



UNIVERSITÄT LEIPZIG

Leipzig University
Institute of Computer Science
Faculty of Mathematics and Computer Science
Database Department

Comparison of Single-Machine and Distributed Calculation of Temporal Degree Metrics

Bachelor Thesis

Submitted by:

Xiaoxi Li

Matriculation number:

3705651

Supervisor:

Prof. Dr. Erhard Rahm

Christopher Rost

© 2022

This work including its parts is **protected by copyright**. Any use outside the narrow limits of copyright law without the consent of the author is prohibited and punishable. This applies in particular to duplications, translations, microfilming and storage and processing in electronic systems.

Abstract

Graph data is not only extremely huge nowadays because of its rapid expansion, but it is also changing over time. Since graph data to be stored and analyzed becomes extremely massive, many graph processing systems are using distributed execution to minimize the runtimes while processing their data in a scalable manner. The question is whether distributed architectures are suitable for all dataset sizes, from small ones with only a few hundred nodes and edges to large ones with billions of edges. This thesis compares the runtime of a distributed implementation to a single machine implementation of a selected graph analysis operation, namely the degree evolution of all vertices of a temporal graph. The distributed implementation in Apache Flink is already finished and executable. Benchmarks of the single machine implementations consist of a self-developed Java implementation as well as an alternative implementation in the graph database management system Neo4j by using the Cypher query language. The results show that distributed computing has a significant runtime advantage for large datasets (several gigabytes), with some smaller datasets slower than the Java single machine implementation. Moreover, the parallelism setting obviously affects the runtime of distributed computing. The runtimes evaluated by the Neo4j implementation consistently lags very behind.

Contents

List of Figures	III
List of Tables	IV
List of Algorithms	V
1. Introduction	1
1.1. Motivation	1
1.2. Goals	2
1.3. Structure	2
2. Related Work	4
3. Background	6
3.1. Graph	6
3.1.1. Property Graph	6
3.1.2. Temporal Property Graph Model	7
3.1.3. Degree	7
3.1.4. Temporal Degree	7
3.1.5. Degree Evolution	8
3.2. Apache Flink	8
3.2.1. Components of A Flink Setup	8
3.2.2. Slots and Parallelism	9
3.2.3. DataSet-API	10
3.3. Java Stream	11
3.4. Neo4j	12
3.4.1. Data Storage Form	13
3.4.2. Query Language Cypher	13
3.4.3. Extending Cypher	13
3.4.3.1. User-defined Procedures	13
3.4.3.2. User-defined Functions	14
3.4.3.3. APOC	15
4. Design and Implementation	16
4.1. Recapture of Degree Evolution Algorithm and Implementation	16
4.1.1. Recapture of Algorithm	16
4.1.2. Recapture of Distributed Implementation	19
4.2. Java Based Single Machine Implementation	20
4.3. Cypher Queries in Neo4j	23
5. Evaluation	27
5.1. Benchmark Setup and Environment	27
5.1.1. Distributed Implementation	27

5.1.2. Java Single Machine Implementation	28
5.1.3. Neo4j	28
5.2. Results Comparison	28
5.2.1. CitiBike	28
5.2.2. Stackexchange	30
5.2.3. LDBC and Stackoverflow	32
5.3. Influencing Factors on Runtime	34
6. Conclusion and Prospect	36
Bibliography	37
Declaration of Authorship	39

List of Figures

3.1. Example of graphs	6
3.2. Bike-sharing network example with temporal attributes <code>valid_from</code> and <code>valid_to</code> . .	6
3.3. An example directed graph	7
3.4. Components of a simple Flink Setup	8
3.5. Example of parallelism in terms of slots	9
4.1. An example graph with 3 vertices and 4 edges with properties <code>valid time</code>	17
4.2. An illusion of constructing a degree tree for vertex id_1 and degree type <code>out</code>	17
4.3. Workflow of Apache Flink implementation	19
4.4. Workflow of single machine implementation	20
5.1. Runtimes of CitiBike SF1	29
5.2. Runtimes of CitiBike SF10	29
5.3. Runtimes of CitiBike SF100	30
5.4. Runtimes of Stackexchange Chess	30
5.5. Runtimes of Stackexchange Writers	31
5.6. Runtimes of Stackexchange Math	31
5.7. Runtimes of LDBC SF1	32
5.8. Runtimes of LDBC SF10	32
5.9. Runtimes of LDBC SF100	33
5.10. Runtimes of Stackoverflow	34
5.11. Speedup of LDBC SF10, SF100 and Stackoverflow	34

List of Tables

3.1. Example Transformations of Apache Flink DataSet-API	10
5.1. DataSets statistics with number of vertices and edges as well as disk size on HDFS .	27

List of Algorithms

1. Recapture of Tree Traversal Algorithm 18

1. Introduction

1.1. Motivation

Real-world graphs keep changing strongly over time, a typical case is the network of relationships between web users. Many applications rely on analysis of increasingly large graph data, such as social media platforms that analyze interests of users, communities that users follow as they evolve over time. And bike share platforms analyze rental information based on the station-to-station relationships.

Graphs are heterogeneous, typically graphs are large and can contain millions or even billions of nodes and edges. A survey was presented by Sahu et al. in [1]. It contains the number of vertices and edges of the graphs that the participants used, as well as the size of the uncompressed graph. The most significant aspect of the survey was the amount of participants using very large graphs. Lots of participants used graphs with vertices greater than 100 million and edges with more than 1 billion, among them are both researchers and practitioners. The survey also noted that more than half of the participants' graphs changed very frequently, that is, graph datasets are dynamic or streaming. Also in [2], when discussing graph processing, it is stated that future graph processing systems are required to be able to cope with dynamic and streaming graph data. Meanwhile, traditional single-machine applications face huge challenges in processing such huge data, thus more and more Systems dealing with graph data analysis base on distributed computing. Another noteworthy part of the survey in [1] is that distributed products were the most preferred alternative among the three architectures (including Single Machine Serial, Single Machine Parallel and Distributed) and that the choice of architecture was highly correlated with the size of the graph, over 50% participants used the distributed architecture had graphs with more than 100 million edges. A survey of numerous distributed graph processing systems is presented as well as in [3].

For example, Gradoop [4] is a distributed graph analysis system based on Apache Flink¹, which is a reference implementation of the Temporal Property Graph Model (TPGM) [5]. The TPGM is a graph data model which allows for modeling the evolution of a graph over time. The analysis of temporal properties enables more intuitively to have a view of the evolution of things.

The experience of users usually depends on the responding speed of computing services. On the one hand, the architecture of the distributed computing system implies a considerable time overhead for inter-component communication. Due to its horizontal scalability, distributed computing allows new nodes to be added to a distributed computing network as needed, that is, more computing devices can be added. Therefore, when using distributed computing to process smaller datasets, the time consumed due to communication of nodes will instead increase the runtime compared to a single machine. On the other hand, single machine solutions have difficulty responding to the increase in dataset size, i.e., they are lack of scalability, thus, large datasets will hard to be processed due to insufficient computational resources. Users are often in a dilemma of whether to choose a single machine or distributed computing. This thesis intends to address this struggle by evaluating both using various datasets of different sizes and finding a boundary of choice.

¹<https://flink.apache.org>

1.2. Goals

The main goal of this thesis is to evaluate whether distributed computing is suitable when processing data of all sizes and whether single machine implementations can be replaced.

In general, a single machine implementation is more appropriate for small datasets, while processing of large datasets is better suited to be executed in a distributed system. Finding how to define this size is thus essential. McSherry et al. propose a new metric COST in [6], that is, the Configuration that Outperforms a Single Thread, which is used to express the configuration (i.e., the number of cores in the paper) required for a distributed computation to be faster than single threaded implementation. In their evaluation, single threads are faster than most other distributions when performing PageRank (a directed graph computation), single thread also consistently leads when performing running times of label propagation. After optimizing the algorithm, a maximum COST of 100 cores is even required to make the distributed computation faster than single-threaded, with one of the distributed implementations requiring only 3.6 seconds of measurement time, but the COST is already 512 cores at this point.

The appropriate choice of technology for different datasets and the configurations when using distributed calculation can greatly reduce the runtime and eliminate unnecessary waste of computational resources, i.e., in this thesis it will be necessary to identify a boundary to classify which technique to use at what size of dataset.

For this purpose, the performance of the existing operator in Gradoop² in the temporal graph properties computation will be benchmarked at different configurations (that is, parallelism settings). Its runtimes in different configurations will be compared with the single machine applications. The distributed implementation in Apache Flink is already finished and executable. The main tasks of this thesis are thus as follows:

- a) The single machine implementation needs to be implemented, it will use the Java 8 Stream-API to produce the identical results as Gradoop.
- b) Implement a Neo4j³ Cypher [7] solution, which also achieves the same results.
- c) Evaluate implementations with n different sizes and types of datasets:
 1. Benchmark the distributed implementation with different parallelism settings;
 2. Benchmarking with single machine implementation;
 3. Benchmarking with Neo4j.

1.3. Structure

The structure of the thesis is as follows. In Section 2 various related research efforts are presented. The thesis-related backgrounds will be introduced in Section 3, containing the definitions of graphs and temporal graphs, the basic knowledge of Apache Flink, Java Stream-API and Neo4j. Section 4 begins with the workflow of the already completed distributed implementation in Apache Flink in

²<https://github.com/dbs-leipzig/gradoop/blob/develop/gradoop-temporal/src/main/java/org/gradoop/temporal/model/impl/operators/metric/TemporalVertexDegree.java>

³<https://neo4j.com>

processing temporal graphs, as well as descriptions about the designs of single machine implementations, including a Java-based implementation and the steps of Cypher queries in Neo4j. Section 5 is devoted to benchmarking and runtime comparison. The last Section 6 concludes this work and the future directions for optimizations.

2. Related Work

For existing Gradoop implementation⁴ built on DataSet-API of Apache Flink, in [5] provides a detailed presentation of the analysis of distributed temporal graph using Gradoop. The finished implementation is a TPGM [8] operator of Gradoop that computes the evolution of vertex degrees for all vertices of the graph and returns a dataset of tuples as a result with the identifier of vertex, time interval, and the value of the degree. An overview for the Gradoop implementation of the temporal degree evolution is presented by Rost et al. in [9], containing the relevant definitions, algorithm and implementation details of the Degree Evolution-Operator. They benchmarked this implementation as well, the results show that the runtime of the algorithm increases sublinearly with a linear increase size of the dataset, and by scaling the number of machines, there were speedups of 10 and 12 when processing the two largest datasets.

Some research has been devoted to the use of graph databases such as Neo4j for storing and querying temporal property graphs. Debrouvier et al. [10] propose a Neo4j-based implementation that solves the problem of modeling, storing and querying temporal attribute graphs, allowing to preserve the history of a graph database. Cattuto et al. [11] present a Neo4j property graph for data modeling of time-varying social network data. They collected time-varying social network data using wearable sensors and investigated the real-world query performance, pointing out the advantages, disadvantages and challenges.

There are already a few components⁵ for benchmarking Gradoop on Github, which should be available for the later work. The benchmarking and runtime comparisons can reference [6], a new metric for Big Data platforms is proposed - COST (Configuration that Outperforms a Single Thread), which is compared and discussed with single-threaded applications and some recent scalable graph processing systems. The benchmarks in this paper compared single-threaded applications to distributed computing, where single-threaded almost always leads when performing the same computation, and distributed computing requires a significant amount of resources (i.e., COST) to outperform single-threaded in performance. However, it is worth noting that only two different dataset sizes were used in their benchmark tests. Verbitskiy et al. [12] show in their work as well as the comparison of single-machine and Flink distributed computing, it puts several different algorithms on a single machine implementation and a distributed implementation for comparison, but the two implementations in this paper are benchmarked on different devices. If the compared benchmarks are run on the same machine, the results will be more comparable. Benchmarking of different distributed systems is also described in detail by Karimov et al. [13].

In the real world, scalability and visualization are the most pressing challenges in processing enormous graphs, in [1] described Sahu et al. their discovery. The dataset sizes of participants in their survey are quite varied, with uncompressed datasets ranging from less than 100 megabytes to larger than 1 terabyte, the number of vertices containing less than 10K to greater than 100 million, and the number of edges ranging from less than 10K to more than 1 billion. It is highly instructive that the dataset sizes used in this work's evaluation should also be as wide as possible. Sakr et al. [2] summarize and motivate the importance of graph processing in one sentence. It discussed the

⁴<https://github.com/dbs-leipzig/gradoop/blob/develop/gradoop-temporal/src/main/java/org/gradoop/temporal/model/impl/operators/metric/TemporalVertexDegree.java>

⁵<https://github.com/dbs-leipzig/gradoop-benchmarks>

shortcomings and challenges of the current graph processing system as well as the future direction, starting from three main elements (i.e. Abstractions, Ecosystems and Performance). Both [1] and [2] show that distributed processing is needed. In [14], the speed of single-machine and distributed system is compared in machine-learning. The results show that at the beginning, the single-machine is faster than the distributed system, but after 10 datapass, the distributed system is faster than the single-machine. In another paper [15], tests showed that Spark, a distributed system for processing data stream, requires more resources to achieve comparable performance to a single-machine.

3. Background

3.1. Graph

A Graph is the non-linear data structure consisting of vertices and edges. It is composed of a set of vertices and a set of edges which can be defined as $G = (V, E)$, where V is the set of vertices and E is the set of edges. The edges of the graph can be directed or undirected (Figure 3.1).

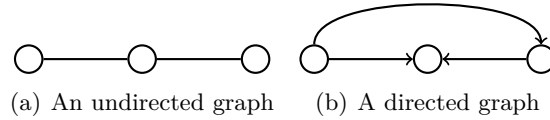


Figure 3.1.: Example of graphs

3.1.1. Property Graph

The graph used in this work is called Property Graphs [16]. A property graph is directed and consists of a set of nodes or vertices and a set of relationships or edges connecting the nodes. Vertices and edges have labels and optional attributes (or properties), which are represented as key-value pairs. The example in Figure 3.2 is a bike-sharing network, which Stations are vertices and Trips are edges. Each Station has the properties such as name, capacity, latitude and longitude, etc. Each edge has a label Trip and properties bikeId, userType, gender, year of birth, valid_from, valid_to, etc.

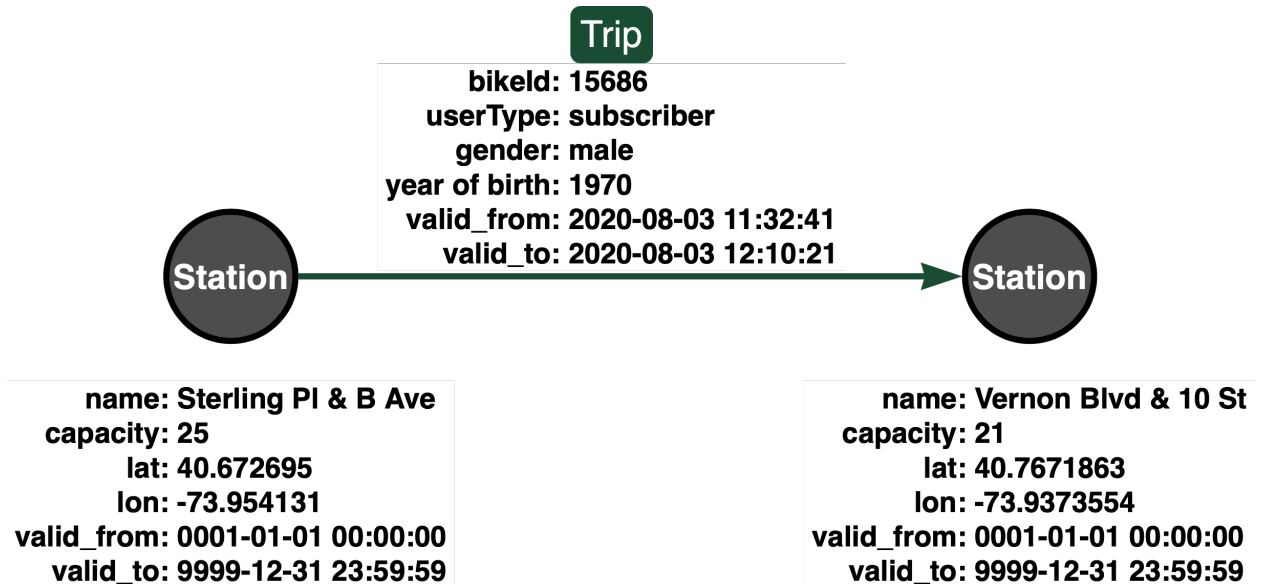


Figure 3.2.: Bike-sharing network example with temporal attributes valid_from and valid_to

3.1.2. Temporal Property Graph Model

Temporal property graph model (TPGM) [8, 17] models the evolution of graphs. It supports bitemporal time management [18], which is known from the relational world. The model provides additional time properties than the extended property graph model (EPGM) [19]. The four additional time attributes in vertices and edges are: **valid from** (`val_from`), **valid to** (`val_to`) of valid time interval and **transaction from** (`tx_from`), **transaction to** (`tx_to`) of transaction time interval. Valid time represent the validity of elements, that is, the validity of an element is from the timestamp `val_from` to the timestamp `val_to`. Transaction times can automatically generate by a system itself. For example, Figure 3.2 shows the start and end time of a Trip with time properties `valid_from` and `valid_to`. Furthermore, the time format used for the actual execution is Unix timestamp.

3.1.3. Degree

Degree is a vertex metric for directed and undirected graphs, denoted as $deg(v)$. For directed graphs, a special case of the degree is indegree, denoted as $deg^-(v)$, and outdegree, denoted as $deg^+(v)$. The outdegree of a vertex is the number of edges that start at the vertex, and the indegree of a vertex is the number of edges whose destination are the vertex. Sum of in- and outdegree is the degree itself, formally:

$$\forall G_{directedGraph} = (V, E), \quad deg(v) = deg^-(v) + deg^+(v), \quad \text{with } v \in V. \quad (3.1)$$

An example shows in Figure 3.3 that $deg(A) = 2$, $deg^+(A) = 2$, $deg^-(A) = 0$, $deg(B) = 2$, $deg^+(B) = 0$, $deg^-(B) = 2$, $deg(C) = 2$, $deg^+(C) = 1$, and $deg^-(C) = 1$.

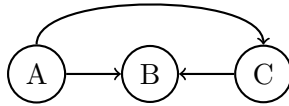


Figure 3.3.: An example directed graph

3.1.4. Temporal Degree

Rost et al. defined in [9] a temporal version of the vertex degree namely *temporal degree*. In a temporal graph $G = (V, E)$, the temporal degree of a vertex $v \in V$ is the degree of this vertex at a given time ω , which denoted as $degt(v, \omega)$. Formally, it is defined as:

$$degt(v, \omega) \begin{cases} deg(v), & \text{if } v \in V(\omega); \\ \text{undefined}, & \text{otherwise.} \end{cases} \quad (3.2)$$

The temporal indegree in a temporal graph is $degt^-(v, \omega)$, i.e., the degree at time ω of vertex v , as well as the temporal outdegree at time ω of vertex v is $degt^+(v, \omega)$.

3.1.5. Degree Evolution

For a vertex, other vertices connected to it at different time will change, i.e., the temporal degree will change. Hence, when a time interval is given, all the relevant values of degrees of the vertex corresponding time interval are the evolution of the degree. Rost et al. defined *degree evolution* in [9] as: The degree evolution $degev(v, \tau) := \{x_1, x_2, \dots, x_m\}$ of a vertex v is a time series of elements $x_i := deg_t(v, \omega)$, with $1 \leq i \leq m$ and $m = \omega_{end} - \omega_{start}$. Every x_i is a temporal degree at time ω_j , that is, x_1 at time point ω_{start} and x_m at $\omega_{end} - 1$, for the time interval $\tau = [\omega_{start}, \omega_{end})$.

3.2. Apache Flink

Apache Flink⁶ is an open-source framework and a distributed streaming data processing engine, which frequently used to handle high-performance, scalable and accurate real-time applications. The purpose of distributed processing is to handle incoming data as efficiently as possible. The exploit of parallelism has the obviously influences on the performances of a Flink application. These performance requirements can be express in terms of latency and throughput [20]. Latency represents the time a task takes from submission to completion. That is, if a task needs to be queued before it can be processed, then the waiting time for processing is included. Throughput indicates the capacity of the system to handle tasks, which is a measure of how many tasks a system can handle in a given time unit.

3.2.1. Components of A Flink Setup

A Flink program consists of multiple tasks, such as transformations, operators, data sources or sinks. A task is split into several parallel instances for execution and each parallel instance processes a subset of the task's input data. The number of parallel instances of a task is called its parallelism. Figure 3.4 is a recapture example in [20], it shows an example of a JobManger claiming slots from the ResourceManager and assigning tasks to the TaskManager.

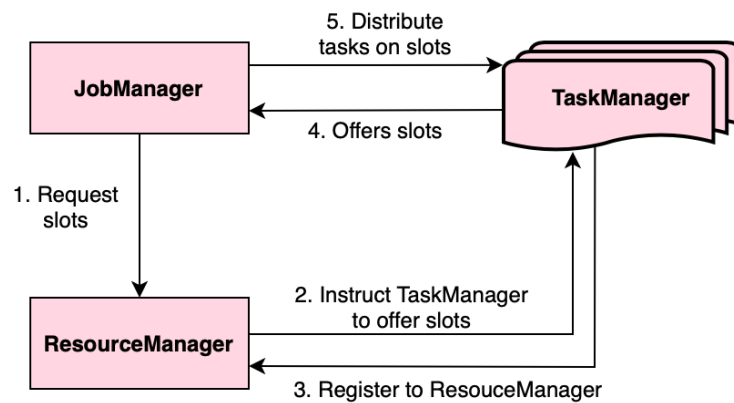


Figure 3.4.: Components of a simple Flink Setup

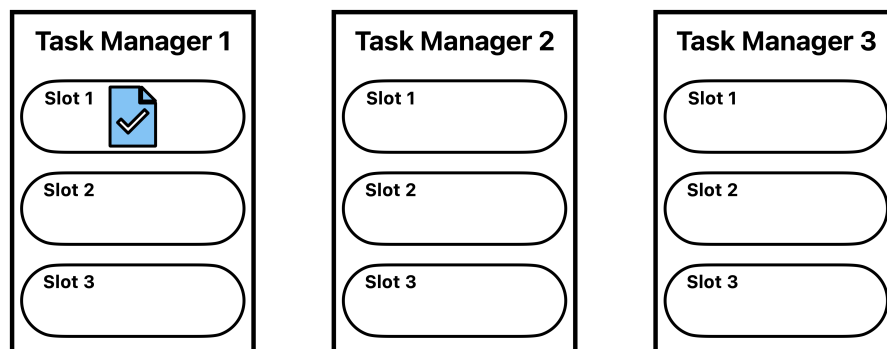
⁶<https://flink.apache.org>

The **JobManager** acts as the master process, the execution of a single application is controlled under a Jobmanager. The application consists of a JobGraph and all the required resources. All programs from higher level APIs are transformed into JobGraphs. A JobGraph can be converted to tasks that are executable in parallel. Then the JobManager requests ResourceManager for the TaskManager slots, as marked by 1. With the enough amount of the TaskMager slots assigns the JobManager tasks into those slots.

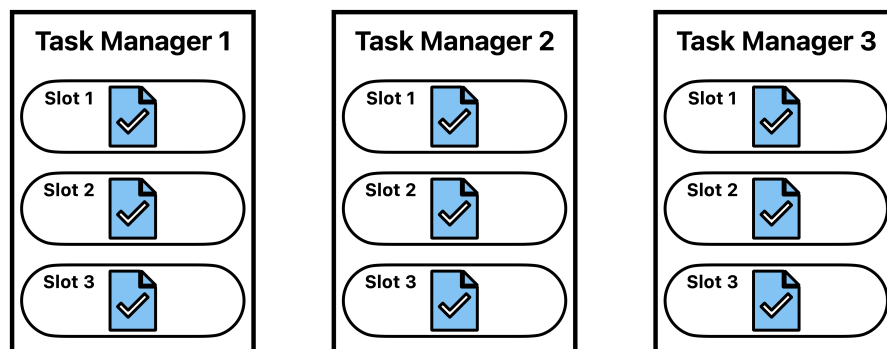
The **ResourceManager** manages all the slots of the TaskManager. On the one hand, after the requirement of a JobManager, the ResourceManager will instruct TaskManagers to provide the available slots to the Jobmanager. Meanwhile, TaskManagers also need to register these slots to ResouceManager as noted 2 and 3. On the other hand, the work of freeing computing resources is also handled via ResourceManager.

Each **TaskManager** holds one or more slots. In Flink, TaskManagers are also called workers. After receiving instructions from the ResourceManager, TaskManagers will provide the currently usable slots to the JobManager, which annotated as 4. Afterwards the JobManager will distribute the tasks in the slots that marked as 5. TaskManagers execute these tasks and exchange the data streams.

3.2.2. Slots and Parallelism



(a) By default parallelism setting, only 1 slot will be used.



(b) Manually set parallelism to 9 to occupy all slots.

Figure 3.5.: Example of parallelism in terms of slots

A TaskManager (or a Worker) is essentially a JVM process. To control how many tasks a worker can receive, the worker is controlled by a task slot (a worker has at least one task slot). Each task slot represents a fixed-size subset of the resources owned by the TaskManager. If a TaskManager has three slots, then it will divide its managed memory into three parts for each slot. If a TaskManager has one slot, it means that each task group runs in a separate JVM, while a TaskManager with multiple slots means that more subtasks can share the same JVM. They may also share datasets and data structures, so that can reduces the load on each task.

Assume that there are 3 TaskManagers as shown in Figure 3.5. Each TaskManager is assigned 3 slots, that is, 9 slots in total and 9 tasks can be distributed. The default setting of parallelism is 1, i.e., the default parallelism of the running program is 1, only 1 of the 9 TaskSlots is used, and 8 are free. Only when set to 9, all slots will be occupied.

Parallelism of operators can be controlled at the execution environment level, or different parallelism can be set for each different operator. When an application is submitted to a running Flink cluster, the parallelism of the execution environment will be set to the default parallelism of the cluster, unless the parallelism is specified.

3.2.3. DataSet-API

`DataSet` and `DataStream` are two peculiar classes that Flink uses to represent data in programs. The data in a `DataSet` is bounded, while for a `DataStream`, the amount of elements can be unlimited. The completed Gradooop distributed implementation in Apache Flink based on `DataSet-API`, thus in this work will focus on the `DataSet`.

Typically, a Flink `DataSet` program can be organized into steps: read datasets from data sources, apply transformations on the data, and return the results via sinks.

`DataSet-API` supports reading datasets from various data sources into Flink system and converting them into `DataSet`, which mainly includes three types: file-based, collection-based and generic data sources. The supported data source methods are listed in the methods provided by the class `ExecutionEnvironment`⁷.

Transformations	Description
Map	Takes one element and produces one element.
FlatMap	Takes one element and produces zero, one, or more elements.
Reduce	Combines a group of elements into a single element by repeatedly combining two elements into one. Reduce may be applied on a full data set or on a grouped data set.
ReduceGroup	Combines a group of elements into one or more elements. ReduceGroup may be applied on a full data set, or on a grouped data set.
Join	Joins two data sets by creating all pairs of elements that are equal on their keys.

Table 3.1.: Example Transformations of Apache Flink `DataSet-API`

⁷<https://nightlies.apache.org/flink/flink-docs-release-1.9/api/java/org/apache/flink/api/java/ExecutionEnvironment.html>

Transformations convert a `DataSet` into another `DataSet`, they are essential for the derivation of data sources to target data. Flink provides a wide range of transformation operators. (see Table 3.1)

Flink's data output in the `DataSet-API` is divided into three types. The first one is based on file implementation, corresponding to the `write()` method of `DataSet`, which realizes the output of `DataSet` data to the file system. The second type is based on common storage media implementation, corresponding to the `output()` method of `DataSet`. The last kind is client-side output, which directly collects `DataSet` data from different nodes to Client and outputs it in the client, for example, the `print()` method of `DataSet`. Listing 3.1 demonstrates a simple example of using the `Dataset-API` in Java to count the number of elements.

```

1  final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
2
3  Integer[] input = {1, 2, 3, 4, 5};
4  DataSet<Integer> data = env.fromElements(input);
5  long numElements = data.count();

```

Listing 3.1: Example Code of counting elements by using `Dataset-API`

3.3. Java Stream

The single machine implementation of this thesis is mainly built on `Stream-API`, a new class added in Java 8 to complement the collection class. `Stream` represents a data stream in which the number of data elements can be finite or infinite.

Consider the collection of elements to be processed as a stream that travels through the pipeline and can be processed at the nodes of the pipeline, such as filtering, sorting, aggregation, etc. A `Stream` of elements is processed through intermediate operations in the pipeline, and the result of the previous processing is finally obtained through the terminal operation. That is, a stream is a queue of elements from a data source that supports aggregation. Data sources are the sources of `Stream`, which can be collections, arrays, I/O channels, generators, etc.

`Stream` is different from `Collection`, all values should have been filled before starting to work with `Collection`. Whereas Java `Stream` is an on-demand data structure. Java `Stream` does not store data, it operates on source data structures (collections and arrays) and produces pipelined data. Iterating over a `Collection` used to be done explicitly outside the collection via `Iterator` or `For-Each`, which is called external iteration. `Streams` provide a way to iterate internally, via the `Visitor` pattern. More specifically, a `Collection` can be traversed any number of times, but streams can only be traversed once. Since `Streams` are consumable, there is no option to create a reference to a stream for future use. That is, since data is used on demand, it is not possible to re-use the same stream multiple times.

Operations such as filtering, mapping, and limiting are called intermediate operations, and they return a new stream, while collection is called a terminal operation. Terminal operations cause the entire stream to be executed and closed, that is, `filter`, `map`, and `limit` are only executed when `collect` is executed. A `Stream` can only be traversed once, as shown in the listing 3.2, this is because

forEach is a terminal operation which closes the stream after execution.

```
1 List<String> list = menu.stream()
2   .filter(d -> d.getCalories() > 300)
3   .map(Dish::getName)
4   .limit(3)
5   .collect(toList());
```

Listing 3.2: Example Code of Stream operations

Furthermore, **Stream** operations use a functional interface, which makes it very suitable for functional programming using **Lambda** expressions. For example, filtering, mapping, or deduplication can be implemented lazily, resulting in higher performance and room for optimization.

3.4. Neo4j

Neo4j is a high-performance NoSQL (Not Only SQL) graph database that stores structured data on graphs rather than tables. Graph databases are a branch of NoSQL that is well suited for the native representation of graph structured data. Unlike traditional relational databases that use two-dimensional tables to store data, graph databases are categorized as a type of NoSQL database, which means that graph databases are non-relational databases.

A general graph database entails at least three functions: graph storage, graph query, and graph analysis. A graph database is a data management system based on the graph that was introduced at the beginning of this section. Data is represented by points and edges, transforming data into points and relationships between data into edges.

Graph databases have strong associative query performance advantages when dealing with highly associative data and natural graph problem scenarios. Since the traditional relational databases need to do expensive table joins (JOIN) when performing associative queries, which involves a lot of I/O operations and memory consumption. The underlying storage mechanisms of various graph databases can be very different. Depending on the storage and processing models, some distinctions are made between graph databases. For example, some graph databases use native graph storage, which is optimized and designed specifically for storing and managing graph data. These databases are generally called native graph databases, such as Neo4j. While some graph databases rely on relational engines to store graph data in tables of relational databases, they interact with data by adding an abstraction layer with graph semantics on top of the underlying storage system where the data actually resides. There are also graph databases that use key-value storage or document-based storage as the underlying storage. These types of graph databases are collectively referred to as non-native graph databases, such as ArrangoDB, OrientDB, JanusGraph, etc. Native graph storage has more performance advantages over non-native. Native graph databases do not rely on third-party storage systems for their underlying storage, and the integration of computation and storage greatly simplifies the system architecture.

3.4.1. Data Storage Form

The data storage in Neo4j is organized by nodes and edges. Nodes can represent entities in graph, and edges can be used to represent relationships between entities, which can be directional and have two ends corresponding to the start and end nodes.

Furthermore, one or more labels (Node Label) can be added to the node to represent the classification of the entity, and a collection of key-value pairs to represent some additional properties of the entity in addition to the relationship properties. Relationships can also be accompanied by additional attributes.

3.4.2. Query Language Cypher

Neo4j uses Cypher as query language. Cypher is a declarative graphical query language for expressive and efficient query updating and management. It is similar to the SQL language for relational databases since many keywords (such as `WHERE` and `ORDER BY`) are inspired by SQL. Listing 3.3 shows an example of using Cypher queries in Neo4j. The `MATCH` clause specifies a node and relationship pattern with two connected nodes, labeled `City` and `Country`, connected with a relationship of type `IN`.

```
1 MATCH (city:City)-[:IN]-(country:Country)
2 WHERE city.name = "Leipzig"
3 RETURN country.name;
```

Listing 3.3: Find nodes with specific relationships in Neo4j

3.4.3. Extending Cypher

Cypher is a very powerful and expressive language with graphical modes and set support. However, occasionally users need to do more than what it currently offers, such as additional algorithms, parallelization or custom transformations. Neo4j and Cypher can therefore be extended with user-defined procedures and functions. In Neo4j, user-defined procedures and functions are required to be developed independently using the Java language, and then generated as `.jar` files to be deployed in the graph database installation directory plugin.

3.4.3.1. User-defined Procedures

User-defined procedures are a mechanism to extend Neo4j by writing custom code that can be called directly from Cypher. Procedures can accept parameters, perform operations on the database, and return results. All procedures are annotated with `@Procedure`. There are three optional arguments the procedure annotation can take: `name`, `mode`, and `eager`. `name` is used to specify a different name for the procedure. If `mode` is specified then `name` must be specified as well. `mode` is used to declare the types of interactions that the procedure will perform. `READ` is the default mode, which means procedure will only perform read operations against the graph. `WRITE` mode will perform batch

read and write operations. **SCHEMA** mode can perform operations against the schema, i.e. create and drop indexes and constraints. A procedure with this mode is able to read graph data, but not write. **DBMS** mode is not able to read or write graph data, procedure will perform system operations such as user management and query management. A procedure will fail if it attempts to execute database operations that violates its mode. That is, for example in **READ** mode, it is impossible to make changes to the database. **eager** is a boolean setting defaulting to false. If it is set to true then the Cypher planner will plan an extra eager operation before calling the procedure. Listing 3.4 shows an example of user-defined procedure.

```

1  package example;
2  ...
3  public class EntityResultExample {
4      ...
5      @Procedure(name = "example.allnodes", mode = Mode.READ)
6      public Stream<EntityContainer> allnodes() {
7          ResourceIterator<Node> nodes = tx.execute("MATCH (n) RETURN n").columnAs("n");
8          return nodes.stream().map(EntityContainer::new);
9      }
10 }
```

Listing 3.4: Example of an user-defined procedure

3.4.3.2. User-defined Functions

User-defined functions are a simpler form of procedures that are read-only and always return a single value. They are often easier to use and more efficient than procedures for many common tasks. User-defined functions are created in a similar way to procedures, except that they are annotated with **@UserFunction**. Listing 3.5 and Listing 3.6 show an example of user-defined function **join** as well as calling the join function using the Cypher language.

```

1  package example;
2  ...
3  public class Join {
4      @UserFunction
5      public String join(@Name("strings") List<String> strings,
6          @Name(value = "delimiter", defaultValue = ",") String delimiter) {
7          if (strings == null || delimiter == null) {
8              return null;
9          }
10         return String.join(delimiter, strings);
11     }
12 }
```

Listing 3.5: Example of user-defined function join

```
1 MATCH (p: Person) WHERE p.age = 36
2 RETURN org.neo4j.examples.join(collect(p.names))
```

Listing 3.6: Example of calling user-defined function join

3.4.3.3. APOC

APOC, meaning Awesome Procedures on Cypher, is a Neo4j extension library that was released to prevent users from developing duplicate functions and procedures, and is considered to be the largest and most widely used extension library for Neo4j. It includes more than 450 standard procedures that provide utilities, transformations, graphical updates, and other features.

There are two ways to use APOC procedures, either in Cypher statements or as regular function calls. The same syntax is used for both methods:

```
CALL <package>.<subpackage>.<procedure>(<argument1>,<argument2>,...);
```

4. Design and Implementation

In Section 1, the main tasks of this thesis has been given, which include the execution of a Java single machine implementation as well as an approach in Neo4j using the Cypher language, both of them generate the same results as the already finished Apache Flink distributed implementation in terms of the temporal degree metrics calculation. Thus, in Sections 4.1, the distributed algorithm and implementation of Rost et al. [9] is recaptured before the design of single-threaded is given in Section 4.2. Afterwards, Section 4.3 describes the design using Neo4j.

4.1. Recapture of Degree Evolution Algorithm and Implementation

4.1.1. Recapture of Algorithm

The original algorithm proposed by Rost et al. in [9] contains the mapping of vertices at the beginning, since this thesis does not use the temporal information of the vertices, vertex mapping is skipped and the default max/min timestamps will be used as input for the later step IV, i.e., the algorithm recapture will start directly from the mapping of edges.

Assuming a temporal graph $G = (V, E)$ where V is a set of temporal vertices and E is a set of temporal edges, is given as input with configurable temporal degree types in, out and both as described for the TPGM model in Section 3.1.2, the algorithm will output a time series that expresses the evolution of the degrees of the vertices, whereby if the degrees do not change in successive times, then this time interval will be merged to simplify the final result. The algorithm is divided into the following steps:

I *Edge mapping*

The vertex information for each edge $e \in E$ is extracted according to the type of the configured degree type and one or two tuple $\langle vid, \omega_{start}, \omega_{end} \rangle$ will be created, where vid is the identifier of the vertex matching the extracted type. In case the degree type is in, vid is the identifier of the target vertex, and for the degree type out, vid is the identifier of the source vertex, if the type of the degree is both, two tuples are created for the target and source vertices respectively. The example graph shown in the Figure 4.1, for vertex id_1 , if the configuration of degree type is out, then the time information of edges e1, e2 will be mapped to $\langle id_1, 1, 3 \rangle, \langle id_1, 2, 4 \rangle$ ⁸.

II *Interval collection*

All the tuples created in step I will be grouped by identifier and a set of time intervals will be mapped for each identifier. For example, suppose there are tuples $\langle id_1, 1, 3 \rangle, \langle id_1, 2, 4 \rangle$ and the type of degree is out from Step I, the mapping result for vertex id_1 is $id_1 \rightarrow I_{id_1} = \{[1, 3), [2, 4)\}$. Note that the set of time intervals is unsorted.

III *Constructing of degree tree*

Each unsorted set corresponding to the identifier of the vertex $vid \rightarrow I_{vid}$ created in step II will be used as input to create a Binary Search Tree (BST) [21] T_v for each vid . Rost et al. in [9] describe

⁸To increase readability, the time points are expressed as integers

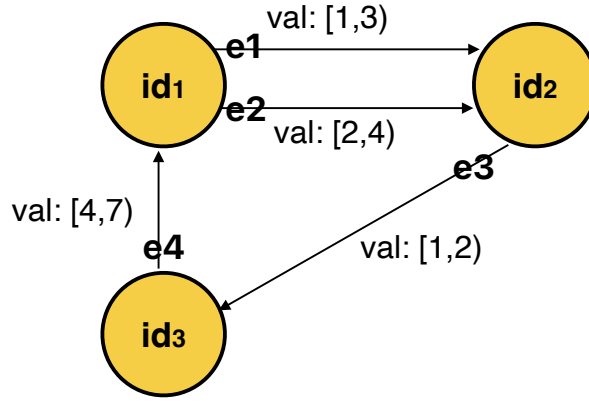
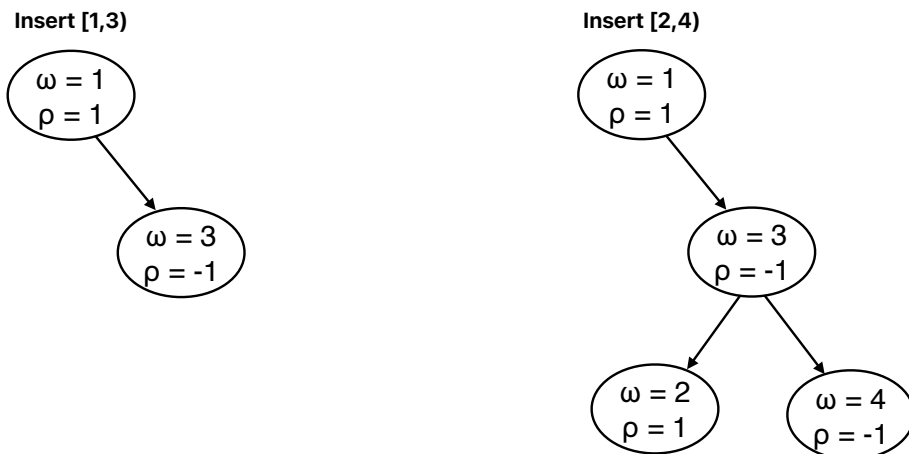


Figure 4.1.: An example graph with 3 vertices and 4 edges with properties valid time

such a BST as a degree tree. Using a degree tree both maintains the ordering of the tree when randomly inserting nodes and reduces memory requirements by avoiding duplicate time points. A time point $\omega \in \Omega$ and a payload $p \in \mathbb{Z}$ with initial value 0 will be used as nodes of the tree, for each time interval $[\omega_{start}, \omega_{end})$, the payload of ω_{start} is increased by 1 and the payload of ω_{end} is decreased by 1. On insertion of nodes, the time point ω_p of the parent node is greater than the time point ω_l of the left child node, and the time point ω_p of the parent node is less than the time point ω_r of the right child node. Figure 4.2 gives an example of constructing a degree tree using the set of time intervals corresponding to vertex id_1 . For the time interval $[1, 3)$, node $\omega = 1, p = 1$ and node $\omega = 3, p = -1$ are inserted, after that node $\omega = 2, p = 1$ and node $\omega = 4, p = -1$ are continued to be inserted for the corresponding interval $[2, 4)$. The final result for vertex id_1 is a degree tree containing four nodes.

Figure 4.2.: An illusion of constructing a degree tree for vertex id_1 and degree type out

IV Tree traversal and results collection

Each degree tree T_v obtained from step III will be accompanied by its corresponding vid as well

as start time ω_{start} and end time ω_{end} as input in this step, which outlines in Algorithm 1. Since the validity of the vertices is not considered, the start and end times of the algorithm are the default minimum value t_{min} and maximum value t_{max} , which will be assigned to ω_{last} and ω_{max} respectively.

The algorithm begins with the traversal of the degree tree T_v from line 3. Rost et al. in [9] use Depth First Search (DFS) [22] and in-order traversal so that the time points of nodes can be processed in ascending order. The nodes are processed in the following ways: Processing the first node (lines 4 to 6), the middle nodes (lines 7 to 11) and the last node (lines 13 to 15).

For the first node, if its time point ω_{node} is identical to the minimum time of initializing ω_{last} , then its payload p_{node} is added to d . For subsequent nodes, if the payload p is not zero (i.e., the degree has changed), the result is collected as $\langle vid, [\omega_{last}, \omega_{node}), d \rangle$ and the time of ω_{last} needs to be updated as the current node ω_{last} , as well as the payload p_{node} of the current node will be added to d . It is also necessary to deal with the time interval from the last visited node ω_{last} to ω_{max} , which has no change of degree (that is, $d = 0$), the result is collected as $\langle vid, [\omega_{last}, \omega_{max}), d \rangle$.

Algorithm 1 Recapture of Tree Traversal Algorithm

```

1: Data:  $vid, T_v, t_{min}, t_{max}$ ;
2: Initial:  $\omega_{last} \leftarrow t_{min}, \omega_{max} \leftarrow t_{max}, d \leftarrow 0$ ;
3: for each (in-order traversal of the DFS) node in  $T_v$  do
4:   if  $\omega_{node} == \omega_{last}$  then
5:      $d \leftarrow d + p_{node}$ ;
6:   end if
7:   if  $p_{node} \neq 0$  then
8:      $collect(\langle vid, [\omega_{last}, \omega_{node}), d \rangle)$ ;
9:      $d \leftarrow d + p_{node}$ ;
10:     $\omega_{last} \leftarrow \omega_{node}$ ;
11:   end if
12: end for
13: if  $\omega_{last} < \omega_{max}$  then
14:    $collect(\langle vid, [\omega_{last}, \omega_{max}), d \rangle)$ ;
15: end if

```

As an example, consider the degree tree T_{id_1} for vertex id_1 of step III, with the degree type out, and assume that the minimum time point is 0 and the maximum time is ∞ . According to the in-order traversal of the DFS, the nodes are processed in the order (1, 1), (2, 1), (3, -1) and (4, -1). Since the time point $\omega = 1$ of the first node (1, 1) is not equal to the minimum time point 0 (i.e., ω_{last}) and payload $p = 1$, following lines 4 to 8 of the algorithm, the first result tuple $\langle id_1, [0, 1), 0 \rangle$ can be collected, next the payload is increased to degree d , i.e., $d = 1$, and ω_{last} is updated to 1. By analogy, the subsequent result tuples $\langle id_1, [1, 2), 1 \rangle$, $\langle id_1, [2, 3), 2 \rangle$ and $\langle id_1, [3, 4), 1 \rangle$ are collected. Now the processing of all nodes is completed. According to lines 13 to 15 of the algorithm, the time interval from the last remembered ω_{last} to the maximum time point ω_{max} also needs to be handled, where the last tuple $\langle id_1, [4, \infty), 0 \rangle$ is collected.

Finally, the result of degree evolution for vertex id_1 in terms of degree type out is: $degev^+(id_1, [0, \infty)) = \{ \langle id_1, [0, 1), 0 \rangle, \langle id_1, [1, 2), 1 \rangle, \langle id_1, [2, 3), 2 \rangle, \langle id_1, [3, 4), 1 \rangle, \langle id_1, [4, \infty), 0 \rangle \}$.

4.1.2. Recapture of Distributed Implementation

Gradoop [8] is an open-source distributed graph processing system which based on Apache Flink⁹ and uses the transformations of Flink’s DataSet for the distributed graph analysis. The operator `TemporalVertexDegree`¹⁰ calculates the evolution of temporal degree metrics in temporal graphs by computing the degree changes of the vertices. Figure 4.3 shows the workflow of the temporal degree evolution operator which calculates temporal degree metrics within the Flink implementation. It follows the algorithm steps proposed by Rost et al. in [9] and has been recaptured in the previous Section 4.1.1, the implementation of the algorithm is recaptured in this Section 4.1.2.

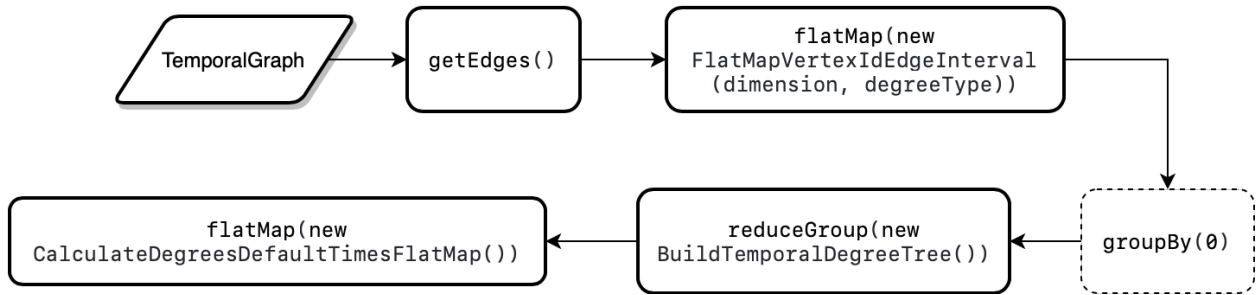


Figure 4.3.: Workflow of Apache Flink implementation

The operator begins with getting all the edges from the input data temporal graph and returns a DataSet holding all the edges with temporal properties (that is, to realize Step I in Section 4.1.1). The identifiers of source and target (that is, the start and end vertex of each edge in the directed graphs), as well as the time interval from the properties of the edges will be extracted from the temporal Edges via the `FlatMap` transformation, which is the Step II *interval collection* in Section 4.1.1. Afterwards, to obtain the degree tree of Step III in Section 4.1.1, the implementation constructs a `TreeMap` with the corresponding temporal degree for each identifier (i.e. sourceId and targetId) by using `GroupBy` and `ReduceGroup` transformations, finally the implementation calculates the temporal degree based on the Algorithm I of Step IV in Section 4.1.1.

For each Temporal Edge in the DataSet, the temporal information needs to be extracted. Thus, it is required to use `FlatMap` transformation to process all the temporal edges data. `FlatMap` applies the `FlatMapVertexIdEdgeInterval` method on each separate Temporal Edge to extract the temporal information, flatten the data and then return them. `FlatMapVertexIdEdgeInterval(dimension, degreeType)` is a user-defined function, which needs to configure the parameters time dimension and degree type. As mentioned in Section 3, there are four time properties (i.e., `val_from`, `val_to`, `tx_from` and `tx_to`) in TPGM, they respectively belong to two time dimensions, that is, valid time and transaction time. The degree types `IN` and `OUT` are the indegrees and the outdegrees of the vertices in a graph, except that there is an additional `BOTH` option in the implementation to calculate both of these two different degrees.

The Flink implementation creates a temporal degree tree for each vertex identifier using these data. In order to construct a temporal degree tree for each id of vertices, the previous data needs to be grouped by the identifier of vertex first. Notably, `GroupBy` does not have a corresponding operator,

⁹<https://flink.apache.org>

¹⁰<https://github.com/dbs-leipzig/gradoop/blob/develop/gradoop-temporal/src/main/java/org/gradoop/temporal/model/impl/operators/metric/TemporalVertexDegree.java>

it is only an intermediate or auxiliary step in generating a DataSet transformation. Therefore, the `groupBy()` method will first be called by passing the parameter 0, which is the first element of the triple, i.e. the identifier of a vertex. The returned data from `GroupBy` is an unsorted group of triples (named as `Tuple3` in Flink), the triples include the identifier and the time interval `from` and `to` of the vertices. Then, the `ReduceGroup` operator of Flink creates an instance of the class `BuildTemporalTree` for each identifier of the vertices. The final result consists of a quadruple in the form of $\langle vid, \omega_{from}, \omega_{to}, degree \rangle$, containing the identifier of the vertex, the start and end time points of the time interval, as well as the degree value for this time interval, which can be represented by the example result of Step IV in Section 4.1.1 as: $\langle id_1, -9223372036854775808, 1, 0 \rangle$, $\langle id_1, 1, 2, 1 \rangle$, $\langle id_1, 2, 3, 2 \rangle$, $\langle id_1, 3, 4, 1 \rangle$ and $\langle id_1, 4, 9223372036854775807, 0 \rangle$ ¹¹.

4.2. Java Based Single Machine Implementation

The primary goal of the single machine applications is to generate the same computational results in terms of temporal degree as the distributed application when processing same data sources. In order to improve the comparability between the single machine application and distributed application, Java programming language will be chosen for designing the single machine implementation of this thesis, and the logic of the programming will be as near as possible to the distributed one. For example, the construction and traversal of the degree tree followed the approximation of the algorithm presented by Rost et al. in [9].

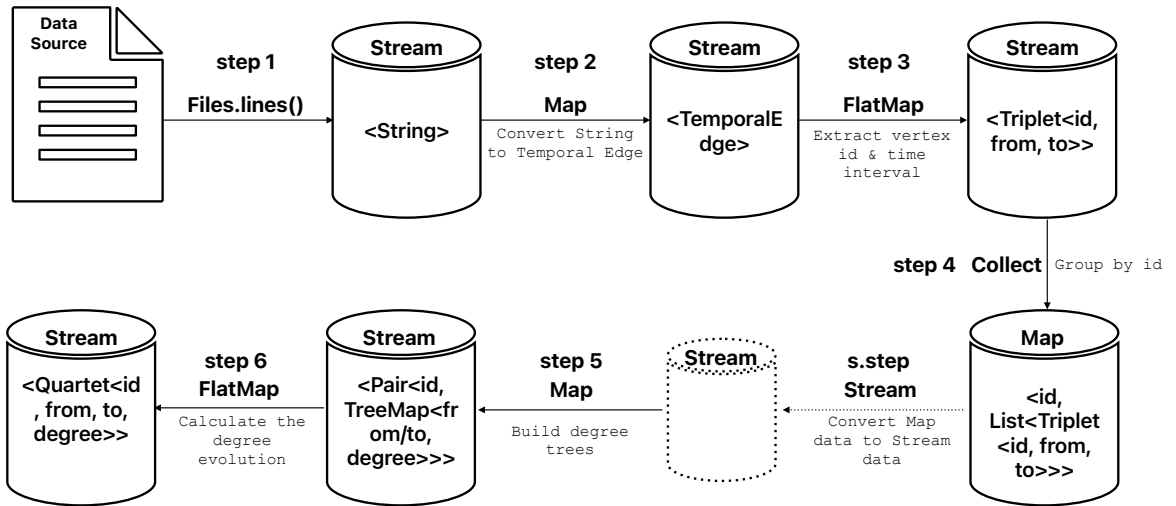


Figure 4.4.: Workflow of single machine implementation

Figure 4.4 shows the flow of data conversion in the single machine implementation. With the `File` class provided by Java, single machine implementation first reads the data from the data source line by line and converts them to Stream of `String`. In the second step, `String` Stream are mapped

¹¹The default minimum and maximum timestamps are the minimum value and the maximum value of long data type, that is, -9223372036854775808 and 9223372036854775807 .

to Stream of `TemporalEdge`, which contains the identifier of the edge, source and target vertices, as well as the time intervals of valid time and transaction time. Similar to Step I *Edge mapping* of Section 4.1.1, the single machine application extracts the vertex information and time intervals from the edge data. Different from the distributed implementation is that after reading the dataset, the single machine implementation needs an additional `StringToEdge` Class to organize the String data, as shown in Listing 4.1. By taking the value of the corresponding position according to the separator, each `String` will be converted into an instance of `TemporalEdge`, including the identifiers of graph, edge, source vertex and target vertex (which are converted to integers using their corresponding hashcodes to reduce memory usage), transaction time and valid time, etc.

```

1  public class StringToEdge {
2      public static TemporalEdge convertToTemporalEdge(String s){
3          String[] temp = s.split(";");
4          String[] timeString = temp[temp.length-1].split("\\\\",\\"(");
5          String valid_time_string = timeString[1];
6          String transaction_time_string = timeString[0];
7
8          // get the time via pattern and matcher
9          Pattern pattern = Pattern.compile("-?\\d+");
10         Matcher valid_matcher = pattern.matcher(valid_time_string);
11         Matcher tx_machter = pattern.matcher(transaction_time_string);
12
13         ArrayList<String> v_timeArray = new ArrayList<>();
14         ArrayList<String> tx_timeArray = new ArrayList<>();
15
16         while(valid_matcher.find()){
17             v_timeArray.add(valid_matcher.group());
18         }
19         while(tx_machter.find()){
20             tx_timeArray.add(tx_machter.group());
21         }
22
23         // the time interval for valid and transaction time
24         Pair<Long, Long> valid = new Pair<>(Long.valueOf(v_timeArray.get(0)),Long.
            valueOf(v_timeArray.get(1)));
25         Pair<Long, Long> transaction = new Pair<>(Long.valueOf(tx_timeArray.get(0)),
            Long.valueOf(tx_timeArray.get(1)));
26
27         return new TemporalEdge(temp[0].hashCode(), temp[1].hashCode(), temp[2].
            hashCode(), temp[3].hashCode(), temp[4], temp[5], transaction, valid);
28     }
29 }

```

Listing 4.1: Converting String data to TemporalEdge

In the following step 3, `FlatMap` will extract the identifiers of the vertices and the time interval from `TemporalEdge` Stream. The type of degree (if `IN`, the identifier of the target vertex will be fetched, if `OUT`, the identifier of the source vertex will be fetched, or if `BOTH`, the identifiers of both

vertices will be extracted) and the type of time dimension (`VALID_TIME` and `TRANSACTION_TIME`) are configurable. The corresponding result `Stream` of triples $\langle id, from, to \rangle$ are obtained according to the configuration.

Step 4 is grouping the `Stream` of the triples based on the identifier of vertex and returns `Maps` with `key` as the identifier of each triple and `value` as a list containing the elements of the group (that is, the triple with `id`, `from` and `to`). It is worth noting that the data stream is interrupted after this step since the `groupingBy()` function provided by `Stream-API` needs to collect all the data stream into memory. Hence, after grouping, the whole `Map` data needed to be reconverted to the stream, which marked as `s.step` in Figure 4.4.

Thereafter, a user-defined function is called in step 5 to construct the received group to a degree tree and assign it to the matching vertex identifier. The result is a `Pair` containing the vertex identifier and its corresponding degree tree. The degree tree is constructed and traversed using the same algorithm as steps III and IV in Section 4.1.1. In the last step, marked as step 6, `FlatMap` is applied on `Pair Stream`, which traverses the degree tree and produces the computed results. For each input `Pair` object, the transformation produces at least one result in the form of quartet that consists of the vertex identifier, the whole time interval, and the corresponding degree.

```

1  public class TemporalVertexDegree_SM {
2      ...
3      // 1) read data from file
4      try (Stream<String> tempStream = Files.lines(file_input)){
5          Stream<Quartet<Integer, Long, Long, Integer>> temporalDegree = tempStream
6          // 2) Get all Edges
7          .map(StringToEdge::convertToTemporalEdge)
8          // 3) Extract vertex id(s) and corresponding time intervals
9          .flatMap(new VertexIdEdgeInterval(degreeType, dimensionType))
10         // 4) Group them by the vertex id
11         .collect(Collectors.groupingBy(Triplet::getValue0))
12         // 4.5) Convert the results to Stream
13         .entrySet().stream()
14         // 5) Build DegreeTrees for each Vertex
15         .map(BuildTemporalDegreeTree::makeTree)
16         // 6) For each vertex, calculate the degree evolution and ouput a tuple4
17         .flatMap(new DegreeCalculatorDefault());
18         ...
19     }
20     ...
21 }

```

Listing 4.2: A draft of the main steps of the single machine implementation

Listing 4.2 shows the main code corresponding to the data processing flow of the single machine implementation. After reading the dataset, lines 7 to 9 take the data read from the dataset (which contains all attributes of the edges) and extract the required vertex identifiers as well as time intervals according to the configured degree type and the temporal dimension type, lines 11 to 13, sort the data by vertex identifiers, as the stream is interrupted after execution in line 11, it will be

converted to a stream again in line 13, lines 15 and 16 are for constructing degree trees of vertices and calculating the evolution of degrees using the degree trees.

4.3. Cypher Queries in Neo4j

In order to implement the graph database operations in Neo4j and to calculate the evolution of temporal degrees, it is necessary to read the data and store them in the graph database of Neo4j, that is, to create a relationship graph in Neo4j. Afterwards, the graph data is manipulated and the final result is calculated. Following are the main steps (note that, the examples used in all steps are bike rental networks with Stations as vertices and trips as edges):

I. Nodes creation

Nodes are divided into start and end nodes. The start and end nodes are respectively the start and end vertices of the edges, which have the identifiers of the vertices as their attributes. The **CREATE** operation imports vertices for each edge sequentially, and creates multiple identical nodes when the edges have the same start or end vertices, thus duplicate nodes need to be deleted manually. In contrast, creating nodes using the **MERGE** operation does not import duplicate vertices, however, if the amount of data is increased, the time spent for creating nodes increases since the **MERGE** operation requires traversing the already existing nodes in the graph database before creating them.

```

1  // creating start_nodes
2  :auto USING PERIODIC COMMIT 500
3  LOAD CSV FROM "file:///gradoop_SF1/edges.csv" AS line
4  FIELDTERMINATOR ';'
5  CREATE (: Station {id: line[2]})
6
7  // creating end_nodes
8  :auto USING PERIODIC COMMIT 500
9  LOAD CSV FROM "file:///gradoop_SF100/edges.csv" AS line
10 FIELDTERMINATOR ';'
11 CREATE (: Station {id: line[3]})
12
13 // deleting duplicate nodes
14 CALL apoc.periodic.iterate(
15 "MATCH (n:Station) WITH n.id AS id, collect(n) AS nodeslist
16  RETURN nodeslist",
17 "WHERE size(nodeslist)>1
18  FOREACH(n in tail(nodeslist) | DETACH DELETE n)",
19 {batchSize:1000, parallel:true, iterateList:true}
20 )YIELD batches, total

```

Listing 4.3: Example of creating start and end nodes

Listing 4.3 is the Cypher statements example for creating the start and end nodes and deleting the duplicate nodes. Nodes are created using the periodic commit once every 500 records (lines 2

to 5 are the start nodes creation and lines 8 to 11 create the end nodes). From lines 14 to 20, the periodic iteration method provided by APOC¹² is called to remove the duplicate nodes.

II. Relationships creation

As shown in Listing 4.4, relationships between nodes are also created using `CREATE`, where the start and end vertices of the edge are matched and the time interval property of the edges is stored as a property of the relationship. The relationships are created using periodic commit, set to a value of 500, and the relationship properties in the dataset are read by line (lines 2 to 7), lines 8 to 9 match the relationships to the corresponding start and end nodes, then in line 10, store the values in the corresponding properties.

```

1  // creating relationships
2  :auto USING PERIODIC COMMIT 500
3  LOAD CSV FROM "file:///gradoop_SF100/edges.csv" AS line
4  FIELDTERMINATOR ','
5  WITH line, split(line[6], "),(") AS time_all
6  WITH line, split(time_all[1], ',') AS valid
7  WITH line, valid[0] AS v_from, replace(valid[1], ',', '') AS v_to
8  MATCH (a: Station), (b: Station)
9  WHERE a.id = line[2] AND b.id = line[3]
10 CREATE (a)-[:Trip {val_from: toInteger(v_from), val_to: toInteger(v_to)}]->(b)

```

Listing 4.4: Example of creating relationships

III. Building "temporal trees"

The example bike-sharing graph inside is created after the nodes and relations are created. Depending on the type of the vertex degree used to calculate the temporal degree evolution, it is necessary to obtain the degree type corresponding nodes from the edges (i.e., relationships in the graph database) and group them according to the vertex identifier, in order to construct a "tree" for each vertex that resembles in the java implementation. The tree construction is to create **DEGREE** relationships for each vertex, each degree relationship connects the vertex and its degree node, the degree node has attributes timestamp and degree, each vertex and all its degree relationships that constitute the corresponding "temporal tree" as Listing 4.5.

Similar to Step II of Section 4.1.1, the vertices and times corresponding to the degree types need to be fetched, thus in Neo4j, different vertices need to be matched depending on the degree type. Assuming that the degree of type both is needed, create vertices for both the indegree (line 3) and the outdegree (line 4), and store the time interval information in the vertices.

In Neo4j, creating new relationships and nodes for each vertex to represent the nodes of the tree structure in Java instead, and using the `count()` method at the corresponding time point to calculate the payload at time points similar to Step III in Section 4.1.1, via the result of `count(from)` to represent the start time point ω_{from} of the interval, and the result of `-count(to)` represents the end time point ω_{to} (lines 9 to 16). In this step, to reduce memory usage, all statements are batched and executed using `apoc.periodic.iterate()`.

¹²<https://neo4j.com/developer/neo4j-apoc/>

```

1  CALL apoc.periodic.iterate(
2  "MATCH (a:Station)-[b:Trip]->(c:Station) RETURN a,b,c",
3  "CREATE (:vertex {vid: a.id, from: b.val_from, to: b.val_to})
4  CREATE (:vertex {vid: c.id, from: b.val_from, to: b.val_to})",
5  {batchsize:5000})
6
7  // building temporal trees
8  CALL apoc.periodic.iterate(
9  "MATCH (n:vertex) RETURN n.vid AS nid,
10  n.from AS FROM, count(n.from) AS FROM_D,
11  n.to AS TO, -count(n.to) AS TO_D",
12  "MATCH (a:Station) WHERE a.id = nid
13  CREATE (b: timeDegree {timestamp: FROM, Degree:FROM_D})
14  CREATE (a)-[:DEGREE]->(b)
15  CREATE (n: timeDegree {timestamp: TO, Degree:TO_D})
16  CREATE (a)-[:DEGREE]->(n)",
17  {batchsize:5000})

```

Listing 4.5: Example of creating trees

IV. Calculating the evolution of temporal degrees

The computation of temporal degree evolution involves the user-defined procedure using Neo4j, due to the difficulty of achieving flow control such as if else or iteration like foreach in the cypher language. The user-defined procedure performs the computation of temporal degree evolution based on the vertices and all degree nodes under their corresponding **DEGREE** relationships. Matching all **DEGREE** relationships in the graph database and passing the node identifier with the degree nodes as parameters produces the same result as the distributed or Java single machine implementation, which is generated by creating new nodes in Neo4j, i.e. the result nodes contain the identifier, the time interval from, to and the corresponding time degree.

By using `apoc.periodic.iterate()`, Listing 4.6 shows the calculation of evolution of temporal degree. The **MATCH** statement in line 2 is used to match the degree tree corresponding to the vertices, because it is not a real tree structure as in Java, the statements in lines 3 to 4 are needed to sort the timestamps and return the order of the nodes as shown in Step IV (line 3 of Algorithm 1) of Section 4.1.1 after traversing the tree structure. The algorithm for degree evolution (lines 7 to 34 of Listing 4.7) is similar to Algorithm 1 (lines 4 to 15), with the difference that the calculation requires calling the user-defined procedure in Neo4j (line 5 in Listing 4.6) and the result is stored as properties in the new nodes.

```

1  call apoc.periodic.iterate(
2  "MATCH (a:Station)-[:DEGREE]->(b:timeDegree) WITH a,b
3  ORDER BY b.timestamp
4  RETURN collect(b) AS ns, a.id AS id",
5  "CALL com.oem.calDegree(ns,id)",
6  {batchsize:5000}
7  )

```

Listing 4.6: Example of calling user-defined procedure


```

1  public class DegreeCalculator {
2      @Context public GraphDatabaseService graphDb;
3      @Procedure(mode = Mode.WRITE)
4      @Description("com.lxx.neo4j.calDegree(nodes, vertexId)")
5      public void calDegree(@Name("nodesList") List<Node> nodes, @Name("vertexId")
        String id){
6          ...
7          for (Node node : nodes) {
8              ...
9              if (lastTimestamp.equals(timestamp)) {
10                 degree += degree_of_timestamp;
11                 continue;
12             }
13             if (degree_of_timestamp != 0) {
14                 ...
15                 Node node_temp = tx.createNode(Label.label("TVD"));
16                 node_temp.setProperty("vid", id);
17                 node_temp.setProperty("from", lastTimestamp);
18                 node_temp.setProperty("to", timestamp);
19                 node_temp.setProperty("degree", degree);
20                 tx.commit();
21                 ...
22                 degree += degree_of_timestamp;
23                 lastTimestamp = timestamp;
24             }
25         }
26         if (lastTimestamp < vertexToTime) {
27             ...
28             Node node_temp = tx.createNode(Label.label("TVD"));
29             node_temp.setProperty("vid", id);
30             node_temp.setProperty("from", lastTimestamp);
31             node_temp.setProperty("to", vertexToTime);
32             node_temp.setProperty("degree", degree);
33             tx.commit();
34         }
35         ...
36     }
37 }

```

Listing 4.7: Example user-defined procedure structure of temporal degree calculation

5. Evaluation

In this Section, the evaluation of distributed execution, Java single machine execution and Neo4j will be performed through benchmarking. All experiments were run on the same physical device and the data sets used for testing were also identical.

5.1. Benchmark Setup and Environment

Running Environment: The running environment for the evaluation is a cluster called Athena, which is composed of 16 worker nodes connected via 1 GBit Ethernet. Each worker node is configured with an E5-2430 6-core 12-thread 2.5Ghz CPU and 48GB RAM.

Dataset: The datasets used for the benchmark are CitiBike¹³ (SF (Scale Factor) = 1, 10 and 100), Stackoverflow¹⁴, Stackexchange (chess, math and writers) and LDBC (SF = 1, 10 and 100). The CitiBike data and the Stackoverflow data are both real world data. LDBC is a synthetical social network generated by LDBC generator [23].

Graphs are stored distributed on the Hadoop Distributed File System (HDFS) [24] by hash partitioning as two datasets V and E . The vertex dataset V will not be used in evaluation, so the data read is the edge dataset E . Table 5.1 shows the statistics of the datasets used.

	$ V $	$ E $	Size (GB)
CitiBike SF1	1100	0.97 M	0.23
CitiBike SF10	1137	9.76 M	2.26
CitiBike SF100	1174	97.6 M	22.60
LDBC SF1	3.2 M	17.3 M	4.20
LDBC SF10	30.0 M	176.6 M	42.30
LDBC SF100	282.6 M	1.77 B	421.90
Stackoverflow	462.9 M	664.8 M	199.00
Stackexchange Chess	150.4 K	159.9 K	0.05
Stackexchange Writers	285 K	310.6 K	0.10
Stackexchange Math	16.6 M	22.3 M	6.68

Table 5.1.: DataSets statistics with number of vertices and edges as well as disk size on HDFS

5.1.1. Distributed Implementation

The benchmark for the distributed implementation runs for 3 rounds, Apache Flink runs with version 1.9.3, the time dimension type is `VALID_TIME`, and the types of time degree `IN`, `OUT` and `BOTH` are measured. Since there are 16 machines in the cluster, that is, the amount of TaskManager is 16 and the number of slots available for each TaskManager is 36, the minimum setting of Flink parallelism for benchmarking is 36 and the maximum setting is 576. Each experiment consists of

¹³<https://citibikenyc.com/system-data>

¹⁴<https://archive.org/details/stackexchange>

reading the graph dataset from HDFS, executing a specific workflow, and finally writing all the results back to HDFS. The evaluation results are averaged over three rounds of running time.

5.1.2. Java Single Machine Implementation

Benchmarking of Java single machine implementation executed in three rounds with Java 8. Maximum memory setting for the JVM is 330GB. The time dimension type is `VALID_TIME`, and the types of time degree `IN`, `OUT` and `BOTH` are used. The different datasets stored distributed in HDFS have been respectively merged into a single CSV file and saved in a local SSD hard disk. Each experiment consists of reading the edge datasets from the hard disk, executing the workflow and writing the final results back to the hard disk. As in the distributed benchmark, the evaluation results are the average of three rounds of runtime.

5.1.3. Neo4j

Neo4j runs with the version `neo4j-community-4.4.10` on the same environment as the single machine implementation and uses the same merged dataset. The time consumed by Neo4j to create nodes and relationships is not included in the runtime, nor is the writing of computation results to the hard disk. Therefore, the runtime only includes the time to operate on the graph database and generate the computation results.

5.2. Results Comparison

The evaluation results for each dataset are presented as three parts, i.e. three line charts with degree types `IN`, `OUT` and `BOTH` from left to right. The vertical coordinate is the runtime in seconds and the horizontal coordinate is the parallelism setting for the distributed implementation. Since such a setting is not expected for the single machine applications, the results for the Neo4j and Java single machine implementation are in horizontal segments, i.e., parallelism unchanged. It should also be noted that since Neo4j's runtimes under the datasets CitiBike with SF1, SF10, math of Stackexchange, and LDBC with SF1 differ significantly from the other two applications, a logarithmic scale is used for the vertical coordinate runtimes to facilitate comparative observations.

5.2.1. CitiBike

Figure 5.1 shows the results for the CitiBike dataset at SF1. For all three different degree types, Neo4j requires the longest runtime, because of the communication overhead, the matching of relationships and the creation of nodes. Java single machine implementation has the shortest runtime, while at type `BOTH`, the runtime of the distributed application Flink is same as Java single machine at the minimum parallelism of 36, but the rest lags behind Java single machine application, the runtime increases with the increase of parallelism.

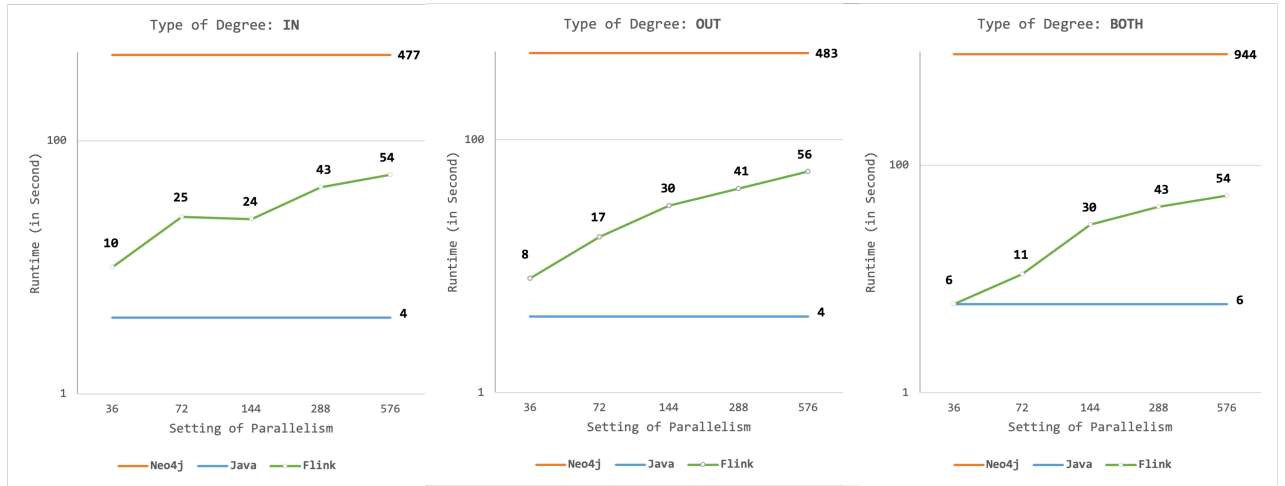


Figure 5.1.: Runtimes of CitiBike SF1

The results for CitiBike at SF10 shows in Figure 5.2. Neo4j is the slowest of all the three programs, runtime of Flink application increases with its increasing parallelism, for degree IN, Flink is consistently faster than Java single machine of 40 seconds when parallelism less than 288. For type OUT, Java single machine also outperforms Flink at parallelism 288. With BOTH type, Flink's runtime is ahead of Java single machine by 60 seconds at all parallelism degrees and still has an advantage of 3 seconds at the maximum parallelism degree of 576.

It is worth noting that for Citibike SF1 and SF10, the runtime increases with the number of workers. This is mainly due to the communication overhead of such small datasets. The optimal result is supposed to be an inverse line, i.e., the runtime decreases as the number of workers increases.

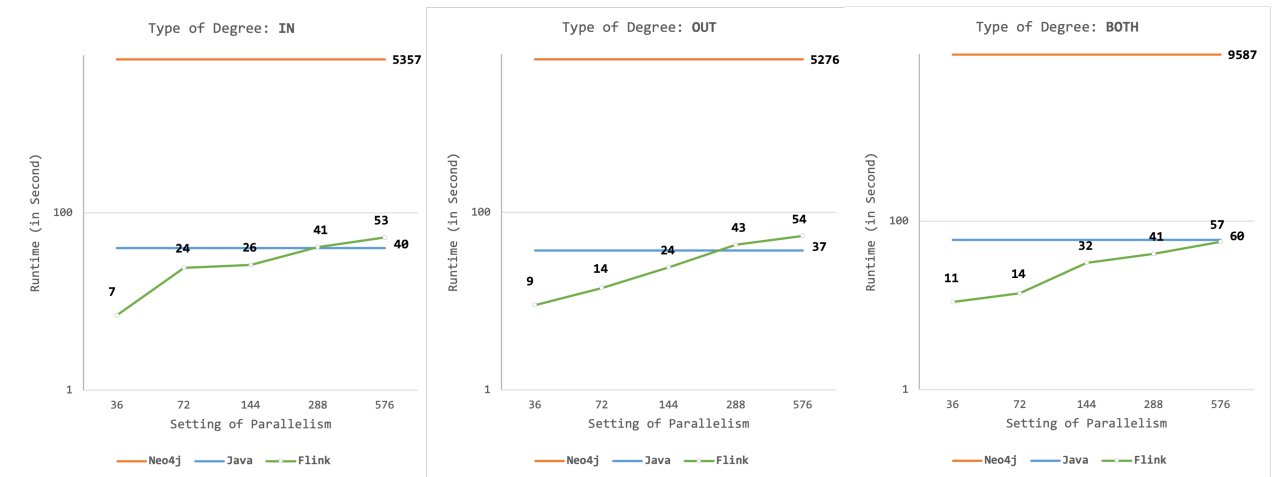


Figure 5.2.: Runtimes of CitiBike SF10

In the evaluation using CitiBike SF100, Neo4j was able to import data and create complete nodes and relationships, however, the number of nodes and relationships was too large during the graph database manipulation, leading to an Out-of-Memory error. Thus the results of Neo4j could not be generated. The results in Figure 5.3 show that the Flink implementation is substantially ahead of the Java single machine for all three degree types, at type IN, the longest running time is at parallelism of 36, the shortest at parallelism of 72, then the running time becomes longer with

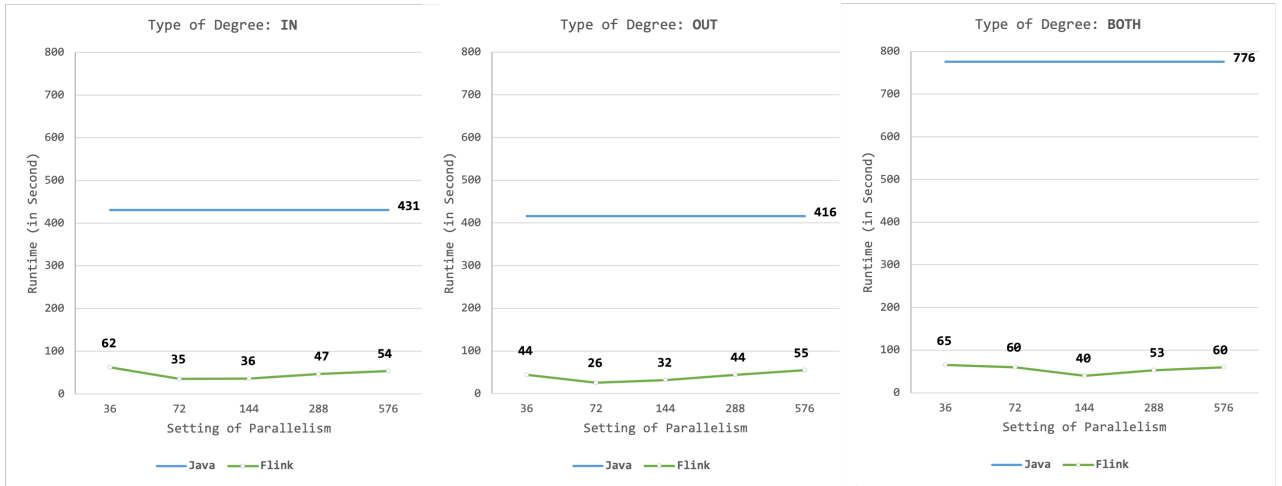


Figure 5.3.: Runtimes of CitiBike SF100

increasing parallelism. For type OUT, the best runtime for Flink is also at parallelism 72, where the runtime decreases with parallelism until then, and increases with parallelism after parallelism 72. The distributed implementation behaves like a single-threaded implementation, i.e., increasing the number of workers does not make a fundamental difference to the change in runtime. For CitiBike SF100, a parallelism setting of 72 is optimal, that is, the number of workers is not as many as necessary, it needs to be adapted to the dataset size.

5.2.2. Stackexchange

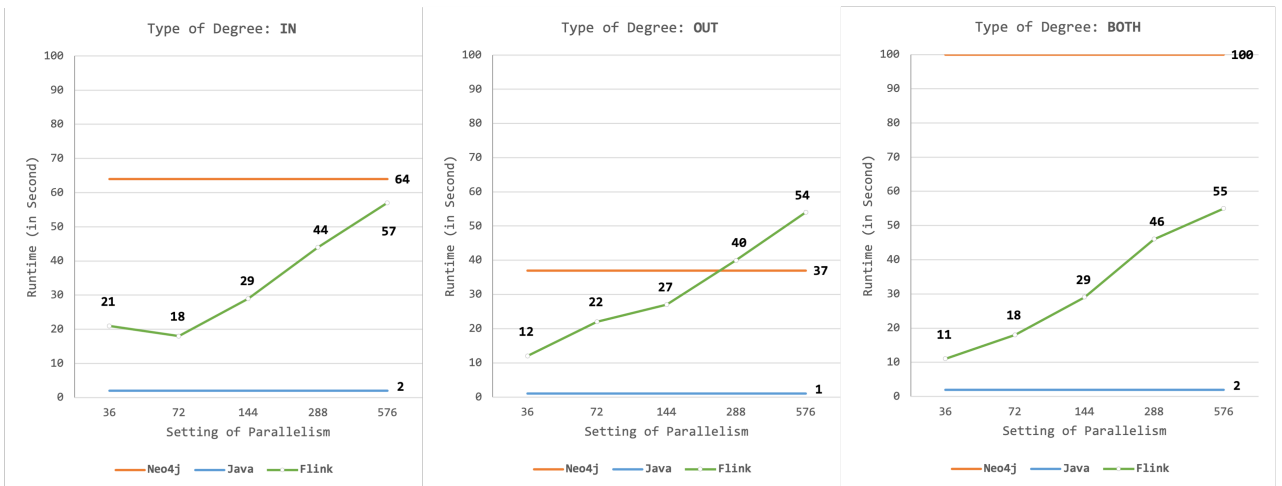


Figure 5.4.: Runtimes of Stackexchange Chess

The chess dataset of Stackexchange is the smallest used in the evaluation, with less than 30 megabytes of its edge dataset. The results in Figure 5.4 illustrate that the Java single machine application has the fastest processing speed of 1 or 2 s for all the three degree types. The running time of the Flink implementation almost always (IN type, slightly faster at parallelism 72 than parallelism 36) grows with parallelism and is faster than Neo4j at all parallelism settings when the degree type is IN and BOTH. Neo4j has a smaller gap between its runtime and the slowest Flink runtime in a dataset of this size and outperforms the distributed execution with parallelism settings

of 288 and 576 at the OUT type.

Figure 5.5 shows the benchmarking results for the dataset Stackexchange writers. While Java single machine execution has the shortest runtime over the remaining two implementations, Neo4j takes the longest time. Flink’s runtime scales up with increasing amount of parallelism.

Stackexchange math evaluation results demonstrate in 5.6 that the distributed computation is faster than both single machine applications. Flink runs at types IN and OUT with increasing parallelism, and at type BOTH, the running time decreases gradually from parallelism 36 to 144, reaching a minimum running time at parallelism 144, after which the running time starts to increase and obtains the slowest result at the maximum parallelism 576. Neo4j consistently lags significantly behind the Java single machine execution and the Flink Distributed execution. Its runtime is about 150 times slower than Java single machine when type IN and OUT, and the difference reaches almost 260 times in type BOTH.

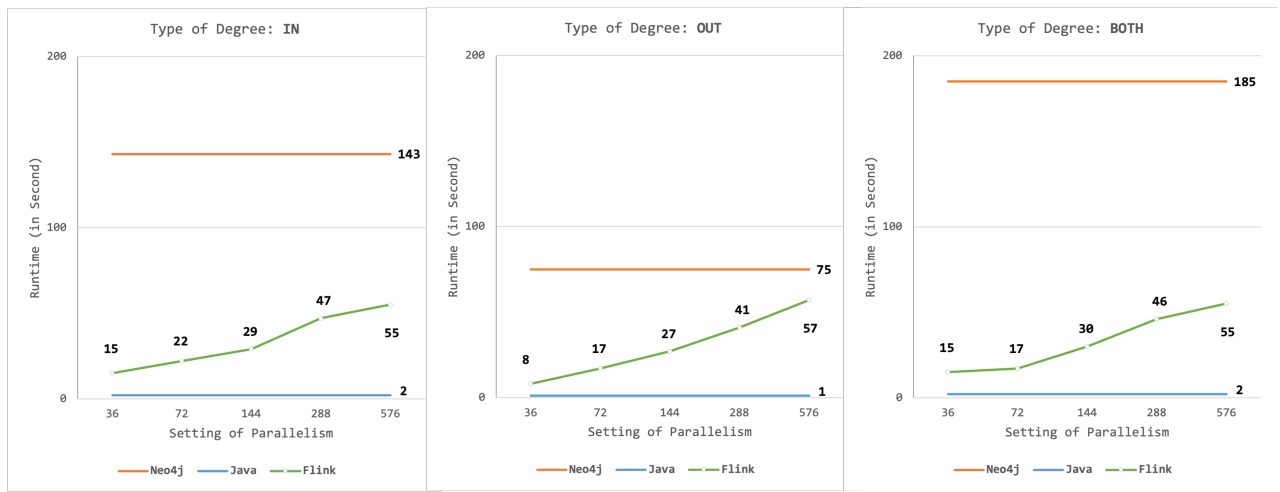


Figure 5.5.: Runtimes of Stackexchange Writers

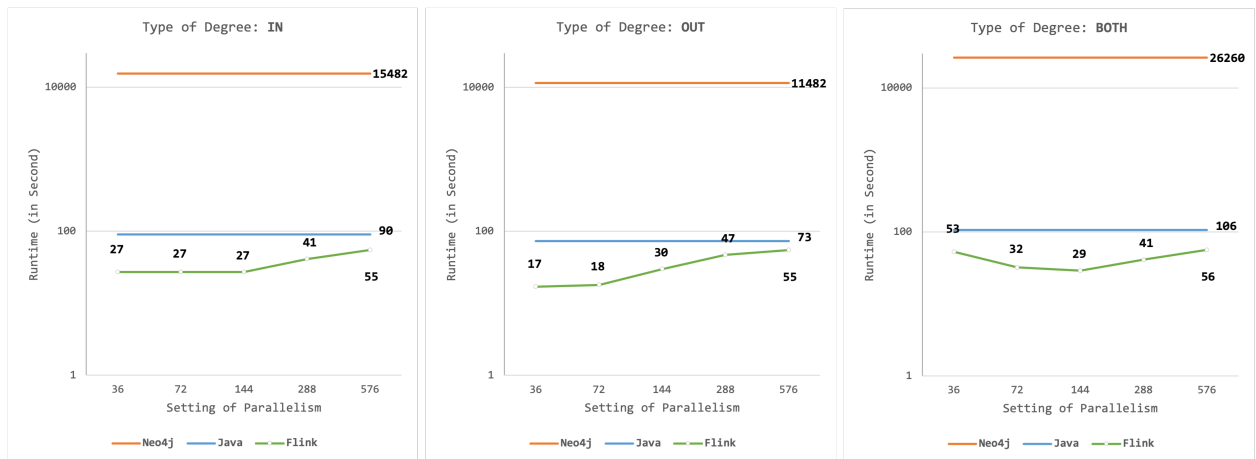


Figure 5.6.: Runtimes of Stackexchange Math

5.2.3. LDBC and Stackoverflow

The results presented in Figure 5.7 are the experiments with LDBC at SF1. Neo4j consumes the most running time in all cases. Running time of the distributed implementation grows with increasing parallelism (except at degree type OUT with parallelism 72, which is one second faster than with parallelism 36) and is faster than the Java single machine under all parallelism configurations, only when the degree type is IN and parallelism reaches a maximum of 576, which is about 10% slower than the Java single thread implementation.

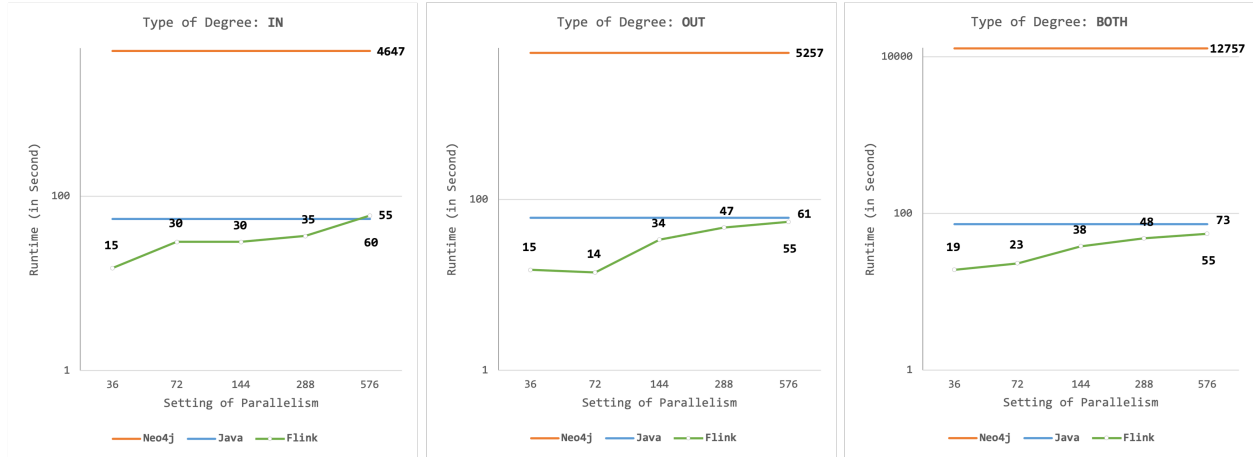


Figure 5.7.: Runtimes of LDBC SF1

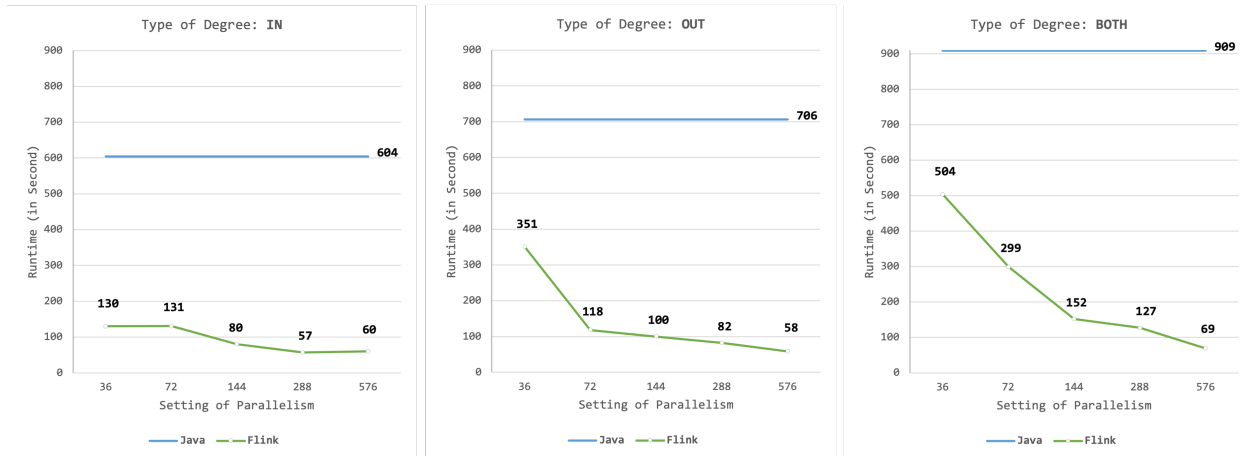


Figure 5.8.: Runtimes of LDBC SF10

In the evaluation with dataset LDBC SF10, Neo4j could not produce results, similar to the situation when CitiBike SF100 was evaluated, Neo4j could only import the data and create the complete nodes and relationships, due to the Out-Of-Memory error generated by intermediate operations that prevented completion of the experiment, hence the runtime of Neo4j is not available in Figure 5.8. In this benchmark, Java single machine application consistently outperforms the distributed application, and the running time of Flink is inversely proportional to the amount of parallelism, i.e., as the parallelism increases, the running time of the Flink application gradually decreases, except for a slight increase of 3 seconds for the IN type and when the parallelism is 576 compared to parallelism 288. It is notable that the running time of the Flink application decreases from 351

to 118 seconds when the parallelism is increased from 36 to 72 when processing the OUT type, which means that doubling the parallelism brings a triple speedup as shown in Figure 5.11. A similar situation occurs for BOTH types, where the parallelism is increased from 36 to 144, reducing the runtime by around 350 seconds, saving more than three times the runtime, of course with a four times increase in parallelism.

For benchmarks using the datasets LDBC SF100 and Stackoverflow, only Flink completed the evaluation, whose results are illustrated in Figures 5.9 and 5.10, respectively. For both datasets, Neo4j was unable to create nodes and relationships completely. Java single machine implementation was terminated early because the grouping operation provided by the Stream API used needed to collect all the data into memory, generating an Out-of-Memory error.

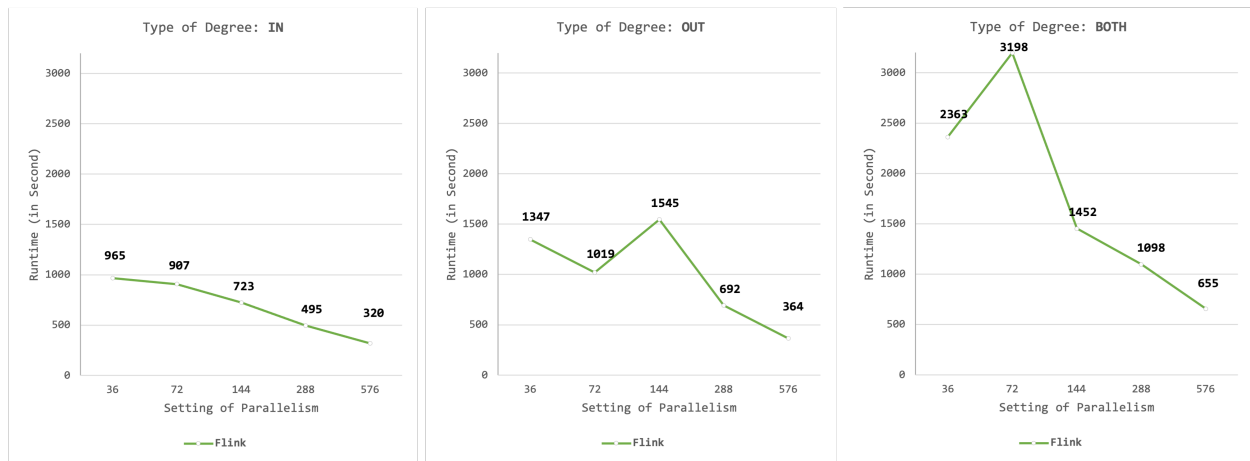


Figure 5.9.: Runtimes of LDBC SF100

In the evaluation of the LDBC SF100 dataset, the runtime reduces with the increasing parallelism when the degree type is IN, with the minimum parallelism being 965 seconds and the maximum parallelism at 320 seconds. With degree type OUT, the overall runtime trend is that increasing parallelism leads to a decrease in runtime, but with a notable exception that when parallelism is configured as 144, the runtime increases from 1019 seconds at parallelism 72 to 1545 seconds, even higher than 1347 seconds at parallelism 36. A similar situation occurs in the evaluation of degree type BOTH, where the runtime increases substantially over the smaller parallelism when the parallelism is 72, i.e., from 2363 to 3198 seconds, and rapidly drops by more than half to about 1450 seconds when the parallelism is increased to 144. Afterwards, the runtime continues reducing slowly as the parallelism increases.

For the same dataset, the number of edges to be processed varies by degree type, and increasing parallelism can lead to an overall decreasing trend in runtime, but at partial parallelism, it may also lead to a worse time overhead.

When benchmarking with Stackoverflow dataset, Flink implementation has a negative correlation between runtime and parallelism setting in all three different degree types, that is, increasing parallelism can speed up the application, the higher the parallelism, the lower the running time. In particular, when processing IN type, the runtime is 1205 seconds at the lowest parallelism degree, and the runtime is reduced to 134 seconds at the largest parallelism, which is a 9 times speedup, where the speedup is most significant at the beginning, when the parallelism degree is increased

from 36 to 72, the runtime is reduced to about 3 times of the initial value, i.e., from 1205 seconds to 449 seconds.

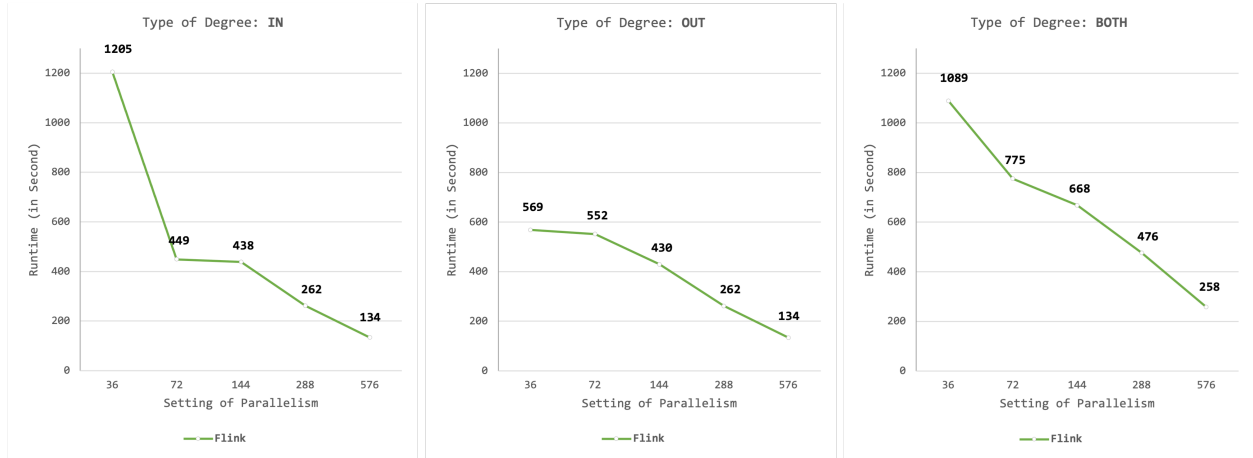


Figure 5.10.: Runtimes of Stackoverflow

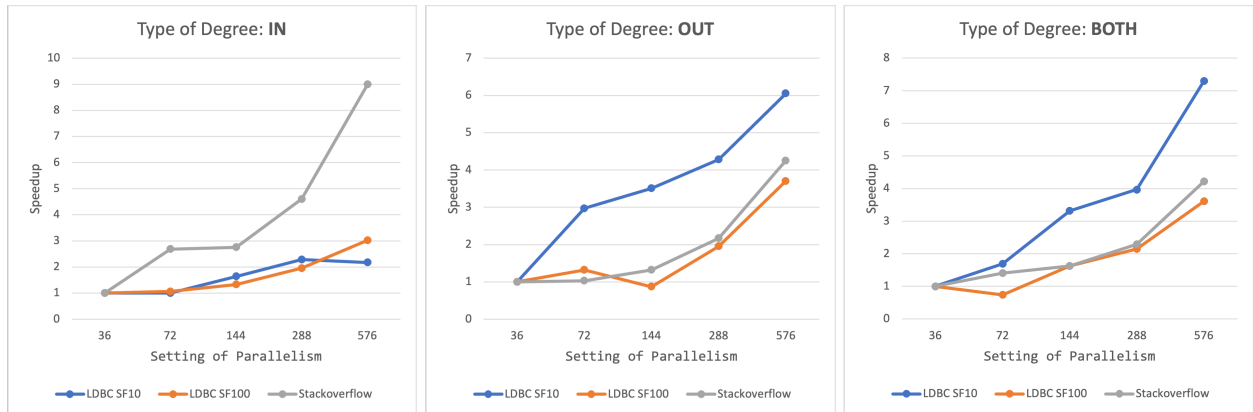


Figure 5.11.: Speedup of LDBC SF10, SF100 and Stackoverflow

5.3. Influencing Factors on Runtime

The analysis of the benchmark results shows that not only the size of the dataset affects applications processing under different technologies, but also the parallelism under different configurations has a significant impact on the runtime when using Flink for distributed processing of data:

I. Impact of dataset size

Despite the feasibility of using Neo4j for temporal degree computation, Neo4j has the longest runtime in almost all benchmarks, with only a partially faster runtime than the distributed computation when evaluated with the chess dataset of Stackexchange. The generation of massive new nodes and relationships during the computation and the processing of those new data are the main factors contributing to the lengthy Neo4j runtime. Memory usage is also an important factor that cannot be ignored. The processing of the hundreds of millions of relationships created in the Neo4j database for large datasets requires a considerable amount of memory, thus some of the datasets evaluated in Neo4j have ended up with the Out-of-Memory error. When the dataset size is within a

few hundred megabytes (e.g. CitiBike SF1, chess and writers of Stackexchange), Java single machine implementation dominates significantly, and its runtime is always faster than Flink's, regardless of how parallelism is tuned. When the dataset size comes in several gigabytes (e.g. CitiBike SF10, LDBC SF1 and the math dataset of Stackexchange) is a boundary, the setting of parallelism in Flink allows it to be faster than Java single machine implementation and also makes it lag behind Java single machine. After evaluating the datasets used over tens of gigabytes, distributed computing is obviously faster than single machine applications. In particular, when processing datasets of hundreds of gigabytes like Stackoverflow and LDBC SF100, a single machine implementation cannot even complete them successfully, because the `groupingBy()` function provided by Java `Stream` is a terminating operation that collects all stream data into memory, so `Stream` will be terminated. Data that exceeds the memory size leads to Out-of-Memory errors for Java single machine application.

II. Impact of parallelism on Flink

Misconfiguring the parallelism will slow down the distributed computation, and the running speed will not always increase as the parallelism increases, that is, the running time and the number of parallelism are not always positively or inversely proportional. For smaller datasets, such as tens to hundreds of megabytes, the processing time of Flink increases from about 10 seconds to about 60 seconds with increasing parallelism, since Flink works in a cluster and communication between nodes in the cluster requires extra time beside the computation itself, such as exchanging intermediate data or operators, increasing parallelism does not bring speedup in these datasets. The evaluation also shows that only for large datasets (e.g. LDBC SF100 and Stackoverflow), setting the parallelism to the maximum achieves the shortest runtime, while the rest of the datasets get the best runtime before the parallelism reaches the maximum.

6. Conclusion and Prospect

When processing graph data, users often face a choice between single machine execution or distributed computation due to the varying sizes of data sets. To explore this dilemma, this thesis designs single machine implementations using Java Stream and Neo4j that can produce the same results as existing Flink distributed application in terms of computing temporal degree metric, as well as using different datasets ranging in size from tens of megabytes to hundreds of gigabytes, benchmarks were conducted to evaluate the running time of each implementation to perform given operations.

The use of Neo4j was considered when designing the Java single machine application, as Neo4j is one of the most widely used graph databases, but the final evaluation results showed that its performance lagged behind the Java single machine and the Java distributed application in most of the datasets used for the evaluation, due to the communication overhead and the matching relationship, as well as the time spent on calling user-defined procedures.

When the Out-of-Memory error occurred in Java single machine application using Stream-API, RxJava¹⁵ was tried instead, but it still ended up with Out-of-Memory error and running time even lagged behind Stream-API, thus it was finally abandoned.

Java single machine implementation designed by using Stream-API shows satisfactory results. When dealing with large datasets, its performance lags behind distributed implementation. However, thanks to the absence of the communication overhead of workers in the cluster, there is a better runtime than distributed implementation when manipulating small dataset.

Further work could be directed at optimizing single machine implementations, for example, by restructuring the processing sequence of `Stream` data, and if a collect operation is unavoidable, attempting to call it as the final operation rather than as an intermediate step, in order to avoid interrupting `Stream` data. Alternatively, libraries other than the Stream API or even other programming languages can be applied. For Neo4j, on the one hand, changing to a commercial version may speed up the process, but there is still potential for optimizing the operation steps involved, for example, whether there is a way to use Cypher queries to implement the computation and avoid using user-defined procedures for degree calculation. On the other hand, a graph database is optimized for path traversals. If having the evolution of degrees as a path, where each edge between two nodes is a time interval holding the value, it could be fast traversed.

For the selection of datasets to be used in the evaluation, more dataset types or larger datasets could be used in the future. As seen in the evaluation of Section 5, the dataset size is an important influence on the runtime, and the number of edges in the dataset should also be noted. If a dataset of several gigabytes contains only a few edges, then the advantages of distributed computing will still not be demonstrated. On the contrary, if the number of edges to be processed in a small dataset reaches a million or more, then the runtime of a single machine implementation can be worrisome.

¹⁵<https://reactivex.io>

Bibliography

- [1] Siddhartha Sahu et al. “The ubiquity of large graphs and surprising challenges of graph processing: extended survey”. In: *The VLDB journal* 29.2 (2020), pp. 595–618.
- [2] Sherif Sakr et al. “The future is big graphs: a community view on graph processing systems”. In: *Communications of the ACM* 64.9 (2021), pp. 62–71.
- [3] Yi Lu et al. “Large-scale distributed graph computing systems: An experimental evaluation”. In: *Proceedings of the VLDB Endowment* 8.3 (2014), pp. 281–292.
- [4] Martin Junghanns et al. “Gradoop: Scalable graph data management and analytics with hadoop”. In: *arXiv preprint arXiv:1506.00548* (2015).
- [5] Christopher Rost et al. “Distributed temporal graph analytics with GRADOOP”. In: *The VLDB journal* 31.2 (2022), pp. 375–401.
- [6] Frank McSherry, Michael Isard, and Derek G Murray. “Scalability! but at what {COST}?” In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. 2015.
- [7] Chris Kemper. *Beginning Neo4j*. Springer, 2015.
- [8] Christopher Rost, Andreas Thor, and Erhard Rahm. “Analyzing temporal graphs with Gradoop”. In: *Datenbank-Spektrum* 19.3 (2019), pp. 199–208.
- [9] Christopher Rost et al. “Evolution of Degree Metrics in Large Temporal Graphs”. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2023)*. LNI. Gesellschaft für Informatik, Bonn, 2023.
- [10] Ariel Debrouvier et al. “A model and query language for temporal graph databases”. In: *The VLDB Journal* 30.5 (2021), pp. 825–858.
- [11] Ciro Cattuto et al. “Time-varying social networks in a graph database: a Neo4j use case”. In: *First international workshop on graph data management experiences and systems*. 2013, pp. 1–6.
- [12] Ilya Verbitskiy, Lauritz Thamsen, and Odej Kao. “When to use a distributed dataflow engine: evaluating the performance of Apache Flink”. In: *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. IEEE. 2016, pp. 698–705.
- [13] Jeyhun Karimov et al. “Benchmarking distributed stream data processing systems”. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 1507–1518.
- [14] Tianqi Chen et al. “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems”. In: *arXiv preprint arXiv:1512.01274* (2015).
- [15] Christoph Boden, Tilmann Rabl, and Volker Markl. “Distributed machine learning-but at what cost”. In: *Machine learning systems workshop at the 2017 conference on neural information processing systems*. 2017.
- [16] Renzo Angles. “The Property Graph Database Model.” In: *AMW*. 2018.

- [17] Christopher Rost, Andreas Thor, and Erhard Rahm. “Temporal Graph Analysis using Gradoop.” In: *BTW (Workshops)*. 2019, pp. 109–118.
- [18] Tom Johnston. *Bitemporal data: theory and practice*. Newnes, 2014.
- [19] Martin Junghanns et al. “Analyzing extended property graphs with Apache Flink”. In: *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics*. 2016, pp. 1–8.
- [20] Fabian Hueske and Vasiliki Kalavri. *Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications*. O’Reilly Media, 2019.
- [21] Jon Louis Bentley. “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* 18.9 (1975), pp. 509–517.
- [22] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.
- [23] Alexandru Iosup et al. “LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms”. In: *Proceedings of the VLDB Endowment* 9.13 (2016), pp. 1317–1328.
- [24] Konstantin Shvachko et al. “The hadoop distributed file system”. In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pp. 1–10.

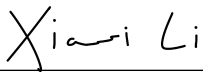
Declaration of Authorship

I do solemnly declare that I have written the presented research thesis:

„Comparison of Single-Machine and Distributed Calculation of Temporal Degree Metrics“

by myself without undue help from a second person others and without using such tools other than that specified. Where I have used thoughts from external sources, directly or indirectly, published or unpublished, this is always clearly attributed. I am aware that infringement can also subsequently lead to the cancellation of the degree.

Leipzig, the 26.12.2022



XIAOXI LI