreover, we select paths with different lengths. By choosing the meta paths along those dimensions we want to evaluate the scalability of the considered approaches with respect to the complexity of meta paths and hereby stress test the examined approaches.

Furthermore, a prototype of the *parallel meta-graph extraction* framework, proposed by Shao et al., will be implemented for the SAP Community graph. Hereby, one state-of-the-art approach for analyzing the graph with respect to meta paths will be applied and the actual applicability with regards to the performance and scalability can be evaluated in this specific real-world context. The implementation needs to be done on a distributed graph processing system, adhering to the vertex-centric model (cf. section 3.5). Therefore, Apache Flink [31] will be chosen and especially Gelly, the Graph API of Flink, will be utilized. SAP will provide resources, i.e. machines, as well as the SAP Community dataset. The dataset itself cannot be published for privacy reasons. Nevertheless, the evaluation is possible without disclosing confidential information. As part of the work of the thesis, the distributed data processing system Apache Flink needs to be set up as a cluster and a distributed file system as the basis for the processing system as well. Moreover, the framework implementation has to be done from scratch and tailored to the SAP Community graph as well as to the considered meta paths.

Hypothesis 2 will be validated against a further implementation using the native query capabilities of Neo4j, as a representative of a popular, open source graph database. Therefore, we will setup a dedicated Neo4j instance hosting the SAP Community graph. Additionally, we will formulate and analyze Cypher queries for the considered meta paths. The runtime of *step 2* in the RS pipeline, i.e. the runtime for the actual meta path analysis, is of special interest. Those measurements will be compared to the respective results of the distributed implementation. Moreover, we will try to relate the results to the underlying core concepts of the used technologies as good as possible. On a high level, core concepts refer to distributed data processing, graph database characteristics and implementation details. Hereby, we can evaluate which approach is better suited for the respective meta paths by comparing the runtimes, i.e. we can validate the *hypothesis 1*.

A quantitative evaluation regarding the performance and scalability of the framework implementation will be conducted based on the results obtained by the experiment executed on the cluster. We will evaluate the scalability of the framework implementation further by scaling the execution from one machine to multiple ones. This will be done for all previously chosen meta paths. On the one hand, the results will supplement the evaluation of *hypothesis 1*. On the other hand, those results will allow the validation of *hypothesis 2*.

Finally, the results will be summarized and a wrap-up will be given, based on the thesis results.

1.3 Contributions

The contributions of this thesis go as follows:

1. We evaluate the capabilities of Neo4j's native graph processing engine for analyzing the SAP Community knowledge graph with regards to meta paths.

2. This thesis is the first to implement the meta-graph extraction framework proposed by Shao et al. on another real-world dataset as the ones used by the authors itself.

3. To the best of our knowledge, we are the first to implement the *parallel meta-graph extraction* framework on top of Apache Flink instead of Apache Giraph.

4. We suggest a meaningful Apache Flink system configuration for analyzing the SAP Community dataset with respect to meta paths.

5. We provide the basis for generating meta path based content recommendations on the SAP Community platform.

6. The results of this thesis will enable SAP to build a state-of-the art graph analysis pipeline for efficiently computing meta paths.

7. As far as we know, we are the first to propose a hybrid approach for addressing the complexity occurring when computing meta path instances covering high degree vertices in the vertex-centric model with Apache Flink. Thus, we suggest to leverage the advantages of Neo4j's native graph processing engine on the one hand as well as the benefit of Flink's distributed runtime on the other.

1.4 Thesis Outline

In Section 2, we will introduce the definitions of elementary terms used within the thesis. Therefore, we will provide graph theoretical definitions as well as distributed processing related ones.

We will take a look at the related work in Section 3. Here, we will discuss, similarity and relevance in HINs, basic graph search algorithms, graph database fundamentals at the example of Neo4j, relevant distributed processing concepts implemented in Apache Flink, the vertex-centric graph processing model as well as the main concepts of the *meta-graph extraction* framework pertinent for our implementation.

Subsequently, we will formalize the problem addressed within this thesis in Section 4 first. Next, we will briefly discuss the general approach for evaluating the considered approaches. Following, we will present the SAP Community HIN, which is the observed dataset in this thesis. Finally, we will define the selected meta paths and approximate its selectivities based on the data distributions in the network.

Section 5 comprises the Neo4j based path count computation experiment. Thus, we will present the approach for evaluating this partial experiment. Moreover, we will talk about the implementation details such as the Cypher query formulation and the resulting execution plans. Afterwards, we evaluate the results and finally draw an interim conclusion for this part.

In Section 6, we will detail the experiment for evaluating the *meta-graph extraction* framework on top of Apache Flink's graph API. Therefore, we will again present our approach for this partial experiment first. Next, we will highlight implementation relevant topics such as the graph representation, system architecture and Path Concatenation Plans derived from the meta paths. Afterwards, we will evaluate the results. We subdivided the evaluation, since we are interested in different aspects such as the absolute runtimes, the strong scalability as well as the effects implied by different meta paths. Eventually, we conclude the results for this partial experiment.

Additionally, we will present a hybrid approach using Neo4j in combination with Apache Flink based on the results gained in the previous experiments in Section 7. Hence, we introduce the main idea and demonstrate the usefulness for all meta paths as a proof of concept.

Finally, we will critically discuss and compare the results from the different experiments conducted for this thesis in Section 8 and we will conclude with a summary of the research results in Section 9.

2 Definitions

In the following section, we provide all necessary definitions relevant for this thesis. Therefore, we define graph theoretical terms first. Next, we give the definitions for heterogeneous as well as homogeneous information networks. Lastly, we briefly introduce elementary parallel data processing subjects, which will be referred to in the evaluation sections.

2.1 Graph Theory

A graph $G = (V, E)$, either directed or undirected, consists of a set of vertices V and a set of edges E that connects vertices with each other. It should be noted that both sets are unordered and an element in E is a pair of vertices from V. If two or more edges connect the same pair of vertices they are called parallel. An edge that connects a vertex with itself is called a self-loop. A graph is called a *directed graph* if additionally the edges are directed. A directed edge points from the 1st vertex in the pair to the 2nd one, since an edge is a pair of vertices. The number of outgoing edges from a vertex is called outdegree. Consequently, the number of ingoing edges is called indegree [50].

A heterogeneous graph $G = (V, E, T_v, T_e, A_v, A_e)$ enhances the simple definition of a graph by adding a vertex type set T_v and and edge type set T_e. There exists a vertex type mapping function $f_v : V \to T_v$, which maps each $v \in V$ to exactly one vertex type $f_v(v) \in T_v$ and an edge type mapping function $f_e : E \to T_e$, that maps each $e \in E$ to one specific edge type $f_e(e) \in T_e$ [21]. In addition, a heterogeneous graph can optionally have a vertex attribute set A_v and an edge attribute set A_e. A heterogeneous graph can be used to model an abstraction of different relations between multi-typed objects in the real world [51].

A *homogeneous graph* [51] is a special case of a heterogeneous graph with $|T_v| = 1$ and $|T_e| = 1$. In other words, there exists exactly one vertex - as well as one edge type. Moreover, a heterogeneous graph with $|T_e| = 1$ and $|T_v| > 1$, i.e. all edges have the same type is called an *edge homogeneous graph*. The result of the *meta-graph extraction* framework is such a graph for example. An edge between to vertices in the extracted graph implies at least one instance of a meta path in the analyzed heterogeneous graph.

2.2 Information Network

An *heterogeneous information network* (HIN) $G = (V, E)$, with a set of nodes V and a set of relations E, is similar to an heterogeneous graph. Therefore, the definition of a *HIN*

is similar as given in section 2.1. Although, both terms are similar we will additionally introduce the term, because it is more common in the context of real world graph analysis use cases. The focus of a *HIN* lies on the semantics encoded in the differently typed objects. Thus, a graph is more abstract and emphasizes the structure, whereas a network emphasizes the semantics. The vertex type mapping is usually referred to as node type mapping $f_v : V \rightarrow A$, which maps each node $v \in V$ to one specific node type $f_v(v) \in A$. In contrast, the edge type mapping is called link type mapping $f_e : E \rightarrow R$, that maps each link or relation $e \in E$ to one specific link -, respectively relation type $f_e(e) \in R$. According to Sun et al., *It turns out that this level of abstraction has great power in not only representing and storing the essential information about the real-world, but also providing a useful tool to mine knowledge from it, by exploring the power of links.* Furthermore, nearly every domain can be modeled as a *HIN*. Popular examples are social networks, e-commerce scenarios, movie databases and many others [56].

A network is called a *homogeneous information network*, if the node type set $|A| = 1$ and the relation type set $|R| = 1$, i.e. when there exists exactly one node type and exactly one link type. Otherwise, it is called a heterogeneous information network [52].

2.3 Property Graph

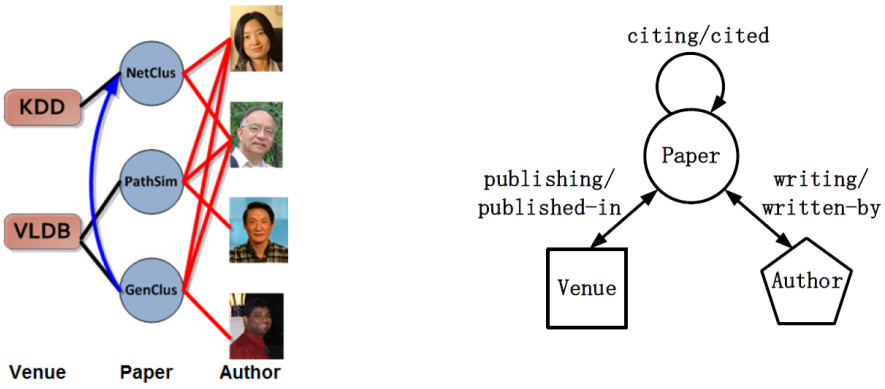
According to Robinson et al., the labeled *property graph* model is the most common graph model implemented in graph databases with the following characteristics [48]:

- Consists of *nodes* and *relations*
- Nodes and Relations can have key-value pairs as *properties*
- Nodes can be labeled*labels*
- Relationships are *named* and *directed*, and always have a start and end node

In section 2.1 we introduced the concept of a heterogeneous graph and in section 2.2 we presented the definition of a HIN. We explained that both concepts are actually similar and differ only in the naming of its objects. Thus, we can represent each HIN with its network schema and each heterogeneous graph respectively as a property graph in a graph database which supports the *property graph* model.

2.4 Network Schema and Network Instance

A *network schema* $S = (A, R)$, with a set of node types A and a set of relation types (R), is a meta template of a network and defines type constraints on the sets of nodes and edges. A network is semi-structured as a consequence of those constraints. The network schema itself is a heterogeneous graph. Given a link type $r \in R$ which connects object types $s, t \in A$, these types are called source - and target object type of link type r.
A network, adhering to the constraints of a *network schema* is called a *network instance* [52].



(a) Network instance (b) Network schema

Figure 2: An example of heterogeneous information network on bibliographic data [56]

Figure 2 (a) shows a concrete network instance following the the network schema visualized in figure 2 (b). The constraints are intuitively visible in the schema. For example, a *Venue* and an *Author* node must not have a direct relation. Thus, in the network instance there does not exist such a relation.

2.5 Meta Path, Path Instance and Path Count

A path P defined on the graph of a network schema $S = (A, R)$ is called a *meta path* and is stated as $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} ... \xrightarrow{R_l} A_{l+1}$. The length l of a *meta path* P is defined as the number of relations in P. A path $p = (v_1 v_2 ... v_{l+1})$ in a network G *follows* the *meta path* P, if $\forall i, f_v(v_i) = A_i$ and $\forall i, e_i = (a_i a_{i+1}), f_e(e_i) = R_i$. f_v denotes the node type mapping and f_e the relation type mapping as presented in section 2.2. v_i refers to the i_{th} vertex in

p and e_i relates to the i_{th} relation in p. The path p is called a *path instance* of the meta path P. Moreover, two meta paths $P = (A_1, A_2, ..., A_l)$ and $P' = (A'_1, A'_2, ..., A'_k)$ can be concatenated if and only if $A_l = A'_1$ [58]. The number of different path instances between A_1 and A_l following P is called the *path count*.

2.6 Data Parallelism and Task Parallelism

We refer to *data parallelism* if we split up the data in chunks and execute the same processing logic in parallel on these subsets [47]. A simple example would be to increment each element of a matrix by 1. We could slice the matrix per row and execute the computation in parallel on each chunk. We can arbitrarily divide the matrix in this example, because the computation is executed on each element and independent of other elements in the matrix. In contrast, we refer to *task parallelism* if we execute different tasks of an algorithm in parallel, i.e. we are executing different computations asynchronously in parallel on either the same or different data [47]. According to Amdahl's law, the degree of parallelism of a program is limited by the serial parts of a program which can not be executed in parallel (cf. section 2.8).

2.7 Actor System

An actor system is a system implemented according to the actor model, which in turn is a model for concurrent computations in distributed systems. Actors are the loosely coupled entities in the distributed system and are able to model shared resources and local state, send messages to other actors and create new ones [2]. Flink uses Akka [33] as its underlying actor model for orchestrating its actors as the *Jobmanager*, *Taskmanager* and *JobClient* asynchronously [14]. The actor system can be seen as a container comprising all actors. The actor system offers shared services like *scheduling*, *logging* and *death watch mechanism* for example. The actor system of Flink needs to be configured according to the actual use case, as we will discuss in section 6.3.2.

2.8 Performance Analysis for Parallel Computing

Scalability is usually evaluated in two different ways in the context of high performance computing [8]: *Strong scalability* refers to changes of the runtime of an algorithm while varying the number of computing nodes but keeping the problem size stable. In other words, strong scalability observes the possibility to solve a problem as quickly as possible.

The scalability of parallel code is mainly evaluated by looking at the strong scalability [55]. The speedup for strong scalability is defined as

$$S = \frac{t_1}{t_N} \tag{1}$$

with the number of parallel processing elements N, the amount of time t_1 one parallel processing element needs to process the fixed sized problem and the amount of time t_N needed by N elements respectively. The perfect scalability would be $S = N$. *Amdahl's law* is a formula used for predicting the theoretical speedup in parallel computing for fixed-sized problems and given by the following formula [4]:

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}} \tag{2}$$

p is the fraction of the code which benefits from the parallelization by a factor of N. Moreover,

$$\lim_{N \to \infty} S(N) = \frac{1}{1-p} \tag{3}$$

shows that the speedup is limited by the non-parallelizable parts $(1-p)$ of a program. Thus, the perfect strong scalability $S = N$ can not be achieved according to Amdahl's law unless $p = 1$, which is not possible practice, since there is always a non-parallelizable part like reading data or control logic for example.

Weak scalability describes how the runtime of an algorithm changes when changing the number of nodes while keeping the problem size per node stable. In other words, scale by solving bigger problems in the same amount of time. The perfect scalability would be constant time independent of the number of processing elements[55]. The speedup from the perspective of weak scalability can be predicted by *Gustafson's law* [22]:

$$S(N) = s + N(1-s) \tag{4}$$

Additionally, we can calculate the *efficiency* as a metric indicating the return on hardware investment based on the speedup S_N and the number of processors N [36]:

$$E(N) = \frac{S_N}{N} \tag{5}$$

3 Related Work

In this section, we will present the related work as the research context of this thesis. Therefore, we briefly present key facts of a predecessor thesis conducted at SAP. Next, we will discuss similarity and relevance in HINs. Moreover, will talk about graph database - as well as Apache Flink internals. Additionally, we introduce the vertex-centric graph processing model at the example of Pregel. Next, we will cover the *parallel meta-graph extraction* framework as the basic concept of the distributed implementation created in the context of this thesis. Finally, we concisely address the topic of performance analysis for parallel computing which will be the methodological foundation for the eventual scalability evaluation.

3.1 Similarity and Relevance in Heterogeneous Information Networks

Recommender systems heavily depend on similarity metrics as a mean to quantify the relevance of items to be potentially recommended. Similarity and relevance are related terms [44]. Similarity refers to the comparison of objects with the same type. *How similar are two documents?* In contrast, relevance is meant when looking at objects with different types. *How relevant is another author when looking at a paper created by another one?* Here we want to recommend relevant authors based on a paper a reader is currently reading [52]. The *Term frequency–inverse document frequency* (TF-IDF) is a similarity measure used in information retrieval - and text mining tasks and it is one of the most popular techniques in content-based recommendation generation [7]. TF-IDF measures similarity based on the importance of words in a document in relation to a collection of documents.

Information networks (cf. section 2.2) provide new possibilities for generating recommendations. There is semantic information hidden in the structure of information networks and especially in HINs. The analysis of such can lead to interpretable results and meaningful conclusions as stated by Sun et al. [56]. A co-author relationship could be encoded in the network structure , when two *Authors* have a *written-by* link to the same *Paper* like in the example of the bibliographic network shown in figure 2. If two *Papers* have a *published-in* link to the same *Venue*, they will likely be content-related. The latter example is more interesting, because a co-author relationship is more obvious and the information can probably be found somewhere else, whereas the contentwise relationship must be explicitly mined by another techniques, like TF-IDF. It is not clear whether TF-IDF would capture this kind of relevance or similarity, because the two authors might have used different terms semantically referring to similar things. Two papers might have been published in the context of graph analysis, whereas the authors of the one might

use the notion of networks with objects and links and the authors of the other use the notion of graphs with vertices and edges. Of course, this is true for the opposite. TF-IDF might capture similarities that a graph-based method might miss. Thus, a combination of different methods is advisable. The concept of meta path (cf. section 2.5) is the basis for different important similarity and relevance measures in HINs, because it captures the semantics hidden in a HIN. Sun et al. introduced *PathSim* [58], which is a similarity metric based on meta path in HINs for finding objects of the same type in a network. They showed that PathSim similarity produces better results for finding peer objects than random walk based similarity measures [56] like SimRank [29] or Personalized PageRank [30]. PathSim is calculated by normalizing the path count between the start - and end node. Random walk based methods are also possible based on meta path as in the example of *Path Constrained Random Walk* (PCRW).

Cheng et al. recently proposed meta structure based relevance measure [12] like Structure Constrained Subgraph Expansion (SCSE). SCSE is based on the meta structure count between the start- and end node analougous to PathSim in the meta path context. Meta structures can contain cylces in contrast to meta path and thus can be used to model more complex relationships.

3.2 Graph Search

The two most common and most fundamental ways to search a graph $G = (V, E)$ are *depth-first search* (DFS) and *breadth-first search* (BFS). Those two algorithms form the basis for many other graph processing algorithms [50], [48].

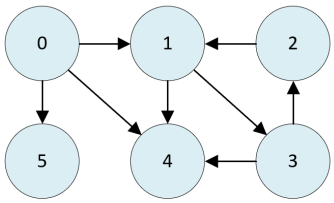
In *BFS*, we start searching from an arbitrarily selected vertex $v \in V$. Next, we explore all direct neighbors of v, before going to the next level relative to v. In other words, we search wide before we going deeper in the graph.

In *DFS*, we start again from v. In contrast to BFS, we explore each branch completely before searching the next one, i.e. we are going deep first instead of wide.

Both algorithms take $O(V+E)$ time, since they will eventually visit all vertices connected to v in time proportional to the sum of their respective degrees [23], [43]. Hence, if the graph is connected, both algorithms will finally visit all vertices. Nevertheless, BFS is preferred when finding the *shortest path* (edge count) between two vertices for example. This makes sense when illustrating how each algorithm would approach this problem. Obviously, BFS is usually more efficient, since it searches the target vertex level by level. Thus, the first match will be the shortest path. DFS would need to enumerate all possible paths between the two vertices to be able to decide which one is the shortest one. Although

DFS could be optimized by restricting the search to only those paths which are shorter than the shortest one currently known, we are still looking for the closest solution. Hence, BFS is preferred here, while having the same time complexity.

Furthermore, it should be noted that the concrete order how the algorithms traverse the graph depends on the concrete representation of the graph. For example, a common data structure would be an array of *adjacency lists*, where the array is indexed by the vertex ID. The array contains one list per vertex and thus represents the full graph. The respective lists are unordered and thus may be inserted randomly. Figure 3 illustrates a simple graph and the functioning of a DFS based algorithm for marking vertices versus a BFS based one.



Depth-First Search	Breadth-First Search
Node 0	Node 0
Node 1	Node 1
Node 3	Node 4
Node 2	Node 5
Node 4	Node 3
Node 5	Node 2

Figure 3: BFS vs. DFS functioning example [32]

Both algorithms are theoretical concepts laying the foundation for further graph algorithms like *Shortest Path* for example and are not specifically designed for computing meta paths and especially not in a data-parallel manner as required by parallel data processing systems. Although, it might be possible to derive such an algorithm inspired by BFS or DFS, we focus on evaluating state-of-the art approaches for computing meta path counts instead of aiming to design completely new algorithms.

3.3 Graph Databases

A *graph database* is a database management system like a relational database management system (RDBMS). The distinctiveness is the structure of the underlying storage system. We will use the term *graph database* as a synonym for graph database management system. It leverages graph structure, i.e. data is represented as vertices and edges [45]. A *graph database* supports all the common CRUD (Create, Read, Update, Delete) methods and exposes a graph data model. As a consequence the CRUD methods will be applied on vertices and edges instead of tables as compared to RDBMS. But *graph databases* are also designed for the usage with OLTP systems and therefore foster transactional performance, - integrity and availability like RDBMS.
Graph databases can be characterized by the underlying storage and the processing engine.

As stated by Robinson et al., those characteristics should be taken into consideration when investigating *graph database* technology. Some databases serialize the graph structure into a relational database model for example. But the most interesting storage type is the so called *native graph storage* (see section 3.3.2). The type of the processing engine in *graph databases* also varies between different systems. Some graph database definitions prescribe that the database must use *index-free-adjacency* (see section 3.3.1) as a graph processing mechanism. In the following, we comply to the definition of Robinson et al., who define every database management system as a graph database, that exposes a graph model to the outside. Furthermore, we adopt the expression of *native graph processing* for graph processing engines that leverage *index-free-adjacency* [48].

In the following, we will present Neo4j's graph processing engine and additionally its graph storage layout as an example of a native graph engine and native graph storage respectively. We base the experiment conducted in the context of this thesis on Neo4j. Moreover, Neo4j is the most popular graph database at the time of writing [54].

3.3.1 Graph Processing with Index-Free Adjacency

As described in the previous section, the property graph model is supported by the majority of graph databases, but the way graph operations are being processed in the database differs more often between the different systems. When analyzing how the operations are being processed, we basically have to look at how the graph is represented in the main memory. In section 3.3, we said that a graph database employs a *native graph processing* engine if it uses *index-free adjacency*. Such a processing engine persists references to all direct neighbor nodes at each node itself. The references can be seen as *local indices*. In contrast, a graph processing engine which solely relies on global indices for joining semantically linked nodes together is called a *non-native graph processing engine*. Traversing the graph is therefore much cheaper when using a *local indices* as compared to using *global* ones since we only need to follow the pointer locally present at the node itself. Consequently, query times are usually independent of the total size of the graph and instead proportional to the size of the part of the graph to be searched as stated by Robinson et al. [48]. This statement is not absolutely true. A graph database commonly uses global indices in addition even though it implements index-free adjacency. Furthermore, the query cost depends on the type of query. For example, if we want to retrieve several vertices just by their ID, we can not leverage index-free adjacency. Instead, this kind of query requires using a global index if available. Yet a lot of graph queries retrieve locally connected parts of the graph, which is the main reason for using a graph data model.
For example one index lookup for traversing to a direct neighbor is usually $O(log(n))$ in complexity when using binary search for seeking a global index, as compared to O(1) when using a direct pointer, where n is the number of elements to be searched. Moreover,

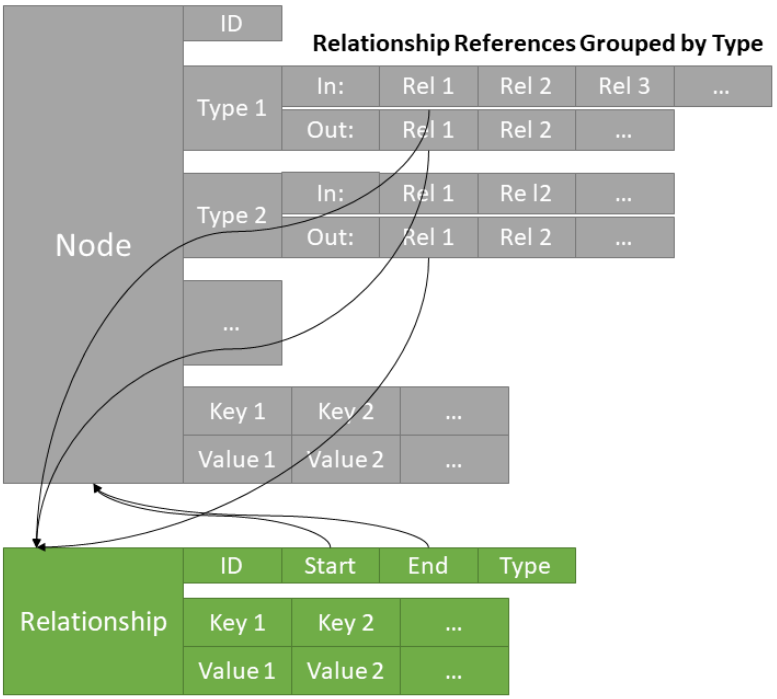


Figure 4: Neo4j's Node- & Relationship Cache [34]

if we want to traverse a path of m steps, the cost would be $O(m * log(n))$ for the global index approach versus $O(m)$ in case of index-free adjacency [48]. Thus, the cost for such queries is significantly lower when using the index-free adjacency mechanism.
The algorithmic complexity for traversing a path instance or relationship between two nodes is equally costly in both direction, although the relations are directed. This is a further characteristic of native graph processing engines. Therefore, local indices for incoming - as well as for outgoing relations need to be kept at each node to support those *bidirectional* traversals. Regarding a non-native processing engine, a lookup in the *opposite* direction from the one the index was originally created from would either require a O(n) complex lookup or a further index needs to be created to support this direction. It should be noted that O(n) corresponds to the complexity of brute force algorithms which is too costly to be applied in this context in practice. Moreover, it should be emphasized that write operations on nodes with a high in- and / or out-degree may incur much higher costs as compared to using global indices. We consider a famous person in a social network as an example. Such a person would be represented as node with a very high in-degree, whereas each ingoing relation could be a follower. If the person decides to leave the network, its node will be deleted with its local pointers to all followers. Additionally, every neighbor node must be updated too. This is a classical trade-off between read and write performance. This fact is negligible in the context of the thesis, since we will only

look at read operations.
So far we assumed in our native graph processing examples that we already found the start node for our queries. But regardless whether we want to traverse from the start node to a direct neighbor or if we want to traverse a path instance, we have to find the starting point first. The starting points for a query are called *bound nodes* or *anchor points*, because the query is *bound* or *anchored* to the those parts of the graph. A global index lookup if available or a full node set scan is necessary to find an anchor node [48].

Figure 4 shows Neo4j's node and relationship cache layout in memory. The relationships are grouped by its type and direction. So if we want to retrieve all neighbors of one specific anchor node of type t, we just need to follow all relationships of type t at cost O(1) each. Hence, the total cost is $O(Rel_t)$, where Rel_t is the number relationships of type t connected to the respective anchor node. As we can see the cost is proportional to the parts of the graph beeing searched and not to the whole graph or at least all relationships of type t in the whole graph.

3.3.2 Neo4j's Graph Storage

We say a graph database embodies a *native graph storage* if the data is represented in a graph structure on disk. Neo4j persists a graph in different *store files* on disk. The graph structure is kept in the *node store* and respectively a *relation store*. Whereas the graph data is kept in a *label store* and a *properties store*. Thus, the graph structure is separated from the data of the property graph. The idea of such a breakup is to support rapid graph traversals. There are further store files beside the ones previously mentioned. Most files have fixed sized records and often use a linked list data structure to connect the different store files.

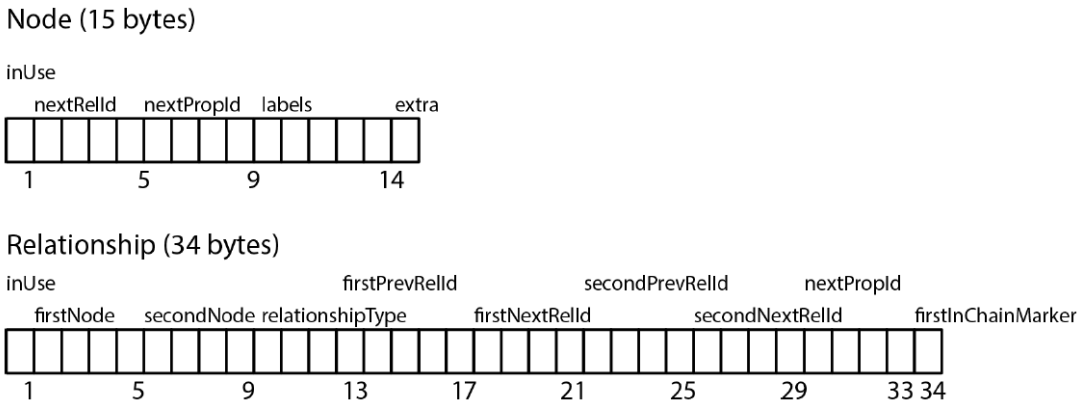


Figure 5: Neo4j node and relationship store file record structure [48]

The upper part in figure 5 shows the layout of one node record. Each node record is 15 bytes long. Consequently, each node created in a property graph has a persisted record within the node store file. The first byte of the record indicates whether the respective record is actively and hence stores a node or whether it can be used to store a new node. There is an additional file which keeps track of those free slots. The next four bytes store the ID of the first relation connected to the node. Each relation points to its successor and predecessor as we will discuss in more detail. The following four bytes point to its first property and the remaining properties are again reachable via an indirection over the first property. The succeeding 5 bytes are reserved for labels and point to the label store file. The final byte is reserved for flags. Currently, the only flag encoded here indicates whether the node is densely, i.e. whether it has high in- and/or out-degrees. It should be noted that there is no node id, but only a few pointers. The node id is inferred based on the position where it is stored. The node stored at the first byte has implicitly the ID 0. The one starting from byte 16 has the ID 1. Thus, the cost of finding a node by its ID is $O(1)$.
The lower part of figure 5 shows the structure of a record in the relation store with a fixed length of 34 bytes. Its structure is more complex as the node record structure. The start - and end node IDs and a pointer to the relationship type are stored first. The ladder is located in the *relationship type store* file. As previously stated, a record in the node store has only a pointer to its first relationship record. Consequently, a mechanism is needed to find the remaining relationships of a node. This mechanism is implemented as a linked list or more precisely as a doubly linked list. Pointers to the previous and next relation record of both start - and end node are stored here. This implementation is referred to as *relationship chain*. Moreover, there is a flag indicating whether the relationship is the first in the relationship chain. Properties of a relationship are also persisted as a linked list like in the same way as the properties of a node. The reason why the relationships are stored as doubly linked lists is to support bi-directional navigation with the same cost of $O(1)$ for one relationship.
Summarizing, the graph is persisted in different store files. Hereby, the graph structure is separated from the property data of the graph objects. These files are based on the idea of fixed-sized records and record IDs acting like pointers realized by linked list data structures. Thus, graph traversals are realized by off-set computations with the complexity of $O(1)$ for each hop between connected records [48].

3.4 Apache Flink

Apache Flink [61] is an open source system for distributed stream and batch data processing. Flink's core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations. Moreover, Flink implements

batch processing on top of its streaming engine, provides native iteration support, managed memory, and program optimization capabilities. Flink provides a Batch API and a Stream API for processing potentially infinite data streams. The systems exposes several further libraries, such as Machine Learning and Gelly the Graph library on top of these APIs. We will base the experiment conducted for this thesis on Apache Flink. Specifically, we will solely use the Batch API and the graph library, because stream processing is not relevant for our research topic. Thus, we will present the necessary background information needed for the interpretation of the results. We recommend the following publications to the interested reader as a deep dive into the concepts of the system: [3], [18]. [10], [11]. Additionally, we recommend to read the appendix 9 which gives a brief introduction to big data processing fundamentals like *Distributed File Systems*, *MapReduce* and *Iterative Parallel Dataflows*.

3.4.1 Distributed System Architecture

Apache Flink is designed in a *master-slave* architecture. Hence, a Flink cluster consists of master nodes called *Jobmanager* and worker nodes called Taskmanager. At minimum, there has to be one *Jobmanager* and one *Taskmanager*. For high availability setups, further nodes are needed. Moreover, a client is needed for submitting a job to the Flink cluster. Flink comes with it's own web based client, which additionally provides monitoring capabilities.

The distributed coordination in Flink is realized by the underlying actor system (c.f. section 2.7) Akka [33]. Hereby, the state and the behavior of the loosely coupled *actors* within the distributed system is managed. This is mainly achieved via asynchronous message passing and tracking of state of the individual actors. Possible states could be the begin of task execution or the successful termination of such. Heartbeats and related timeouts are main concepts of Akka's death watch mechanism for detecting failed nodes or ensuring ensuring the availability of non-failing ones [14].

The **Jobmanager** [63] coordinates the distributed execution of a Flink program. It allocates the required resources on the Taskmanagers and schedules the individual tasks of a dataflow job on those resources. Moreover, it is responsible for fault tolerance and therefore coordinates checkpointing and recovery in case of actual failures. Since a Jobmanager is an actor within the actor system, it comprises an actor component too.

The **Taskmanagers** [63] represent the worker nodes within a Flink cluster and thus execute the actual computations, buffer parts of the data and exchange data streams with further worker nodes within the cluster. More specifically, a worker node executes the subtasks of a dataflow. A Taskmanager itself is a JVM process and consists of 1-n

Taskslots, the *Memory Manager*, the *Network Manager* and again the *Actor System*. The resources available to a Taskmanager are slotted. The execution of the individual dataflow tasks are scheduled and executed in those slots. The memory manager takes care about the memory management (c.f. section 3.4.2) within a Taskmanager. Taskslots are threads within the Taskmanager's process. Flink divides its Managed Memory (c.f. section 3.4.2) evenly between the slots. Thus, it is possible to control how subtasks are isolated from each other. For example, a Taskmanager with one slot implies the execution of a task group in a separate JVM, whereas multiple slots per Taskmanager entail multiple threads within the same JVM working on its own dedicated part of Flink managed memory, while sharing TCP connections via multiplexing and heartbeat messages . It should be noted, that no CPU isolation is happening between the different Taskslots of one specific Taskmanager at the time of writing.

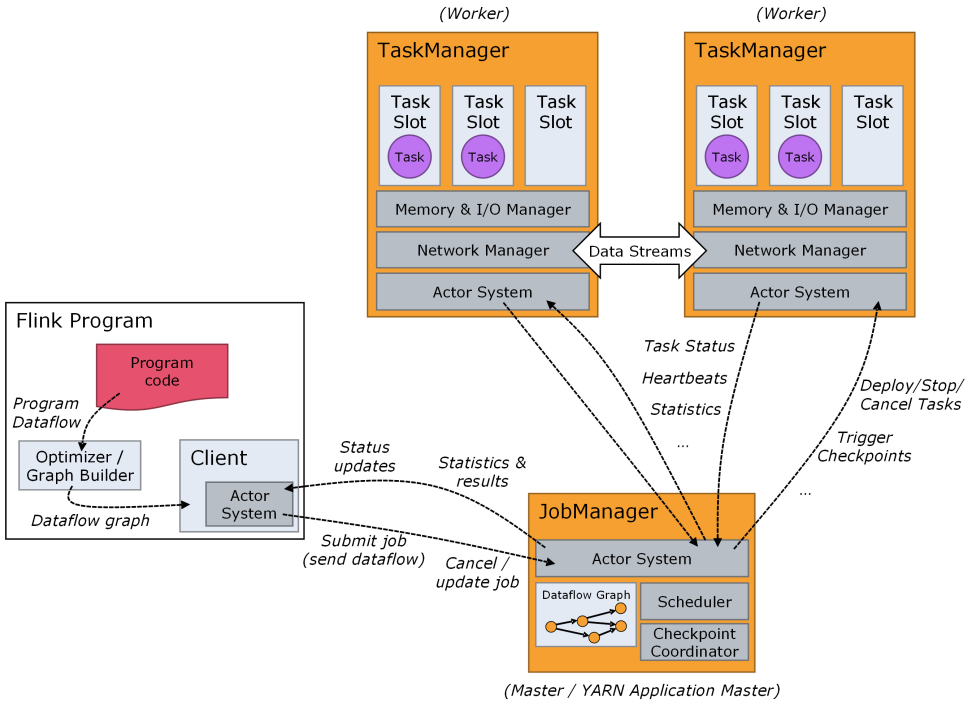


Figure 6: Apache Flink's Distributed Runtime [63]

Figure 6 shows a comprehensive view on the different roles and interactions within the Flink cluster. The Flink program is submitted to the Jobmanager via a client. Next, the Jobmanager itself allocates the required resources for executing the job and orchestrates the distributed job execution. Resources mainly refer to the *Taskslots* of the available Taskmanagers. The worker nodes exchange data streams between each other since the data is partitioned across the different Taskmanager nodes.

3.4.2 Memory Management

It is crucial to understand the memory management [15] of Flink's runtime to be able to configure the system for the individual demands arising in different use cases. Flink

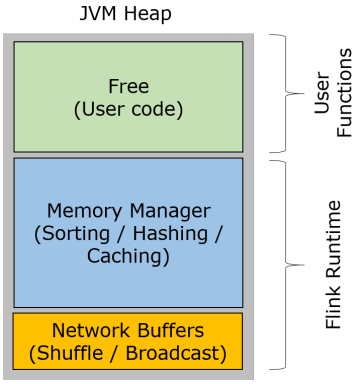


Figure 7: Heap Division in Apache Flink's Batch API [15]

employs the JVM and stores large amounts of data in memory. As already said a Taskmanager is a JVM process. Flink divides the available JVM Heap space of a Taskmanager into three regions as illustrated in figure 7.

Network Buffers: A small portion is used as buffers for the data transmission over the network of 32 KB size each. Per default Flink uses 2.048 buffers. The number can be configured via the parameter *taskmanager.network.numberOfBuffers*.

Managed Memory: Flink has it's own style of memory management and therefore uses a large part of the heap (per default 70%) as Managed Memory. This part is organized in *Memory Segments* of 32 KB each per default. Flink serializes large parts of a computation into a byte representation and stored in the managed memory. All runtime algorithms such as sorting or hashing will buffer their data here. This means, all Records stored by Flink are actually serialized into those Memory Segments. The memory manager handles the access to this memory region. Therefore, it is also called *Memory Manager Pool*. Moreover, the runtime algorithms using this memory pool, explicitly request and release parts of it. The memory manager assigns a strict memory budget to the requester. If the memory is fully utilized, then the algorithm will execute further in an out-of-core manner, which means that it is swapped to disk. Since the data is already serialized into memory pages, it is relatively efficient to move the data between memory and disk. The bottleneck will be the disk I/O here.

3.5 Pregel - Vertex-centric Graph Processing

Pregel is a special purpose system developed by Google for large-scale distributed graph processing [35]. Its master slave architecture and design goals are similar to MapReduce (cf. appendix 9), but with a *natural* graph API and improved support for iterative computations. The graph representation is partitioned across the so called worker nodes and kept in memory. Hence, Pregel exhibits data-parallelism too, but without persisting intermediate results on disk as compared to Hadoop. The idea for developing Pregel was to built a system for the efficient processing of large graphs, since MapReduce is ill-suited especially for iterative graph algorithms. Moreover, it is not intuitive to implement graph algorithms in MapReduce, whereas Pregel treats edges and especially vertices as fist-class citizens. Hence, the implementation of graph algorithms is naturally simplified. The framework is inspired by Vitalian's *Bulk Synchronous Processing* model as illustrated in figure 8. The parallel computation is executed on each vertex in iterations called *supersteps*. The actual computation takes place in a user-defined function (UDF) which gets executed on each vertex by the framework. The results are communicated via message passing between vertices. Finally, a superstep ends with a global synchronization point. The next superstep therefore begins when all messages have been delivered.

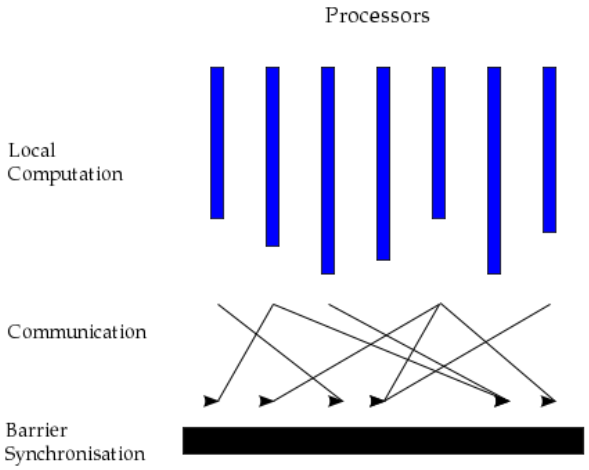


Figure 8: Bulk Synchronous Processing Model [67]

A graph is represented as vertices and edges in Pregel and thus developers think in those terms instead of more abstract data types as in MapReduce. A Pregel job gets a directed graph as input. A vertex consists of a unique ID and a modifiable, user defined value, whereas each directed edge is represented by its source -, target vertex ID and

its modifiable, user-defined value. Since the computation is executed on each vertex, the computational model is called *vertex-centric graph processing* and inspired further distributed graph processing systems like Apache Giraph [59] or libraries as found in Apache Flink or Apache Spark [60] for example. It is often said, that developers of Pregel programs need to *think like a vertex*. Additionally, each vertex has an associated state that can be either *active* or *inactive*. Initially, every vertex is active.
The UDF gets all messages sent to the respective vertex in the previous superstep as input. A message consists of the target vertex ID and a user-defined message value. The message type is fixed for the respective computation. In other words, a message will always have the same type across all supersteps of one algorithm. Furthermore, the UDF encapsulates the algorithmic logic of the graph computation. Hereby, a vertex can change its value or the value of its outgoing edges. Even the graph topology can be changed by deleting or creating edges for example. Moreover, the vertex can send messages to other vertices to be received in the next superstep. Lastly, a vertex can set itself to inactive by *voting to halt*. This means it will not participate in the computation of the next supersteps unless it will become active again by receiving a further message. To become inactive again, a vertex must explicitly vote for halt once more. The computation finishes when all vertices are inactive. A Pregel program outputs the values of explicitly selected vertices.

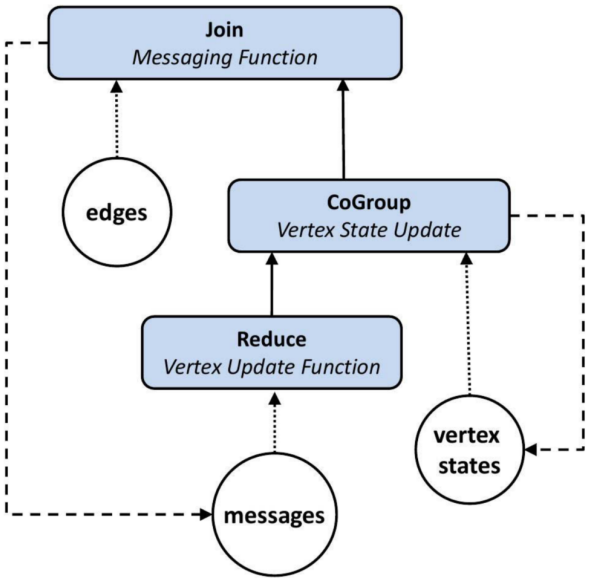


Figure 9: Vertex centric graph processing as iterative dataflow plan in Pact [49]

Moreover, it was shown by the research community that such programs can easily be executed in general dataflow systems supporting iterative dataflows [49]. Figure 9 illustrates exemplary how Apache Flink executes vertex-centric graph computations as a parallel dataflow using three different datasets. The *vertex state* represents the vertices with its

values and will be the solution set of the iteration. Furthermore, the *edges* of the graph and the *initial message set* will be passed to the Pact program. The messages correspond to the workset in Flink's Delta Iterations (cf. appendix 9). The vertex update function is realized by the Reduce operator applied on the message set, in other words the workset of Flink's Delta Iterations. Followed by a CoGroup to reflect possible changes in the vertex state set. Lastly, a new workset is created as a result of joining the edges with the CoGroup result and executing the messaging UDF here [65].

3.6 Meta-Graph Extraction Framework

Shao et al. [51] recently proposed an approach for analyzing heterogeneous graphs by extracting *edge homogeneous graphs*, i.e. all edges have the same type. They state that graph extraction is a basic preprocessing step for analyzing heterogeneous graphs, because it eliminates the gap between heterogeneous graphs and classical graph analysis, because it is possible to execute graph algorithms like PageRank on the extracted graph for example. An edge in the extracted graph is defined by a *line pattern* in the heterogeneous graph and an edge value is calculated by two user-defined aggregate functions. The definition of a line pattern given by Shao et al. corresponds to the definition of meta path given in section 2.5. The proposed approach is called *parallel meta-graph extraction* framework and is based on the idea to search for instances of short paths in the heterogeneous graph and to concatenate those paths to generate the final paths determined by the actual meta paths in the vertex-centric model (cf. section 3.5). The analysis of the heterogeneous graph with respect to one specific meta path results in one homogeneous graph per meta path. The framework compiles a meta path into a so called *Path Concatenation Plan* (PCP), which determines the order of concatenating *primitive patterns* for generating final paths in a divide-and-conquer manner.

3.6.1 Homogeneous Graph Extraction Problem

Shao et al. formally define the homogeneous graph extraction problem [51] as follows:

> *Given a* $G_{he} = (V, E, L_v, L_e, A_v, A_e)$*, a* $G_p = (V_p, E_p, L_{pv}, L_{pe})$ *and user-defined aggregate functions* $\otimes$ *and* $\oplus$ *generate an edge homogeneous graph* $G = (V', E', A'_e)$*. Vertex set* V' *is the union of vertices who match the start vertex* v_s *and end vertex* v_e *of* G_p*, i.e.,* $V' = \{v|f(v) = fp(v_s)\} \cup \{v|f(v) = fp(v_e)\}$*. Each edge* $e = (u, v) \in E'$ *indicates there exist at least one path matched by* G_p *between vertices* u *and* v*. The attribute* $a \in A'_e$ *of an edge* $e = (u, v)$ *is com-*

puted from all the paths between vertices u and v by the user-defined aggregate functions $\otimes$ *and* $\oplus$*.*

3.6.2 Pairwise Aggregation

The edge values in the extracted homogeneous graph is calculated on two levels by pairwise aggregation using two binary operators $\otimes$ and $\oplus$. First, the value $val(p)$ of one specific path instance p is calculated based on the edge values $w(e_i)$ for all $e_i \in p$ using the aggregation function $\otimes$. Next, the final edge value $val(u, v)$ of the new edge (u,v) in the homogeneous graph is calculated by aggregating all path instances $p_i \in P_{uv}$ applying the second user-defined aggregate function

$$val(p_i) = \forall_{p_i \in P_{uv}} \otimes w(e_i) \tag{6}$$

$$val(u, v) = \forall_{p_i \in P_{uv}} \oplus val(p_i) \tag{7}$$

For example, we could choose $\otimes$ and $\oplus$ such that both compute the sum and thus are simple addition operators. This means $val(p_i)$ would be just the sum of all edge values of one specific path instance and $val(u, v)$ is the final sum aggregated from all path instance values $val(p_i)$ of all path instances between u and v.

3.6.3 Extended Label Sets

The concepts of *extended vertex label set* and *extended edge label set* are introduced for concatenating path in a vertex-centric way. The *extended vertex label set* is defined as $L_{ev} = L_v \cup id_i$ with the vertex label set L_v and additional vertex labels id_i, which are the IDs of the primitive patterns and used to concatenate primitive patterns for generating longer paths. The *extended edge label set* $L_{ee} = L_e \cup \emptyset$ includes the edge label set L_e and additionally the empty edge label $\emptyset$. The original labels are called *native label* (NL) and the new ones *query label* (QL).

3.6.4 Primitive Pattern

A *primitive pattern* $P_p = (id, V_{pp}, E_{pp}, L_{ev}, L_{vv})$ is a meta path of length two with the extended label sets and the *id* as unique identifier. $V_{pp} = \{v_s, v_p, v_e\}$ is the vertex set of a primitive pattern and consists the *start vertex* v_s, the *pivot vertex* v_p and the *end vertex* v_e. The pivot vertex is the middle node in a primitive pattern. $E_{pp} = \{e_l, e_r\}$ denotes the edge set of the primitive pattern containing the left edge $e_l = (v_s, v_p, dir)$ and the

right edge $e_r = (v_p, v_e, dir)$ with dir as the edge direction. We denote a primitive pattern by PP_{id}, when referring to a specific pattern. For example the primitive pattern with $id = 1$ is denoted by PP_1. Furthermore, we distinguish four types of primitive patterns as illustrated in figure 10, namely NL-QL, QL-QL, NL-NL, QL-NL. NL-QL means for example that $f_v(v_s), f_v(v_e) \in L_{ev}$.

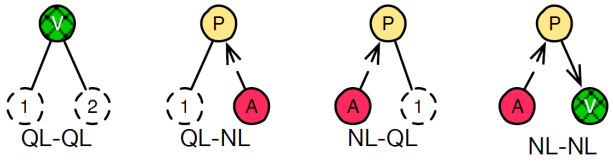


Figure 10: Four types of primitive patterns with $L_v = \{V, P, A\}$ [51]

The leftmost primitive pattern in figure 10 implies that the result of PP-1 is matched by v_s and v_e matches the result of PP-2 consequently. Those matched instances are then concatenated on v_e of PP-1 and v_s of PP-2, which both have P as vertex label. The explanations for the other PP types is similar.

3.6.5 Path Concatenation Plan

A general meta path can be compiled into a *Path Concatenation Plan* (PCP) by breaking a meta path up into several primitive patterns. Thus, a PCP is a set of primitive patterns and dictates how those patterns or instances of such can be concatenated to form the general meta path or an instance of the path respectively. The dependence between the primitive patterns and hence the concatenation order forms a binary tree. Each node in the tree is a primitive pattern and each leaf node is always a NL-NL pattern. Moreover, a node X is the parent of a node Y if and only if Y's primitive pattern ID is a vertex label of X's primitive pattern. The height of a tree is declared as H. Shao et al. showed that the height of a PCP is at least $log_2(l)$ given a meta path of length l with $l \geq 2$ [51].

3.6.6 PCP Evaluation Algorithm

Figure 11 illustrates the execution flow of the graph extraction framework based on a sample meta path derived from the bibliography network schema as shown in figure 2. The master node gets a meta path and two aggregate functions $\otimes$ and $\oplus$ as user input (cf. section 3.6.2). The meta path is then compiled into a path concatenation plan of length two. Consequently, the computing nodes generate complete path instances within two iterations and finally create the homogeneous graph by executing the aggregate

functions. We explained in section 3.6.5 that the dependence between the individual

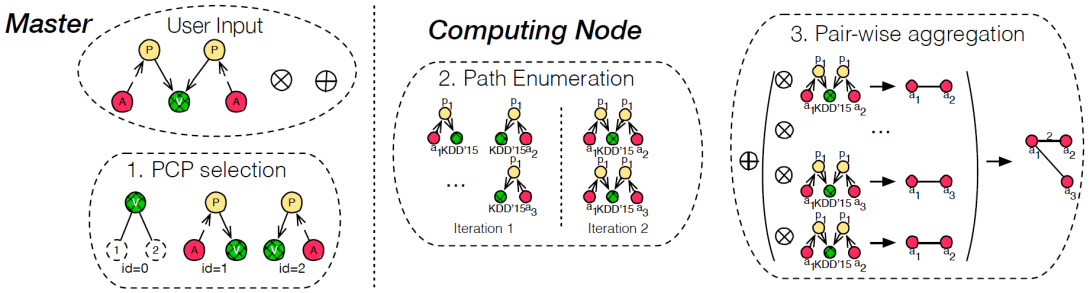


Figure 11: Example Execution flow of parallel graph extraction framework [51]

primitive patterns in a PCP forms a binary tree. Using this intuition, a PCP is evaluated from leaf to root. Primitive patterns on the same tree level can be computed in parallel, because they are independent of each other. The final paths will just be generated when evaluating the root PCP. This is why the evaluation algorithm shown in figure 11 requires at least two iterations depending on the concrete implementation. A primitive pattern has length two and the start - and end vertex are both neighbors of the pivot vertex consequently. Thus, it is possible to concatenate partial paths based on the pivot vertex by neighbor exploring instead of global searching. Iteration 1 of the example flow matches all instances of PP_1 and PP_2. The final path will be generated in the second iteration by concatenating the instances $pp_1 \in PP_1$ and with the respective instances $pp_2 \in PP_2$ where $v_{e_{pp_1}} = v_{s_{pp_2}}$. In other words, where the end vertex of pp_1 is equal to the start vertex of pp_2.

The pseudocode for the PCP Evaluation algorithm and hence the basis of our implementation can be found in the appendix section 9.

3.6.7 PCP Evaluation Cost Analysis

Shao et al. state that the cost is mainly dependent on the path enumeration as well as the the pairwise aggregation for computing the final edge values.

The cost for enumerating all path instances is the cost for evaluating a PCP:

$$S_{PCP} = \sum_{pp_i \in PCP} S_{pp_i} \tag{8}$$

which is in turn the sum of the cost of the primitive pattern evaluation. The cost S_{pp_i} for evaluating the primitive pattern pp_i is given by:

$$S_{pp_i} = \sum_{v:f(v)=f_p(v_p)} d_{vl} * d_{vr} \tag{9}$$

where d_{vl} is either the number of edges matched by e_l of pp_i or the number of intermediate paths matched by another primitive pattern in case v_s is of type QL. Correspondingly, d_{vr} is either the number of edges matched by e_r of pp_i or the number of intermediate paths matched by another primitive pattern if v_e has a query label. Moreover, d_{vl} relates to *leftRes* in the pseudocode given in appendix 9 and d_{vr} to *rightRes* respectively.

3.6.8 PCP Selection Strategies

A PCP consists of different primitive patterns which define how the full path can be constructed based on partial paths. A meta path can be divided in different primitive patterns and thus different PCPs can be derived from a meta path, if the path is long enough. Shao et al. present three different PCP selection strategies:

Iteration Optimized Strategy
The goal of this strategy is to select a PCP which has height $log_2(l)$, because this is the minimal height possible given a meta path of length l and implies the minimal number of supersteps. The meta path is recursively divided into two line patterns with the same length until each of these sub-line patterns has length less than three. If there are several possibilities for dividing a line pattern into two line patterns with the same length, than we randomly pick one.

Path Optimized Strategy
This plan aims to find a PCP which minimizes the number of intermediate results, because the total cost for evaluating a PCP is dependent on the number of intermediate paths as shown by Shao et al. A path size estimation is necessary for this strategy.

Hybrid Strategy
Since the number of intermediate paths and the height of a PCP do not exhibit positive correlation Shao et al. present a further strategy, which aims to select a PCP with minimum number of intermediate paths from the ones requiring least iterations. Hence, the minimization of the number of iterations is prioritized, because they claim that this is the main performance driver.

4 SAP Community Graph Analysis

Now, we want to evaluate approaches for efficiently computing meta paths in the SAP Community graph. Therefore, we first formalize the task by defining the path count computation problem (cf. section 4.1). Next, we present the SAP Community network schema. Additionally, we define multiple meta paths on this network schema and give a semantic interpretation of these. Moreover, we analyze the meta paths by looking at the distribution of relations and partial paths and approximate the selectivity.

4.1 Problem Definition

Given the SAP Community network G following the network schema S (cf. section 4.3) and a meta path $P = (a_1 a_2 ... a_{l+1})$ (cf. section 2.5) with length l defined on S. We want to compute the path count c_{uv} between vertices $u, v \in G$, with an object type mapping $f_v(u) = a_1$ and $f_v(v) = a_{l+1}$. Hereby, we define the *path count computation problem* (PCCP).

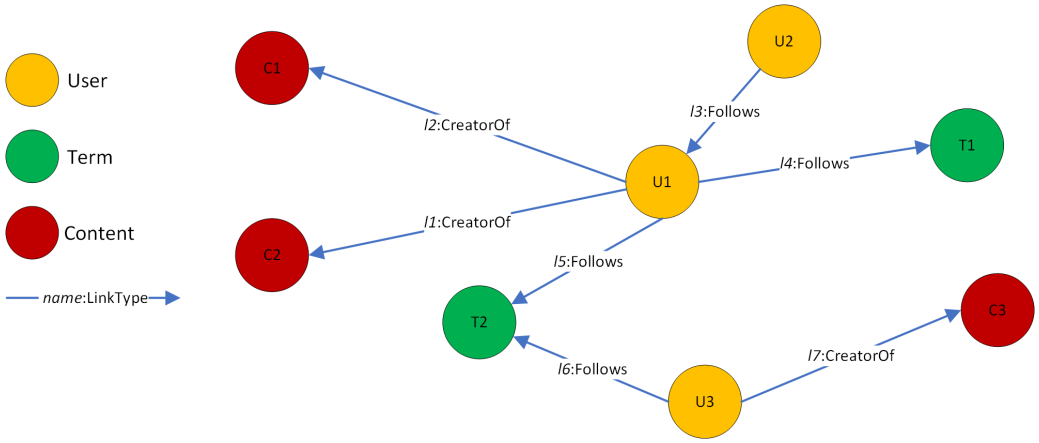


Figure 12: Toy Graph Instance Following the SAP Community Network Schema

We illustrate the PCCP based on a toy graph shown in figure 12 and the meta path *CUTUC* (cf. section 4.4.6). The graph adheres to the SAP community network schema presented in 4.3. Table 2 shows the final result of the solved PCCP for meta path CUTUC on this specific graph, where the start node corresponds to u and the end node to v. The path count for *C1* and *C3* is 1 as well as for *C2* and *C3*, while the count for *C1* and *C2* is two. This means there exists exactly one path instance between *C1* and *C3*, exactly one instance between *C2* and *C3* and exactly two instances between *C1* and *C2* respectively.

Content	$\xleftarrow{CreatorOf}$	**User**	$\xrightarrow{Follows}$	**Term**	$\xleftarrow{Follows}$	**User**	$\xrightarrow{CreatorOf}$	**Content**
C1	*l2*	U1	*l5*	T2	*l6*	U3	*l7*	C3
C1	*l2*	U1	*l4*	T1	*l4*	U1	*l1*	C2
C1	*l2*	U1	*l5*	T2	*l5*	U1	*l1*	C2
C2	*l1*	U1	*l5*	T2	*l6*	U3	*l7*	C3

Table 1: Path Instances Following Meta Path CUTUC

To be able to calculate the path count, we first need to find all meta path instances following the meta path *CUTUC* in the graph. Table 1 shows the exact path instances. As in the example of the the nodes *C1* and *C2*, it is possible that different path instances only slightly differ. The two path instances are quite similar and only differ in the term and the respective edges.

Start Node	**End Node**	**Path Count**
C1	C3	1
C1	C2	2
C2	C3	1

Table 2: Path Counts for Meta Path CUTUC in Toy Graph

4.2 General Approach

The SAP Community graph is kept in Neo4j. Consequently, we evaluate the applicability of the native query capabilities of a graph database for solving the path count computation problem at the example of Neo4j. For this reason, we formulate Cypher queries solving the path count computation problem for all considered meta paths. Cypher is a query language invented by Neo4j. Next, we execute those queries in a dedicated Neo4j instance in order to measure the runtimes of those. These runtimes will be our baseline for the comparison with the distributed processing approach.

Furthermore, we create a distributed implementation for solving the path count computation problem, since parallel data - and respectively parallel graph processing are scalable approaches by design. Especially the vertex-centric graph processing model (cf. section 3.5) is inherently scalable, since the actual computation is executed on each vertex in a data-parallel manner. Thus, the approach is horizontally scalable and it should be possible to reduce the runtimes on the current data set by scaling up the number of parallel worker nodes. The implementation is conceptually inspired by the meta graph extraction

framework (cf. section 3.6) and follows the vertex-centric model (cf. section 3.5). We measure and compare the runtimes for both approaches and evaluate the scalability of such additionally. On the one hand, we examine how the respective approaches might handle increased graph sizes, which might occur in the future. On the other hand, we study the scalability by varying the length of the meta paths.

4.3 The SAP Community Graph

In the following we are going to describe the SAP Community graph. The SAP Community knowledge graph is a *HIN instance* and defined as $G = (V, E)$, with:

- V, the set of 14,708,372 nodes
- E, the set of 30,810,46 relations

The network instance G follows the *network schema* $S = (L_v, L_e)$ with:

- $L_v = \{content, term, user\}$, the set of node types
- $L_e = \{CreatorOf, Follows, PartOf, RelatedTo, ResponseTo, TaggedWith\}$, the set of link types

The SAP Community network schema consists of three different node types: *content, term, user*. The content node can be further specified as *blog, discussion, question or comment*. This is achieved by an additional node attribute. Figure 13 shows the network schema of the SAP Community and consequently illustrates the type constraints on the set of nodes and links. We omitted the node attributes in figure 13 for the sake of simplicity. The *entitySubType* node attribute is the only relevant attribute anyway in the context of this thesis. Moreover, it should be noted that we do not have any link attributes. The node type *user* represents all kind of users registered on the SAP Community platform, like consultants, developers and so on. The node type *term* represents what is usually called a category or a tag of a specific content item. A link between *term* and *content* may be inferred by text analysis or be manually assigned by a user. The reflexive relationships on the *term* can be interpreted as subcategories or tags related to a specific category. *Comments* and *answers* do not have direct links to *terms*.

In order to keep the dataset stable we set up an individaul Neo4j instance with a copy of the data.

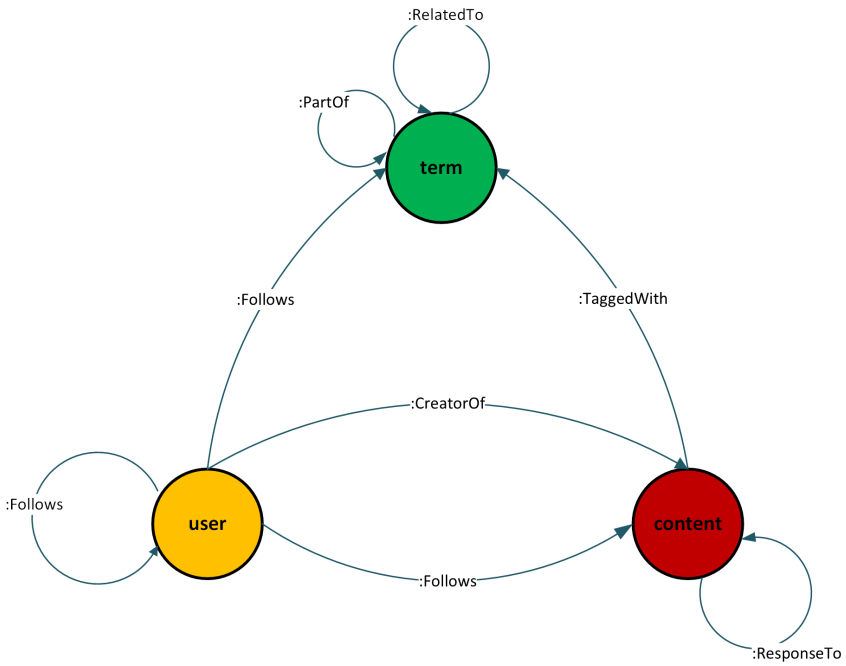


Figure 13: SAP Community Network Schema

Node Type	Node Subtype	Node Count
content		13,326,223
	blog	101,407
	question	108,903
	discussion	2,444,245
	comment	10,520,540
	answer	129,498
term	-	108,117
user	-	1,274,032

Table 3: SAP Community Node Counts per Type

4.4 Selected Meta Paths

A meta path encapsulates semantic information. Hence, the choice of those is essential for eventually generating meaningful meta path based recommendation results. In general, there are two possibilities for selecting meaningful paths. Either machine learning can be used for automating the selection process or the process can be done manually based on domain knowledge. We are using domain knowledge in the context of the thesis.

We will not consider the machine learning approach, because this is not in scope of the thesis. Moreover, we do not want to evaluate which approach might generate meta paths leading to the best recommendation results, but we want to evaluate efficient and scalable approaches for solving the path count computation problem. Anyway, the insights gained in this experiment do not directly depend on the concrete meta paths chosen here and can be interpreted more generally. In the following experiments the graph data will be analyzed with respect to those meta paths. The meta path were chosen such that the length will be varied as well as the selectivity. Now, the selected meta paths will be presented and an interpretation will be given. We selected the paths according those two dimensions, to be able to evaluate how the respective approaches might scale when selecting a meta path.

The *term* nodes are of special interest for generating meaningful recommendations, because they imply important semantic. The *terms* connected to *blogs* describe the item contentwise. For example, a *blog* that is tagged with the *term* with value *database* will most likely contain information related to this topic. Moreover, *users* following specific *terms* indicate interest into those topics. Therefore, we include the node type *term* in each meta path.

4.4.1 Blog-Term-Blog BTB

The meta path BTB is a line pattern of length two and can be interpreted as two different *blogs* being similar, if they are connected to the same term.

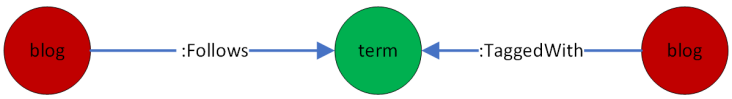


Figure 14: BTB

Figure 14 visualizes the meta path BTB. Both *blog* nodes have an outgoing relation of type *TaggedWith* to the same *term*.

4.4.2 Content-Term-Content CTC

The meta path CTC, shown in figure 15, is quite similar to the BTB path. Consequently, the semantic interpretation is similar.

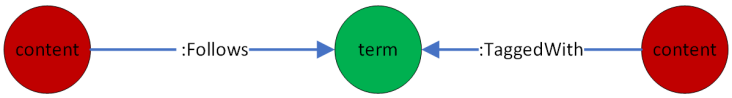


Figure 15: CTC

In section 4.3 we explained that a *content* node can be expressed as a *blog* among others. This means that a *blog* node is actually a *content* node with a constraint on the *entitySubType*. Hence, the selectivity of this path is significantly higher as in the more specific BTB case and the result set is a superset of the BTB selection. Nevertheless, this path leads does not lead to more meaningful recommendations, since the CTC path will consider questions, answers and comments which are actually not suitable for recommendations, whereas recommending *blogs* is useful in the considered domain.

4.4.3 Blog-User-Term-Blog BUTB

BUTB is a meta path of length three, i.e. it has one more edge as the previously presented paths.

Figure 16: BUTB

The most left node in the path refers to a *blog* again. It has an ingoing *CreatorOf* relation from a *user* node. This can be interpreted that the *user* is the creator of the *blog*. The same *user* has an outgoing *Followed* relation to a *term* node. This connection indicates that the user is interested in this specific *term* or maybe he is a blogger related to the topic which is defined by the *term*. The last node is again of type *blog* and has an outgoing relation to the same *term* that is connected to the user. The full meta path can be interpreted as two *blogs* being similar if the 2nd *blog* is flagged with the same *term* that the *user*, who created the first *blog*, follows. In other words, a blog $blog_B$ is considered similar to another one called $blog_A$, if the author (user) of $blog_A$ follows the same topic (term) to which $blog_B$ is also connected to.

4.4.4 Content-User-Term-Content CUTC

This meta path is again of length three and it is similar to the BUTB path. Instead of matching only *blogs* we are considering all *content* nodes, i.e. the selectivity is significantly higher as compared to BUTB and the result will be a superset of it. Moreover, the remark that recommending blogs is more meaningful as considering all content nodes is valid here as well. Nevertheless, we can hereby stress test the observed approaches.

Figure 17: CUTC

4.4.5 Blog-User-Term-User-Blog BUTUB

The meta path BUTUB is of length four and the longest one we will consider in the context of this thesis. Due to its length the encapsulated semantic is especially interesting. Hereby, more *hidden* connections between peer objects can be uncovered. The semantic interpretation of the short path BTB for example is simple and the similarity more obvious. Provided that the *terms* describe the content of the a blog quite good, it can be assumed that a similarity measure like TF-IDF (cf. section 3.1) leads to the similar or maybe even better results as BTB based similarity.

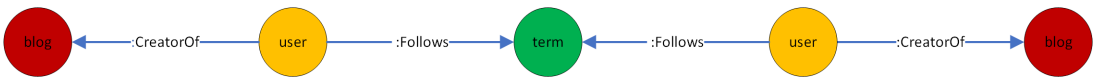


Figure 18: BUTUB

The interpretation of the first three nodes with its relations is the same as described in section 4.4.3 BUTB path. Since the last two relations have the same type and therefore the interpretation is equivalent. Thus, the whole meta path can be interpreted as two *blogs* are considered similar, if they were created by *users* who follow the same *term*. In other words, two *blogs* are similar if their *creators* are interested in the same topic (term). For example, $user_1$ created $blog_A$ and follows $term_T$. Additionally, $user_2$ created $blog_B$ and follows $term_T$ as well.

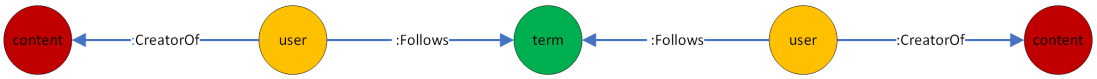


Figure 19: CUTUC

4.4.6 Content-User-Term-User-Content CUTUC

The meta path CUTUC is a derivation of the path BUTUB while omitting the constraint to only consider *blogs*. Thus, the interpretation is also similar to the one given in section 4.4.5 BUTUB. The semantical relevance is low for the same reason as discussed for the other *content* recommendation paths.

4.4.7 Meta Path Selectivity Approximation

We presented the selected meta paths in section 4.4.1 - 4.4.6 and additionally gave a semantic interpretation. In this section, we will approximate the path instance counts for the respective meta paths to be able to understand the implied complexity.

The following three relation types appear in the selected meta paths:

1. *CreatorOf*: This relation can exist between a *user* and a *content* node. Each *content* usually has only one incoming connection of this type, while the user can have multiple of such outgoing relations. A user creates on average 27 *content* items or 5 *blogs* respectively. This number only considers users having created at least one blog.

2. *Follows*: This connection can occur between a *user* following a *term* or a *content* item. The latter case is not relevant for the considered paths. Although, a user can follow multiple *terms*, the term node will usually exhibit a higher node degree than a *user* node, since many users can follow the same term and especially trending terms might be followed more often.

3. *TaggedWith*: The explanation here is similar to the one given for the *Follows* type, just that it exists between *content* and *term* nodes.

We observed that *term* nodes exhibit the highest degree nodes, and which is why we will approximate the path instance counts based on partial paths connected to *terms*. Hereby, we make use of the path concatenation property of meta paths (cf. section 2.5).

BTB & CTC

Both meta paths are symmetric in the *term* node, because the first relation and its start node have the same types as the second relation with its respective start node. The number of meta path instances per term can be calculated by squaring the number of *blogs* connected via a *TaggedWith* relationship to the considered *term* node. Hereby, we get all possible path combinations. We need to subtract this number by the number of aforementioned connections, if we do not want to have the path instance between a blog itself. Moreover, we need to divide this number by 2 in order to not count the mirrored result, like $blog_A, term_1, blog_B$ and $blog_B, term_1, blog_A$. Thus, the number of BTB path instances C_{BTB} per *term* can be calculated as:

$$C_{BTB} = \sum_{t_i \in T} \frac{c_{BT_{t_i}}^2 - c_{BT_{t_i}}}{2} \tag{10}$$

where $c_{BT_{t_i}}$ is the number of *TaggedWith* relationships per *term* i or the Blog-Term (BT) path counts connected to *term* t_i in other words. The square in equation 10 shows that the number of path instances grows significantly for high degree terms, i.e. when $c_{BT_{t_i}}$ is large. For example, if we have three terms, each having five BT connections, this will result in 30 distinct path instances. In contrast, if have the same amount of BT relations connected to just one term instead of equally distributed over three *terms*, then those 15 relations will lead to 105 instances. Thus, the distribution heavily impacts the resulting number of path instances. We will approximate the selectivity by looking at the the top 30 *terms* regarding the vertex degree, while only considering *TaggedWith* relationships coming from *blogs* or *content* nodes as required by the meta paths. Figure 20 shows the top 30 distribution of *terms* with BT paths. The horizontal axis identifies the *term* by its position in the ranking. This means the data point for $TermRank = 1$ is the *term* with most relations to *blogs*. Figure 21 visualizes the respective distribution of terms for the partial path CT, i.e. direct *content* connections. We can observe that most *blogs* or *content* nodes respectively are connected to the the respective top 5 terms. As a consequence, the approximation based on the top 30 will be relatively accurate. It should be noted that the *term* ranked at position one in distribution BT is not necessarily ranked at the same position in distribution CT. The ranked lists with the term labels can be found in appendix 9. Table 5 shows the top 5 when considering all content vertices and table 4 for *blogs* respectively. Those are the peak nodes resulting in the largest portion of path instances.

When calculating the instance counts based on the previously presented formula based on the top 30 terms, we get 4,762,581,338 BTB instances and 37,769,523,487 CTC instances.

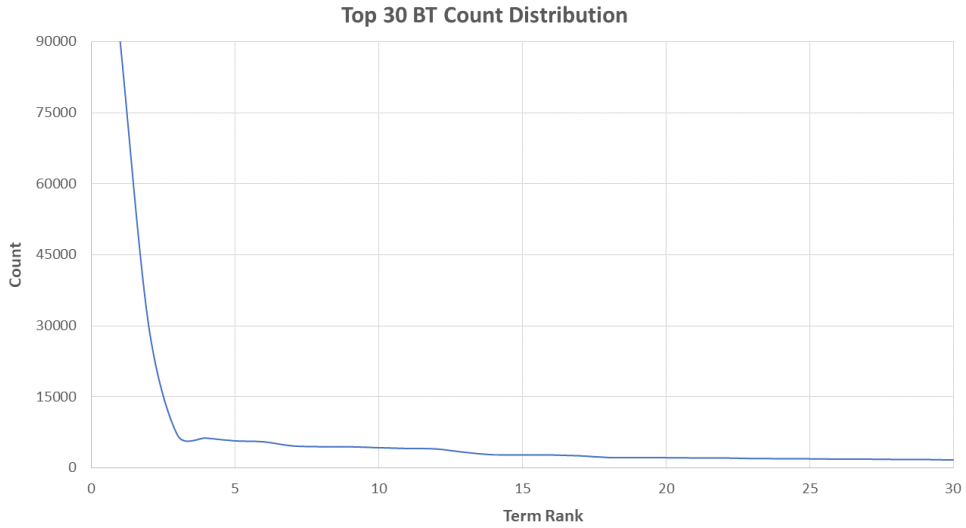


Figure 20: Top 30 BT Path Count Distribution

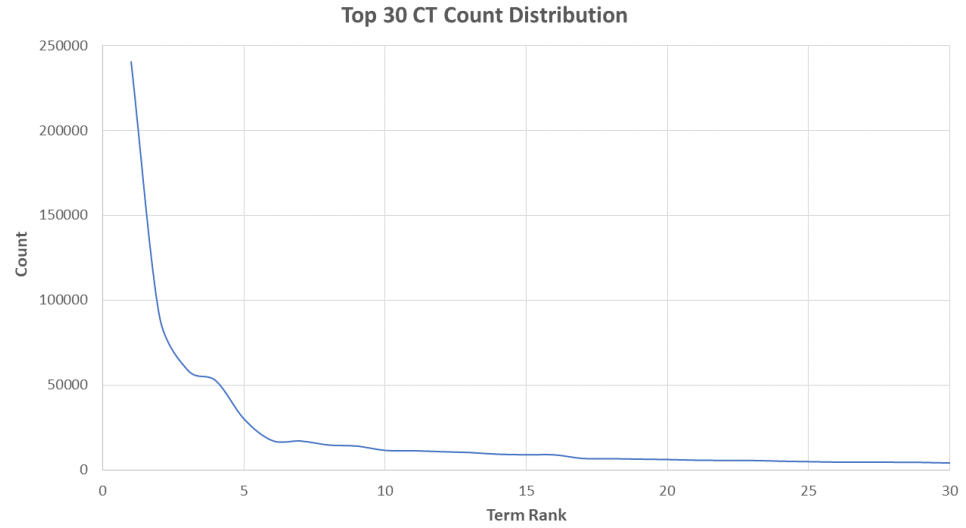


Figure 21: Top 30 CT Path Count Distribution

BUTUB & CUTUC

These meta paths are again symmetric in the *term*. Moreover, a *blog* has one *CreatorOf* relation to its creating *user* and a user can follow potentially many terms. Thus, again we will approximate the complexity of these meta paths by splitting the path in the *term*

Term	BT Count
English	90,919
Retagging Required	30,218
SAP	6,958
SAP HANA	6,269
HANA	5,658

Table 4: Top 5 Terms Tagged to Blog Nodes in the SAP Community Network

Term	CT Count
ABAP	240,472
English	90,923
Customer Relationship Management	59,091
Enterprise Resource Planning	52,756
Retagging Required	30,218

Table 5: Top 5 Terms Tagged to Content Nodes in the SAP Community Network

node into the partial paths BUT and TUB and concatenate the results in T. Since BUT implies the same path instances as TUB, we can again calculate the path instance count C_{BUTUB} for the meta path BUTUB:

$$C_{BUTUB} = \sum_{t_i \in T} \frac{c^2_{BUT_{t_i}} - c_{BUT_{t_i}}}{2} \tag{11}$$

where $c_{BUT_{t_i}}$ is the path instance count of BUT in *term* t_i. The same explanation is valid for the CUTUC case. The path concatenation for BUTUB and CUTUC is explained in detail in section 6.2.4, because we make use of this property in the distributed graph processing implementation as well. Next, we extract the terms that have the most path instances of the form Blog-User-Term (BUT) connected. In other words, we look at the *terms* having the most indirect connections to *blogs* via the *CreatorOf* relation to the *user*, who himself follows the respective term. Figure 22 shows the top 30 term distribution based on the connected BUT instances. The distribution for the CUT case is visualized in figure Figure 23 and the full top 30 ranking tables with the mapping from *Term Rank* to the corresponding term value can be found in appendix 9. The shape of the distributions looks similar to the BT / CT distributions. But the tail of the distributions is longer, which means that the approximation will not be as accurate as the other ones. Nevertheless, this is sufficiently good for our selectivity estimation.

Consequently, we approximate 15,093,281 BUTUB path instances and 91,717,224,313 CUTUC path instances,

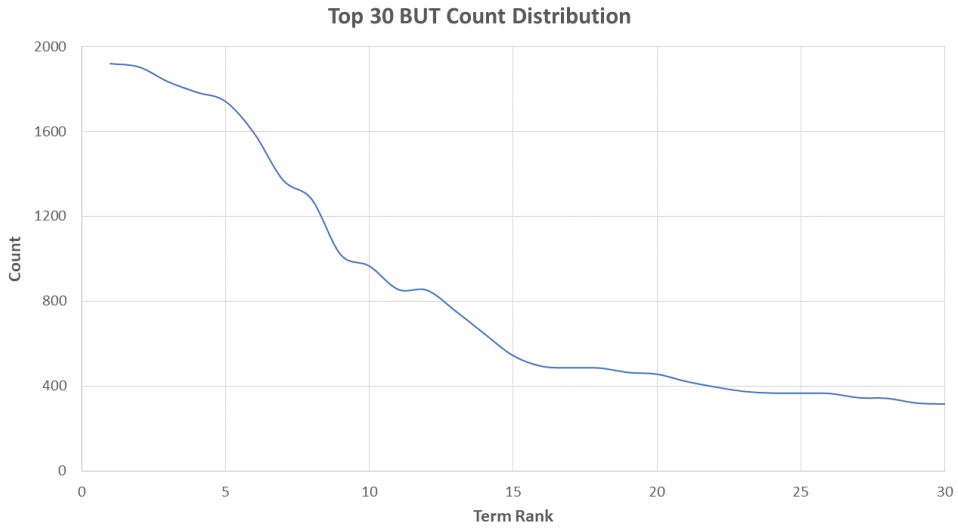


Figure 22: Top 30 BUT Path Count Distribution

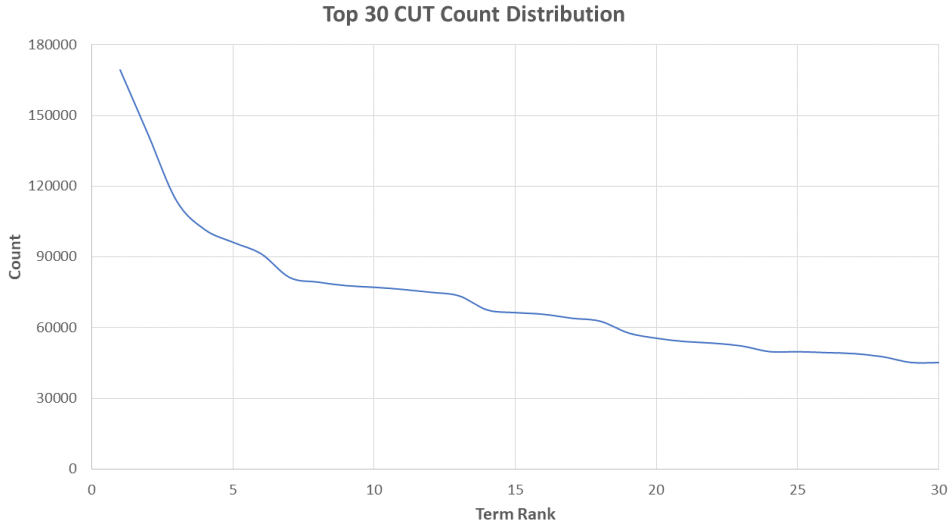


Figure 23: Top 30 CUT Path Count Distribution

BUTB & CUTC

The previously regarded meta paths were all symmetric in the *term* node. This is not the case for BUTB and CUTC, but the *term* is again included in the paths and as we already

Term	BUT Path Count	BT Count
SAP BusinessObjects Cloud	1,922	148
SAP BusinessObjects Business Intelligence platform	1,906	1,724
SAP BusinessObjects Design Studio	1,837	944
SBOP Web/Desktop Intelligence (client/server)	1,787	480
SAP BusinessObjects Lumira	1,744	1,396

Table 6: Top 5 Terms with Blog-User-Term Relationship in the SAP Community Network

Term	CUT Path Count	CT Count
Using SAP.com	169,244	2,841
ABAP Development	141,591	11,793
MM (Materials Management)	113,666	4,264
SAP BusinessObjects Business Intelligence platform	101,500	3.198
Coffee Corner	96,137	43

Table 7: Top 5 Terms with Content-User-Term Relationship in the SAP Community Network

stated these nodes exhibit the highest node degree. Hence, we will again approximate the selectivity in a similar way as for the other meta paths. Therefor we look again at the top 30 terms regarding the BUT - and CUT connections respectively. Additionally, we calculate the BT path instances for the same *terms* considered in the top 30 BUT and CUT distribution. Those values are listed in detail in appendix 9 and exemplary shown in table 6 and 7. The Blog-Term count (BT count) column gives additionally the number of the directly connected *blogs* via a *TaggedWith* relationship to the respective terms and the Content-Term count (CT count) column analogously for the *content* recommendation case. Here, we make use of the meta path concatenation property again. The meta path concatenation for BUTB and CUTC is explained in detail in section 6.2.4 too. The total number of BUTB path instances C_{BUTB} can be upper bounded by the Cartesian product between the number of BUT paths instances $c_{BUT_{t_i}}$ including *term* t_i and the count of directly connected *blogs* $c_{BT_{t_i}}$ to t_i, which is:

$$C_{BUTB} = \sum_{t_i \in T} C_{BUT_{t_i}} * C_{BT_{t_i}} \tag{12}$$

The reason why we have an upper bound in this case is that a $blog_A$ which is *TaggedWith* t_i could be possibly created by a *user* who himself *Follows* the *term* t_i. The same calculation

and explanation is valid for CUTC. Following this approach we get an approximation of 30,934,318 BUTB instances and 6,946,928,797 CUTC path instances.

We classified the meta paths according to the selectivity as *normal*, *high* and *very high* based on the approximated path instance counts. Moreover, the length defined as the number of relations per path and the semantic relevance is given. Table 8 summarizes the generated meta paths and provides a comprehensive overview of such. The recommendation quality for the content meta paths, i.e. CTC, CUTC, CUTUC, will most likely not be very good, because it is not meaningful to recommend *comments* or *answers*. But those entities constitute the biggest fraction of the *content* nodes in the SAP Community network (cf. table 3) with approx. 80%, while blogs represent nearly 1% of the content items. For example, a user will not be interested in getting a *comment* recommended, when reading a *blog* post. In contrast, blog recommendations are very useful, since a user will probably be interested in similar blog posts to read. As already stated, we are not directly interested in the quality of the recommendation result, but instead in evaluating the scalability and the limits of the evaluated approaches. Thus, by including the content recommendations in our evaluation we are able to put very high workloads on the respective systems which can be seen as stress tests. The actual paths of interest for use in the recommendation engine are the blog recommendation ones, namely BTB, BUTB, BUTUB. BUTB and BUTUB are of special interest, because of the more complex structure as compared to BTB.

Meta Path	**Selectivity**	**Approximation**	**Length**	**Semantic Relevance**
BUTUB	normal	15,093,281	4	strong
BUTB		33,346,280	3	strong
BTB	high	4,762,581,338	2	normal
CUTC		6,946,928,797	3	low
CTC	very high	37,769,523,487	2	low
CUTUC		91,717,224,313	4	low

Table 8: Meta Paths Overview

Although, we need to aggregate the path instances for all paths between the start - and end vertices for solving the PCCP (cf. section 4.1), we will need to enumerate all paths beforehand. Thus, in order to understand the complexity, we need to look at the actual path instance counts as we did here. We observe path instance approximations of several billions for all content recommendations as well as for the BTB meta paths. These numbers are very high when considering the total size of the SAP Community network with roughly 30 million relations.

5 Path Count Computation with Neo4j

The SAP Community graph is stored in a *Neo4j* graph DB instance. Neo4j's implementation is conceptually realized as a native graph storage (cf. section 3.3.2) and a native graph processing engine (cf. section 3.3.1). We will analyze the performance behavior when solving the PCCP using the native query capabilities of Neo4j, i.e. we will use the *Cyper* query language. We are interested in evaluating the general applicability of Neo4j for the computation of meta paths 4.4. Therefore, we are especially interested in the runtimes of the different queries. An important benefit of this approach is that we do not need to create a lot of code, because the database engine will take care about the actual execution strategy. In contrast, the parallel graph processing approach (cf. section 6) requires the implementation of a dedicated algorithm. Hence, the desirable property of the graph DB approach is its simplicity an the advertised efficiency for graph queries.

5.1 Approach

First, we will formulate Cypher queries for all considered meta paths (cf. section 4.4). Moreover, we will analyze the generated query plans for the respective Cypher queries to better understand Neo4j's execution strategy. Next, we will evaluate the runtimes and the behavior when putting high workload onto the system through executing Cypher statements for high selective meta paths. Therefore, we will setup a dedicated Neo4j instance using a snapshot of the SAP Community Network. Moreover, we are interested in the effects implied by the length and selectivity of a meta path. Thus, we will compare the results of the different paths with each other in order to evaluate the impact.

5.2 Implementation

We will present and explain the Cypher statements for all considered meta paths in this section. Furthermore, we sate the resulting query plans to be able to understand how Neo4j translates the Cypher queries into Neo4j operators.

BTB & CTC

Figure 24 shows the Cypher query for solving the PCCP for the meta path BTB. In other words, it computes all paths and aggregates the results based on the start and end nodes.

Figure 26 gives a visual representation of the resulting execution plan for the respective Cypher statement. Neo4j builds up two similar sets. Therefore, the *term* nodes will be

```
MATCH (blogA:content)-[rel1:TaggedWith]->(aTerm:term)<-[rel2:TaggedWith]-(blogB:content)
    WHERE blogA.entitySubType= 'blog'  AND blogB.entitySubType = 'blog'
    RETURN ID(blogA), ID(blogB), COUNT(*);
```

Figure 24: Cypher Query - BTB

```
MATCH (blogA:content)-[rel1:TaggedWith]->(aTerm:term)<-[rel2:TaggedWith]-(blogB:content)
    RETURN ID(blogA), ID(blogB), COUNT(*);
```

Figure 25: Cypher Query - CTC

retrieved first as anchor nodes. Next, the *ExpandAll* operator is applied on these nodes and only the ones matching the required meta path are kept by applying the necessary filter critera. This means, that all incoming relationships of type *TaggedWith* per anchor node will be iterated and the ones which are not having the *TaggedWith* type will be disregarded. Moreover, the start node of the relationship will be evaluated to be of type *content* and Subtype *blog*. At this point we have two identical sets of *terms* with all its connected *blogs*, i.e. we have enumerated the BT instances. Now, these sets are combined via the NodeHashJoin operator on the *term* node. Hereby, we get the full set of path instances follwoing the meta path BTB. Next, Neo4j filters out the path instances where both *blogs* are the same. It should be noted that we do not need to formulate this is in the Cypher statement explicitly, since Neo4j enforces uniqueness of the edges per default while matching patterns [37]. Finally, the path instances are grouped by the ID_{aBlog} and ID_{bBlog} and summed up via the *EagerAggregation* operator. Hereby, we get the path count between both *blog* nodes is calculated.

The Cypher statement for the meta path CTC is shown in figure 25. It is actually the same without constraining the query to *blogs* and hence considering all *content* nodes. The query execution plan is also similar to the BTB one and just does not enforce the filtering on the SubType. Therefore, we omit to display the repsective execution plan again.

BUTB & CUTC

Figure 27 shows the cypher statement for the meta path BUTB and figure 28 for the path CUTC. Additionally, the query execution plan for BUTB is given in table 9. The execution plan for CUTC is again similar to the BUTB one just without the extra filter for *blogs*.

First, all *term* nodes are selected as anchor points. Now, the meta path is inflated in the left direction relative to the anchor nodes. Therefore, all relations are expanded and only the relations of type *Follows* starting from a *user* node are kept. Again the *ExpandAll*

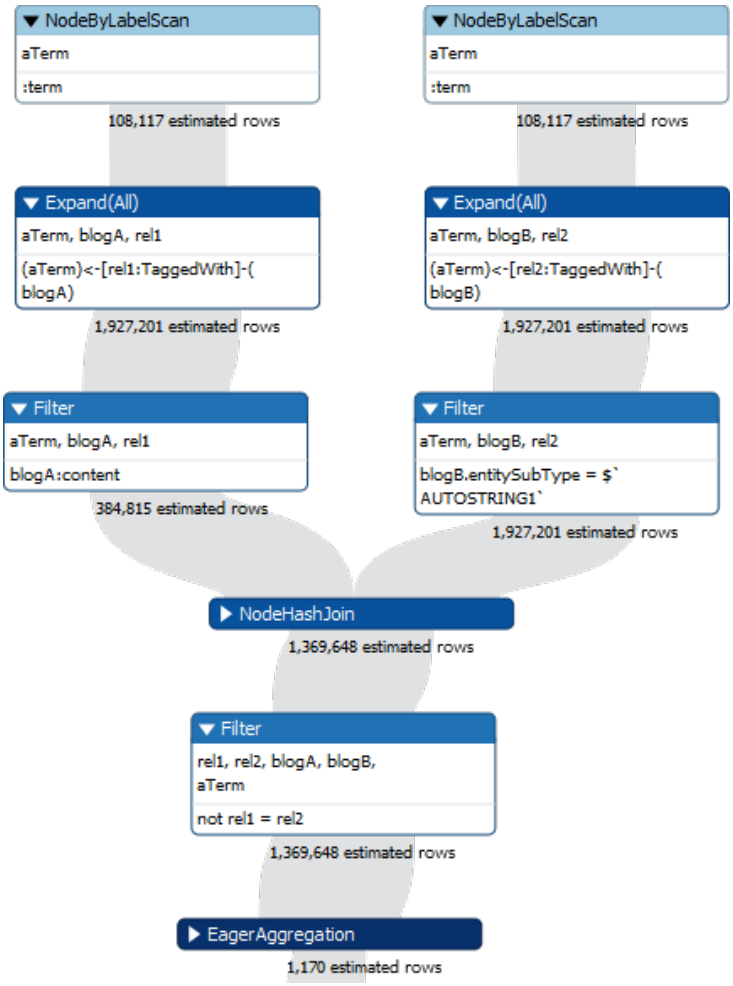


Figure 26: Neo4j Query Execution Plan - BTB

```
MATCH (aBlog:content)<-[rel1:CreatorOf]-(aUser:user)-[rel2:Follows]->(aTerm:term)<-[rel3:TaggedWith]-(bBlog:content)
    WHERE bBlog.entitySubType= 'blog' AND aBlog.entitySubType = 'blog' AND ID(aBlog) <> ID(bBlog)
    RETURN ID(aBlog), ID(bBlog), COUNT(*);
```

Figure 27: Cypher Query - BUTB

```
MATCH (aBlog:content)<-[rel1:CreatorOf]-(aUser:user)-[rel2:Follows]->(aTerm:term)<-[rel3:TaggedWith]-(bBlog:content)
    WHERE  ID(aBlog) <> ID(bBlog)
    RETURN ID(aBlog), ID(bBlog), COUNT(*);
```

Figure 28: Cypher Query - CUTC

and *Filter* operators are used for inflating the first relationship of the meta path including the starting nodes. In other words, we built the set of BUT path instances. Consequently,

the last edge of the meta path is expanded, i.e. all ingoing *TaggedWith* relations coming from *blogs* to the *terms* are inflated, whereby the full paths are generated. Finally, the *EagerAggregation* operator calculates the final path counts between all pairs of *blogs* or *content* nodes respectively.

#	Operator	Identifiers	Other
1	NodeByLabelScan	aTerm	:term
2	Expand(All)	aTerm, aUser, rel2	(aTerm)<-[rel2:Follows]-(aUser)
3	Filter	aTerm, aUser, rel2	aUser:user
4	Expand(All)	rel1, aBlog, rel2, aUser, aTerm	(aUser)-[rel1:CreatorOf]->(aBlog)
5	Filter	rel1, aBlog, rel2, aUser, aTerm	aBlog:content
6	Expand(All)	bBlog, rel1, aBlog, rel2, aUser, rel3, aTerm	(aTerm)<-[rel3:TaggedWith]-(bBlog)
7	Filter	bBlog, rel1, aBlog, rel2, aUser, rel3, aTerm	bBlog:content
8	EagerAggregation	COUNT(*), ID(aBlog), ID(bBlog)	ID(aBlog), ID(bBlog)

Table 9: Neo4j Query Execution Plan - BUTB

BUTUB & CUTUC

The Cypher statements needed for solving the PCCP for the meta paths BUTUB and CUTUC are visualized in figure 29 and 30. These statement need two *MATCH* expressions instead of one. The reason is that Cypher ensures that an relationship won't occur twice in the same path instance while pattern matching [37]. This feature is not desired for the *CreatorOf* relationship in our case. For example, if we have two different *blogs* that were created by the same *user*, who itself is following a specific term, then we want to match these path instance too. But the *Uniqueness* constraint of Cypher would filter out paths using the same *Follows* relationship multiple times. Therefore, we need to use two *MATCH* statements and enforce the desired uniqueness constraint manually only for ensuring that the *blogs* matched are not identical using the *WHERE* clause of the statement, i.e. *aBlog* must not be *bBLog*.

```
MATCH (aBlog:content) <-[rel1:CreatorOf]- (aUser:user) -[rel2:Follows]-> (aTerm:term)
MATCH (bBlog:content) <-[rel4:CreatorOf]- (bUser:user) -[rel3:Follows]-> (aTerm)
    WHERE bBlog.entitySubType= 'blog' AND aBlog.entitySubType = 'blog' AND ID(aBlog) <> ID(bBlog)
    RETURN ID(aBlog), ID(bBlog), COUNT(*);
```

Figure 29: Cypher Query - BUTUB

The query execution plan for BUTUB is given in table 10. The plan starts again by selecting all *term* nodes as anchor nodes via a *NodeByLabelScan*. As already stated, the meta path is symmetric in the term node. From here, all direct neighbors are expanded and filtered first, i.e. both relations of type *Follows* with its *user* nodes are selected. Starting from the retrieved *user* nodes all relations are expanded once again and only the ones of type *CreatorOf* pointing to *blog* nodes are kept. Hereby, Neo4j has all path

```
MATCH (aBlog:content) <-[rel1:CreatorOf]- (aUser:user) -[rel2:Follows]-> (aTerm:term)
MATCH (bBlog:content) <-[rel4:CreatorOf]- (bUser:user) -[rel3:Follows]-> (aTerm)
    WHERE ID(aBlog) <> ID(bBlog)
    RETURN ID(aBlog), ID(bBlog), COUNT(*);
```

Figure 30: Cypher Query - CUTUC

instances available. Finally, the aggregation is executed for counting all path instances between each pair of *blogs* or *content* nodes respectively.

#	Operator	Identifiers	Other
1	NodeByLabelScan	aTerm	:term
2	Expand(All)	aTerm, aUser, rel2	(aTerm)<-[rel2:Follows]-(aUser)
3	Filter	aTerm, aUser, rel2	aUser:user
4	Expand(All)	bUser, rel2, aUser, rel3, aTerm	(aTerm)<-[rel3:Follows]-(bUser)
5	Filter	bUser, rel2, aUser, rel3, aTerm	bUser:user
6	Expand(All)	rel1, bUser, aBlog, rel2, aUser, rel3, aTerm	(aUser)-[rel1:CreatorOf]->(aBlog)
7	Filter	rel1, bUser, aBlog, rel2, aUser, rel3, aTerm	aBlog.entitySubType = 'blog'
8	Expand(All)	bBlog, rel1, rel4, bUser, aBlog, rel2, aUser, rel3, aTerm	(bUser)-[rel4:CreatorOf]->(bBlog)
9	Filter	bBlog, rel1, rel4, bUser, aBlog, rel2, aUser, rel3, aTerm	bBlog:content
10	EagerAggregation	ID(aBlog), ID(bBlog), count(*)	ID(aBlog), ID(bBlog)

Table 10: Neo4j Query Execution Plan - BUTUB

5.3 Evaluation

We used Neo4j in version 3.4 for the evaluation of the path count computation capabilities of the graph database. The graph database was setup with 170GB heap space on a cloud server with 240GB memory in total and 16 CPU cores with 2.1GHz each. Moreover, we exclusively executed one query at a time to be able to get comparable runtimes. Next, we will discuss the observations in detail.

The path counts for the meta path BUTUB could be successfully computed in 2h38m. The result set consisted of 16,143,814 rows. The computation of all path instances without the aggregation took 2h31m and produced 33,726,074 rows, i.e. path instances. The result contained mirrored results. This means if the result contains the BUTUB path count c between start node $blog_A$ and end node $blog_B$, then the result set will always contain the mirrored result where $blog_b$ is the start node and $blog_A$ the end node with the same path count value c. The reason for this behavior is that the meta path is symmetric and we can not put further constraints on the query without unintentionally excluding further path instances. Anyways, it is trivial to remove the duplicates in a postprocessing step if needed. Consequently, we need to divide the previously given numbers by 2 in

order to get the actual number of path instances, which is 16,863,037 (cf. equation 11). Accordingly, we observed that there are 8,071,907 distinct pairs of *blogs* in the SAP Community dataset, which are connected by at least one path instance following the meta path BUTUB. Furthermore, Neo4j was capable to compute all distinct 38,294,760 path counts for the meta path BUTB in 19m. The query does not contain mirrored results since it is not symmetric in any way an therefore no postprocessing is necessary in this case. In other words, we found out that there are over 38M pairs of blogs having at least one path instance following the meta path BUTB in the graph. The computation of all 39,902,383 distinct path instances took only 13m.

The results of the query executions for the meta path BTB are documented in table 11. The direct execution of the query (cf. figure 24) was not possible since the memory requirements were to high. In order to address this problem, we first tried to execute a similar query but without the aggregation. In other words, we tried to enumerate all path instances without computing the final path counts. By omitting the aggregation, it is generally possible to output the results while still processing further paths and thus reducing the memory requirements. The other way round, when using the aggregation all path instances must be computed first to be able to finally aggregate these values. The idea was to evaluate whether Neo4j uses such an optimization. It was still not possible to execute this query completely. We manually aborted the query execution since Neo4j was unresponsive due to JVM garbage collection stalls, because the system did not produced intermediate results. Moreover, we tested whether it is possible to execute the results in batches by using Cypher's *SKIP* and *LIMIT* statements. We tried various batch sizes and could determine that Neo4j could handle batch sizes of 600M. When using larger batch sizes Neo4j got stuck in Garbage Collection stalls (cf. section 9). Moreover, we observed that Neo4j ran into an endless loop when executing the BTB query in batches. Interestingly, batching was possible when for the BUTB and BUTUB queries. A prerequisite for batching is a that Neo4j produces the results in the same order. This constraint is not necessarily fulfilled, because Neo4j does not guarantee the order unless the result set is sorted [38]. But sorting the result set implies that the full result set needs to be available first. Thus, we assume that Neo4j can only reliably determine the order of the result set under certain conditions. We assume that a too low heap size influences the reliable batch execution on the one hand and the query execution plan on the other. The BTB execution plan (cf. figure 26) involves a hash join. This may be already a potential problem. Further research is needed to validate this statement. For answering our research question, it is sufficient to understand that the execution order is not guaranteed. In table 4 we showed the top 5 peak terms leading to the largest portion of BTB instances. The first two terms are semantically meaningless. In a production setting those *terms* need to be removed, because the recommendation result will not be useful or might even distort the recommendations. Therefore, we executed the BTB query while excluding the top 2 terms. Hereby, we could succesfully compute all remaining

481,747,176 BTB path instances in 1h39m (cf. equation 10). Thus, we demonstrated exemplary the impact of exploring and cleaning the data set in advance. The matched amount of instances supports our assumption that Neo4j can reliably compute up to 600M path instances in the observed setting.

#	Batch Size	Aggregation	Runtime	Observation	Remark
1	-	Yes	-	java.lang.OutOfMemoryError	GC Overhead Limit Exceeded
2	-	No	-	Garbage Collection stall	Manually aborted
3	5M	No	3d4h	860M instances	Manually aborted
4	50M	No	20h	1.25B instances	Manually aborted
5	1B	No	-	Garbage Collection stall	Manually aborted
6	800M	No	-	Garbage Collection stall	Manually aborted
7	600M	No	6d	26B instances	Manually aborted
8	-	No	1h39m	481,747,176 instances	2 Terms excluded
9	-	Yes	-	Garbage Collection stall	2 Terms excluded

Table 11: Neo4j Results for BTB

The result set of the meta path BTB is already the largest one of the considered paths used for recommending blogs. All content related paths have a higher selectivity and could therefore not be executed in this setting. In other words, Neo4j was not capable to directly compute the path counts for the high - and very high selective paths.

Additionally, we observe that the runtimes for the meta path BUTUB is considerably higher as for the meta path BUTB. The actual amount of path instances is one cost factor, since we need to iterate all final path instances at least once. But the final amount of BUTUB instances is slightly lower than the number of BUTB instances and it follows that this can not be the main influencing factor in this scenario. We can not reliably determine the actual cause, because we did not found further publications about the concrete implementation of Neo4j. Thus, we give a possible explanation based on the query execution plans. We assume that Neo4j builds intermediate result sets after the execution of each operator and the *ExpandAll* operation gets executed iteratively on each row. This would imply, that the *ExpandAll* operator will be executed proportionally often to the number of occurrences of the respective node, if the intermediate result contains a node multiple times which needs to be expanded further. Considering the graph given in figure 12 and the query execution plan (cf. table 10) for meta path CUTUC. To simplify the example we assume that term $t1$ does not exist in the graph. After executing the first operator the result set will contain one row with just the term t_2. Thus, the ExpandAll (Operator #2) will only be executed once. Now, the result set contains two instances, since there are two user connected to t_2. The next *ExpandAll* (Operator #4) again scans all ingoing *Follows* relations starting for the term t_2. Since t_2 is contained in both rows of the intermediate result the same relations might be called twice and hence the cost is doubled. Depending on the meta path and the distribution of the relations this will

lead to a multiple of the processing time of the previous operator. The path instance computation for a meta path BUTU took just 21 minutes while the execution of BUTUB took already 2h31m.

Meta Path	Instance Count	Runtime
BUT	53,097	32s
BT	672,239	17s
CT	1,927,201	27s
CUT	6,024,590	1m24s

Table 12: Neo4j Results for Partial Paths

Therefore, we split all considered meta path in the term node. Thus we got the partial paths BT, BUT, CT, and CUT. The results are presented in table 12. CUT could be executed in roughly 1.5m and all other paths below 33s. The meta path BUT could be computed in 32s, BUTU in 21m and the full path BUTUB in 2h31m.

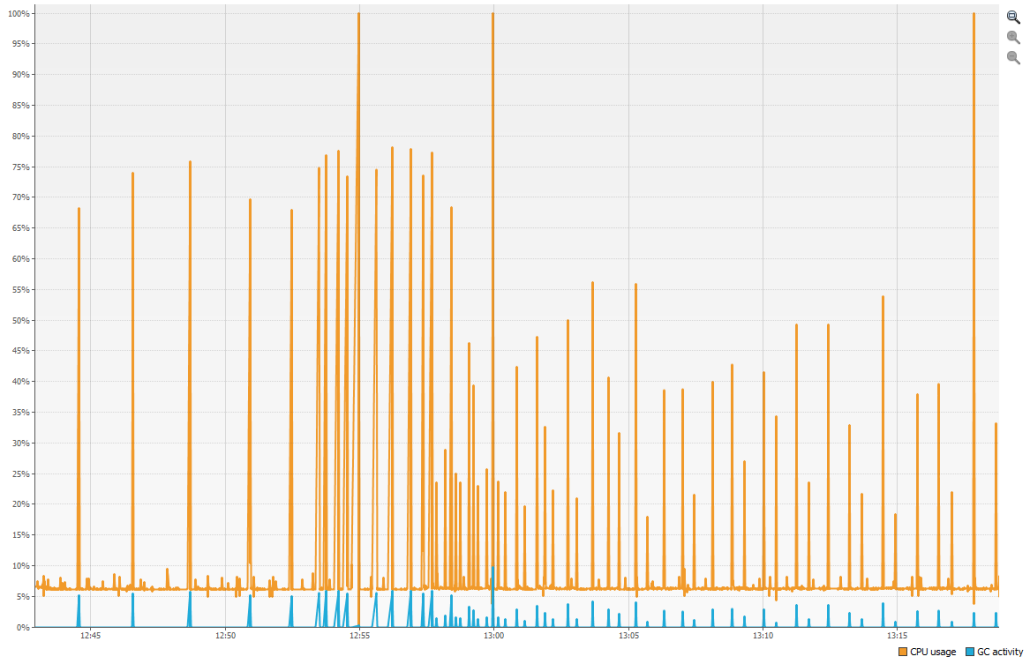


Figure 31: CPU utilization during BUTUB Cypher query execution

Furthermore, we could determine that the query execution just used a single CPU core by monitoring the CPU utilization. Only the JVM garbage collection used multiple cores. This shows that the query execution using is efficient in terms of resource consumption.

In contrast, it does not scale very good. We can increase the available memory to be able to compute larger result sets but we can not distribute the workload to multiple cores in this approach. Figure 31 shows exemplary a snapshot of the CPU utilization during the execution of the Cypher query for BUTUB. 100% CPU utilization implies the full usage of all 16 cores in this setting. The peaks in the blue line indicate garbage collection activity. At the same time of garbage collection the CPU utilization significantly increases, while it is around 6-8% in the meantime. This percentage value corresponds roughly to one CPU core.

5.4 Interim Conclusion

We successfully executed the meta paths with a normal selectivity, while the high - and very high selective paths were not computable, since the memory requirements were too high. Thus, we identified the JVM heap size as the main limiting factor. Table 13 provides a comprehensive overview about the experimental results.

Meta Path	Aggregation	Result Size	Runtime	Remark
BUTUB	Yes	16,143,814 rows	2h38m	-
BUTUB	No	33,726,074 rows	2h31m	-
BUTB	Yes	38,294,760 rows	19m	-
BUTB	No	39,902,383 rows	13m	-
BTB*	No	481,747,176 rows	1h39m	w/o Top 2 Terms
BTB	-	-	-	Not executable
CUTC	-	-	-	Not executable
CTC	-	-	-	Not executable
CUTUC	-	-	-	Not executable

Table 13: Neo4j Results Summary

Moreover, we demonstrated the impact of data exploration and data cleaning exemplary at the BTB case. Hereby, we could drastically remove the total number of path instances and demonstrated that Neo4j can handle up to 600M path instances in the evaluated setting. Thus, we could compute the BTB path instances in a practical relevant way. Furthermore, we showed that partial meta paths can be computed very efficiently, as long as the *term* is either the start - or end node of the meta paths.

6 Path Count Computation with Parallel Graph Processing in Flink

The pregel-like programming model (cf. section 3.5) is based on the idea to iteratively execute a UDF on each vertex in parallel and communicate the results to the respective receiver vertices by message passing. The messages are guaranteed to be available at the beginning of the next superstep. It is non-trivial to design algorithms according to this paradigm. At the time of writing there exists only one publication proposing an approach for analyzing a heterogeneous graph with respect to meta paths. We presented the *parallel meta-graph extraction* framework in section 3.6 and with it the homogeneous graph extraction problem (cf. section 3.6.1). The main concepts are the *path enumeration* and the *two-level aggregate model*. Path enumeration is actually the same task as finding all path instances of a meta path P in a heterogeneous graph. This is also necessary for the PCCP as explained in section 4.1. Moreover, the two-level aggregate model is based on two user-defined functions $\oplus$ and $\otimes$. The aggregation model (cf. section 3.6.2) calculates the edge value $val(u, v)$ in the extracted homogeneous graph by first calculating an aggregated value per path instance $p \in P_{uv}$ based on each individual edge value of the path instance, whereas this value is defined as $val(p_i) = \forall_{p_i \in P_{uv}} \otimes w(e_i)$. In other words, the path instance value is calculated by applying the aggregate function $\otimes$ to each edge value $e_i \in p_i$. The final value val(u,v) is then $val(u, v) = \forall_{p_i \in P_{uv}} \oplus val(p_i)$, i.e. the function $\oplus$ is applied to all path instance values between u and v.
We choose $\otimes$ such that it always returns *1*, independently of the input value. As a consequence $val(p_i) = 1$ for $\forall_{p_i \in P_{uv}}$. Moreover, we elect $\oplus$ to be the summation of its input values. It follows, $val(u, v) = \forall_{p_i \in P_{uv}} \oplus val(p_i) = \sum_i 1 = c_{uv}$, where c_{uv} denotes the path count between u and v. Hereby, we reduced the homogeneous graph extraction problem to the PCCP and showed that we can create an implementation in the style of the framework proposed by Shao et al. for solving the PCCP.

6.1 Approach

We create a distributed graph processing application based on the concepts of the *meta-graph extraction* framework presented in section 3.6. The framework is presented in the *Fast Parallel Path Concatenation for Graph Extraction* paper published 2017 by Shao et al. [51]. Moreover, we manually derive PCPs (cf. section 3.6.5) for all considered meta paths (cf. section 4.4).

We choose Apache Flink as a state-of-the-art big data processing system comprising the Graph API called Gelly. Gelly provides API methods for executing iterative graph com-

putations in the vertex-centric model (cf. section 3.5). The reason why we chose Flink instead of a specialized system for distributed graph processing like Pregel is that we also need to pre- and post-process the dataset. Since Flink natively supports general data transformations, we do not need a further system for these tasks. Distributed processing systems need to be configured individually for different use cases. Hence, we evaluate a meaningful system configuration first.

Next, we measure the absolute runtimes in wall time for all meta paths. This means, we take several individual measurements for each meta path using different hardware configurations and varying the Flink cluster size. In other words, we observe different hardware configurations and scale each configuration up to multiple Flink Taskmanagers. The maximum number of used worker nodes depends on the hardware specifications of the used machines. We execute the Flink application twice in each setting for each meta path. The average runtime is then calculated and used for the evaluation. For example, we execute the Flink job twice with one Taskmanager of type *Config A* for meta path BUTUB and calculate the average runtimes based on the two runs. Afterwards, we scale the number of Taskmanagers and calculate the average runtimes repeatedly as explained before. By using an average value we aim to circumvent possible outlier measurement values and thus getting a more representative result. These results will be compared to the Neo4j results later on in section 8.

The scalability evaluation will be done based on the approach presented in section 2.8. Therefore, we calculate the *Speedup* and the *Efficiency* based on the measured runtimes in order to evaluate the strong scalability. Since we are dealing with a fixed size graph data set it is not directly possible to evaluate the weak scalability. The main reason is that we can not vary the dataset size without possibly changing the graph structure significantly. Nevertheless, we aim to get similar insights by considering high selective meta paths as when actually scaling up the graph size.

6.2 Implementation

In this section, we present the implementation details of our *meta-graph extraction* prototype implementation. Therefore, we highlight the conceptual differences of our implementation and the one proposed by Shao et al. first. Next, we discuss the Gelly graph representation as used in our implementation. Additionally, we present the system architecture of the complete graph analysis pipeline. Finally, we present and discuss the PCPs derived for the considered meta paths.

6.2.1 Conceptual Differentiation to the Parallel Graph Extraction Framework

The parallel graph extraction framework is a generic concept for extracting homogeneous graphs from a heterogeneous graph based on meta paths. Generic in a sense that a user specified meta path is compiled by the framework to a PCP. Although, such a functionality is desirable from a user perspective in a productive setting, we manually derive one PCP (cf. section 6.2.4) for each considered meta path (cf. section 4.4). Moreover, we used an extensible design for the PCP generation module, such that it is possible to easily implement new manually derived PCPs.

6.2.2 Graph Representation

A graph in Flink is represented as a dataset of vertices and a dataset of edges. A vertex is specified by an ID and a generic vertex value, whereas an edge is specified by the ID of the start - and end vertex plus a generic edge value as well. We need to model the graph such that the SAP Community property graph data, i.e. the semantic information, is encapsulated in the generic values and additionally the metadata used for the execution of the analysis algorithm.

Vertex Value

Two fields are used to capture all relevant semantic information for a vertex. This is the vertex *label* on the one hand, and a vertex *subtype* on the other. We modeled the subtype as a generic field used to be able to capture more information as the vertex label. Currently, we make use of it to further distinguish between content vertices (cf. section 4.3). Nevertheless, the field can potentially be used more generically in the future and is not restricted to content vertices only.
The implemented algorithm adheres to the vertex-centric model (cf. appendix 3.5). It is based on the idea to find instances of a primitive pattern in the graph by evaluating vertex labels and communicating the results to neighbor vertices. But the results must only be send to those neighbors matching the structure given by the Primitive Patterns, which means where the object types match the one specified in the primitive pattern. In Flink, we only have the outgoing edges without the target vertex type at hand during the execution of the vertex UDF. Thus, we additionally need to materialize the information about all ingoing edges, i.e. the edge type and the vertex labels and subtypes of the source vertices, at each vertex. Hereby, we are able to evaluate in a vertex-centric-manner whether an edge matches the respective edge of the primitive pattern or the meta path respectively. Furthermore, we need to be able store the final analysis result. Since we only have the generic vertex value available, we need to consider this when modeling the vertex value.

Edge Value

The edge value has a simpler structure as the vertex one, because we only have to store the edge type as semantic information. Moreover, there is less algorithm specific metadata to be stored at the edges as compared to the vertex value. We explained the need to have all edges incl. the vertex label and the subtype at hand during the execution of the vertex update UDF. The ingoing edges to a vertex are materialized at the target vertex itself as described above. Access to the outgoing edges including the edge type is provided by the framework itself. But we also need the target vertex label and the subtype to be able to evaluate the primitive pattern for outgoing edges. Therefore, we include the vertex label and the vertex subtype in the edge value. Alternatively, We could have persisted this data at each vertex too. Since the data is semantically closer to the edge than to the source vertex, we decided to store the data in the edge value.

All IDs, labels, types and thus all semantically relevant data is mapped to integer representations. Initially, we started by using strings, because the respective data in Neo4j is represented as strings. Hereby, we were able to significantly reduce the memory consumption since an integer in Java takes 4 bytes in total as compared to 2 bytes for a single character in a string.

6.2.3 Architectural Overview

The system architecture and hence the complete graph analysis pipeline is illustrated in figure 32. There is one *Neo4j v.3.4.8 Community Edition* graph database instance (cf. 3.3) which is installed on one dedicated server. The *SC graph* (cf. section 4.3) is stored as a property graph within this instance. Several physical machines were connected via network as a cluster. The HDFS version 2.8.3 (cf. 9) was installed on top of this physical layer. Moreover, the distributed data processing system *Apache Flink v.1.4* (cf. 3.4) is also installed on the cluster hardware. In other words, we have machines acting as Flink - and as HDFS master nodes in the cluster as well as servers acting as slave nodes of the respective systems.

The actual programmed application is a Flink program. This application executes two different jobs. The first job (cf. figure 33) handles the initial data loading and needs to be executed every time we want to have the latest update of the graph data. We executed this job only once in the context of this experiment, because we need to have an immutable graph dataset to be able to get comparable runtimes. Initially, we query the graph data, i.e. all vertices and edges including its relevant labels and properties, via Flink from Neo4j. The respective activities are shown in figure 33. First, we are calling Neo4j's REST API to request the graph data (1). In Flink, we then transform the

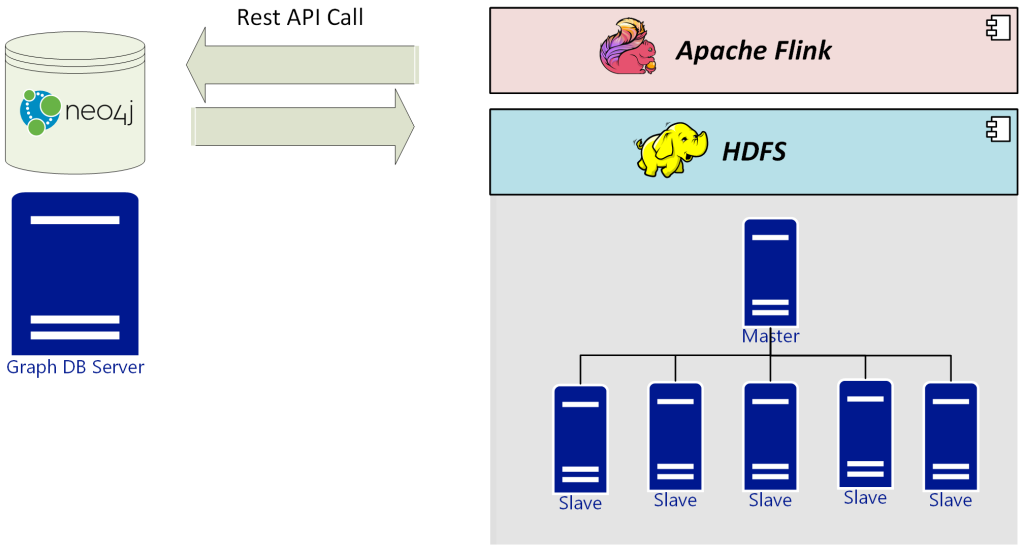


Figure 32: High Level System Architecture

loaded data to two datasets (2). One datasets stores the edges incl. its properties and the other stores the vertices with its labels. Finally, we persist the transformed result in the distributed file system (3). Hereby, we are eliminating the need to query the complete data from the graph database for each analysis run again and instead benefit from the data locality when using a distributed file system in combination with Flink (cf. section 9).

Figure 33: Initial Data Load

The second job (cf. figure 34) executes the actual graph analysis algorithm. This job needs to be fully executed for each meta path again. In other words, if we want to analyze the graph with respect to one specific meta path for example CUTUC (cf. 4.4.4), then we first need to read the edge - and vertex files from the HDFS (1) and create the in-memory graph representation (2) in Flink based on those. As discussed in section 6.2.2, we additionally need to store algorithm specific metadata, i.e. store information about ingoing edges at each vertex value as well as the target vertex label and subtype of an outgoing edge at each edge value, (3) to be able to execute the graph analysis. The actual analysis, this means the execution of the PCP evaluation algorithm (4) is done afterwards. Finally, the result is written back to the HDFS (5).

Figure 34: Graph Analysis Pipeline

6.2.4 Path Concatenation Plans

We presented the concept of PCP and PCP selection strategies in section 3.6.5. Moreover, we presented the meta paths used in our experiments in section 4.4. Now, we present the PCPs used in our implementation. Therefore, we show how to split up the chosen meta paths into individual primitive patterns, such that they can be concatenated to form the initial meta paths again. We manually developed the PCPs used here according to the *iteration optimized strategy*. A PCP is iteration optimized, if the height of the PCP tree is $log_2(l)$, where l is the length of the respective meta path. An iteration corresponds to a superstep in the vertex-centric graph processing model (cf. 3.5). A superstep ends with a synchronization barrier. If most parallel tasks finished already in a superstep, then those task slots would be idle due to the *synchronization barrier*. It follows, that the iteration optimized strategy is indeed beneficial, because the number of synchronization points is minimized. Moreover, it is easy to manually build up PCPs according to this strategy for the meta paths used here.

Since we chose 3 pairs of meta paths, the PCPs of the paths belonging to one pair have similar PCPs, where one PCP requires vertices with label *Blog* instead of label *Content*.

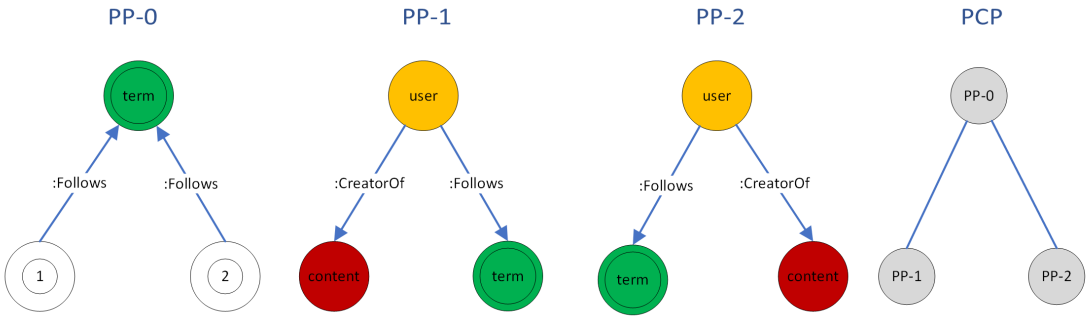


Figure 35: PCP - CUTUC

Figure 35 & 36 show the primitive patterns and PCPs for the meta paths CUTUC and BUTUB respectively. Both meta paths have the length 4. Thus, an iteration optimized PCP will have the height $log_2(4) = 2$ as the ones presented here. We explain the PCP for

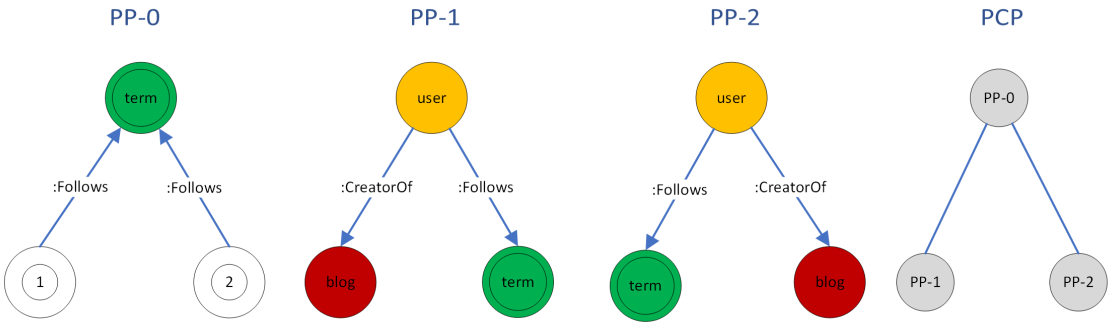


Figure 36: PCP - BUTUB

the meta path CUTUC now. But the explanation is valid for the BUTUB PCP too. The PCP is a full balanced binary tree with height 2. The root node is the PP-0, with PP-1 as left child and PP-2 as right child. Because all primitive patterns on the same level can be evaluated in parallel in the same superstep, the PCP requires 3 iterations. Instances of PP-1 and PP-2 are searched in the first superstep. The concatenation takes places in the second superstep based on the PP-0. The start - and end vertex of the meta path, i.e. the content vertices receive the final meta path result in the last superstep, where the result is persisted in the vertex values. The last superstep is the same for all PCPs. Therefor, we will not explain it for the remaining PCPs again.

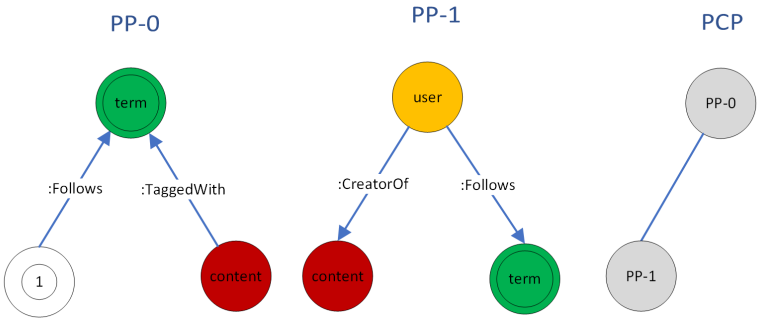


Figure 37: PCP - CUTC

The PCP for the meta path CUTC is illustrated in figure 37. The meta path has length 3 and $1 < log_2(3) < 2$. Moreover, the height of a tree can only be a positive natural number. Thus, an iteration optimized plan must have the height 2 as presented here. The first iteration searches for instances of PP-1. The second superstep finds all instances of patterns of the form $(Term) < -(Content)$ and concatenates those with the respective

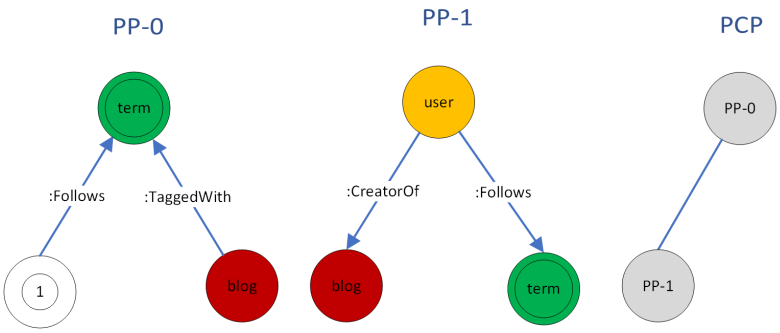


Figure 38: PCP - BUTB

instances of PP-1 on the *Term* vertex. Figure 38 shows the PCP for meta path BUTB. The explanations are analogue to the previous PCP discussed.

The PCP for the meta path CTC and BTB respectively consists only of one primitive pattern, whereas the primitive pattern is the meta path itself. The meta path has length 2 which equals the length of one PP. Thus, no path concatenation is necessary here and 2 supersteps are needed in total.

6.3 Evaluation

In this section, we discuss the evaluation results. First, we present the experimental setting. Next, we cover the Flink system configuration evaluation. Consequently, we present the runtime evaluation as well as the scalability evaluation. Finally, we conclude the different results gained in this experiment.

6.3.1 Experimental Setting

The experiment was conducted on SAP's internal cloud infrastructure and executed on virtualized servers with Intel Xeon E5-4650 v3 processors. Each CPU core has 2,1 Ghz. We used Apache Flink in Version 1.4.2 and measured the runtimes in different server configurations. Therefor, we varied the number of CPU cores per Server, the number of Taskslots per Taskmanager and the amount of RAM available to each Taskmanager. The Flink documentation [64] recommends to set the number of Taskslots per Taskmanager to the number of available CPU cores. This means the number of physical CPU cores

equals the number of Taskslots. The available Managed Memory of a Taskmanager will be equally divided by the number of Tasklots for exclusive usage of this memory spaces by the respective slots. Thus, we additionally reduce the number of slots to evaluate the impact especially in memory intensive scenarios. All configurations are listed in table 14. The stated amount of RAM is the configured amount each Taskmanager allocates in advance. The actual available RAM of the respective servers is slightly higher and used by the operating system. Taskmanager is abbreviated by TM and Taskslost by TS in the following listing.

	Config A	**Config B**	**Config C**	**Config D**	**Config E**
Cores / Server	16	8	8	16	16
TS/TM	16	8	4	16	8
RAM/TM	55 GB	110 GB	110 GB	240 GB	240GB

Table 14: Apache Flink Taskmanager Config Overview

We take two different runtime measurements per job execution for this experiment. Namely the *Total* runtime of the graph analysis pipeline as illustrated in figure 34 and additionally the *PCP Evaluation* runtime, i.e. the runtime of just the graph analysis algorithm (cf. figure 34 (4)). As explained in section 6.2.3, the complete analysis pipeline starts with reading from the HDFS as data source, entails data transformation steps, the actual analysis and ends with writing back to HDFS. We are only able to individually measure the runtimes of the graph analysis part, because it is a vertex-centric job. Flink instantiates the complete operator graph and pipelines the results of one operator into the next one. Thus, it is not possible to simply state how long each operator executed, although the runtimes can be read from the Flink UI. But pipelining implies that one operator does not use the whole cluster exclusively. Since the PCP Evaluation is a vertex-centric job and is executed in synchronized supersteps, we are able to get the runtimes for just this job individually.

6.3.2 System Configuration Evaluation

The system configuration can have a significant impact on the performance of a system. Moreover, it can be key to even get a system successfully running especially in resource constraint environments. We faced different issues during initial trial runs using the default system configuration, which emphasized the need to evaluate different configuration scenarios. Thus, we combined system knowledge with a *trial-and-error* approach for identifying important configuration parameters in the first step. Next, we want to derive different concrete configuration parameter settings for a further evaluation of meaningful

parameter settings for our specific use case and data set based on the insights gained in the first step. The goal is not to find the best system configuration, but to better understand the influence of relevant configuration parameters for this specific use case and dataset.

We started the trial-and-error runs on one machine with 32 CPU cores and 512GB RAM. Therefore, we instantiated one Jobmanager and one Taskmanager with 400GB heap space and 32 Taskslots. The Flink documentation [64] recommends to set the the number of Taskslots per core to the number of CPU cores. There is no extra explanation given. A Taskslot is a JVM Thread (cf. section 3.4.1) and hence using the recommended setting enables the executing of all Taskslots in parallel on a dedicated CPU. We assume, that this fact is the reasoning behind the recommendation. Hence, from a performance perspective, it seems to be desirable to set the number of Taskslots to the number of available cores in order to best utilize the CPU core. We mainly observed exceptions related to lost Taskmanagers or Jobmanagers. When analyzing the cause of those exceptions, we determined that these are related to Flink's deaths watch mechanism (c.f. section 3.4.1). A dead actor could possibly have two reasons. Either an actor shut down due to a concrete failure or an actor is wrongly detected as dead. The latter would mean a false positive detection induced by the actor system. Although, the lost Taskmanagers were not detected anymore by the Jobmanager, which we could verify in the Flink Dashboard and using the logs, we could also observe that the respective nodes were still running. We validated this statement by analyzing the Flink logs of the employed Taskmanagers. Thus, we confirmed a false positive detection of an actor by Akka's death watch mechanism, which resulted in a gated Taskmanager. For this reasons, the exceptions and the Dashboard indicated a dead node, while it was actually still running. We enabled the memory debug options in each Taskmanager by setting the config parameters *taskmanager.debug.memory.logIntervalMs* and *taskmanager.debug.memory.startLogThread*. Hereby, we were able to observe long garbage collection runtimes of the JVM. Oracle stated that "The process of locating and removing the dead objects can stall any application [...]" [40]. So, we determined that mainly Taskmanagers were falsely considered dead due to long garbage collection stalls.

Next, we wanted to know whether we can improve the garbage collection behavior by employing more Taskmanager Docker container and consequently more JVMs in parallel, while keeping the total amount of Taskslots, i.e. one Taskslot per core, stable. For this reason, we used three Taskmanager. Two of them with 11 cores and one with 10 cores. Next, we tried four Taskmanager with eight cores each and finally eight worker nodes with 4 Taskslots per worker. We could not observe any improvement here. Further possibilities for reducing the garbage collection impact would be to either reduce the total number of objects created, trying give more memory to the JVMs or configuring the heap division of the Taskmanager JVMs such that the unmanaged memory is increased (sf. section 3.4.2). The latter option implies a trade-off between the availble *Managed Memory* and

the available *User Memory*, because the available memory of a Taskmanager is mainly split up in the Managed Memory part and the User Memory part. The latter one is mainly used for the instantiation of objects in UDFs. We will evaluate different memory division settings within the next step. Furthermore, we are aware that we need to create a lot of Java objects in the vertex update function and hence, we probably have to deal with a higher garbage collection workload as compared to other graph algorithms and especially "simple" data transformation tasks, like joining two datasets, we we only produce a new tuple based on the join and do not execute a costly algorithm within the UDF. Therefore, we increased relevant timeout parameters related to the death watch mechanism to values higher than the default values [64]. Specifically, this is the *akka.ask.timeout*, which is related to Futures (cf. section 9) and all blocking Akka calls. Additionally, we increased the *akka.watch.heartbeat.pause*, which specifies the acceptable heartbeat pause for Akka's death watch mechanism. Lastly, we modified the *heartbeat.timeout*, which is the timeout for requesting and receiving heartbeat for both sender and receiver sides. As a consequence of the previous configuration adjustments, we were able to eliminate the errors related to garbage collection stalls and successfully execute the algorithm for the meta paths BUTB and BUTUB. This does not mean, that we optimized the execution but instead determined the influencing factors and find settings in which those two paths could be successfully executed.

For the remaining paths, we mainly experienced *Out-of-Memory* errors, resulting either in a killed Taskmanager process or in failed subtasks leading to the cancellation of the whole job, while the respective Taskmanager process is not getting killed. Flink divides up the heap space mainly in two regions. The managed memory, which can be swapped to disk in many cases if the available space is fully consumed and the user memory used for allocating short-lived Java objects (cf. section 9) primarily allocated in UDFs. The effect of failed subtasks implies that the memory resources available to one or more Taskslots were not sufficient for computing a subtask. We mainly observed two memory exceptions. Namely the *java.lang.RuntimeException* with the additional message "Memory ran out" and furthermore the *java.lang.OutOfMemoryError* including the information "GC overhead limit exceeded". We could determine the actual cause of these exceptions by varying the division of the Taskmanager JVM Heap spaces, i.e. varying the fraction of the available Flink Managed Memory and the User Memory (cf. figure 7) by adjusting the configuration parameter *taskmanager.memory.fraction*. When increasing the available managed memory and thus reducing the available amount of user memory, we observed the OutOfMemoryError. Vice versa, we noticed the RuntimeException when configuring a bigger fraction of the heap to be the user space resulting in a diminished Flink managed memory space. Hence, we could determine that the OutOfMemoryError is caused by a shortage of user memory and the RuntimeException is evoked as a consequence of insufficient Managed Memory. The Flink community coincides with our conclusion regarding those exceptions as read in a conversation found in the Flink *mailing list* archive

[16]. The Flink *mailing list* is the primary place where all Flink committers are present. Moreover, the Out-of-Memory error means that due to the insufficient User heap space a lot of Garbage Collection is necessary in order to free up memory as stated by Oracle [39].

Flink has a sophisticated memory management (cf. section 3.4.2) and keeps a lot of its objects in its own managed memory either on-heap or off-heap as controllable via configuration. Mostly Java objects instantiated in UDFs are kept in non-serialized form on the Java Heap. The workload of the PCP evaluation algorithm is executed in the UDF exceuted in parallel on each vertex. The runtime as well as the space complexity within the vertex UDF is high compared to many data transformations use cases or typical graph algorithms like *Graph Coloring* for example. We explain the reason for the high memory requirements in the scalability evaluation in section 6.3.5.

	Memory Fraction	**Off-Heap Usage**	**Solution Set in Managed Memory**
1	0.7	No	Yes
2	0.7	Yes	Yes
3	0.5	No	Yes
4	0.5	Yes	Yes
5	0.7	No	No
6	0.7	Yes	No
7	0.5	No	No
8	0.5	Yes	No
9	0.3	No	No
10	0.3	Yes	Yes
11	0.3	No	Yes
12	0.3	Yes	No

Table 15: Apache Flink Memory Configurations

In order to evaluate a meaningful memory configuration, we identified three important parameters. First, the memory fraction (*taskmanager.memory.fraction*). This is the fraction of the available main memory to a Taskmanager which is used as Flink *managed memory*. Hence, $(1 - MemoryFraction)$ is roughly the fraction used as *User Memory*. Second, the *Off-Heap Usage* indicates, whether the *managed memory* is kept on the JVM heap or off-heap. The Off-Heap usage is recommended for large JVM heap sizes and can improve the garbage collection behavior [17], [15]. The vertex-centric iterations in Flink is implemented on top of Flink's *Delta Iteration* (cf. section 9). It is possible to decide whether you want to keep the *Solution Set* of the Delta Iteration in *Managed Memory* or as Java Objects in the *User Memory*, which constitutes the last parameter we take into consideration here. Table 15 shows the different parameter configurations, that we want

to evaluate. Therefore, we execute all eight memory configuration settings for each meta path on a cluster using one Jobmanager and 6 Taskmanager each having 55 GB RAM and 16 cores. We set the number of Taskslots per Taskmanager equal to the number of cores as recommended in the Flink documentation. We successfully executed the implementation for the meta paths BUTUB and BUTB. Interestingly, the configuration 3 and 9-12 are the configuration that was not executable for any meta paths in this setting. When either enabling the off-heap usage (configuration 4) and/or keeping the Solution Set on the heap (configuration 7/8) we could successfully run the implementation for paths BUTUB and BUTB. Thus, we can confirm the usefulness of the off-heap usage in this context. Moreover, we observed that configuration 6 is the most efficient one in terms of runtime. We further determined configuration 6 as the fastest one when varying the number of worker nodes.

All other meta paths, i.e. the paths with high and very high selectivity (cf. table 8), were not executable in the considered setting. We still observed the previously described issues and mainly out of memory errors. Either related to a lack of managed memory or insufficient JVM heap space. Since we already employed 6 Taskmanager with 330 GB memory in total for the configuration evaluation, we saw the need to try to reduce the memory footprint by making software design improvements. The cost of evaluating a PCP is actually the sum of the cost of evaluating all primitive patterns contained in a PCP, which is highly depended on the number of intermediate paths [51]. The required memory for executing the PCP evaluation highly depends on the number of intermediate paths too, because we need to instantiate Java objects on the JVM Heap proportional to the number of intermediate paths. Moreover, the number of intermediate paths is also strongly depended on the degree of a vertex (cf. equations 10-12). Our approach is based on primitive pattern matching by neighbor exploring since we are working in the vertex-centric model. In order to be able to evaluate the neighbor nodes and thus the edges, we needed to persist the edge data incl. the neighbor vertex label at each vertex as explained in section 6.2.2. Moreover, we we need to communicate the intermediate paths to the neighbor vertices and persist the final results at each vertex.

Thus, we identified on the one hand the temporary instantiation of Java objects in the vertex-update function as a bottleneck and a significant memory consumer leading to *OutOfMemory* errors or garbage collection stalls and the increasing underlying datasets of the graph leading to a shortage of Flink Managed Memory on the other hand. Consequently, we reworked the design such that we tried to circumvent nested Java classes and instead used primitive data types as much as possible. This is a trade-off between the readability of the code and the performance. In typical Java programs a sophisticated object oriented design and hence the readability is preferred because the performance disadvantages are too small in many use cases. In our use case and setting we determined the usage of primitive data types instead of nested Java classes as an important design goal.

We reworked the design of the Java objects used in the UDF according this observation and were able to reduce the memory consumption by approx. 29%. We calculated this value by comparing the *Bytes Sent* and the *Bytes Received* measurements by the Flink Subtasks of the graph analysis job.

6.3.3 Configuration Evaluation Summary

We identified the timeouts of the underlying actor system as important configuration parameters in the first place. Those need to be adjusted in order to avoid false positive detection of dead Taskmanagers. This is mainly related to high garbage collection efforts during the PCP evaluation in the vertex-centric UDF.

Moreover, we saw that a that insufficient JVM heap space either leads to *OutOfMemory* errors or to Garbage collection stalls because the application spends most of the time for the collection of old Java objects. Vice versa, a bigger JVM heap reduces the probability of garbage collection stalls, while leading to JVM crashes in case of insufficient Manged Memory. Furthermore, we identified memory configuration 6 (cf. table 15) to be the fastest setting for the meta paths BUTUB and BUTB of the examined configurations. This implies further that off-heap memory usage is beneficial in this context. Consequently, we will use this memory configuration for the actual runtime measurements.

6.3.4 Absolute Runtime Evaluation

In this section, we present the measured runtimes for the meta paths BUTUB and BUTB in the considered configurations in detail. The remaining paths were not executable in this approach as we discuss in section 6.3.5. The scalability - (cf. section 6.3.6) and efficiency evaluation (cf. section 6.3.7) will be based on this results. The results are based on the average of two individual runtime measurements per *meta path / #Taskmanager* combination in order to smooth the effect of runtime variations. We use the *memory configuration 6* (cf. table 15) as discussed in the previous section.

Config A

We conducted the experiment on up to seven worker nodes for each considered meta path in this configuration. The available memory resources were not sufficient when using just one Taskmanager. Hence, we measured the runtimes for two up to seven Taskmanagers. The experimental results for meta path BUTUB are presented in figure 39. We could also observe a significant lowering of the *Total runtimes* from approx. 31 minutes down

to approx. 8 minutes. The *PCP evaluation* using two Taskmanagers took approximately 14 minutes, while only lasting 3m8s minute using all seven worker nodes.

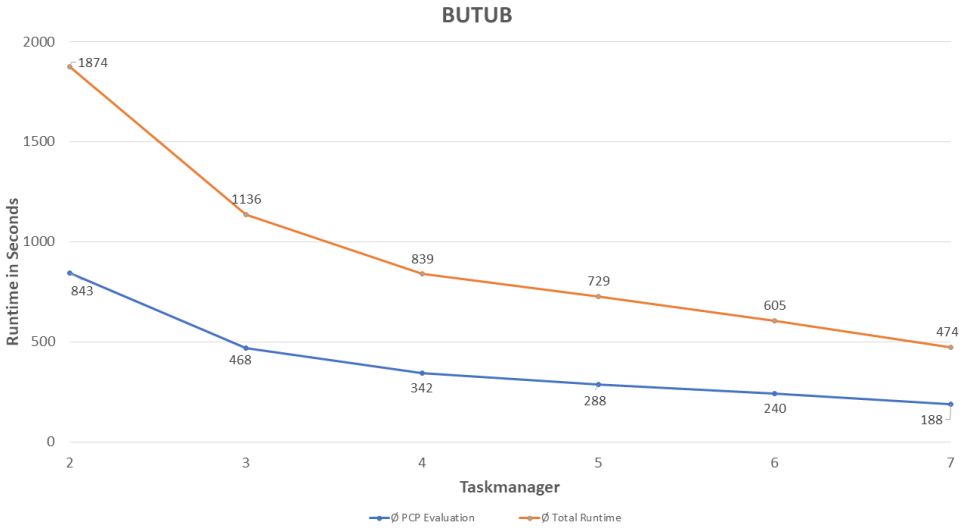


Figure 39: BUTUB Runtimes with Config A

The results for meta paths BUTB are visualized in figure 40. We were able to successfully reduce the *Total* runtime from approx. 33 minutes using two Taskmanagers to approx. 9 minutes for all seven worker nodes. The *PCP Evaluation* runtime, i.e. the runtime of the vertex-centric analysis job, was reduced from approx. 15 minutes to approx. 3m30s.

Config B

This setting was scaled up from one to six Taskmanagers. Figure 41 illustrates the results for meta path BUTUB. We observe a *Total* runtime of approx. 110 minutes when using one worker and approx. 18.9 minutes when using all six. Analogous, we observe initial *PCP Evaluation* runtimes from approx. 44.2 minutes, which we were able to reduce to approx. 7.9 minutes.

The *Total* runtime for solving the PCCP for the meta path BUTB is approx. 111 minutes when using just one worker node and approx. 19.3 minutes for all six Taskmanagers. Again, the *PCP Evaluation* took approx. 46.9 minutes at most and approx. 7.9 minutes at lowest in the examined setting.

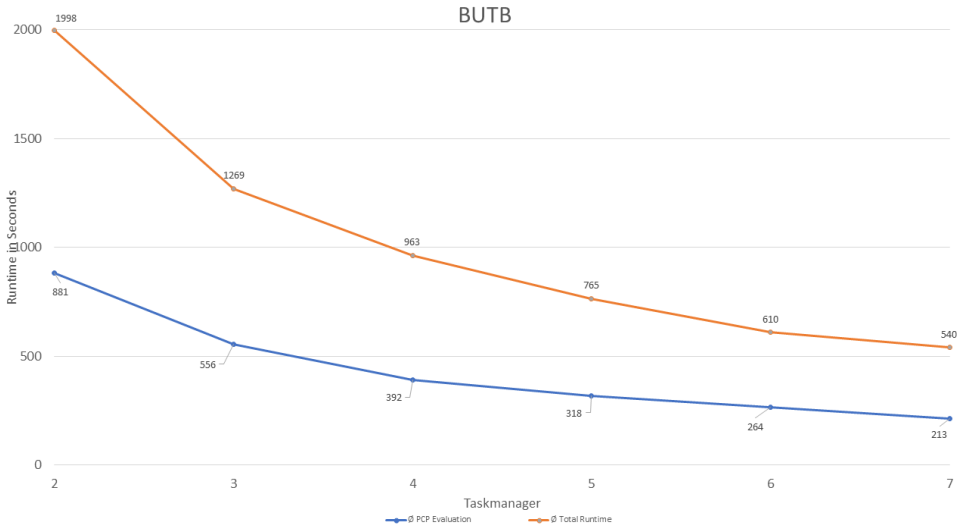


Figure 40: BUTB Runtimes with Config A

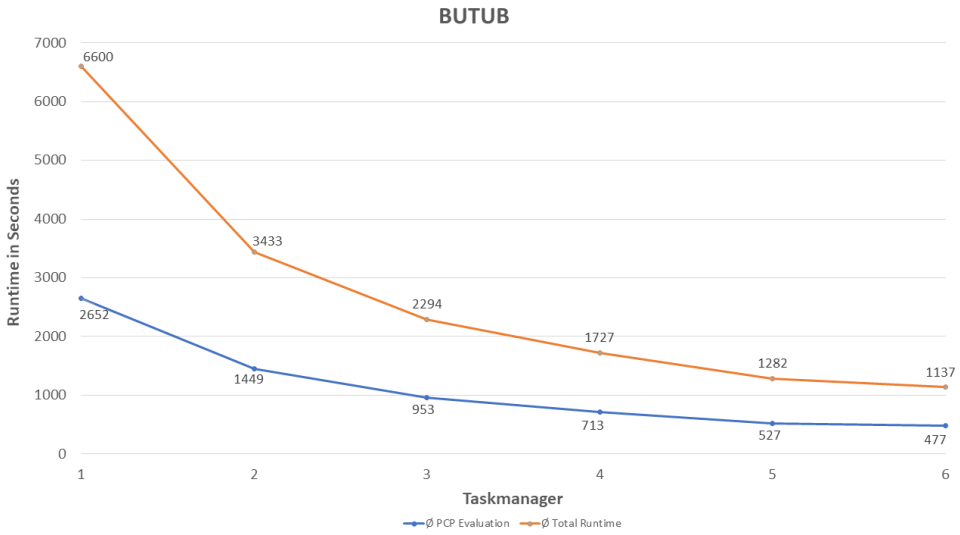


Figure 41: BUTUB Runtimes with Config B

6.3.5 Meta Path Scalability Evaluation

We are interested in the scalability of the implementation with regards to the characteristics of the meta path to be able to better understand the performance impact implied by such characteristics. We therefore examine the influence of the meta path *length* as well

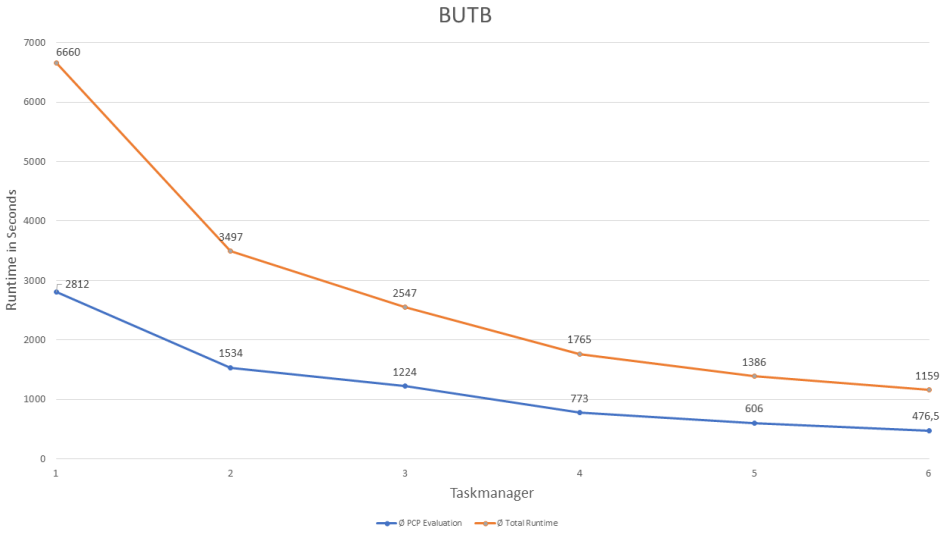


Figure 42: BUTB Runtimes with Config B

as the *selectivity*. The observed meta paths were presented in section 4.4 and summarized with its characteristics in table 8.

Our initial assumption was that the length and the selectivity of meta paths positively correlate with the runtime of the PCP evaluation algorithm. The reason for this initial assumption with respect to the path length is that the length of a meta path positively correlates with the required supersteps for the PCP evaluation (cf. section 6.2.4). The reason for the initial assumption related to the selectivity of the meta path is that a lower selectivity suggests lower computational cost, lower memory consumption during the computation in a superstep and moreover a lower messaging workload at the synchronization barriers of a superstep, since the result of a successfully matched primitive pattern needs to be communicated to the respective neighbor vertices.

First, we analyze the results of the successful computed meta paths *BUTUB* and *BUTB*. Table 16 shows the runtime differences between those two meta paths in Config A and table 17 for Config B respectively. The numbers are calculated based on the runtime measurements presented in section 6.3.4. The runtimes for the shorter meta path BUTB are consistently higher than for the longer path BUTUB. The *PCP Evaluation Runtime* of *BUTB* is 12 % higher on average in Config A and 11 % in Config B. The total runtimes are still higher but the difference is slightly smaller. This is because only the PCP Evaluation will be effected and not the preprocessing. Moreover, the result set consists of over 33M BUTUB path instances and nearly 40M BUTB instances. These numbers indicate that BUTB produces a higher number of intermediate paths and final paths leading to

higher runtimes as compared to BUTUB.

	2 TM	**3 TM**	**4 TM**	**5 TM**	**6 TM**	**7 TM**	**ø**
PCP Eval. Diff.	5 %	19 %	15 %	10 %	10 %	13	12 %
Total Diff.	7 %	12 %	15 %	5 %	1 %	14 %	9

Table 16: Config A - Procentual Runtime Differences from BUTUB to BUTB

	1 TM	**2 TM**	**3 TM**	**4 TM**	**5 TM**	**6 TM**	**ø**
PCP Eval. Diff.	6 %	6 %	28 %	8 %	15 %	0 %	11 %
Total Diff.	1 %	2 %	11 %	2 %	8 %	2 %	4 %

Table 17: Config B - Procentual Runtime Differences from BUTUB to BUTB

We observe that a longer meta path does not necessarily result in longer runtimes. This has two main reasons. First, the respective PCPs (cf. section 6.2.4) both have height two and consequently the same required amount of supersteps. Nevertheless, the computational cost within a superstep is different, because this cost is mainly affected by the number of Primitive Patterns to be evaluated in the same Superstep in parallel and moreover by the number of intermediate paths as a result of the Primitive Pattern evaluation as shown by Shao et al. (cf. equations 8&9).

We needed to materialize all edges including the vertex types of the source vertex for ingoing edges and the destination vertex types for the outgoing edges to be able evaluate a Primitive Pattern on the pivot vertex by neighbor exploring (cf. section 6.2.2). Table 5 shows the top 5 *terms* where *content* vertices are tagged with. We can see that the term *abap* is the most used one for tagging content vertices. This means that the *term* node with the value *abap* has 240,472 ingoing edges of type *TaggedWith*. Hence, the respective vertex value of the *abap term* node in our Flink graph representation initially stores 240,472 edge data entries. Since we have 14,708,372 nodes in total in the SAP Community network as shown in table 3, the memory consumption for the graph representation increases significantly after the preprocessing. Considering, that each edge in the graph needs to be materialized at its source and destination vertex including the respective vertex types, this results in an increase of *two* times the *total number of edges* in the SAP Community network times the bytes needed to materialize one edge including neighbor vertex types. To capture the edge information, we need to store one additional vertex ID, since an edge is defined by its source and end vertex as well as the edge type. Moreover, we need to materialize the vertex *type* and *subtype* of either the source or destination vertex depending on the edge direction. We already compressed the graph representation by mapping all

types and IDs to integers. Consequently, we will need at least 4 additional integer values at each vertex per incoming and outgoing edge. An integer in Java is represented by four bytes. Thus, we need to store 32 additional bytes per edge. The considered network has approx. 30.8 million edges which results in an increase of approx. 930 megabytes. A graph in Flink is represented by a dataset of vertices and edges, whereas the vertex set consists of one *ID*, one *type* and one *subType* per vertex and the edge set comprises two *IDs* and the edge *type*. All attributes are represented as integer. Thus, our initial graph comprises approx. 520 megabytes (350 megabyte for the edges and approx. 170 megabytes for the vertices) graph specific data. This analysis excludes additional data structures needed by Flink for representing the datasets itself. Nevertheless, we observe that our initial graph representation increases at least by a factor of 3 after the preprocessing.

We analyze the PCP for the meta path *Content-Term-Content* (cf. section 6.2.4) exemplary for illustrating the memory requirements in the actual PCP Evaluation and hence in the vertex-centric UDF. The meta path CTC has length two, which means it is a Primitive Pattern of type *NL-NL* itself. This means again that the meta path itself is its PCP. The pivot vertex is of type *term*. The UDF is executed in parallel on each vertex and the full Primitive Pattern evaluation is executed on all vertices of type *term*. We use again the *term* vertex with the value *abap* exemplary. Since both edges of the meta path are incoming relative to the pivot vertex and both neighbor vertices are of type *content*, we need to evaluate the cross product for the set of all 240,472 incoming edges with the same set (cf. equation 10). This will result in over 57 billion intermediate results just in this one UDF execution. Hence, we observe that the memory requirements as well as the computational cost in one UDF are highly influenced by the respective vertex degrees. The Taskmanager or more precisely the Taskslot executing the UDF on a vertex with a very high degree will require significantly more memory than a Taskmanager which is mainly executing UDFs on vertices with a relatively low degree. This fact is a bottleneck and a clear drawback of the used approach for datasets not exhibiting a uniform distribution of edges and partial paths in the considered graph. Analogous explanations are valid for the other meta paths as well. We presented the relevant distributions in section 4.4.7. Consequently, it was not possible to compute the path instances and the respective path counts for the other considered meta paths.

We determine the previously discussed observations as the main reasons for the *OutOfMemory* errors and JVM Garbage Collection stalls when computing high and very high selective paths in the considered settings as discussed in section 6.3.2.

6.3.6 Strong Scalability Evaluation

Now, we will analyze the strong scalability (cf. section 2.8) based on the runtime measurements presented in section 6.3.4. The results are visualized by line charts in the figures 43 - 45. We need to set the runtime t_N of N workers in relation to the runtime t_1 using just one worker (cf. equation 1). Since, we do not have results for using one Taskmanager in *Config A*, we just defined the $t_1 = 2 * t_2$. This means we artificially assume linear scalability here and consequently the strong scalability is exactly 2 here.

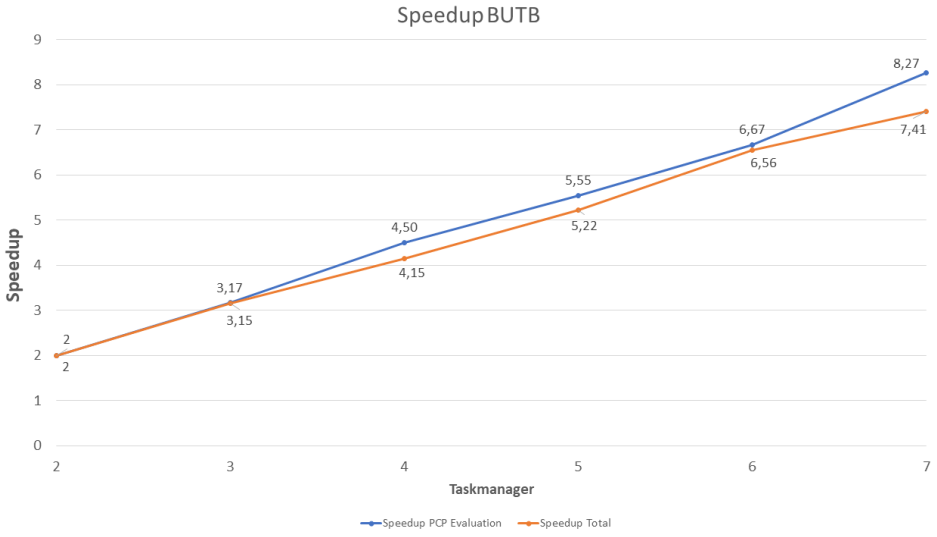


Figure 43: BUTB Speedup with Config A

We call the strong scalability to be perfect, if $S_n = n$ with $n \in N$. We observe nearly perfect scalability for all retrieved results. Interestingly, the scalability for the meta path BUTB in *Config A* is slightly higher than the perfect scalability. Theoretically, the scalability S_n can not be greater than n, because this would imply that the resource utilization for $n = 1$ is lower as the resource utilization of one node when $n > 1$. Moreover, the distributed execution always implies a overhead for the distributed coordination, data shuffling and the communication of intermediate results. Since one Taskmanager already parallelizes its workload via its Taskslots, we do not have a serial execution for t_1. Nevertheless, from a practical point of view this evaluation is sufficient. Otherwise, we would receive a result that perfectly matches the theory without a practical benefit for our actual research question. Moreover, the runtimes vary slightly between repeated executions, which consequently influences the scalability calculation. Thus, we interpret the results to exhibit approximately perfect scalability for the meta paths BUTUB and BUTB.

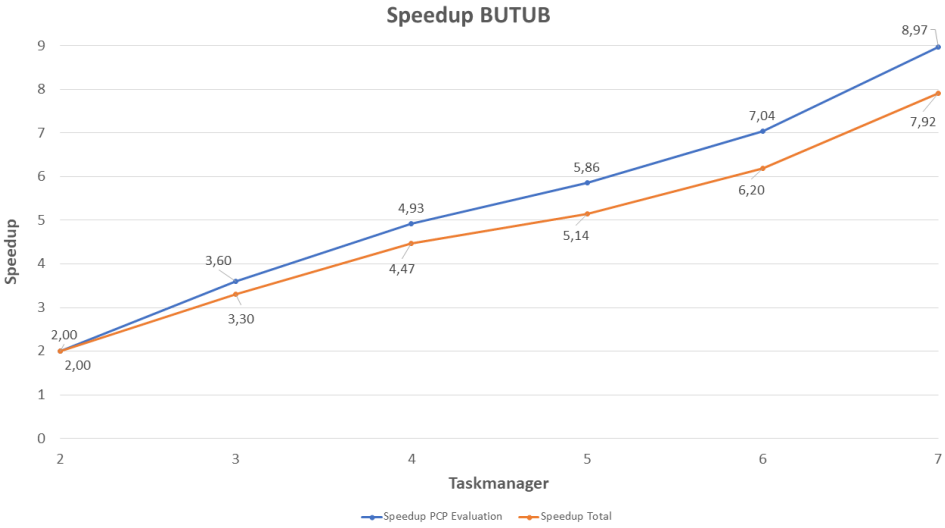


Figure 44: BUTUB Speedup with Config A

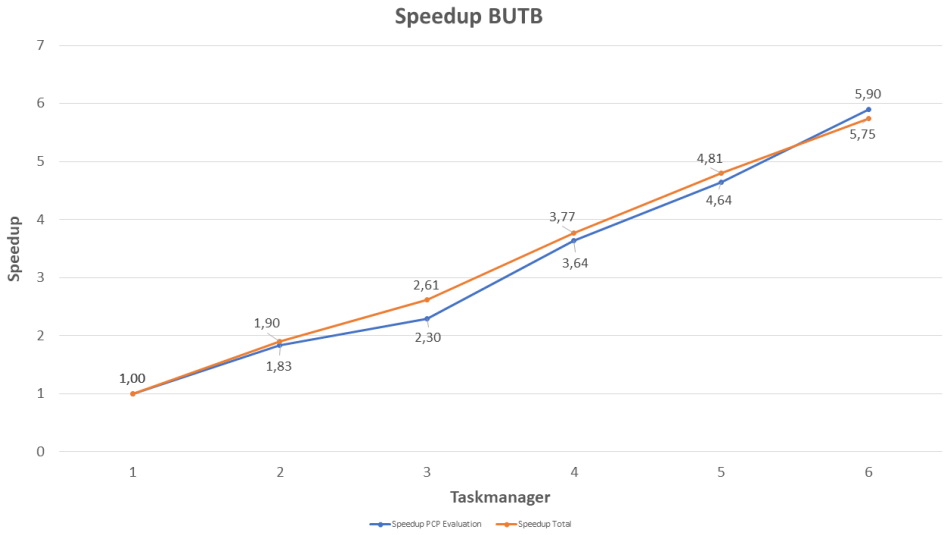


Figure 45: BUTB Speedup with Config B

In conclusion, we observe that the approach scales indeed very good for the meta path BUTB and BUTB. This is also due to the fact that the vertex-centric model itself scales well, because the UDF is invoked in a data-parallel manner on each vertex. The positive results also indicate that the communication overhead is negligible. The main reason for this observation is that a significant fraction of the total computational cost is done in

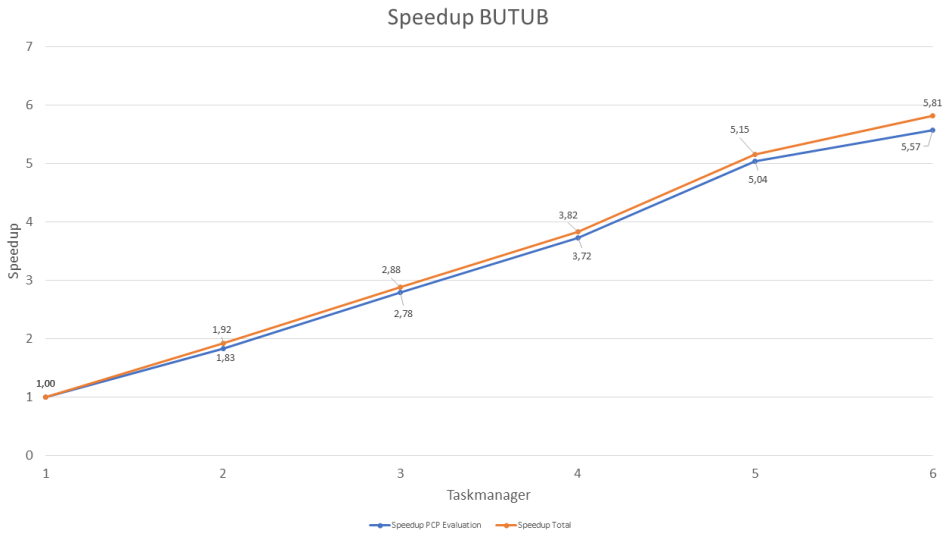


Figure 46: BUTUB Speedup with Config B

the UDFs and the amount of data that needs to be shuffled between the Taskslots is low for the respective meta paths. In section 6.3.5 we explained that the UDF of the PCP Evaluation is the bottleneck of the implemented approach. This is why we were not able to execute the meta paths which imply an extraordinarily high workload in individual UDFs using the resources available for the experiment.

6.3.7 Efficiency Evaluation

The efficiency values in *Config A* are consistently higher as for the Config B. It is not clear from the data whether this means that this configuration is indeed the more efficient one. The strong scaling efficiency results are listed in the tables 18-21. The efficiency is calculated according to equation 5, and consequently it is depended on the speedup. We explained in section 6.3.6 that we artificially set the base calculation value t_1 for the speedup calculation in *Config A* to twice the runtime of t_2, because one Taskmanager was not capable of executing the algorithm. When looking at the efficiency results for *1 TM* in *Config B*, we can observe values for *2 TM* already lower than the values for *3 TM* in *Config A*. Those inaccuracy most likely effects the further values as well. The reason why we compare the 3rd and 2nd values is that the first efficiency value will always be one since it is calculated in relation to itself. Moreover, we observe the best efficiency when using seven TM in *Config A* as shown in Table 18. Those value is based on two runtime measurements of 427 seconds and 520 seconds. This is an untypical difference of more than 1.5 minutes. Assuming that higher value is close to an average value when executing

the same setting more often, than the respective efficiency will also be closer to the other ones.

	2 TM	3 TM	4 TM	5 TM	6 TM	7 TM
Efficiency PCP Evaluation	1.00	1.20	1.23	1.17	1.17	1.28
Efficiency Total	1.00	1.10	1.12	1.03	1.03	1.13

Table 18: BUTUB Efficiency with Config A

	2 TM	3 TM	4 TM	5 TM	6 TM	7 TM
Efficiency PCP Evaluation	1.00	1.06	1.13	1.11	1.11	1.18
Efficiency Total	1.00	1.05	1.04	1.04	1.09	1.06

Table 19: BUTB Efficiency with Config A

We observe very good efficiency values close to an efficiency of 100%. The strong scalability as well as the efficiency indicate that we could add further worker nodes and efficiently further reduce the runtimes, because we do not see an decrease in the efficiency so far and instead observe approximately linear scalability. It would be interesting to scale the approach further until a significant decrease in the efficiency will occur. Hereby, it would be possible to find a sweet spot, where the runtimes are lowest while having a good efficiency and hence a good return of hardware investment.

	1 TM	2 TM	3 TM	4 TM	5 TM	6 TM
Efficiency PCP Evaluation	1.00	0.92	0.93	0.93	1.01	0.93
Efficiency Total	1.00	0.96	0.96	0.96	1.03	0.97

Table 20: BUTUB Efficiency with Config B

	1 TM	2 TM	3 TM	4 TM	5 TM	6 TM
Efficiency PCP Evaluation	1.00	0.92	0.77	0.91	0.93	0.98
Efficiency Total	1.00	0.95	0.87	0.94	0.96	0.96

Table 21: BUTB Efficiency with Config B

The server configuration *Config A* has half as much memory available compared to *Config B* while doubling the number of available Processors and Tasksslots respectively (cf. 6.3.1). We observe that the number of processors is the actual performance driver since the runtimes for *Config A* are approximately the same as the runtimes for *Config B* with

twice as much Taskmanagers and thus the same amount of CPUs. For example the *Total runtime* with 4 worker nodes, i.e. 48 cores, for meta path BUTUB in *Config A* is 1136 seconds while the corresponding runtime in *Config B* with 6 TM is 1137 seconds. Thus, the available memory needs to be sufficient but does not directly impact the runtimes in our experiment. Consequently, *Config A* is preferable because we get along with a fourth of the memory provided by server *Config B* for achieving similar runtime results. But we pointed out the drawback of the approach for high degree nodes. Thus, those observation are not representative for every meta path defined on the SAP Community schema, since the distribution of partial paths and the total amount of path instances impact the results.

6.4 Interim Conclusion

We were able to successfully compute the path instances for meta paths with normal selectivity, while it was not possible to execute the high - and very high selective paths (cf. table 8) at all. The main reason is that the applicability and scalability is strongly dependent on the data distribution. More specifically, the usefulness is negatively impacted by high degree nodes. Those peak nodes lead to very high processing and storage requirements within one specific UDF. Moreover, Flink can not spill intermediate results to disk when computing in the vertex-centric model in case of insufficient memory. Especially not the memory allocated in the vertex-centric PCP Evaluation function, since it is user-defined and thus cannot be optimized by Flink. Moreover, increasing the heap size may prevent *OutOfMemory* errors but will lead to garbage collection stalls because of the instantiation of too many Java objects in the UDF and hence in the user heap space, which is not managed by Flink (cf. section 3.4.2). The *term* nodes in the SAP Community network exhibit a distribution leading to the previously mentioned problems, when contained within a meta path. Nevertheless, the result for the normal selective paths show that the approach scales linearly and that the runtimes could be efficiently further reduced by adding more machines. Moreover, we saw that a shorter meta path does not necessarily imply shorter runtimes as compared to a longer one.

7 Path Count Computation with Hybrid Approach

In the following we propose an alternative approach for coping the problem of high degree vertices leading to JVM gargabge colletion stalls and *OutOfMemory* errors when computing high selective meta paths with our *meta-graph extraction* framework implementation as presented in section 6.

7.1 Main Idea

The goal is to be able to distribute the workload executed in one vertex-centric UDF for high degree vertices. It is based on the idea of using Flink's dataset transformations (cf. section 9) instead of the graph processing API, i.e. the vertex-centric model. We identified the *term* node in the considered meta paths as the potential high degree vertices. Moreover, we showed that the amount of generated path instances explodes around the peak *term* nodes (cf. section 4.4). Hence, we propose to first built up the two sets of matching path instances around the *term* nodes. This can be achieved either via Neo4j or in the vertex-centric model as we presented in section 5 and 6. We recommend to use Neo4j for this purpose, since we demonstrated that all relevant partial paths having the *term* either as start - or end node can be computed efficiently within the graph database (cf. table 12). As a result, we get a list of partial paths containing the respective vertex IDs. Consequently, we only operate on the partial paths list instead of the full graph including the additionally materialized edges at each vertex (cf. section 6.2.2).

Next, we can join both sets based on the *term* with Flink's join operator (cf. appendix 9). Hereby, the full paths are created using the path concatenation property of meta paths. In other words, we are generating the full paths by joining the respective datasets. Figure 47 illustrates the proposed apporach at the example of the toy graph given in figure 12. First, we query all CUT partial paths instances from Neo4j to Flink. Next, we join the CUT partial path set with itself with the *term* as join key. Hereby, we are generating the full CUTUC paths. Each red box represents one UDF invocation. The red and filled boxes indicate the UDFs, which produce the final path instances, while the empty ones would only produce duplicates. Those duplicates can easily be filtered out within the UDF.

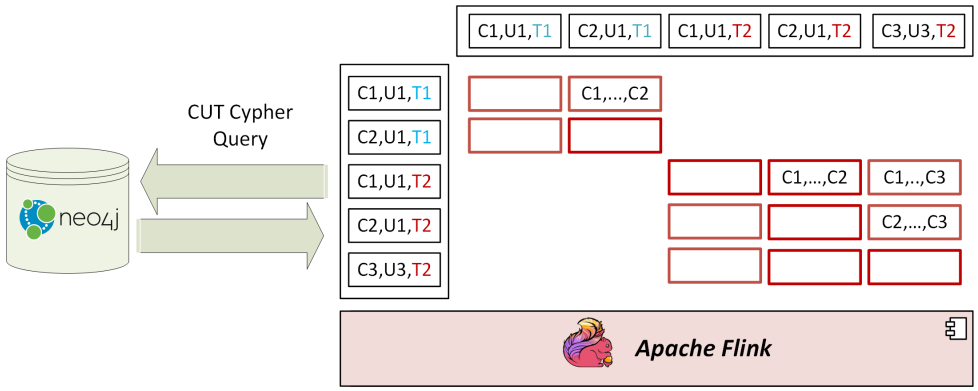


Figure 47: Hybrid Approach for Path Concatenation - CUTUC

The important difference here is that the UDF of Flink's Join operator is executed on each joined tuple. As a consequence, the workload per UDF invocation is the same for all parallel invocations, since we only need to concatenate two intermediate path instances, whereas we needed to concatenate an unpredictable amount of intermediate paths in one UDF when using the vertex-centric model. This implies further that the main workload is executed in Flink's Managed Memory instead of in the User Memory (cf. section 3.4.2). Consequently, Flink can operate more efficiently on the data. Furthermore, Flink is able to spill data to disk when using the join operator [25], whereas out-of-core processing is not possible with Flink when using vertex-centric iterations [26], because it is based on Flink's Delta-Iteration (cf. section 3.5 & 9). Moreover, we can easily implement a mechanism within the UDF for not computing the mirrored path instances implied by symmetric meta paths, which was not easily possible with Neo4j as well as with the meta-graph extraction framework. Thus, the result set is smaller by a factor of 0.5 while comprising the same semantic information. Hence, the negative effects of workload peaks for high-degree vertices is circumvented as a consequence of the before mentioned advantages. In addition, it is possible to execute much higher workloads, since the memory can be swapped to disk if necessary and the main part of the workload is efficiently executed in Flink's managed memory.

Finally, the results can be aggregated based on the start and end nodes for computing the path counts. This can be achieved by using a simple *MapReduce* job for example (cf. appendix 9). The *Mapper* UDF creates a new tuple with a key composed of the start - and end node of the final path instance. The *Reducer* UDF aggregates the instances with the same key. Alternatively, Flink provides more high-level operators in its Dataset API which could be used for aggregating datasets based on user defined keys and hence calculating the final path counts between the start and end node.

7.2 Proof of Concept

We implemented the hybrid approach as a proof of concept for all meta paths. We could demonstrate that the computation of very high workloads is ensured and even result sizes of over 135B path instances were generally possible when using just one Taskmanager. Moreover, we constantly observed a good Garbage Collection behavior of Flink for all meta paths. Table 22 summarizes the results taken for the proof of concept using 1 Taskmanager with server configuration D (cf. table 14) ordered by the selectivity. We did not consider the Neo4j computation time since this can be done offline. Analogously, we did not consider the runtimes for the initial graph extraction from Neo4j to Flink while evaluating the *meta-graph extraction* framework implementation for the same reason. Nevertheless, all relevant partial paths computation runtimes are listed in table 12 and are especially for the very high selective paths negligible.

Meta Path	Aggregation	Result Size	Runtime	Remark
BUTUB	Yes	8,071,907 rows	2m20s	-
BUTUB	No	16,863,037 rows	2m03s	TS finished equally
BUTB	Yes	38,294,760 rows	2m27s	-
BUTB	No	39,902,383 rows	2m01s	TS finished equally
BTB*	No	481,747,176 rows	3m02s	w/o Top 2 Terms
BTB	Yes	4,141,799,768 rows	3h30m	-
BTB	No	4,830,509,062 rows	52m01s	Second last TS finished in 16m
CUTC	Yes	9,063,260,152 rows	3h44m	-
CUTC	No	9,520,695,353 rows	1h42m	Second last TS finished in 31m
CTC	Yes	29,027,729,553 rows	14h53m	-
CTC	No	38,568,559,156 rows	16h1m	Second last TS finished in 1h
CUTUC	Yes	85,910,037,852 rows	11h59m	-
CUTUC	No	135,535,185,297 rows	9h12m	Second last TS finished in 8h4m

Table 22: Hybrid Approach Result Summary

The normal selective meta paths could be computed efficiently and all used Taskslots finished rather equally. The BUTUB computation lasted 2m20s and the BUTB one took 2m3s.

All 4,830,509,062 BTB path instances could be enumerated in 52m. We observed that 14 of 16 Taskslots finished already after 2m30s at most. The second last slot finished after about 11m30s, while the last one required nearly 52m. We assume that the last two Tasklslots executed the path concatenation for the partial paths on the two most occuring *terms* (cf. table 4) among others. The BTB path instance computation without the partial paths containing the top 2 *terms* took only 3m2s and resulted in 481,747,176 instances. This result supports our assumption. The reason is that Flink hash partitions the join keys over the used Taskslots, since it used the *Hybrid-Hash-Join* strategy. As a consequence all tuples with the same key will be joined in the same Taskslot. Thus, this approach is also affected by peak nodes. But in this case it only impacts the runtime, while it is possible at all to compute the high selective meta paths, since individual partitions can be spilled to disk [25] and the path concatenation is executed in serialized form in Flink's Managed Memory instead as regular JAVA objects in Flink's User Memory. The computation of BTB with aggregating the results, i.e. including the final path count computation, endured 3h30m. We could observe that the Taskslots were equally utilized for in the aggregation phase. Hence, we could observe that an uneven data distribution negatively impacts the resource utilization for the path concatenation but not the aggregation phase. We could observe similar resource utilization behavior for the remaining meta paths as for BTB, while the longer paths, namely CUTC and CUTUC, exhibit a slightly better resource utilization as BTB and CTC.

8 Discussion

Our findings show that the *meta-graph extraction* framework proposed by Shao et. al. indeed outperforms the meta path count computation with Neo4j and Cypher. We observed approximately linear scalability for all successfully computed paths via the implemented framework, i.e. the normal selective meta paths BUTB and BUTUB. Neo4j as well as the *meta-graph extraction* framework implementation in Flink were capable of computing the path instances for normal selective meta paths (cf. table 8), while it was not possible to compute the instances of the high and very high selective meta paths (cf. table 8) in both approaches. We expected the Neo4j approach to be easy to implement on the one hand, because we can use a query language, and efficient on the other, since it implements a native graph processing engine using index-free adjacency. Furthermore, we expected that Neo4j will not scale out and hence exhibit limitations in its generic applicability for the path count computation. We could confirm this expectation and thus the *hypothesis 2* holds true. Also batch processing via Cypher's *Skip* and *Count* operators is not reliably possible as a manual way of out-of-core processing. Contrary, we expected to be able to compute all meta paths using the distributed framework implementation on top of Flink, because we assumed that out-of-core processing features ensure at least the successful execution, although the performance might decrease. We detected that out-of-core processing is not possible for the vertex-centric *PCP Evaluation* algorithm, since the main workload executes within a UDF, which can not be optimized by Flink.

The computation for BUTB took 19m with Neo4j and 3m30s at best in the considered setting when using the Flink based *meta-graph extraction* framework implementation. The BUTUB computation lasted 2h38m in Neo4j while only enduring 3m8s in the distributed setting. We observed that runtimes are strongly dependent on the selectivity in both approaches. Additionally, the Cypher query execution can be impacted more significantly by the length of a meta path as the considered distributed approach. It should be emphasized that the actual query execution used only one CPU core in Neo4j. In contrast, the previously mentioned Flink results required seven worker nodes, i.e. 112 CPU cores. Thus, the execution in Neo4j is more efficient in terms of CPU resource consumption due to its efficient graph processing engine using *index-free-adjacency*. The speedup and efficiency evaluation of the *meta-graph extraction* framework approach shows that the measured runtimes can be reduced further by adding more Taskmanager instances to the Flink cluster. This means that the distributed framework implementation scales linearly for the meta path BUTB and BUTUB in the observed setting. It would be interesting to see how far the runtimes can be pushed down further until the speedup decreases. Since, we still obtained linear scalability when using seven worker nodes, we assume that the runtimes can be significantly reduced further. Furthermore, we identified the data distribution as a critical factor determining the general applicability for the computation of specific

meta paths. Especially the Flink based framework implementation in the vertex-centric model can not deal with high degree nodes in the SAP Community network, resulting in *OutOfMemory* errors or Garbage Collection stalls due to either an uneven distribution of the workload in the cluster or in other words the instantiation of too many Java objects on the JVM heap. Further research is needed to prove whether this is a Flink specific behavior or whether the same errors will occur when using Apache Giraph for example. We assume Giraph will exhibit similar problems since both systems are JVM based and the *PCP Evaluation* takes place in a UDF, which implies that at least the JVM Garbage Collection effort should be equal. Likewise, the distributed system architecture of both systems is similar, since Giraph uses slotted resources as well, so an uneven workload distribution will occur. Shao et al. [51] assume a uniform distribution of relations and intermediate paths over the nodes of their observed HINs, which implies a more evenly distributed workload over the processing slots in the cluster. In contrast, the *term* node distribution as well as the partial path distributions around the *terms* in the SAP Community network exhibits a distributions similar to a long-tail distribution. Furthermore, the *terms* are semantically very relevant in this domain and consequently we included this node type in every meta path. Moreover, big JVM heap sizes are in general challenging for Java based big data processing systems, because of the consequent Garbage Collection overhead [17]. Therefore, optimizations such as system managed memory and off-heap usage are implemented in those systems. Thus, we determine the data distribution and the selectivity of a meta path as the main reasons that Shao et al. were not confronted with too high workloads within individual UDF invocations. Since the high number of instantiated Java Objects within a UDF is a critical factor determining the general applicability of the *meta-graph extraction* framework approach, a dedicated evaluation of optimization techniques of the *PCP Evaluation* algorithm could be conducted as future work. Flink implements it's algorithms used for the execution of parallel dataflows not against Java objects but instead serializes the data into byte representations. It would be interesting to see, whether it is possible to convert the *PCP Evaluation* algorithm such that it operates in a similar way and how the performance can be improved by this means.

One main advantage of implementing the *meta-graph extraction* framework for a HIN is that it compiles a meta path to a PCP and finally computes the result based on this plan. Consequently, it is conceptually generic applicable to all meta paths defined on the respective network schema, similar to the capabilities of a query language like Cypher. We compromise the general applicability, because of its dependence on the actual workload within individual UDFs and thus on the selected meta paths and the data distribution as well. We confirmed this statement at the example of Flink. As stated before we claim that this a JVM system based problem and not limited to Flink. Further research is needed to prove this hypothesis.

Furthermore, we showed that it is beneficial in this context to use Neo4j for the partial paths computation and additionally leveraging Flink's *Join* operator for the partial path concatenation on the *term* nodes. The benefit is that we are using Flink's memory optimization and *out-of-core* processing capabilities instead of generating full paths in the *PCP evaluation* algorithm executed in Flink's User Memory. Hereby, we are able to successfully compute the path counts for all considered meta paths. We showed that the runtimes and thus the scalability is still impacted by *peak* nodes in the SAP Community network, while the successful execution is possible for potentially arbitrary selective meta paths in this way. We suggest to examine how to distribute the workload of peak nodes more evenly within the cluster in order to ensure an even resource utilization and consequently good scalability characteristics. We observed runtimes of 2m3s for the meta paths BUTUB and 2m27s for BUTB in this approach, whereas the computation lasted 24m for BUTUB and 25m for BUTB when using the framework with a similar resource employment. Hence, the computation in the hybrid approach is more efficient by a factor of ten for the two aforementioned paths and enables the computation of all paths, which were not computable at all with the implemented framework. The drawback of this approach is that it needs to be implemented for each meta path individually and hence, we are loosing the generic query capability. As we pointed out before, this generality is not unrestrictedly given anyway at least for the considered dataset and the used distributed processing system. Moreover, our framework prototype implementation consists of over 2700 lines of code (excl. comments, newlines etc.), while we do not yet have implemented the *automated PCP creation* feature, which would be the basis for the generic applicability. In contrast, less than 100 lines of code were needed for implementing the alternative approach for the meta path BTB. The meta paths used within the recommendation system won't change daily. Thus, it is reasonable to give up the generic query feature provided by a full implementation of the *meta-graph extraction* framework. Since the recommendation generation is just one use case of meta paths, it is eventually a business decision how much flexibility is needed.

Additionally, we demonstrated the usefulness of data exploration and data cleaning at the example of the meta path BTB. We showed that the two most occurring *terms* are semantically not meaningful, while resulting in approximately 90% of all BTB path instances. It would be interesting to see, how the removal of high degree nodes actually impacts the final recommendation quality. It could be theoretically possible, that the recommendation quality is negatively impacted by those nodes and the exclusion of such is beneficial. As a consequence, the data distribution is more uniformly and the resource utilization in the cluster is improved.

9 Conclusion

In this thesis, we addressed the problem of efficiently computing meta paths defined on the SAP Community network schema. The result sets can then be represented as edge homogeneous graphs with the respective meta path counts as edge values. The actual computation of meta path based similarities within the Recommender System is strongly simplified, because the graph analysis was executed beforehand. We therefor formalized the Path Count Computation Problem and analyzed the applicability and the limits of Neo4j for computing the path counts of the considered meta paths. We saw that Neo4j was indeed capable of computing semantically interesting meta paths, while it is limited by its available memory and thus exhibits its limits when computing high selective paths. Thus, we could confirm hypothesis 2. Furthermore, we assumed that we will be able to compute arbitrary meta paths by using a *meta-graph extraction* framework implementation based on Apache Flink that will outperform Neo4j by scaling the Flink cluster horizontally. We could confirm *Hypothesis 1* to the extent that the framework implementation indeed exhibited shorter runtimes and better scalability characteristics as the graph DB based approach, since we observed linear scalability for the meta path BUTB and BUTUB. But the general applicability of the framework on the SAP Community knowledge graph could not be determined, since it is prone to Garbage Collection Stalls and *OutOfMemory* errors for high selective meta paths. We pointed out that all considered meta paths exhibit a long tail distribution of partial paths and relations around the *term* node, which results in a significantly higher number of path instances around those peak nodes and consequently in too high workloads within individual UDFs.

In order to address the problems implied by peak nodes, we demonstrated the usefulness of data exploration and data cleaning to significantly reduce the total amount of path instances and a more even resource distribution in the cluster on the one hand. On the other hand, we propose a hybrid approach for computing path counts leveraging Neo4j for efficiently computing partial paths in combination with Flink's dataset transformation operators for concatenating these paths on the *term* nodes and eventually computing the final path count values in a data-parallel manner. Hereby, the computation for all defined meta paths is possible, even if the available memory in a specific worker node is not sufficient. Furthermore, we demonstrated that the hybrid approach outperformed the *meta-graph extraction* framework implementation approximately by a factor of 10 for the meta paths BUTB and BUTUB.

Although the *meta-graph extraction* framework proposed by Shao et al. did not fully met our expectations, the results of this thesis enable SAP to build a state-of-the art graph analysis pipeline for efficiently computing meta paths counts on the SAP Community knowledge graph in a distributed manner. Consequently, we provide the basis for generating meta path based content recommendations on the SAP Community platform.

References

[1] A. Agarwal, M. Chauhan, and Ghaziabad. Similarity measures used in recommender systems : A study. *International Journal of Engineering Technology Science and Research*, 4(6), 2017. ISSN 2394–3386.

[2] Agha and G. Abdulnabi. Actors: A model of concurrent computation in distributed systems, 1985. URL `https://dspace.mit.edu/handle/1721.1/6952`. Last visited 19-11-2018.

[3] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014. ISSN 1066-8888. doi: 10.1007/s00778-014-0357-y.

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In Unknown, editor, *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, page 483, New York, New York, USA, 1967. ACM Press. doi: 10.1145/1465482.1465560.

[5] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5): 1–40, 2017. ISSN 03600300. doi: 10.1145/3104031.

[6] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts. In J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing*, page 119, New York, NY, 2010. ACM Press. ISBN 9781450300360. doi: 10.1145/1807128.1807148.

[7] J. Beel, B. Gipp, S. Langer, and C. Breitinger. Research-paper recommender systems: a literature survey. *International Journal on Digital Libraries*, 17(4):305–338, 2016. ISSN 1432-1300. doi: 10.1007/s00799-015-0156-0.

[8] C. Boden, A. Spina, T. Rabl, and V. Markl. Benchmarking data flow systems for scalable machine learning. In Unknown, F. Afrati, J. Sroka, and P. Koutris, editors, *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR 2017)*, pages 1–10, New York, NY, 2017. The Association for Computing Machinery, Inc. ISBN 9781450350198. doi: 10.1145/3070607.3070612.

[9] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998. ISSN 01697552. doi: 10.1016/S0169-7552(98)00110-X.

[10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38 (4):28–38, 2015. URL http://sites.computer.org/debull/A15dec/p28.pdf.

[11] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017. ISSN 21508097. doi: 10.14778/3137765.3137777.

[12] R. Cheng, Z. Huang, Y. Zheng, J. Yan, K. Y. Wong, and E. Ng. Meta paths and meta structures: Analysing large heterogeneous information networks. In L. Chen, C. S. Jensen, C. Shahabi, X. Yang, and X. Lian, editors, *Web and Big Data*, pages 3–7, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63579-8.

[13] J. Dean and S. Ghemawat. Mapreduce. *Communications of the ACM*, 51(1):107, 2008. ISSN 00010782. doi: 10.1145/1327452.1327492.

[14] S. Ewen. Apache Flink internals: Akka and actors, 2015. URL https://cwiki.apache.org/confluence/display/FLINK/Akka+and+Actors. Last visited 21-09-2018.

[15] S. Ewen. Apache Flink internals: Memory management (batch api), 2015. URL https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=53741525. Last visited 22-09-2018.

[16] S. Ewen. Apache Flink user mailing list archive: Gelly ran out of memory, 2015. URL http://apache-flink-user-mailing-list-archive.2336050.n4.nabble.com/Gelly-ran-out-of-memory-td2462.html. Last visited 29-09-2018.

[17] S. Ewen. Off-heap memory in Apache Flink and the curious jit compiler, 2015. URL https://flink.apache.org/news/2015/09/16/off-heap-memory.html. Last visited 05-10-2018.

[18] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment (PVLDB)*, 2012. URL http://arxiv.org/pdf/1208.0088v1.

[19] Gartner Inc. Gartner IT Glossary - Big Data, 2018. URL https://www.gartner.com/it-glossary/big-data. Last visited 03-12-2018.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In M. L. Scott and L. Peterson, editors, *SOSP'03*, page 29, New York, N.Y., op. 2003. Association for Computing Machinery. ISBN 1581137575. doi: 10.1145/945445.945450.

[21] A. Grove, editor. *Information science with impact: research in and for the community: ASIST 2015 : proceedings of the 78th ASIS & T Annual Meeting, volume 52, 2015*. Association for Information Science and Technology, 2015. ISBN 087715547X.

[22] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5): 532–533, 1988. ISSN 00010782. doi: 10.1145/42411.42415.

[23] W. Hochstättler. *Algorithmische Mathematik*. Springer-Verlag, Berlin and Heidelberg, 2010. ISBN 3642054218.

[24] W. Huber, V. J. Carey, L. Long, S. Falcon, and R. Gentleman. Graphs in molecular biology. *BMC bioinformatics*, 8 Suppl 6:S8, 2007. doi: 10.1186/1471-2105-8-S6-S8.

[25] F. Hüske. Apache Flink: Peeking into Apache Flink's engine room, 2015. URL `https://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html`. Last visited 01-11-2018.

[26] F. Hüske. Apache Flink user mailing list archive: Delta iteration not spilling to disk, 2017. URL `http://mail-archives.apache.org/mod_mbox/flink-user/201710.mbox/%3CCAAdrtT1SKV=Tmd+ap3WuLMyB9HX5LedY0WveGGx5Zsw9eXO6GA@mail.gmail.com%3E`. Last visited 01-11-2018.

[27] IBM Corporation. The state of graph databases: Worldwide adoption and use case characteristics. 2017. URL `https://public.dhe.ibm.com/common/ssi/ecm/im/en/imm14205usen/hybrid-cloud-analytics-platform-im-e-book-imm14205usen-20180426.pdf`. Last visited 01-06-2018.

[28] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi. Big data and its technical challenges. *Communications of the ACM*, 57(7):86–94, 2014. ISSN 00010782. doi: 10.1145/2611567.

[29] G. Jeh and J. Widom. Simrank. In D. J. Hand, D. Keim, and R. Ng, editors, *KDD-2002*, page 538, New York, New York, USA, 2002. ACM Press. ISBN 158113567X. doi: 10.1145/775047.775126.

[30] G. Jeh and J. Widom. Scaling personalized web search. In G. Hencsey, B. White, Y.-F. R. Chen, L. Kovács, and S. Lawrence, editors, *Proceedings of the twelfth international conference on World Wide Web - WWW '03*, page 271, New York, New York, USA, 2003. ACM Press. ISBN 1581136803. doi: 10.1145/775152.775191.

[31] A. Katsifodimos and S. Schelter. Apache Flink: Stream analytics at scale. In I. I. C. o. C. Engineering, editor, *2016 IEEE International Conference on Cloud Engineering workshops*, page 193, Piscataway, NJ, 2016. IEEE. ISBN 978-1-5090-3684-4. doi: 10.1109/IC2EW.2016.56.

[32] G. Laakmann McDowell. *Cracking the coding interview: 189 programming interview questions and solutions*. CareerCup, Palo Alto, 6th ed. edition, 2015. ISBN 9780984782857.

[33] Lightbend Inc. Akka, 2018. URL `https://akka.io/`. Last visited 21-09-2018.

[34] T. Lindaaker. An overview of neo4j internals, 2012. URL `https://www.slideshare.net/thobe/an-overview-of-neo4j-internals`. Last visited 19-11-2018.

[35] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the 2010 international conference on Management of data*, page 135, New York, New York, USA, 2010. ACM Press. ISBN 9781450300322. doi: 10.1145/1807167.1807184.

[36] M. D. McCool, A. D. Robison, and J. Reinders. *Structured parallel programing: Patterns for efficient computation*. Elsevier Morgan Kaufmann, Amsterdam, 2012. ISBN 9780124159938.

[37] Neo4j Inc. Developer manual: Uniqueness, n.d.. URL `https://neo4j.com/docs/developer-manual/current/cypher/introduction/uniqueness/`. Last visited 22-10-2018.

[38] Neo4j Inc. Developer manual: Skip clause, n.d.. URL `https://neo4j.com/docs/developer-manual/current/cypher/clauses/skip/`. Last visited 07-11-2018.

[39] Oracle Corporation. Virtual machine garbage collection tuning, n.d.. URL `https://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#par_gc.oom`. Last visited 29-09-2018.

[40] Oracle Corporation. Sun java system application server 9.1 performance tuning guide: Managing memory and garbage collection, n.d.. URL `https://docs.oracle.com/cd/E19159-01/819-3681/6n5srlhqf/index.html`. Last visited 27-09-2018.

[41] Oracle Corporation. Java platform se 8: Future, n.d.. URL `https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/Future.html`. Last visited 27-09-2018.

[42] Oracle Corporation. Java garbage collection basics, n.d.. URL `https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html`. Last visited 03-11-2018.

[43] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, Heidelberg, 5. aufl. edition, 2012. ISBN 3827428033.

[44] V. Patil, R. Vasappanavara, and T. Ghorpade. Comparative analysis of similarity measures in heterogeneous information network. In I. C. o. I. S. a. Control, editor, *ICSO 2017*, pages 297–301, [Piscataway, NJ], 2017. IEEE. ISBN 978-1-5090-2717-0. doi: 10.1109/ISCO.2017.7856002.

[45] J. Pokorný. Graph databases: Their power and limitations. In K. Saeed and W. Homenda, editors, *Computer information systems and industrial management*, volume 9339 of *LNCS sublibrary: SL 3 - Information systems and application, incl. Internet/Web and HCI*, pages 58–69. Springer, Cham, 2015. ISBN 978-3-319-24368-9. doi: 10.1007/978-3-319-24369-6_5.

[46] A. Rajaraman and J. D. Ullman. Large-scale file systems and map-reduce. In A. Rajaraman and J. D. Ullman, editors, *Mining of Massive Datasets*, pages 18–52. Cambridge University Press, Cambridge, 2011. ISBN 9781139058452. doi: 10.1017/CBO9781139058452.003.

[47] J. Reinders. Understanding task and data parallelism, 2007. URL `https://www.zdnet.com/article/understanding-task-and-data-parallelism-3039289129/`. Last visited 19-11-2018.

[48] I. Robinson, J. Webber, and E. Eifrem. *Graph databases*. O'Reilly, Sebastopol, Cal., second edition edition, 2015. ISBN 9781491930892.

[49] S. Schelter. *Scaling Data Mining in Massively Parallel Dataflow Systems*. Dissertation, Technische Universität Berlin, 2016.

[50] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, Boston, Mass. and London, 4th ed. edition, 2011. ISBN 032157351X.

[51] Y. Shao, K. Lei, L. Chen, Z. Huang, B. Cui, Z. Liu, Y. Tong, and J. Xu. Fast parallel path concatenation for graph extraction. *IEEE Transactions on Knowledge and Data Engineering*, 29(10):2210–2222, 2017. ISSN 1041-4347. doi: 10.1109/TKDE.2017.2716939.

[52] C. Shi, Y. Li, J. Zhang, Y. Sun, and P. S. Yu. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering*, 29(1):17–37, 2017. ISSN 1041-4347. doi: 10.1109/TKDE.2016.2598561.

[53] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In M. G. Khatib, editor, *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010*, pages 1–10, Piscataway, NJ, 5/3/2010 - 5/7/2010. IEEE. ISBN 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496972.

[54] solidIT GmbH. Db-engines ranking: popularity ranking of database management systems, 2018. URL https://db-engines.com/en/ranking. Last visited 18-07-2018.

[55] STFC Computational Science and Engineering Department. The weak scaling of dl_poly 3, n.d. URL https://web.archive.org/web/20140307224104/http://www.stfc.ac.uk/cse/25052.aspx. Last visited 23-08-2018.

[56] Y. Sun and J. Han. Mining heterogeneous information networks. *ACM SIGKDD Explorations Newsletter*, 14(2):20, 2013. ISSN 19310145. doi: 10.1145/2481244.2481248.

[57] Y. Sun and J. Han. Meta-path-based search and mining in heterogeneous information networks. *Tsinghua Science and Technology*, 18(4):329–338, 2013. doi: 10.1109/TST.2013.6574671.

[58] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *ACM SIGKDD Explorations Newsletter*, 2011. ISSN 19310145. URL http://www.vldb.org/pvldb/vol4/p992-sun.pdf.

[59] The Apache Software Foundation. Apache Giraph, 2018. URL http://giraph.apache.org/. Last visited 20-07-2018.

[60] The Apache Software Foundation. GraphX - Apache Spark's API, 2018. URL https://spark.apache.org/graphx/. Last visited 03-12-2018.

[61] The Apache Software Foundation. Apache Flink, n.d.. URL https://flink.apache.org/. Last visited 20-09-2018.

[62] The Apache Software Foundation. Apache Flink 1.4 documentation: Iterations, n.d.. URL https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/batch/iterations.html#delta-iterate-operator. Last visited 25-07-2018.

[63] The Apache Software Foundation. Apache Flink 1.4 documentation: Distributed runtime environment, n.d.. URL https://ci.apache.org/projects/flink/flink-docs-release-1.4/concepts/runtime.html. Last visited 19-09-2018.

[64] The Apache Software Foundation. Apache flink 1.4 documentation: Configuration, n.d.. URL https://ci.apache.org/projects/flink/flink-docs-release-1.4/ops/config.html#distributed-coordination-via-akka. Last visited 14-06-2018.

[65] The Apache Software Foundation. Apache Flink 1.4 documentation: Iterative graph processing, n.d.. URL https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/libs/gelly/iterative_graph_processing.html. Last visited 20-05-2018.

[66] J.-F. Weets, M. K. Kakhani, and A. Kumar. Limitations and challenges of hdfs and mapreduce. In ICGCIoT, editor, *Proceedings of the 2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, pages 545–549, [Piscataway, NJ], 2015. IEEE. ISBN 978-1-4673-7910-6. doi: 10.1109/ICGCIoT.2015.7380524.

[67] Wikipedia. Bulk synchronous parallel, n.d. URL `https://en.wikipedia.org/w/index.php?oldid=849793142`. Last visited 23-07-2018.

Appendix

Big Data Processing Fundamentals

We are now at the beginning of the so called big data era. The term refers to the fact that high volumes of high-variety information assets being generated with high velocity [19]. A lot of the economical sectors progressively change to a *data-driven* decision making model based on the analysis of large datasets [28]. The demands for data processing changed with the rise of the Internet in the 90s. At the beginning, the search engine companies generated a demand for large scale data processing to index and rank web pages. Google developed the *MapReduce* framework in this context, which can be seen as the god father of distributed data processing. It was based on the idea to develop a fault-tolerant framework for large scale, distributed data processing on clusters consisting of potentially 1000s of nodes. This means that those big data processing systems scale horizontally by parallelizing the load on multiple machines based on commodity hardware instead of scaling vertically by using *super computers* [46]. Furthermore, the framework should abstract and hide the *messy details of parallelization, fault-tolerance, data distribution and load balancing in a library* [13]. The popular *Apache Hadoop* Framework was developed as an open-source project based on the MapReduce publication by Google. Apache Hadoop comprises the parallel data processing framework MapReduce and moreover the Hadoop Distributed File System (HDFS). The HDFS is a scalable, distributed file system and was developed after the publication of the Google File System (GFS) [53], [20]. As a 1st generation big data processing system, Hadoop had several drawbacks. Although, the MapReduce paradigm is simple, it is not easy to program, since the developer needs to find an algorithm for its problems that can be expressed by the low level operators. A further drawback is the lacking support for iterative computations [66]. The increasing popularity of distributed data processing for use cases like machine learning and graph analysis led to the development of next generation big data systems like Pregel, Apache Flink or Apache Spark for example. Such use cases often rely on iterative computations [18].

Distributed File System

In the following, we will briefly present the *Google File System* (GFS) and the *Hadoop Distributed File System* (HDFS), which is modeled after the GFS. The basic concepts are

therefore similar in both systems. The HDFS is as an popular, open-source example of a distributed file system (DFS) whereas the GFS is an proprietary solution from Google. Distributed processing frameworks usually work on a DFS like MapReduce works sits on top of the HDFS. The HDFS is designed to reliably store large datasets distributed across multiple nodes in a shared-nothing cluster, i.e. nodes in the cluster are connected via a network and do not share any resources. Moreover, single machine failures are expected and anticipated by the design. Files can potentially be very large which means possibly multi-terabyte in size. The data is split up in chunks of the same size (several mega bytes) and can be replicated to several nodes, to realize fault tolerance and possibly improve data locality for the processing systems usually sitting on top of it. Instead of perfectly supporting random access to files or the frequent modification of such, the HFDS is designed for sequentially writing and stream reading large files. The write operations High throughput for bulk reads is preferred over low latency for individual file accesses [49].

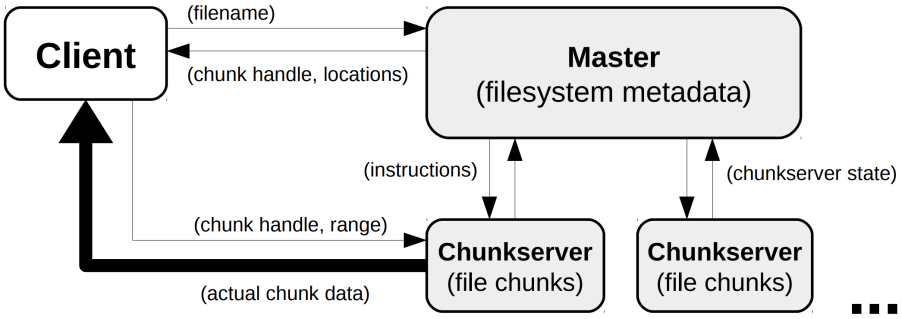


Figure 48: Communication patterns in GFS [49]

The GFS is modeled in a master-slave architecture. The master is in charge of the file system meta data, handles client requests as first point of contact and coordinates its slaves. The slaves store the actual data chunks. The degree of replication of the data and the master server can be freely configured to improve fault tolerance and availability. The number of slave nodes can be easily scaled by design. Figure 48 shows the communication pattern in the DFS. The master instructs the slaves to replicate the data and monitors the health via heartbeat mechanism. A client requests a file by its name from the master node. The master responds by sending the location of the data back. Next, the client will contact the respective chunk servers and stream reads the requested file. The master is called *Namenode* and the slaves *Datanodes* in HDFS instead of *Master* and *Chunkserver* in GFS [53], [20].

MapReduce

MapReduce is a programming model for distributed data processing in a data parallel manner and a concrete implementation of such developed by Google [13]. As stated before, Apache Hadoop is the open-source implementation consisting of MapReduce and the HDFS. The processing framework is based on two second-order functions *Map* and *Reduce* that take two user-defined first-order functions (UDF) f_m and f_r as input [49]. MapReduce operates on a simple *key-value* data model. The function f_m gets one key/value-pair with key k_1 and value v_1 as input and generates a set of 0-n key/value-pairs as output with a different key k_2 and a different value v_2. It should be noted that f_m can produce multiple tuples each having a different key and value.

$$f_m : (k_1, v_1) \rightarrow list(k_2, v_2)$$

The function f_r gets all tuples with set same key as input in the form $(k_2, list(v_2))$ produces a condensed output.

$$f_r : (k_2, list(v_2)) \rightarrow list(v_2)$$

Many real world tasks can be mapped to this model and it was originally used by Google to rank web sites, count URL access frequencies and create inverted indices [13].

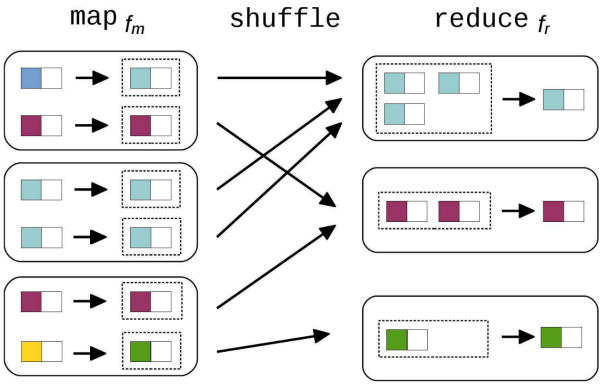


Figure 49: Dataflow of key-value tuples in MapReduce [49]

MapReduce is designed as *master-slave* architecture with similar design goals like the GFS good scalability, fault tolerance and consequently high parallel processing capabilities. The master schedules the Map and Reduce tasks on the slaves and responds to possible node failures. The slave machines are called worker nodes in MapReduce. Each worker node may execute multiple tasks in parallel. A MapReduce job consists of three phases: 1. *Map* 2. *Shuffle* 3. *Reduce*
The data is usually read as chunks from a DFS, partitioned by the second-order function

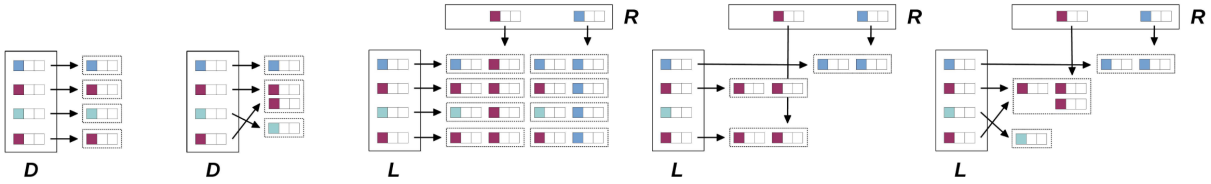
Map and handed over as key-value pair (k_1, v_1) to f_m. The master tries to exploit data locality and therefore schedules the tasks where the data is actually stored if possible. Typically, the input key to f_m is irrelevant for the execution. The intention is to to be able to chain the execution of multiple jobs such that the successor job can work on the result of its predecessor. Each Mapper may produce multiple key-value pairs (k_2, v_2) with potentially different keys. The key k_2 of those output pairs is defined by the UDF. The Shuffle phase sorts the created tuples by the newly created key and sends all tuples with the same key to one Reducer. Hence, f_r works on a full set of tuples with the same key. This means, the Reduce UDF is applied on each grouped subset $list((k_2, v_2))$ in parallel and creates a condensed output $list(v_2)$.
The maximum degree of parallelism for each phase can be inferred. We can have at most as much Mapper tasks in parallel as the total number of input tuples, since the Map UDF is executed on a per tuple basis. Likewise the maximum amount of parallel reduce tasks is limited by the count of different keys created in the Map phase. It should be noted that it is not always the best to use the maximum possible degree of parallelism especially for the reducer. On the one hand there is overhead associated with each parallel task and on the other the distribution of tuples in the grouped sets is often significantly skewed. As a consequence the runtimes for the different Reducers might vary. Thus, it might make sense depending on the actual distribution to have less Reducers than the count of distinct keys to better utilize the available resources [46].

A simple example for illustrating the MapReduce principles is the counting of occurrences per word in a file. We have an input file where each line contains multiple words. We want to conceptually crate a MapReduce job for counting how often each word appears in the whole file. Each Map task will get one line as input with its line number as key or simply a constant since the job will not use this key. The Map UDF will split up the line per word and creates output tuples, such that each word will be the new key and the value is simply set to 1. As a consequence, each Reduce task will get all tuples with the same word as key. The Reduce UDF will then just count the number of tuples and creates one output tuple with the respective word as key and the number of occurrences as value. Regarding the degree of parallelism we can have at most as much Mappers as lines in the file in parallel and at not more Reducers than the count of distinct words in the whole file.

Although, we said a lot of problems can be expressed as MapReduce jobs, there are still uses cases which can not or at least not efficiently be dealt with by just using a Map and a Reduce function. Moreover, MapReduce uses a *disk-backed execution* strategy, i.e. the result of a job is always written to disk which implies a reduced performance when scheduling a successor task based on the result of a previous one as compared to keeping the result in memory.

Parallel Dataflows at the Example of Apache Flink

In the following, we will present the concept of *Parallelization Contracts* (Pacts), which is the core abstraction for parallel data processing of *Apache Flink* [3], [49]. On the one hand, Flink is a representative of a state-of-the-art big data processing system and on the other, we will base our further experiments on it. As a state-of-the-art system, Flink offers in-memory data processing, program optimization capabilities and native support for iterative computations.
The Pact programming model extends the MapReduce paradigm discussed in section 9 and additionally provides further operators to work on two datasets as compared to just one in Map and Reduce. Flink treats UDFs as first-class citizens and works in a data-parallel manner like MapReduce. The data is partitioned in memory across worker nodes and the computation is sent to those. A Pact operator is a second-order function that takes a first-order UDF as input and moreover specifies the partitioning of the dataset in memory as illustrated in figure 50. The UDF is then applied to each group retrieved from the partitioning.



(a) Map (b) Reduce (c) Cross (d) Join (e) CoGroup

Figure 50: Second-order functions forming the Pact operators [49]

Flink offers the same unary operators like MapReduce, which work on a single dataset $D = \{t_1, ..., t_n\}$, with n tuples. A tuple $r_i = \{v_1, ..., v_m\}$ itself consists of m unordered values. The types of the values is only relevant in the UDF and not for the respective Pact operator.

Figure 50 shows how the data will be partitioned when using the respective second-order functions. The resulting partitioning can be found at the target of the arrows. The first-order UDF will be applied on each partition visually grouped by a container at the respective arrow heads. Thus, those sets are the most granular level for parallelization and the number of sets determines the maximum degree of parallelism possible.

$Map : D \times f \rightarrow \{f(t_1), ..., f(t_n)\}$

The Map operator (cf. figure 50a) applies the UDF to each tuple $t_i \in D$, hence the partitioning can be freely chosen on a per tuple basis. The maximum degree of parallelism is equal to the number of tuples in D.

$$Reduce : D \times f \times K \rightarrow \{f(t_{1_1}^{(k_1)}, ..., t_{n_1}^{(k_1)}), ..., f(t_{1_t}^{(k_t)}, ..., t_{n_t}^{(k_t)})\}$$

The reduce operator (cf. figure 50b) works on a user defined key set $K = (k_1, ..., k_t)$ defined over attributes of D, where t is the number of distinct keys and $r.K = K$. The UDF is applied on each subset of D, while all tuples in a subset share the same key $k_i \in K$. The maximum parallelism is determined by t.

Moreover, Flink provides several binary operators. In other words, those operators work on two datasets $L = \{l_1, ..., l_p\}$ with n as the number of elements in L and $R = \{r_1, ..., r_q\}$ with n as the number of elements in R. Consequently, the respective UDF will get two inputs instead of one compared to the unary operators.

$$Cross : L \times R \times f \rightarrow \{f(l_1, r_1), ..., f(l_1, r_q), f(l_2, r_1), ..., f(l_p, r_q)\}$$

The Cross operator (cf. figure 50c) applies the UDF on each tuple pair resulting from the Cartesian product of L and R.

$$Join : L \times R \times K_L \times K_R \times f \rightarrow \{f(l, r) \mid l.K_L = r.K_R\}$$

The Join operator (cf. figure 50d) works on two key sets K_L and K_R. Each set is defined analogously to K for the Reduce operator. K_L is defined over attributes of L and K_R is defined over attributes of R. The UDF is applied on each tuple resulting from an *equi join* between the 2 datasets where $l.K_L = r.K_R$.

$$CoGroup : L \times R \times K_L \times K_R \times f \rightarrow \{f(l_{1_1}^{(w_1)}, ..., l_{n_1}^{(w_1)}, r_{1_1}^{(w_1)}, ..., r_{n_1}^{(w_1)}), ..., f(l_{1_t}^{(w_1)}, ..., l_{n_t}^{(w_t)}, r_{1_t}^{(w_t)}, ..., r_{n_t}^{(w_t)})\}$$

The CoGroup operator (cf. figure 50e) is the binary equivalent to Reduce. L and R will be grouped by their respective key spaces. The UDF is then applied on each group pair from both sets with the same key $w_i \in W$. If there is one key W_i which is only available in one group, then the UDF is applied only the set with the respective key, similar like the working of an outer join.

A Flink job, i.e. a Pact job or more generically a parallel dataflow can be illustrated as a *directed acyclic graph* (DAG), where the vertices correspond to the Pact operators with its respective UDF and the edges illustrate the data flow between them. Based on such a representation Flink will optimize the execution strategy. Flink's optimization techniques are based on those developed for parallel databases, namely logical execution plan equivalences in combination with a cost based optimization model. Moreover, Flink

tries to incorporate further interesting properties like partitioning or sort order to finally create a globally efficient plan [6].

Many relevant use cases as present in machine learning or graph processing require iterative computations. Generically stated, an iterative computation is a step function f with an initial state s^0. The function f is repeatedly executed until $s^* = f(s^*)$. We distinguish between two types of iterative computations [18]:

Bulk Iterations apply the step function to the complete partial solution set S^i, such that $S^{i+1} = f(S^i)$. Thus, a complete new partial solution set is created after each iteration based on the processing of the complete result set of from the previous iteration.

Incremental Iterations in contrast are based on the idea to only work on parts of an intermediate result to reduce the workload. Many use cases imply an algorithmic property called *sparse computational dependency*. In other words, we usually do not create e complete new partial solution, instead typically an update of an element only effects some other ones. As a consequence, different parts of the computation converge at different velocities. Therefore, one can use the concept of a *working set* W in addition to the solution set s. The solution set holds the intermediate or final result, whereas the working set holds data algorithm-specific data for computing updates to S. Ewen et al. introduce two functions for that reason, instead of just one single step function [18]. The function u which computes the next partial solution $S_{i+1} = u(W^i, S^i)$ and a function δ for computing the working set $W^{i+1} = \delta(W^i, S^i)$. The iteration terminates, when the working set is empty.

Depending on the use case, the efficiency between the two iteration types can vary significantly. For example, the machine learning algorithm gradient descent is well suited for the bulk iteration, since the complete partial solution is updated based on the previous results. Hence, incremental iterations won't make sense here. In opposition, graph algorithms like Connected Components will benefit from the incremental model, because the working set will often be smaller than the result of the previous partial solution.

As mentioned earlier, a Pact program can be represented as a DAG. Flink offers dedicated support for iterative parallel dataflows by specifying parts of the graph as iterative [18]. The output of the last operator will be fed into the first operator of the iterative part. Obviously, the graph won't be acyclic anymore. The iterative part can be seen as the step function, that consists of potentially multiple Pact operators. Flink supports bulk iterations as well as incremental ones called *Delta iterations*.

Figure 51 visualizes bulk iterations in Flink. The step function will be applied on the full initial input dataset or on the partial solution (from the previous iteration) and finally

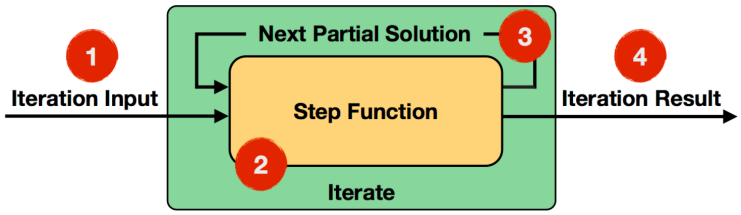


Figure 51: Bulk Iterations in Apache Flink [62]

produces the iteration result. The iteration ends either after a fixed number of iterations or when a custom defined convergence criteria is met

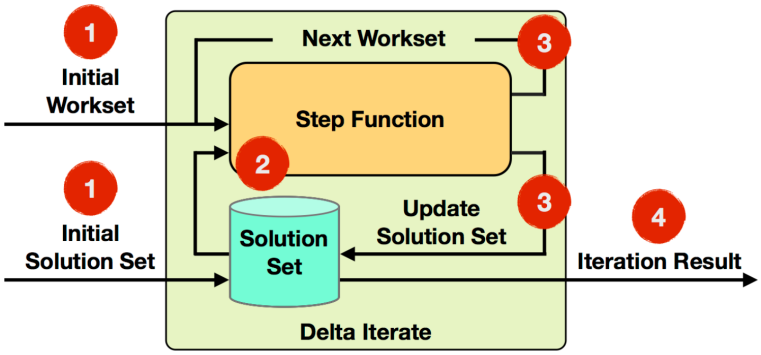


Figure 52: Delta Iterations in Apache Flink [62]

Figure 52 illustrates Delta Iterations in Apache Flink. We have to datasets. The *workset* and the *solution set*. The step includes encapsulates the functionality of u and δ as introduced by incremental iterations. The solution is evolved by selectively updating elements instead of fully recomputing a partial solution. The default convergence criteria is either an empty workset or a specified number of maximum iterations.

PCP Evaluation Algorithm

Algorithm 1 PCP Evaluation Pseudocode

Input: PCP $plan$, The height of PCP H, Aggregate functions $\otimes$ and $\oplus$, Heterogeneous graph G_{he}
Output: The extracted homogeneous graph G_{ho}

foreach vertex v in G_{he} **do**
  preprocessing of materializing both out/in neighbors of v at local;
end foreach{preprocessing}
foreach $h \leftarrow H$ to 1 **do**
  $ppSet \leftarrow plan.getPPbyLevel(h)$;
  foreach vertex v in G_{he} **do**
    foreach primitive pattern pp in $ppSet$ **do**
      evaluate pp on vertex v by Algorithms 2;
    end foreach
  end foreach
end foreach{path eumeration}
foreach vertex v_e in G_{he} **do**
  $PS \leftarrow$ get paths end with v_e and index the paths by their start vertices v_s;
  foreach vertex v_s **do**
    $path_{v_s \to v_e} \leftarrow$ get paths start with v_s from PS;
    $aggVal \leftarrow \phi$;
    foreach path pa in $path_{v_s \to v_e}$ **do**
      $pathVal \leftarrow \otimes w(e_i), e_i \in pa$;
      $aggVal \leftarrow val \oplus pathVal$
    end foreach
    insert extracted relation $(v_s, v_e, aggVal)$ into G_{ho}.
  end foreach
end foreach{pair-wise aggregation}
return G_{ho};

Algorithm 2 Primitive Pattern Evaluation Pesudocode

Input: Vertex $v \in G_{he}$, Primitive Pattern pp

initialize v_s, v_p, v_e, e_l, e_r by pp;
if $f(v) = f_p(v_p)$ **then**
  $leftRes \leftarrow NULL; rightRes \leftarrow NULL$;
  if $f_p(v_s) \in NL$ **then**
    $leftRes \leftarrow$ matching e_l to the v's neighbors in G_{he};
  else
    $leftRes \leftarrow$ retrieving the results of primitive pattern $id = f_p(v_s)$;
  end if
  if $f_p(v_e) \in NL$ **then**
    $rightRes \leftarrow$ matching e_r to the v's neighbors in G_{he};
  else
    $rightRes \leftarrow$ retrieving the results of primitive pattern $id = f_p(v_e)$;
  end if
  concatenate $leftRes$ and $rightRes$;
  if pp is the left child **then**
    send the new paths to the end vertex;
  else
    send the new paths to the start vertex;
  end if
end if

Figure 53: PCP Evaluation Algorithm Pseudocode [51]

Data Distributions

Term	BT Count	BTB Count
English	90919	4133086821
Retagging required	30218	456548653
sap	6958	24203403
SAP HANA	6269	19647046
hana	5658	16003653
Cloud	5467	14941311
Mobile	4601	10582300
code	4418	9757153
Analytics	4411	9726255
sapmentor	4218	8893653
Business Trends	4050	8199225
Training	3954	7815081
abap	3243	5256903
ABAP Development	2718	3692403
Solution	2708	3665278
Business Intelligence	2683	3597903
innovation	2506	3138765
HCM (Human Capital Management)	2143	2295153
BW (SAP Business Warehouse)	2122	2250381
sapmentors	2102	2208151
SAP ERP	2051	2102275
Human Resources	2048	2096128
University Alliances	1920	1842240
SAPPHIRE NOW	1893	1790778
mobility	1862	1732591
SAP Enterprise Portal	1804	1626306
Big Data	1789	1599366
SAP TechEd	1727	1490401
SAP BusinessObjects Business Intelligence platform	1724	1485226
enterprise resource planning	1617	1306536

Table 23: BTB Distribution Approximation

Term	CT Count	CTC Count
abap	240472	28913271156
English	90923	4133450503
Customer Relationship Management	59091	1745843595
enterprise resource planning	52756	1391571390
Retagging required	30218	456548653
sap	17556	154097790
SAP Business One	17329	150138456
sap erp financials	14953	111788628
sap erp logistics materials management	14315	102452455
ABAP Development	11793	69531528
Master Data Management	11606	67343815
SAP HANA	11005	60549510
sap erp sales and distribution	10519	55319421
Mobile	9534	45443811
SAPUI5	9220	42499590
hana	9142	41783511
Database	7035	24742095
Cloud	6912	23884416
SAP Solution Manager	6626	21948625
sap erp manufacturing production planning	6405	20508810
Business Intelligence	6034	18201561
sap erp human capital management	5880	17284260
SAP Business Planning and Consolidation	5858	17155153
SAP Process Integration	5450	14848525
Web Dynpro	5168	13351528
fiori	4875	11880375
Analytics	4871	11860885
code	4819	11608971
sapmentor	4711	11094405
SAP NetWeaver	4364	9520066

Table 24: CTC Distribution Approximation

Term	BUT Count	BUTUB Count
SAP BusinessObjects Cloud	1922	1846081
SAP BusinessObjects Business Intelligence platform	1906	1815465
SAP BusinessObjects Design Studio	1837	1686366
SBOP Web/Desktop Intelligence (client/server)	1787	1595791
SAP BusinessObjects Lumira	1744	1519896
SAP BusinessObjects Analysis edition for Microsoft Office	1593	1268028
Using SAP.com	1373	941878
SAP HANA Cloud Platform	1282	821121
SAP HANA	1022	521731
SAP Fiori	968	468028
ABAP Development	857	366796
SAPUI5	854	364231
Business Trends	755	284635
SAP S/4HANA	648	209628
SAP HANA platform edition	545	148240
Internet of Things	495	122265
SAP Process Integration	488	118828
SAP Gateway	487	118341
Training	466	108345
SAP Mentors	458	104653
SAP Screen Personas	424	89676
Internationalization and Unicode	398	79003
SAP Mobile Platform	377	70876
SAP Inside Track	369	67896
SAP NetWeaver	368	67528
Security	367	67161
SAP TechEd	347	60031
SAP Enterprise Portal	344	58996
Careers	322	51681
SAP HANA Cloud Portal	317	50086

Table 25: BUTUB Distribution Approximation

Term	CUT Count	CUTUC Count
Using SAP.com	169244	14321681146
ABAP Development	141591	10023934845
MM (Materials Management)	113666	6459922945
SAP BusinessObjects Business Intelligence platform	101500	5151074250
Coffee Corner	96137	4621113316
SAP ERP	91206	4159221615
LE (Logistics Execution)	81217	3298059936
FIN (Finance)	79274	3142143901
SAP Inventory Manager	77762	3023425441
SAP Material Availability	77071	2969930985
SAP Process Integration	76182	2901810471
SAP HANA Cloud Platform	74947	2808488931
SAP Crystal Reports	73421	2695284910
SAP Fiori	67447	2274515181
SAP HANA	66309	2198408586
SD (Sales and Distribution)	65609	2152237636
PLM Enterprise Asset Management (EAM)/Plant Maintenance (PM)	63933	2043682278
SBOP Web/Desktop Intelligence (client/server)	62662	1963231791
HCM (Human Capital Management)	57684	1663693086
SAPUI5	55434	1536436461
support services	54051	1460728275
BW (SAP Business Warehouse)	53365	1423884930
SAP BusinessObjects Lumira	52185	1361611020
Careers	49759	1237954161
SAP Crystal Reports version for Visual Studio	49751	1237556125
SBOP SDK	49365	1218426930
FIN Controlling	48916	1196363070
FIN Asset Accounting	47612	1133427466
SAP Invoice and Goods Receipt Reconciliation	45180	1020593610
SAP AR Warehouse Picker	45131	1018381015

Table 26: CUTUC Distribution Approximation

Term	BUT Count	BT Count	BUTB Count
SAP HANA	1022	6269	6406918
SAP BusinessObjects Business Intelligence platform	1906	1724	3285944
Business Trends	755	4050	3057750
SAP BusinessObjects Lumira	1744	1396	2434624
ABAP Development	857	2718	2329326
Using SAP.com	1373	1515	2080095
Training	466	3954	1842564
SAP BusinessObjects Design Studio	1837	944	1734128
SAP HANA Cloud Platform	1282	1136	1456352
SAPUI5	854	1601	1367254
SBOP Web/Desktop Intelligence (client/server)	1787	480	857760
SAP Fiori	968	849	821832
SAP Enterprise Portal	344	1804	620576
SAP TechEd	347	1727	599269
SAP Process Integration	488	1212	591456
SAP NetWeaver	368	1237	455216
Internet of Things	495	879	435105
SAP BusinessObjects Analysis edition for Microsoft Office	1593	238	379134
Careers	322	1113	358386
SAP S/4HANA	648	553	358344
SAP Gateway	487	731	355997
SAP Mobile Platform	377	841	317057
Security	367	860	315620
SAP BusinessObjects Cloud	1922	148	284456
SAP Mentors	458	460	210680
SAP Inside Track	369	451	166419
SAP Screen Personas	424	258	109392
SAP HANA Cloud Portal	317	202	64034
SAP HANA platform edition	545	68	37060
Internationalization and Unicode	398	34	13532

Table 27: BUTB Distribution Approximation

Term	CUT Count	CT Count	CUTC Count
ABAP Development	141591	11793	1669782663
SAP HANA	66309	11005	729730545
SAPUI5	55434	9220	511101480
MM (Materials Management)	113666	4264	484671824
Using SAP.com	169244	2841	480822204
SAP Process Integration	76182	5450	415191900
SAP BusinessObjects Business Intelligence platform	101500	3198	324597000
SAP ERP	91206	3385	308732310
FIN (Finance)	79274	3620	286971880
SAP Crystal Reports	73421	3717	272905857
SAP Fiori	67447	3289	221833183
BW (SAP Business Warehouse)	53365	4149	221411385
SD (Sales and Distribution)	65609	3250	213229250
HCM (Human Capital Management)	57684	3158	182166072
PLM Enterprise Asset Management (EAM)/Plant Maintenance (PM)	63933	1919	122687427
SAP HANA Cloud Platform	74947	1140	85439580
Careers	49759	1529	76081511
SAP BusinessObjects Lumira	52185	1396	72850260
FIN Controlling	48916	1199	58650284
LE (Logistics Execution)	81217	630	51166710
SAP Crystal Reports version for Visual Studio	49751	996	49551996
FIN Asset Accounting	47612	795	37851540
SBOP Web/Desktop Intelligence (client/server)	62662	480	30077760
SBOP SDK	49365	201	9922365
support services	54051	174	9404874
SAP Inventory Manager	77762	72	5598864
SAP Material Availability	77071	70	5394970
SAP Invoice and Goods Receipt Reconciliation	45180	98	4427640
Coffee Corner	96137	43	4133891
SAP AR Warehouse Picker	45131	12	541572

Table 28: CUTC Distribution Approximation

Java Fundamentals

In this section, we briefly explain Java's *Garbage Collection* mechanism and *Futures*. Especially Garbage Collection is an important topic in Java based big data processing systems such as Apache Flink.

Garbage Collection

Garbage collection [42] refers to the automated identification of used and an unused Java objects on the Java heap memory in the Java Virtual Machine (JVM), which is also called Java HotSpot heap. Moreover, the process of garbage collection comprises the automated deletion of the unused objects by the garbage collector in order to free up heap space. An object is in use if it is still referenced somewhere in the program. Vice versa, the program contains no pointers to objects that are marked as unused anymore.

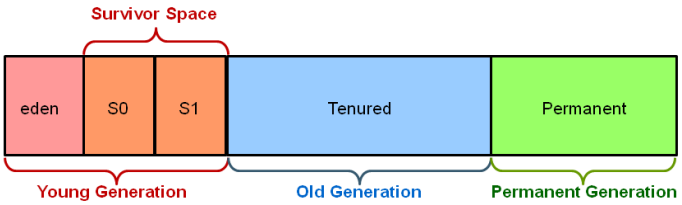


Figure 54: Java Heap Memory Structure [42]

The heap space is divided into so called generations (cf. figure 54) in order to optimize the garbage collection execution. Most objects are allocated in the *young generation*. A *minor collection* is triggered in a *stop-the-world* manner when this part fills up. In other words, all application related threads are stopped until finishing the collection. Surviving objects are moved to *old generation*. All long surviving objects are stored here. During a *major collection* the unused objects in this generation are collected as well in a stop-the-world manner. Meta data that is needed by the JVM to describe classes for example is kept in the *permanent generation*.

Future

A *Future* in Java represents the result of an asynchronous computation and as such it is a result which might eventually appear in the future after the asynchronous computation is finished [41].