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

Windowed Graph Stream Pattern Matching using Subgraph-Isomorphism

Bachelor Thesis

Submitted by:
Sung Geun Yun

Matriculation number:
3735467

Supervisor:
Prof. Dr. Erhard Rahm

Abstract

Graph pattern matching has a wide range of applications. However, because of its complexity as an NP-complete problem [1], matching a large graph requires a distributed system to mitigate its difficulty. To conduct graph pattern matching in a distributed system, Apache Flink[1] was adopted by Gradoop [2] as a batch processor using relational algebra operations and by SGraPMa [3] as a stream processor using the Dual Simulation algorithm. This thesis extends the use case of Flink as a stream processor by adopting Ullmann's Subgraph-Isomorphism algorithm [4]. The main contributions of this thesis are the modification of Ullmann's algorithm to fit property graph stream pattern matching with Flink and the evaluation on various scenarios in terms of the number of elements in the graph stream, length of the window and degree of the distribution. The result of evaluation shows the inherent difficulty of the graph pattern matching problem and the limitation of the applicable situation of property graph stream pattern matching with the modified Ullmann's Subgraph-Isomorphism. The evaluation demonstrated that the modified Ullmann's Subgraph-Isomorphism algorithm is capable of conducting low-latency stream pattern matching in small graphs, with up to 1000 events per second throughput and a degree of parallelism up to 144. However, increasing the degree of parallelism too much degrades its performance, causing bottlenecks.

[1]https://nightlies.apache.org/flink/flink-docs-master/, accessed February 2023

Sung Geun Yun
3735467

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

Graph pattern matching is one of the problems in graph theory that has a wide range of applications for solving many real-life problems. For example, a pattern matching program can detect a specific route of travelers in a transportation network or identify a predefined abnormal pattern of transactions in a banking network.

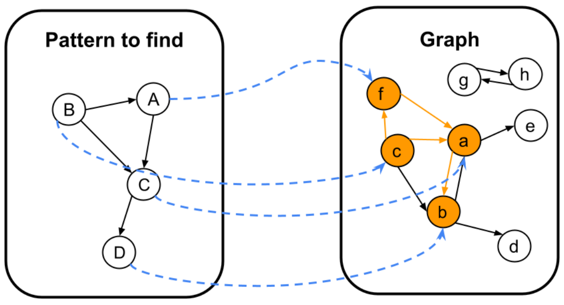


Figure 1.1.: Graph Pattern Matching Problem

In property graphs, where detailed descriptions of graph elements are embedded, it is possible to define the pattern and graph with more detailed information. Adopting the property to graphs leads to a broader scope of graph pattern matching applications.

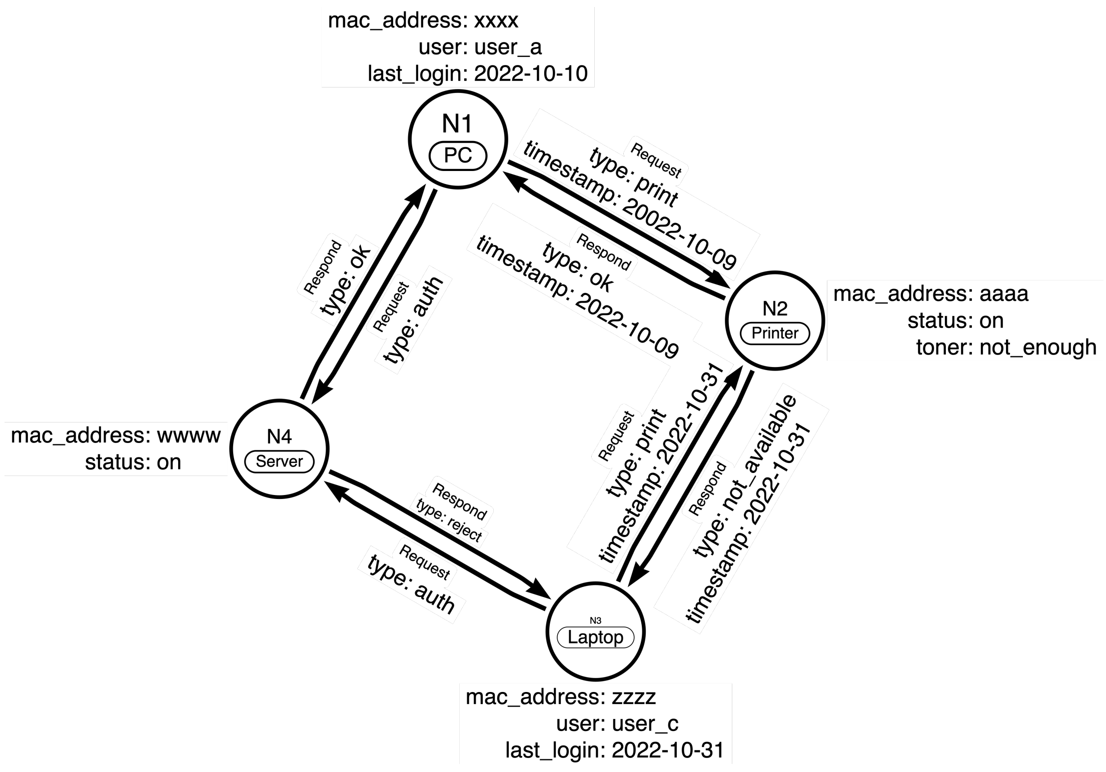


Figure 1.2.: Property Graph

However, conducting graph pattern matching on a large static graph requires a lot of computation with a powerful system to process it. Scalable distributed systems with a static database could be one of the solutions to conduct the task with shorter amount of time. As well as the ability to process a large graph, on the other hand, low-latency processing is also important for the stream pattern matching task. This task takes its input not as a static graph, but as a stream of graph elements. Because of its ability to handle real-time input streams, stream pattern matching applications can solve more real-life graph problems than a static pattern matching application. A distributed stream processing system can provide more ability to stream pattern matching applications. However, it has to be verified whether the distributed stream system can produce the pattern matching result with low-latency.

The Figure 1.3 illustrates a stream of graph elements with their assignments to the time windows. The rounded boxes are time windows of ten seconds in every five seconds, and the circles are vertices of the graph stream and the arrows are the edges between two vertices. The positions of the edges on the time axis are the timestamps of the edges. From this Figure, the next Figure 1.4 demonstrates an example output of the graph stream pattern matching on a time window of [12:02:10 PM - 12:02:20 PM].

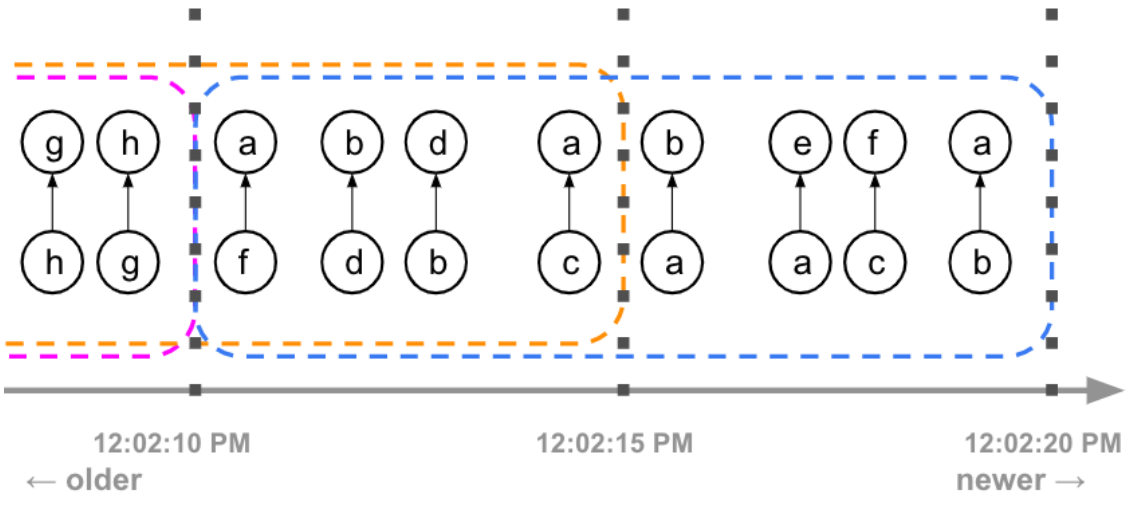


Figure 1.3.: Graph Stream and Time Window

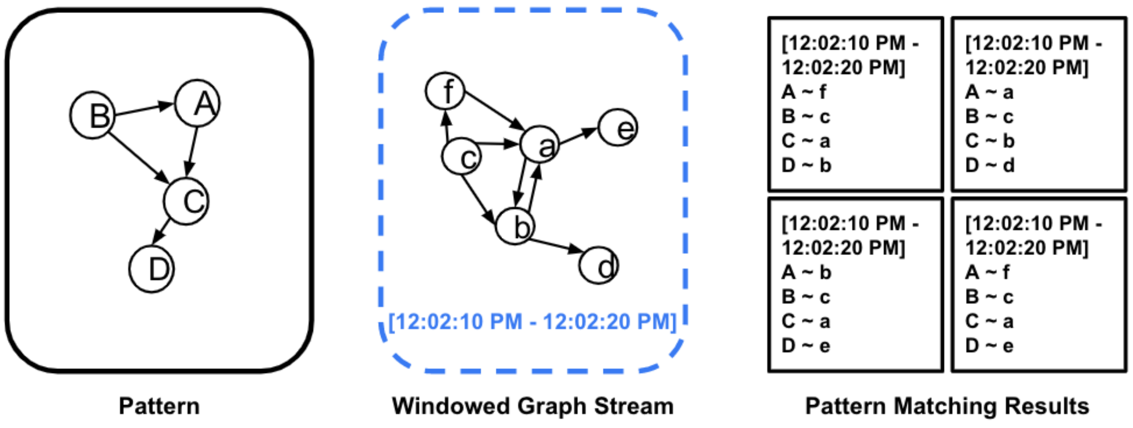

B
A
C
D
Pattern
f
e
c
a
b
d
[12:02:10 PM - 12:02:20 PM]
Windowed Graph Stream
[12:02:10 PM - 12:02:20 PM] A ~ f B ~ c C ~ a D ~ b
[12:02:10 PM - 12:02:20 PM] A ~ a B ~ c C ~ b D ~ d
[12:02:10 PM - 12:02:20 PM] A ~ b B ~ c C ~ a D ~ e
[12:02:10 PM - 12:02:20 PM] A ~ f B ~ c C ~ a D ~ e
Pattern Matching Results

Figure 1.4.: Result of windowed graph stream pattern matching

1.2. Goal of the Thesis

This thesis aims to modify Ullmann's Subgraph-Isomorphism algorithm [4] to be able to perform *Windowed Graph Stream Pattern Matching using Subgraph-Isomorphism* in distributed Apache Flink[2] clusters and to examine if adding distributivity to the modified algorithm can enhance its performance to produce the result with low latency. Ullmann's algorithm will be brocken down into details and will be tailored to benefit from the distributed system Flink. Discussions also include the limitations of distributed computations. At the end, the result of evaluation will find the relation between the degree of parallelism and the difficulty of pattern matching job in terms of the amount of graph elements in a time window.

1.3. Structure of the Thesis

This thesis is structured into 8 sections beginning with this introduction to the motivation and goal of the thesis. Section 2 provides the necessary background on graph theory especially on graph pattern matching, Ullmann's Subgraph-Isomorphism, property graph, and windowed stream processing with Apache Flink. The related work in Section 3 is followed by Section 4 which lists analysis of problems associated with modifying Ullmann's algorithm into distributed property graph stream pattern matching algorithm. In Section 5, a design of the algorithm for graph pattern matching on property graph streams is presented based on the analysis in Section 4. Section 6 describes a prototypical implementation of the algorithm, and Section 7 presents an evaluation of the algorithm's performance. Finally, Section 8 concludes the thesis with discussion on the contributions, limitations, and potential future work of the implemented algorithm.

[2]https://nightlies.apache.org/flink/flink-docs-master/, accessed February 2023

2. Basics

2.1. Graph

Graph G is a tuple (V, E) with a set V of vertices and a set $E \subseteq V \times V$ of edges. An edge $e \in E$ is a connection between two vertices $v_i, v_j \in V \quad i, j \in \{1, 2, ..., |V|\}$. Graphs can be used to represent relations between objects(vertices). For example, a graph of network(G) with several machines(V) and their physical connections(E) or a graph of public transportation(G) with bus stops(V) and the bus lines(E) connecting the bus stops. An edge $e \in E$ can be directed or undirected. When edges have direction from one vertex to other vertex, the graph is called directed graph. Otherwise it is called an undirected graph. An undirected edge can be replaced by two directed edges in opposite directions. Thus, directed graphs can also express undirected graphs by replacing an undirected edge with two edges of opposite directions. This thesis assumes directed graphs, if not explicitly others stated.

In this thesis, two representation for Graphs are mainly used: Adjacency Matrix and Adjacency List. Let us assume that vertices are distinctively labeled with $i \in \{1, 2, ..., |V|\}$. Adjacency Matrix is a square Matrix of $Mat_{|V|\times|V|}$ where i-th element of both row and column refer the same vertex $v_i \in V$. In Adjacency Matrix, the value of each element e_{ij} means the following.

$$e_{ij} = \begin{cases} 1 & \text{if an edge from } v_i \text{ to } v_j \text{ exists} \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

Adjacency List L of a graph $G(V, E)$ is a list of vertex tuples (v_i, v_j) where $i, j \in \{1, 2, ..., |V|\}$. For a e_{ij} in Adjacent Matrix, $e_{ij} = 1 \Leftrightarrow (v_i, v_j) \in L$ and $e_{ij} = 0 \Leftrightarrow (v_i, v_j) \notin L$. Adjacency Matrix and Adjacency List can represent a single graph equivalently, where a single graph means that there exists maximum one edge between two vertices. Otherwise it is called multi graph. Depending on the context of their usage, each way offers different opportunities to utilize. For example, by assigning the number of edges between v_i and v_j to e_{ij} in Adjacency Matrix, a multi graph can be expressed. On the other hand, to express a multi graph with Adjacency List, multiple tuples with same source and target vertex can be added to the list. An advantage of using Adjacency List is that it does not need to keep the set of vertices separately from the set of edges. More over, with a $L' \subseteq L$, an incomplete subgraph of L can be expressed, which is maybe still evolving.

2.2. Graph Pattern Matching

Graph Pattern Matching is one of graph problems. It seeks for a predefined pattern in a given graph. A pattern(or query) is also a graph with a set of vertices and a set of edges. There are two types of graph pattern matching rule - Homomorphism and Isomorphism.

Definition 1 (Graph Homomorphism)**.** Graph Homomorphism is a function $h : G \to H$ between graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ such that if (u, v) is an edge in E_G, then $(h(u), h(v))$

is an edge in E_H. Homomorphism requires to preserve adjacencies, but allow mapping of multiple vertices to the same vertex.

Definition 2 (Graph Isomorphism)**.** Isomorphism is a bijective homomorphism between graphs G and H. An isomorphism is a one-to-one mapping $h : V_G \to V_H$ such that (u, v) is an edge in E_G if and only if $(h(u), h(v))$ is an edge in E_H. Isomorphism requires to preserve adjacencies and injectivity of vertex mapping on both graphs.

In this thesis, mere homomorphism as excluded because it does not require injectivity of vertex mapping function thus distinct pattern vertices could be mapped to one graph vertex [3]. Graph Pattern Matching is also called Subgraph Isomorphism and it is a task to find a subgraph of the original graph that is topologically identical to the predefined pattern. An isomorphism(bijective function) is found when both the vertices and edges of the pattern could be mapped to subset of both vertices and edges of graph respectively. By definition of isomorphism, the bijection between the pattern and the subgraph of the graph requires to meet the none-existency of edges as well. However, one can require only **injective** function from pattern edges to graph edges and allow additional edge in the graph which are not mapped from the pattern. This condition is defined as Injective Homomorphism.

Subgraph Isomorphism as a decision problem is known to be an NP-Complete problem [1]. This means no efficient algorithm has been found to be able to solve this problem in polynomial time. However, when a solution is given, testing the correctness can be done in polynomial time. To find a solution of an NP-Complete problem, 'Brute forcely trying all the candidates' could be the simplest approach. Nevertheless this inevitable difficulty, if someone is trying to find an optimized algorithm for this problem, possible goals of the algorithm would be to reduce the number of candidates to test and to distribute some of its tasks to reduce the running time of the algorithm. The following famous algorithm is an approach to these goals.

2.3. Ullmann's Subgraph Isomorphism

In 1976, J. R. Ullmann [4] introduced the Subgraph-Isomorphism algorithm to find all matching patterns in a given graph with back-tracking search strategy and heuristics to improve the algorithm's efficiency. Even though this algorithm has been researched in many other papers, it is worth explaining all the components and steps of the algorithm here in detail. It is because every single steps of this algorithm will be analyzed and modified in different contexts and assumptions. Although Ullmann named his algorithm Subgraph-Isomorphism, the isomorphism condition, he introduced in his paper, is injective homomorphism. In this thesis, the term isomorphism can also mean injective homomorphism unless otherwise stated. This algorithm assumes that a graph is single and each vertex has unique identifier. It utilizes two types of matrices. One is the adjacency matrix B to represent graph $G_\beta(V_\beta, E_\beta)$ and the adjacency matrix A for pattern $G_\alpha(V_\alpha, E_\alpha)$. The other type is the matrix $M \in Mat_{|V_\alpha| \times |V_\beta|}$ that helps finding all isomorphisms from G_α to G_β.

[3]This condition can be used in Graph Coloring Problem

2.3.1. Search Space

For a subgraph isomorphism problem from G_α to G_β, there are ${}_{|V_\alpha|}C_{|V_\beta|}$ candidates without any prior information. In order to systemically find all candidates, Ullmann's Subgraph-Isomorphism uses a matrix $M^0 \in Mat_{|V_\alpha| \times |V_\beta|}$ as a search space to enumerate all the candidates. Each row of M corresponds to a vertex in V_α and each column corresponds to a vertex in V_β.

The following rule applies to initialize the M^0.

$$m_{ij} = \begin{cases} 1 & \text{if } v_j \text{ is a candidate for } v_i \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

Using Ullmann's backtracking search strategy, the tree search starts from M^0, where each 1 in the matrix is a node, and every 1 in the row immediately below it is a child node. A child node can be a child node of many parent nodes in different paths. Figure 2.1 illustrates an example of trees derived from a M^0, with the matrix expressed with the table form, and 18 candidate isomorphisms obtained from it.

	b1	b2	b3	b4
a1	1	1	0	0
a2	1	1	1	0
a3	0	1	1	1

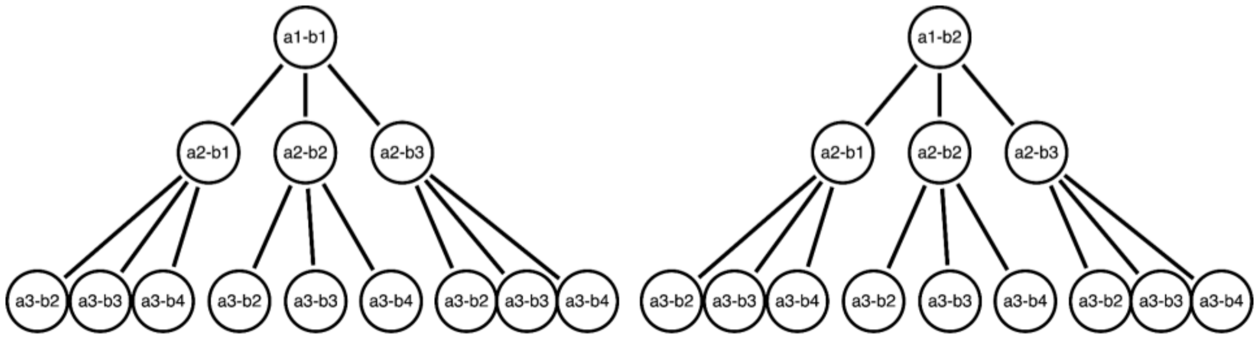


Figure 2.1.: Derived trees from initial search space

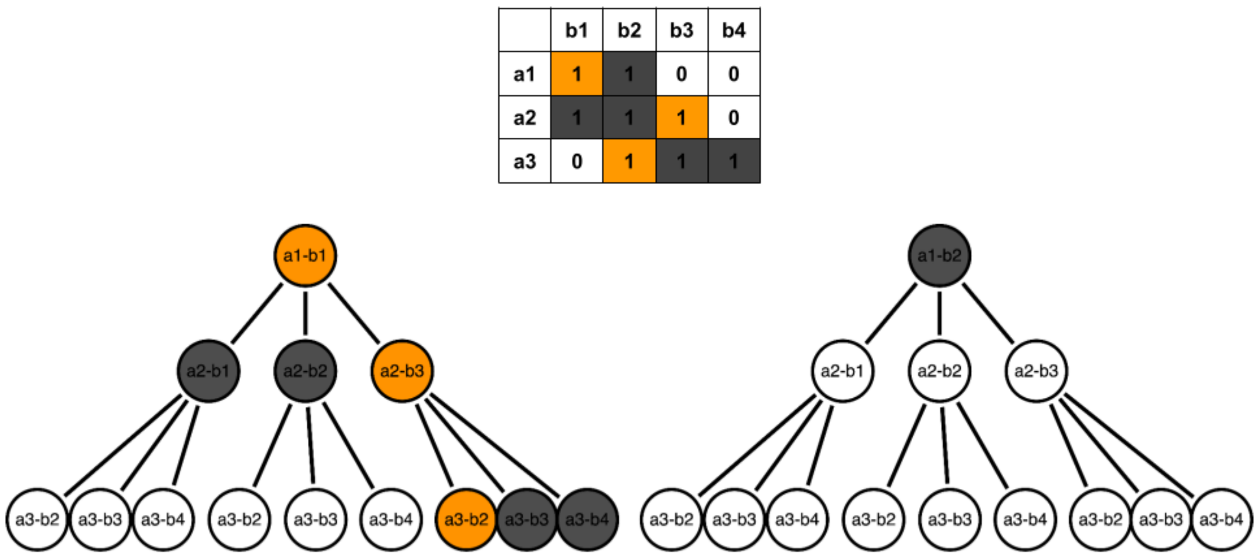
Definition 3 (Incomplete Mapping Matrix M)**.** M is a incomplete mapping matrix(incomplete tree search path).

Some rows in M may contain more than one 1 during the search. By leaving only one 1 in every row, the search will be completed and will output a mapping matrix M'. A complete mapping(path) M' meets the following conditions.

$$\begin{aligned} &\text{(a) Each row has exactly one 1.} \\ &\text{(b) Each column has maximum one 1.} \end{aligned} \tag{2.3}$$

Definition 4 (Complete Mapping Matrix M')**.** M' is a matrix that satisfy the condition 2.3.

M' is a representation of an isomorphism. An element $m_{ij} = 1$ ($i \in \{1, ..., |V_\alpha|\}$, $j \in \{1, ..., |V_\beta|\}$) in a M' is a mapping from $v_i \in V_\alpha$ to $v_j \in V_\beta$. An example of a complete path and M' is demonstrated in the Figure 2.2.



	b1	b2	b3	b4
a1	1	1	0	0
a2	1	1	1	0
a3	0	1	1	1

Figure 2.2.: M' and a complete tree search path

To enumerate all candidates, Ullmann used Depth-First-Search(DFS) with backtracking. The enumeration starts from m_{11}. As the DFS goes one depth forward, it selects(colored in orange) a $m_{ij} = 1$ and automatically excludes(colored in grey) all the other 1s in the i-th row and the 1s in the j-th column in current M respectively. When a path is completed, subgraph isomorphism test in the condition (2.6) will be conducted on the resulting M'. After the test, it backtracks to the most recent split and keeps searching for the next M' until there is no more available path in M^0. With the help of a stack, which stores incomplete paths M, splits can be stored and restored during enumeration. One thing to notice is that this stack is used globally. However tree search with this single global stack would limit the benefit of distributed computation. Section 4.10 introduces other tree search strategy to enable distributed search for a better performance.

A M' can be used to derive a subgraph C from the original graph G_β. The adjacency matrix C of a subgraph of B can be calculated as following [4].

$$C = ((BM')^T M')^T \in Mat_{|V_\alpha| \times |V_\alpha|} \tag{2.4}$$

When a permutation matrix is multiplied once to another matrix, it changes the size of the matrix and switches the order of rows and columns. By combining with the matrix transpose, a permutation

[4]In Ullmann's paper [4], the permutation was introduces as $C = (BM')^T M'$. This holds because Ullmann assumed pattern matching between undirected graphs at the beginning of his paper. It was not necessary to conduct the additional transpose of the result matrix C. However, to conduct the pattern matching on two directed graphs, additional transpose operation is required to preserve the directions of edges.

matrix can be used as a representation of injective function from V_α to V_β. If the condition (2.2) is satisfied, M' can be considered as a permutation matrix(injective function). When this calculation $((BM')^T M')^T$ is applied to the graph adjacency matrix B, it returns a matrix C with the same size of pattern adjacency matrix A. One important characteristic of this permutation is that it not only permutes the order of the vertices(index of C) and reduce the size(selecting a subset of vertices from V_β), but also keeps the adjacency of mapped vertices in the graph(1s in the B).

2.3.2. Initializing Search Space

Ullmann suggested setting as many element to 0 as possible when initializing the M^0 if any prior information is available. With the help of degree of vertices, exclusion of some of the unnecessary candidates from M^0 becomes possible. Each m^0_{ij} can be set by the condition below.

$$m^0_{ij} = \begin{cases} 1 & \text{if } degree(v_i) \leq degree(v_j) \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

In a directed graph, in and out degree can be used instead.

When a vertex of graph has smaller degree than that of a vertex in pattern, it cannot be a valid candidate for this pattern vertex thus can be excluded from the search space. This can be done by setting 0 to the corresponding element in the M^0 before starting the enumeration. The paper not mention how this step can be done, and the effort to do this step was not taken into consideration in the evaluation of the algorithm. This is maybe because the focus of the paper was to provide an algorithm to enumerate and efficiently search all matching subgraphs by applying heuristics during tree search (see section 2.3.4): Refinement. The implement design of this thesis in the section 5.1 includes the steps to construct M^0 in the context of stream environment.

Ullmann suggested to reduce as many 1s as possible to reduce the size of search tree M^0 if there exists any prior information that some v_i cannot correspond to some v_j. Since this thesis discusses about a pattern matching on a property graph, there exists more information than degree information to determine the feasibility for each vertex. In section 4.5, the discussion will cover how to reduce 1s in the context of property match and in the graph stream.

2.3.3. Condition for Subgraph Isomorphism

As test condition for subgraph isomorphism, it compares the pattern adjacency matrix A with C (2.4) which is the permuted subgraph adjacency matrix from B by the permutation matrix M'. Ullmann's condition for subgraph isomorphism is the following.

$$\forall i \forall j (a_{ij} = 1) \rightarrow (c_{ij} = 1) \tag{2.6}$$

It does not test the other direction of the condition. In C, there can be more 1s(edges) than A. Therefore, it is not an isomorphism, but an injective homomorphism. However, according to the Section 2.2, this will be assumed as a condition for subgraph isomorphism.

This permutation (2.4) requires 2 matrix multiplications and 2 transposes. It is worth considering to compute this computation distributively. Elements for an internal calculation could be distributed to each physical or logical node, and then the results will be further aggregated(for multiplication) and put together(for both multiplication and transpose). But the matrices in the equation (2.4) are likely to be a sparse matrix because vertex is connected only to some of other vertices [5]. The discussion in section 4.4 will present an efficient alternative to the matrix permutation by using a different graph representation.

2.3.4. Refinement

The essence of Ullmann's Subgraph-Isomorphism is the Refinement step. This heuristic method enables early termination during the search.

Definition 5 (Refinement)**.** Refinement is an iteration of pruning until there is no more to prune in the context of the current search path.

Definition 6 (Pruning)**.** Pruning changes a $m_{ij} = 1$ to 0 when it turns out that v_i cannot be mapped to v_j.

The scope of pruning is all $m_{ij} = 1$ at the moment of refinement. Refinements will be conducted after every forwarding in DFS. In every forwarding in a DFS, the selected $m_{ij} = 1$ not only map a graph vertex to a pattern vertex, but also exclude(prune) the rest unnecessary mapping candidates in the current intermediate search space M. From this exclusion, the ongoing search path(current search space M) can acquire more information about its feasibility of the isomorphism in the middle of a DFS. Therefore, pruning removes impossible paths and enables early termination of a path during the DFS. The refinement will conduct prunnings until there is no more to prune in M, and the result of the refinement is either an early termination of DFS or a pruned M. With the help of this step, the difficulty from NP-Complete problem could be mitigated.

Definition 7 (Neighbor Condition)**.** For a $m_{ij} = 1$, all adjacencies(direct neighbors) of a pattern vertex v_i and the adjacencies of its mapped graph vertex v_j should be also mapped in the current M.

Adjacencies of a vertex v in graph G refers to the adjacent vertices found in the adjacency list L_G. When checking the existence of mapping of directed graphs, the directions of edges have to be considered.

Definition 8 ($Neighbor_G$ object)**.** $Neighbor_G \in \{(V_{source}, V_{target}) \mid V_{source}, V_{target} \subseteq V_G\}$ is an object with two attributes. One is a set of source vertices and the other is a set of target vertices.

$Neighbor_G$ can represent direct adjacent vertices of a vertex v in G for both edge directions.

Definition 9 (*neighbor* function)**.** A function $neighbor_G : V_G \to V_G \times V_G$ is defined by
$neighbor_G : V_G \to \{(V_{source}, V_{target}) \mid V_{source}, V_{target} \subseteq V_G\}$
$neighbor_G(v_i) := (source_G(v_i),\ target_G(v_i))\ where\ v_i \in V_G,$
$source_G : V_G \to V \subseteq V_G, \quad source_G(v_i) := \{v_j \mid (v_j, v_i) \in L_G\},$
$target_G : V_G \to V \subseteq V_G, \quad target_G(v_i) := \{v_j \mid (v_i, v_j) \in L_G\}.$

The neighbor function finds all adjacencies from the adjacency list L_G and returns a $Neighbor_G$ object of the given vertex v in G. The condition in 7 can be tested with this function. If the condition is not satisfied, the mapping m_{ij} can be pruned in the rest of DFS. Following algorithms show the Refinement and Pruning step.

Algorithmus 1 : Refinement

```
1  Function refinement
       Data : Pattern $A$, Graph $G$, current Mapping $M$
       Result : Pruned $M$ or $NULL$ for early termination
2      /* initialization                                              */
3      $M_{prev} \leftarrow M$
4      $M_{pruned} \leftarrow M$
5      repeat
6          $M_{prev} \leftarrow M_{pruned}$
7          foreach $m_{ij} = 1$ in $M_{pruned}$ do
8              $M_{pruned} \leftarrow$ prune($A, G, M_{pruned}, v_i, v_j$)
9              if a row of $M_{pruned}$ has none of 1 then
10                 return $NULL$
11             end
12         end
13     until $M_{pruned} \neq M_{prev}$/* until there is no more to prune   */;
14     return $M_{pruned}$
15 end
```

Algorithmus 2 : Pruning

```
1  Function prune
       Data : Pattern $A$, Graph $G$, current Mapping $M$, $v_i$ in $A$, $v_j$ in $G$
       Result : Pruned $M$
2      /* test if neighbors have mapping in current M                    */
3      foreach $s_A \in source_A(v_i)$ do
4          if $s_A$ is not mapped to $s_G \in source_G(v_j)$ in $M$ then
5              $m_{ij} \leftarrow 0$
6              break
7          end
8      end
9      if $m_{ij}$ is 1 then
10         foreach $t_A \in target_A(v_i)$ do
11             if $t_A$ is not mapped to $t_G \in target_G(v_j)$ in $M$ then
12                 $m_{ij} \leftarrow 0$
13                 break
14             end
15         end
16     end
17     return $M$ /* M is pruned                                         */
18 end
```

Ullmann suggested to conduct the refinement asynchronously with several computers. This could be done for the line 7 to 12 in algorithm 1 by distributing each $m_{ij} = 1$, but the problem is that Flink does not allow for the distributed streams to access a global status which is M_{pruned} in this case.

However, there is still possibility to benefit from distribution. Because the refinement step is done after every forwarding in the DFS, it accompanies many repetitive calculation for each i and j of $m_{ij} = 1$. By storing the results of the repetitive computations for each $m_{ij} = 1$ in M^0, more efficient refinement step can be achieved. This will be discussed in section 4.9

2.3.5. Alternative Condition for Subgraph Isomorphism

Once a DFS has reached a leaf and the Refinement step changes nothing in this M', then this is a clue that there are only correct 1s in the resulting M'. This also means that the mapping of all vertices satisfies all the adjacencies(neighbors) and it is equivalent to the subgraph isomorphism. Ullmann introduced this as an alternative condition of subgraph isomorphism. By using Refinement in the course of DFS, matrix permutation (2.4) will be no longer necessary, which is originally required when testing the subgraph isomorphism (2.6).

For the property graph pattern matching, however, a decision to the matching has not been accomplished because above condition tests only the topology matching. After the topology matching, property matching can finally determine the property graph matching.

2.4. Property Graph Stream

Definition 10 (Property Graph)**.** A Property Graph is a 7-tuple $(V, E, L, \tau, K, A, \kappa)$.

(V, E) is a graph with a set of vertices V and the set of edges E. L is the set of labels, and $\tau : V \cup E \rightarrow L$ is a function that maps a label to each graph element. K and A denote the set of property keys and the set of property values. The function $\kappa : (V \cup E) \times K \rightarrow A \cup \{\epsilon\}$ maps a value by a key of a graph element. Graph elements may not have a property value. Assignable type of values may vary.

The Figure 2.3 illustrates a property graph stream of messages in a network with four vertices and six edges.

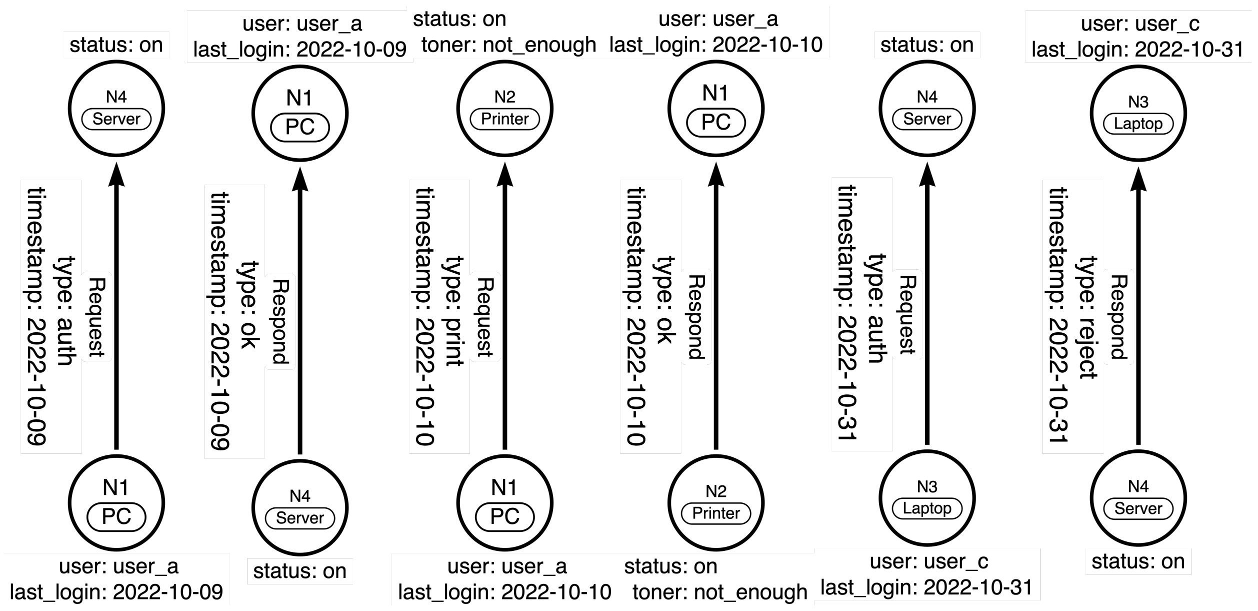


Figure 2.3.: Example of a Property Graph Stream

Below Listing 2.1 is an GDL[5] expression of the first Triple(Source-Edge->Target) from the left in the Figure 2.3.

```
(N1:PC {user: user_a, last_login: 2022-10-09})
    -(:Request {type: auth, timestamp: 2022-10-09})->(N4:Server {status: on})
```

Listing 2.1: GDL expression of a triple

2.5. Windowed Stream Processing with Flink

In a stream environment, elements of the stream flow through the system *unbounded.* For example, elements in a stream can be messages over network, bank transactions, events in a social network service or real-time data from sensors. By contrast, in a static environment, data are *bounded* within

[5]GDL(Graph Definition Language) is a language to express a graph in a plane text. It supports the syntax for property graphs and the MATCH clause for a query.(github.com/s1ck/gdl, accessed February 2023)

a system and an application can have an access without limitation in terms of the completeness of the data in the system at any given time.

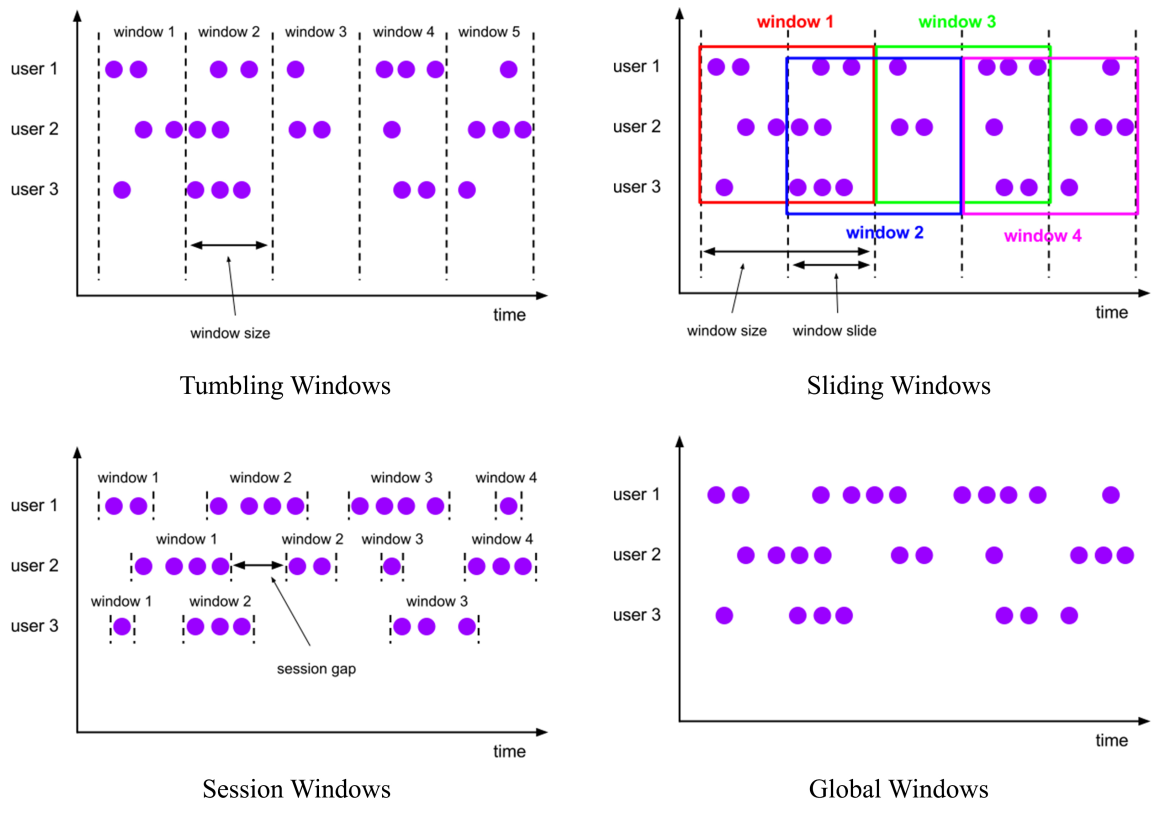
Unlike a static environment, unbounded stream system cannot have an overall access to the elements as a whole. Any application for a stream system has to define in advance how to handle streams and their elements. Flink offers many operators for stream transformation, but in this thesis, the scope is restricted to the four types of transformation as below.

- Type 1: 1 to 1 transformation to single stream.(`Map`)
- Type 2: 1 to n transformation to distributed stream.(`FlatMap`)
- Type 3: n to 1 transformation to single stream.(`Join`)
- Type 4: n to m transformation to distributed stream.(`FlatJoin, CoGroup`)

Type 1 and Type 2 are appropriate for real-time processing and being unbounded is not a problem. Type 3 and Type 4 need definition on how to capture n elements in a stream. To this end, Flink provides `Windows` [6]. A Window is a split of stream elements into groups. Flink provides 4 Windows - Tumbling Window, Sliding Window, Session Window and Global Window(see figure 2.4). Tumbling Window and Sliding Window group the stream elements within a predefined duration(window size). While Tumbling Window splits the time axis without overlap, Sliding Window can split the time axis with overlap. Session Window groups the stream elements between two sessions and a session is defined by the period between inactivities in the stream. Global Window groups the steam elements by user-defined key. All these 4 Windows offer different scopes to capture elements in the stream. Windows can be combined consecutively or joined to apply functions to elements of the streams together for further transformation into a different type of stream.

For Tumbling, Sliding, Session Window, a timestamp assignment rule needs to be defined. Flink offers two notions of time: Event Time and Processing Time. Event time refers to the timestamp of a stream element itself and is already embedded in the data before entering the Flink system. Processing Time, on the other hand, refers to the system time of the Flink application. Event Time is useful when the application has to treat the data based on their occurred time. It enables a simulation of the data in accordance with their original occurrence and Flink processes the data independent from the wall clock. Supposing someone conducts a simulation and extracts some statistics on a 10-year long period of data based on Event Time. The processing may be done within a few seconds or minutes. On the other hand, Processing Time is appropriate for a real-time application. If a graph pattern matching application is running on a real-time data and the size of the window is set to 3 minutes, the application has to capture the data also for 3 minutes in wall clock which is the Processing Time.

Another aspect of stream processing is the keyed and the non-keyed window. By using the keyed window, each group of elements in the stream with the same key will be processed in separate task slots distributively. It is useful when grouping of elements are necessary in a window. The keyed window enables to process a group with the same key distributively without affecting the windows with different keys. For example, a stream of user events can be divided by user's identifier and a pattern matching algorithm can be conducted on to each user. On the other hand, non-keyed



Tumbling Windows

Sliding Windows

Session Windows

Global Windows

Figure 2.4.: Assignable Windows in Flink

window treats all elements in the stream in the same window operation without considering group. To decide whether to use keyed or non-keyed window, following aspects need to be taken into consideration. The keyed window offers a more detailed access to the data, but this also necessitates more computation and memory. If grouping does not offer any better chance to apply an operation, using the non-keyed window can be a simpler selection. Some of the pipelines in the implementation of modified Subgraph-Isomorphism can utilize the keyed window for a better performance. This will be discussed in section 5.1.

2.6. Flink Architecture

Flink is a distributed system that can run on a number of shared-nothing clusters(workers, *TaskManagers*). The TaskManagers are connected to a single *JobManager* that controls all the connected TaskManagers. The JobManager is responsible for the execution of the *dataflow* of the program and scheduling of its tasks. When starting a Flink application, a client loads the program code to *JVM*, and then the dataflow is built. The dataflow will be sent to a single *JobManager*. The client has no more role during the *Job* is running. Some Flink streaming applications may never stop until the client commands the JobManager to stop the running job explicitly. Each TaskManager is a process in a JVM and executes *task(s)* in *task slot(s)*. The predefined number of task slots is the maximum degree of parallelism of a Flink system. Each task slots in a TaskManager will separately use the memory of a TaskManager.

Definition 11 (DAG)**.** DAG(Directed Acyclic Graph) is a directed graph without any cyclic sequence of edges.

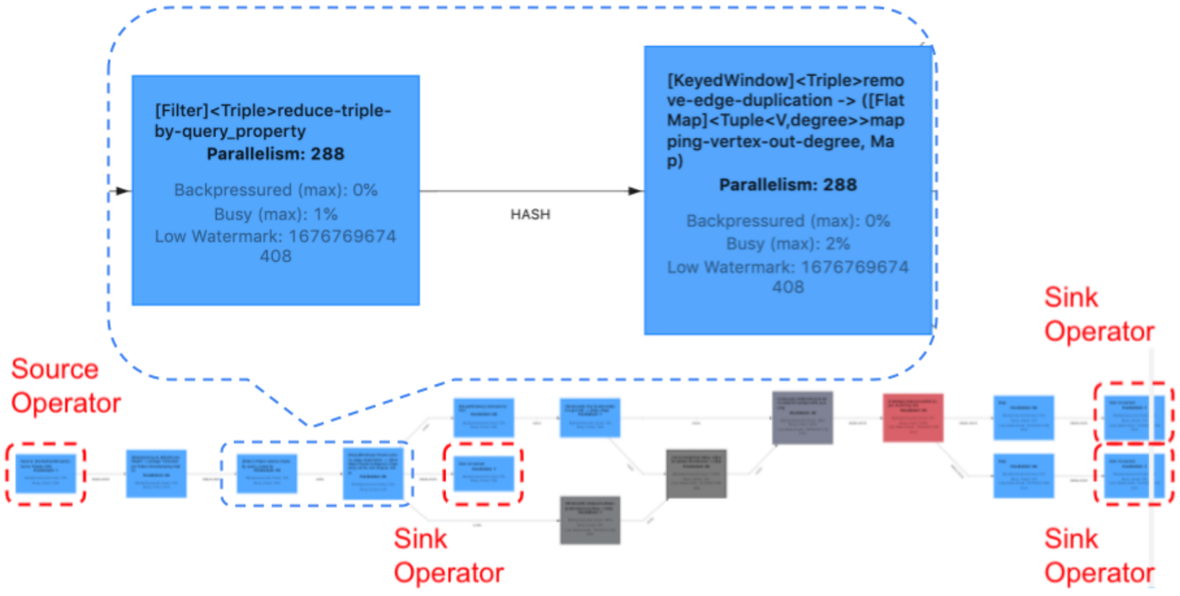

[Filter]<Triple>reduce-triple-by-query_property
Parallelism: 288
Backpressured (max): 0%
Busy (max): 1%
Low Watermark: 1676769674408
HASH
[KeyedWindow]<Triple>remove-edge-duplication -> ([Flat Map]<Tuple<V,degree>>mapping-vertex-out-degree, Map)
Parallelism: 288
Backpressured (max): 0%
Busy (max): 2%
Low Watermark: 1676769674408
Source Operator
Sink Operator
Sink Operator
Sink Operator

Figure 2.5.: Example Dataflow from Flink Web Dashboard

Flink dataflow can be represented by a DAG[6] consisting of **operators**(*Source(s)*, *Transformation operator(s)* and *Sink(s)*) as a set of vertices and **streams** between operators as a set of directed edges. The figure 2.5 is an actual screenshot from Flink Web Dashboard. One Source operator and many transformation operators with three Sink operators are displayed in a DAG.

The result can be stored or directly output to some device depending on requirements of a user. It is also possible to pipeline the result stream to another Flink application as an input stream via TCP Socket.

Flink API offers Table API and SQL on top of DataStream API. DataStream API enables programmers to fluently handle the stream with transformation, join, window, state, etc, and to produce an output also as a stream. For more declarative usage, Table API and SQL provide relational algebra operations such as select, project, join, group-by, aggregate, etc. With the help of Table API and SQL, querying on continuously changing graph stream can be possible [7]. Ullmann's algorithm is, however, less relevant to relational algebra, but more about matrix calculations, search strategy and heuristics. The implementation in this thesis uses only DataStream API and distribution strategy is designed based on the analysis of data transformations and their need for distributed computation.

[6]There exists some exceptions such as Iteration operator in the DataStream API.

[7]To implement a stream graph pattern matching solution, using Table API could be worth considering because it supports relational algebra operations. A solution to graph pattern matching problem with relational algebra with Flink can be found in the research [2] and it is implemented in Gradoop. Gradoop has a graph pattern matching implementation using relational algebra and enables querying on the static distributed dataset with Cypher. The implementation uses DataSet API.

2.7. Distribution in Flink

Type 2 and Type 4 transformation in DataStream API can transform a stream into distributed streams. Although the streams are distributed, API allows programmers to access the streams as if they enter a transformation operator as one logical stream. The distributed execution of the transformation operation task is automatically coordinated across the JobManager and TaskManagers. Task slots keep the JobManager updated on their recent *Watermark*, which reflects the status of their operation in terms of their recently processed *Timestamp*. If one task slot is taking long to complete its operation, the other task slots in the same stream will stop proceeding until they receive a progressed watermark.

Distributed computation can solve problems with a large volume of data by partitioning them into smaller problems. However, in a streaming system, the system must provide effective load-balancing, so that one overloaded task slot does not delay the entire application [7]. Flink offers several partitioning strategies such as rescale, rebalance and broadcast to ensure fair use of task slots.

One of the critical considerations in this thesis is determining what should be distributed. While high parallelism could reduce computation time, it can also result in a larger amount of data transfer to TaskManagers and task slots. This thesis evaluates this trade-off by comparing two scenarios: one involves a large amount of object distribution to achieve less computation in the successor operators, and the other will not distribute objects to minimize the communication costs.

An important note about the distribution is that the join transformation cannot be applied to an already distributed stream. If two heterogeneous objects needs to be processed together in a transformation operator, they must first be joined and then distributed together by being packed in a single object.

3. Related Work

Ullmann's Subgraph-Isomorphism [4] focuses on the enumeration of all M' and refinement during searching for isomorphisms. How to construct A, B, M^0 with a computer was not mentioned in the paper. Discussions in this thesis include these tasks in terms of stream graph and distributivity. In a stream system, these constructions will be conducted repeatedly, thus it is worth considering this cost into the pattern matching algorithm together. In addition to that, the design and implementation will also include property matching. Ullmann suggested asynchronous computation on some of the steps in the algorithm. Discussions in this thesis include what and how his suggestion can be achieved.

Stonebraker et al. [7] presented eight requirements of real-time stream processing. The paper compared the requirements of stream processing with relational DBMS and online analytical processing. It highlighted the importance of low-latency processing, the ability to handle high throughput of data, scalability to handle increase of data volume, fault tolerance, querying on stream, etc. Some of the designs and evaluations in this thesis are inspired from these requirements.

Alkamel, A. [3] used Dual Simulation algorithm in Flink as a graph pattern matching algorithm for property graph stream. In that thesis, several possible matching models were introduced as well as different matching steps that could be applied in different stream processing steps. Furthermore, a prototypical implementation of the pattern matching system, SGraPMa, was introduced with an evaluation method and its result. Basic ideas in the paper on property matching and implementation prototype have inspired the implementation of this thesis.

Other Subgraph Isomorphism algorithms such as VF2 [8], QuickSI [9], GraphQL [10], GADDI [11], and SPath [12] were introduced and compared in the study by Lee, J. et al. [13]. These algorithms improve the performance of pattern matching by using different join orders, pruning rules and neighborhood information. In the study [13], the algorithms from originally different programming environments were implemented in the same environment and tested with the same real-world datasets(undirected graphs) in a static graph system. And then it was followed by an in-depth comparisons of subgraph isomorphism algorithms.

The usage of query language was previously available in graph databases but not well supported in the distributed graph data processing system. In the study by Junghanns et al. [2] in 2017, the declarative graph query language Cypher was implemented to enable non-programmers to use it more comprehensibly in the distributed graph analysis platform Gradoop. It included pattern matching in a large graph data with the query language. It also demonstrated the scalability of the operations in the queries in Gradoop. At the time this paper was written, Flink was selected as distributed data processing system, but not as stream processor. Gradoop used DataSet API of Flink, which has been soft-deprecated since November 2020(Flink 1.12) and replaced by DataStream API, which is now used with backwards compatibility.

Ullmann's enumeration strategy was the DFS and the possibility to distribute this enumeration was not introduced. Makki et al. [14] presented an efficient distributed DFS algorithm. Nodes and edges are distributed in a communication network. Nodes can communicate with other nodes by sending messages over network connection. Messages about visited and unvisited nodes are sent and they enable more efficient tree search. Like other DFS algorithms, it also uses stack but with more functionality - inspecting entire stack and removing some elements directly. The performance is proven to be better than other DFS algorithms. However this distributed DFS is not appropriate to be adopted in Flink. The distributed DFS is designed to use communications between distributed nodes in a network. However Flink does not provide such communications between distributed nodes for a stream. Therefore another distributable search strategies are analyzed in this thesis.

4. Problem Analysis

This chapter analyzes the subgraph isomorphism problem in the context of property graph streams in distributed environment Flink. The analysis begins with discussion on the complexity of the problem, followed by discussions focusing on the challenges of Ullmann's Subgraph-Isomorphism on graph stream. This problem analysis and discussions on potential solutions set the foundation of the implementation and evaluation of the proposed approaches to the goal of this thesis: Windowed Graph Stream Pattern Matching using Subgraph-Isomorphism.

4.1. NP-Complete

Subgraph isomorphism as a decision problem is known to be an NP-Complete [1], meaning that there is no known algorithm to find a solution within polynomial time. Ullmann's Subgraph-Isomorphism can be categorized as a backtracking based algorithm for the subgraph enumeration problem, which is computationally more expensive than the decision problem. The heuristics in the algorithm are not designed to produce an efficient algorithm, but rather to mitigate the difficulty of the problem by reducing the size of search space M^0 and enabling early termination via the Refinement step. The goal of this thesis is to prototypically implement the algorithm in the streaming system Flink and to add further efforts to mitigate the difficulty of the problem by distributing some of its processing steps. Presenting an efficient subgraph isomorphism algorithm is not the goal of this thesis.

4.2. No static database

In a static graph system, the set of vertices and the set of edges can be stored in separate tables within a relational database system. Every edge requires two reference keys to map its source and target object in the vertex table. To solve a graph problem in a static graph system, relational algebra operations such as Join and Select are used to find vertices of edges. In a stream graph system, by contrast, accessing static database should be avoided if possible. One of the essential requirements of stream processing is that the data should keep flowing through the system without having a bottleneck. If stream elements are edges and there is a static database with vertex and edge tables with keys to join them, the edges in stream will require at least two Join operations. For analytical purpose, this is not acceptable because streaming system demands low-latency processing. However continuous storage and polling in relational database system can lead to higher latency and cause problems such as bottlenecks in operations. For this reason, static database systems are not well-suited for stream systems [7].

An edge in static system can be transformed into a triple of $(s, e, t) \in V \times E \times V$ that can equivalently represent an event in the graph stream. Source and target objects contain all of their description even if they can appear repeatedly in the stream. Unlike static system, redundant occurrences are not considered as a problem in stream systems. If it is necessary to capture aggregated data, the operations in the graph stream system should store data in memory within a window's operator.

From this context, this thesis assumes that there is no prior information or the use of a database. A set of triples is considered incomplete until the defined time window is closed. To conduct computations such as building matrices A, B, and M^0, or applying matrix permutation (2.4), all necessary information needs to be stored without the help of a database. For example, to construct an adjacency matrix, it requires the indices of vertices and information about their connections. Although there is no database, the index are stored similar to a database system, for example, by hashing with an available key.

4.3. Challenges from Using Traditional Graph Representations

One problem of constructing adjacency matrix in the graph stream is that it first requires complete set of vertices, edges and their index. To obtain a complete set of vertices and edges in the stream, it is necessary to wait until the end of each time window. Constructing the adjacency matrix incrementally is also possible. However, it is inappropriate to build an adjacency matrix for an incomplete, evolving graph. Adding a new row and column to the adjacency matrix at every appearance of new vertex requires expanding the size of the matrix each time. Additionally, 2D-array is not a good solution because it requires fixed size upon initialization. Another problem of adjacency matrix is that it stores $|V|^2$ information about edges for a graph $G(V, E)$. For a sparse matrix, it would be waste of memory because it also stores 0s for unconnected vertices. A vertex in a graphs is likely to be connected to only some of other vertices, therefore its adjacency matrix would be a sparse matrix [5]. For this reason, using adjacency list with `List` object is more preferable to represent a stream graph.

While adjacency list offers a better opportunity to represent a stream graph than 2D-Array, there is also a disadvantage. Subgraph isomorphism problem requires to meet the mapping of both vertex and adjacency. To verify mapping of a number of adjacencies, it would demand high computational cost to iterate on the graph several times. To select an edge or a vertex in an adjacency list, an iteration on the list is unavoidable. If such iterative operations must occur frequently, another alternative can be chosen: Hashing. With the help of `HashMap`, it is possible to store edges with indexing the key with `SourceAndTargetId`, which is `Tuple2<SourceId, TargetId>`. This enables lower cost of selecting an edge in the graph. The implementation of the thesis represents stream graphs with `HashMap<SourceAndTargetId,Triple>` and it is named `AdjacencyMap`. `AdjacencyMap` also contains additional index to allow accessing the graph elements by the order and id of vertex. Briefly, the `AdjacencyMap` is an expansion of `HashMap<SourceAndTargetId,Triple>` with additional indices based on vertex order and vertex id respectively.

Definition 12 (AdjacencyMap)**.** AdjacencyMap is a representation of a stream graph that includes three types of indices to facilitate easier access to graph elements: `the tuple of (Source Id, Target Id), Vertex Id, and Vertex Order`

4.4. Subgraph from AdjacencyMap: an Alternative to Matrix Permutation

AdjacencyMap needs to support the permutation operation in formula 2.4 to derive a subgraph from the stream graph. An advantage of using AdjacencyMap is that deriving subgraph with M' accompanies less computation than adjacency matrix. This is because, for a $m'_{ij} = 1$, the j is the index of vertex of the graph and i is the order of the selected vertex in the isomorphism. With the help of index in the `AdjacencyMap`, deriving subgraph from the stream graph requires, in general, constant complexity of $O(1)$, which is more efficient than $O(N^3)$ of matrix multiplication[8]. By avoiding matrix multiplications, higher efficiency could be achieved in terms of the time complexity.

4.5. Reducing Search Space using Property

The inherent complexity of the subgraph isomorphism increases significantly as the number of graph elements in a window grows. Therefore, reducing the number of candidates in the search space plays an essential role in solving this problem. Properties and predicates can contribute to this effort in the first two steps among the three steps below. This lists the usage of properties and the predicates within the algorithm.

- Step 1. Filtering out irrelevant triples from input stream
- Step 2. Initializing the search space M^0
- Step 3. Property matching on an isomorphic subgraph

The step 1 filters irrelevant triples out from the input stream of triples. To achieve this, the *tripleFilter* (function 3) is applied on the input stream. The *soloMatch* in the function refers to the *Solo check* in the paper [3], which tests the properties and also self-predicates of a single graph element against that of an element in the query. Predicates could be tested on the subgraph after isomorphism test(Step 3). However, by testing the self-predicates in this step 1, reducing more triples becomes possible in earlier stage of the algorithm. The step 1 contributes to reduce the number of columns in the M^0 by filtering only the relevant vertices from the property graph stream. Only the relevant vertices will construct the columns of M^0 as a result of step 1.[9]

Out of this filtered triple stream, an additional stream will be produced to compute the degree of each vertex. Each triple will be transformed(Type 2 in 2.5) into the stream of `(source vertex id, 1)` and `(target vertex id, 0)` tuples, and they will be distributed to keyed window stream by each vertex key(identifier). The sum of the 1s in each of keyed window indicates the out degree of each vertex in the windowed graph stream. Although this computation was not considered in Ullmann's algorithm, the vertex degree can be efficiently computed with the help of Flink's distributed keyed window stream.

[8]There are many algorithms for matrix multiplication including distributed algorithms that have lower complexity than $O(N^3)$, but they are still higher than $O(1)$

[9]As explained in the Limitation (see section 5.3), the column index will not be ordered.

After this, the step 2 determines the value of each row in the columns out of step 1. This step follows the suggestion of Ullmann by reducing more 1s in the M^0 with prior information, which is the properties and self-predicates besides the vertex degree. Three conditions(degree, properties and self-predicates) of each query vertex v_i(a row of M^0) test each stream vertex v_j(a column of M^0), and assign $m^0_{ij} = 1$ only when the v_j passes all the conditions of v_i.

By applying Step 1 and Step 2, the search space M^0 becomes smaller than the M^0 without properties and self-predicates.

Algorithmus 3 : Triple Filter function

```
Function tripleFilter
    /* test if the triple $t_G$ has a matching triple in $A$ */
    Data : A Triple $t_G$ in Graph $G$, Pattern $A$
    Result : Boolean
    foreach $t_A \in A$ do
        if $soloMatch(t_G.source,\ t_A.source)$ then
            if $soloMatch(t_G.target,\ t_A.target)$ then
                if $soloMatch(t_G.edge,\ t_A.edge)$ then
                    return True /* $t_G$ has a matching triple in $A$ */
                end
            end
        end
    end
    return False /* $t_G$ has no matching triple in $A$ */
end
```

4.6. Multi Graph into Single Graph

In a property graph, two edges are considered distinct if they have the same source and target vertices but different labels or properties. This distinction results in a multi-graph by definition, where multiple edges can exist between the same pair of vertices. For a GDL query below, for example, there will be four matching patterns in the figure 4.1.

```
MATCH (:City)-[type:"regional"]->(:City)-[:Train]->(:City)
```

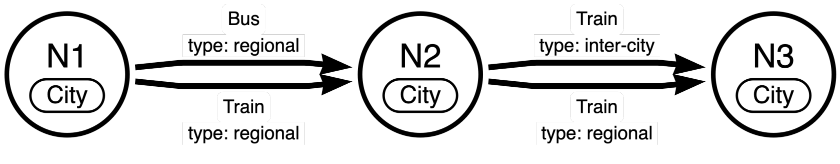


Figure 4.1.: Example of a multi property graph

In this thesis, the prototypical implementation and evaluation are restricted to single graphs. Because of this restriction, it is necessary to ensure that all edges of the triples in a time window have to be unique for each pair of source and target vertices. To solve the problem of removing edge duplication, Keyed Window can be used. To split the stream, key has been assigned by `(s.id,` `t.id)` to each triple `(s,e,t)`. The number of splits is maximum $|V_\beta|^2$, which is the number of elements in adjacency matrix B. After applying Window to this Keyed Stream, the triples in a Keyed Window have the same key `(s.id, t.id)`, but there can be a number of different edges in terms of their property. To remove edge duplication, following assumptions were considered.

1. Choose the first one in the window.
2. Choose the most recent one in the window.
3. Choose one with matching property.

In this thesis, the first alternative was chosen for simplicity. Because Flink offers access to `Iterable` object and getting the first one can be done without iterating objects in the window. On the other hand, choosing the last one in the `Iterable` requires one full iteration. Computing property match between graph edges and a pattern edge may provide the best result in terms of finding isomorphism, but it is hard to define the match because it is not know yet which pattern edge should be compared to the edge in the window.

4.7. Configuring Multiple Window Operations

This thesis limits its scope to Tumbling and Sliding Windows because the evaluation assumes that the input stream flows with a predefined throughput per second. Session Window, which requires gaps in the stream to split it, is not suitable. Global Windows is flexible to use, but it offers no benefit to graph pattern match because there is no prior information about a key to group stream elements. The available information to group the stream elements is only the timestamp. Therefore, the thesis focuses on Tumbling or Sliding Window to compose a graph using the timestamp in the stream. Besides the selection of Window, it is required to define how to assign timestamp for each stream element.

For the dataset, which is used for the evaluation, the embedded event timestamp could have been used. However, to enable more realistic evaluation, additional assumption and modification were necessary. If the timestamps in the record are used, the number of triple among time windows will fluctuate unpredictably, which will make the evaluation less meaningful. For this reason, it has been first decided to use the processing time. But this did not fully solve the problem because the implementation uses several consecutive windowing operations [15] to construct the pipelines of the stream Subgraph-Isomorphism. One thing to note is that the elements out of a window will get the maximum timestamp of the window.

Sliding Windows requires more care when a Flink application contains consecutive window operations. If multiple Sliding Windows are used, duplicated elements will be observed by downstream Sliding Windows. In the figure 1.3, the Sliding Window is configured with 10 seconds window length and 5 seconds for the slide. The timestamp of all elements out of the Sliding Window `[12:02:05PM, 12:02:15PM)` will be `12:02:14.999PM`. If a successor window operator is configured also with the length of 10 seconds, the window will also contain **all** elements out of the overlapping previous Sliding Window. This is because the timestamp `12:02:14.999PM` falls into the window `[12:02:10PM, 12:02:20PM)`. Therefore, all elements in `[12:02:10PM, 12:02:14.999PM]` will appear duplicated in the successor window operator for the period of `[12:02:10PM, 12:02:20PM)`. To avoid this, if a Flink application uses Sliding Window and multiple window operations, the successor windows must use Tumbling Window as below.

- Place the Sliding Window in the front of the dataflow.
- Tumbling Window's size must be the sliding size of the Sliding Window.

With this configuration, the timestamps of elements will be always the maximum timestamp of the **first** Sliding Window and they will not change in the course of the dataflow. The intended operations will occur on every **length of slide** without any duplicated stream elements in the successor windows.

4.8. Tracking Flink Operations

One advantage of stream system is the real-time computation. However, the graph pattern matching requires windowed operations. With increasing amount of elements in windows, the system may not process streams with low latency and predictable results. A bottleneck in a window operation delays not only the successor operations, but also the predecessor operations such as source operator, especially if the bottleneck is not making a progress on its recent watermark. In that case, the source operator cannot accept any new input until there are available task slots. There are some solutions for this problem by automatically scaling up the degree of distribution and increasing the system capacity. But this thesis excludes discussions about this.

It is obvious that window operations such as Type 3 and Type 4 in 2.5 require more capacity. One challenge is that it is difficult to know how much capacity it would require. The impact of the amount of elements in the stream needs to be analyzed in terms of the system capacity and the throughput of input.

To analyze this, the evaluation will focus on the time consumed by operations in the implementation. To measure this, a `Tracker` is designed and attached to stream objects. `Tracker` will keep the timestamps of some of the operations throughout the system. Observing the `Tracker` records as in the table 4.1 will enable to identify which operations in the implementation are causing issues.

Operation	End Time	Duration
WindowAll:graphAdjacencyMap	2023-01-09T22:25:30.731Z	0
WindowAll:get-M0.begin	2023-01-09T22:25:30.820Z	89
WindowAll:get-M0.end	2023-01-09T22:25:30.821Z	1
Join,FlatMap:M0distribution	2023-01-09T22:25:30.967Z	146
Map:neighboursMapSubset	2023-01-09T22:25:31.079Z	112
CoGroup:merge-NeighboursMap	2023-01-09T22:25:32.618Z	1539
CoGroup:BFS-queuing.begin	2023-01-09T22:25:32.770Z	152
FlatMap:DFS.begin	2023-01-09T22:25:33.096Z	326
apply-permutation	2023-01-09T22:25:33.102Z	6
property-match	2023-01-09T22:25:33.103Z	1

Table 4.1.: Records in a Tracker in a Window

4.9. Distributed Neighbor Computation

During searching in the M^0, a number of repetitive computations occur. The largest overhead emerges from Refinement step, which iteratively tests the existence of direct neighbors mapping in M. Within a path, the line 7 of the algorithm 1 causes problem because the Refinement is conducted at every forwarding in the path. More over, the entire search will visit the same node($m_{ij} = 1$) in the search tree many times from different paths(see figure 2.1). To avoid this repetition, `NeighborsMap` can be computed only once for each window.

Definition 13 (NeighborsMap)**.** *NeighborsMap* object is a HashMap of *Neighbors* object by hashing $(i,\ j)$ as the key, where $m^0_{ij} = 1$.

Definition 14 (Neighbors)**.** *Neighbors* is an object with 4 Lists: query targets, query sources, graph targets, graph sources depending on following arguments: a query vertex, a target vertex in the context of Query and Graph.

A naive approach to compute the NeighborsMap is to iterate on every $m^0_{ij} = 1$. However, because the computations on each (i, j) are independent from each other, a distributed approach can be considered. By evenly partitioning the set of $m^0_{ij} = 1$ in M^0, the subsets of the NeighborsMap can be distributively computed. After the completion of distributed computation, all subsets of the NeighborsMap in the window can be merged(joined) because the resulting subsets have no intersecting elements with the same (i, j). This step is not mentioned in Ullmann's paper, but it is worth implementing and verifying the new approach. Whether the distribution of M^0 for the calculation of NeighborsMap improves the performance or not will be discovered in the evaluation part.

4.10. Distributed Tree Search

The NP-Completeness arises from searching for all isomorphisms in the M^0. If possible, this step should be conducted distributively to mitigate the inherent difficulty. Ullmann's method of enumeration cannot be distributed because of the backtracking with the global stack. This requires a distributed search strategy as demonstrated in the Figure 4.2.

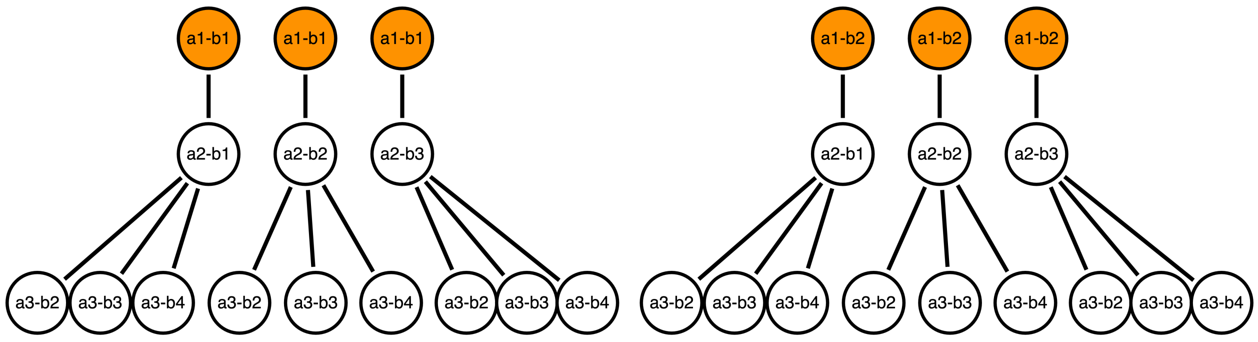


Figure 4.2.: SubTrees derived from Figure 2.1

This search strategy is a mix of Breadth-First Search(BFS) and Depth-First Search. To partition the SubTrees with the similar amount of search spaces, BFS strategy is used first: all the 1s in the row with lowest i in the M^0 will be first selected and split until the desired degree of distribution is achieved. Each of the partitions is the `SubTree` and it can be now considered as an independent search space. The set of `SubTrees` and the original tree are equivalent in terms of the result of the subgraph isomorphism, therefore they can be evenly distributed to each task slots for further tree search with the same Refinement and NeighborsMap as before.

The next consideration is how to remove the global backtracking and this can be achieved with the Recursive Depth-First Search. By copying the SubTrees for each recursions on the same level, the global stack can be replace. But, this does not reduce the calculation or increase the performance because the copying the SubTrees are fundamentally as same as storing the splits in the stack.

Although the search space is evenly partitioned, some of the SubTrees may finish its search earlier with the help of the Refinement. In fact, the partitioned SubTrees do not contain similar amount of isomorphisms. However, knowing it in advance is not possible due to the nature as the NP-Complete problem.

5. Distributed Subgraph Isomorphism on Property Graph Stream

This chapter integrates problem analysis and discussions from the previous chapter to design a solution for the problem of property graph subgraph isomorphism in the stream environment Flink. The chapter also includes a list of assumptions and limitations. In addition, to benefit from distributed system following basis of strategy has been established to address the challenges from the problem analysis.

- Rule 1. Most time consuming operation should be distributed at the last stage of the algorithm.
- Rule 2. Select distribution when there is a trade-off between distribution and volume of data to distribute.
- Rule 3. Reduce Join transformation as much as possible.

As a result, following strategy has been set. For the Rule 1, the searching in M^0 will be distributively conducted at the end of the pipeline because this process is where the NP-Completeness originates. To enable distributed search, M^0 will be partitioned fairly to each task slots as illustrated in the Figure 2.1. As a result of the Rule 2, AdjacencyMap itself will be distributed to all task slots by attaching it to partitioned M^0 together with pre-computed Refinement object. This will require large volume of data communication, but at the same time, it will enhance the searching performance. In order to comply with the Rule 3, the AdjacencyMap is joined to distributed stream of Sub-Trees before computation of distributed NeighborsMap.

5.1. Design

This section illustrates the design of *Windowed Graph Stream Pattern Matching using Subgraph-Isomorphism* based on the problem analysis and discussions in the section 4.

In contrast to the *dataflow* in the figure 2.5, this DAG in figure 5.1 represents streams as vertices and transformation operations as edges. When joined streams map to a stream of Tuple with the original or mapped data type, they are represented in the form of `<XXX, YYY>`. It is important to note that the query graph from the GDL Query is not reflected in this DAG, as the query object is not a stream element. However, it can be easily accessed by any operation by providing it as an argument to the operator. The dataflow in Figure 2.5 together with the DAG in Figure 5.1 illustrates how Subgraph-Isomorphism on a property graph stream is designed with Flink DataStream API.

The source operator(Type 1) accepts a stream of triples as input from outside the Flink application. This stream is then filtered by the next operator(Type 1) to remove irrelevant data. The filtered stream is then transformed by multiple operators. One operator(Type 2) computes the degree of each vertex in a keyed window stream in a distributed manner and constructs matrix M^0 using

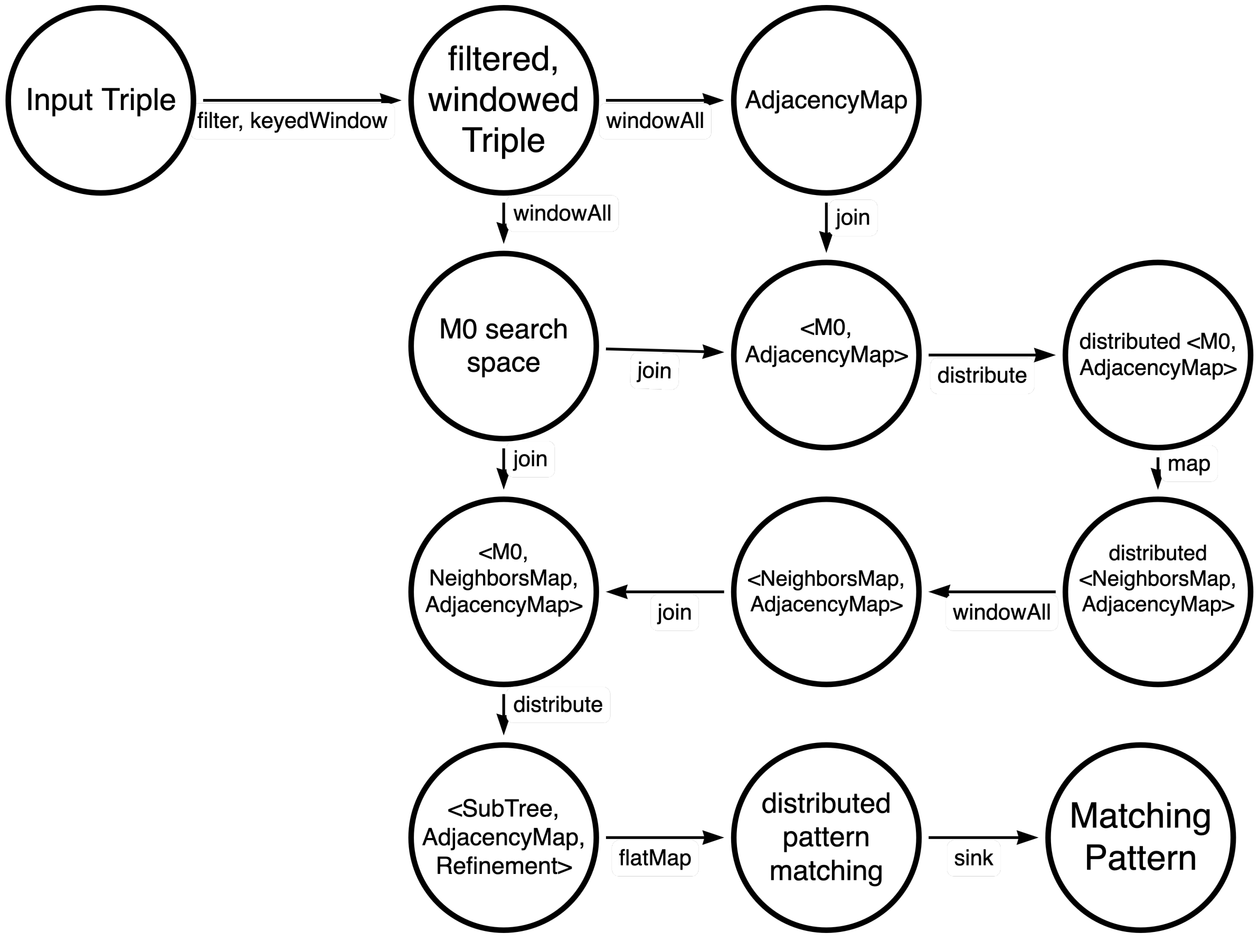

Input Triple
filter, keyedWindow
filtered, windowed Triple
windowAll
AdjacencyMap
windowAll
join
M0 search space
join
<M0, AdjacencyMap>
distribute
distributed <M0, AdjacencyMap>
join
map
<M0, NeighborsMap, AdjacencyMap>
join
<NeighborsMap, AdjacencyMap>
windowAll
distributed <NeighborsMap, AdjacencyMap>
distribute
<SubTree, AdjacencyMap, Refinement>
flatMap
distributed pattern matching
sink
Matching Pattern

Figure 5.1.: Design of Windowed Stream Subgraph-Isomorphism

windowAll, while the other operator(Type 3) constructs an AdjacencyMap with all graph elements in a pre-filtered, non-keyed window without distribution.

Before the streams from the previous operators are joined, the NeighborsMap is distributively(Type 2) constructed(Type 1) and grouped in a window(Type 3). The M^0 stream is then joined(Type 3) with the AdjacencyMap and NeighborsMap before being distributed(Type 2) by partitioning the M^0 into disjoint Sub-Trees (see section 4.10).

The distributed Subgraph-Isomorphism begins with the operator(Type 1), which performs DFS Refinement independently in each task slot following Ullmann's algorithm. After this, the correct results obtained from Ullmann's algorithm are processed again with the property matching, and the resulting stream is produced at the sink operators(Type 3).

5.2. Assumption

The design of the *Windowed Graph Stream Pattern Matching using Subgraph-Isomorphism* assumes the followings.

1. Database is not available.

2. Elements of input stream are triples of $(source, edge, target)$ with unique identifier in each element of a triple.

3. Edges are directed.

4. Elements have label and properties.

5. Pattern matching will be applied on a single graph. Among duplicated edges, only the first edge will be considered.

6. The condition for the subgraph isomorphism requires injectivity, not bijectivity.

5.3. Limitation

The results of the algorithm are indistinguishable in terms of the set of matching subgraph isomorphisms regardless of the order of search in the search space. However, Ullmann suggested to sort the index of vertex in adjacency matrix in the descending order of degree to enhance the refinement step. But the computational effort it requires to sort the vertex index was not taken into account in the paper [4]. The implementation and evaluation of this thesis also do not consider this. As sorting the vertex index was identified as a bottleneck for the design, it was excluded from both the implementation and evaluation stages of this thesis.

6. Prototypical Implementation

This chapter presents the prototypical implementation of the *Windowed Graph Stream Pattern Matching using Subgraph-Isomorphism* including the illustration of the workflow for the evaluation. In addition, a custom *Stream Server* is implemented specifically to help the prototypical implementation with following reasons:

1. To restrict the scope of the thesis merely to Flink.
2. To ensure the deterministic order of input data from the dataset for the evaluation.
3. To configure the throughput of the input stream for the evaluation.

6.1. Technologies

Used technologies are listed in table 6.1 with their usages.

Technology	Version	Usage in the thesis
Java	jdk1.8	Development and Runtime Environment for Java Application
Flink	1.14	Distributed stream processing environment for the Property Graph Stream Pattern Matching
Apache Maven	3.8.3	Build automation for the Flink application
GDL	0.3	Graph and Query definition
TCP/IP	-	Network communication protocol between the Stream Server and Flink
Netcat	0.7.1	Configuration of the Stream Server

Table 6.1.: Used technologies

6.2. Workflow of the Prototypical Implementation

The prototypical implementation of the graph pattern matching system is structured into three levels: client, socket, and Flink application, as illustrated in Figure 6.1. The overall workflow consists of the following steps:

1. The Stream Server runs on the client and writes graph stream to a socket.
2. With the given arguments in the Table 6.2, the Flink application connects to the socket and reads from the socket as the source of stream.
3. The Flink application applies the graph pattern matching algorithm to the stream, and generates a stream of matching results.
4. The outputs will be sent to the client over sockets.

The Benchmark program controls the execution of the Flink application in various scenarios by configuring the throughput of the Stream Server and assigning different sets of arguments for the Flink application as listed in Table 6.2. For the purpose of the evaluation, the prototypical implementation outputs three streams in total: Result of pattern matching to the `Socket1`, Timestamp records in the `Tracker` to the `Socket2`, Filtered triples in a window to the `Socket3`.

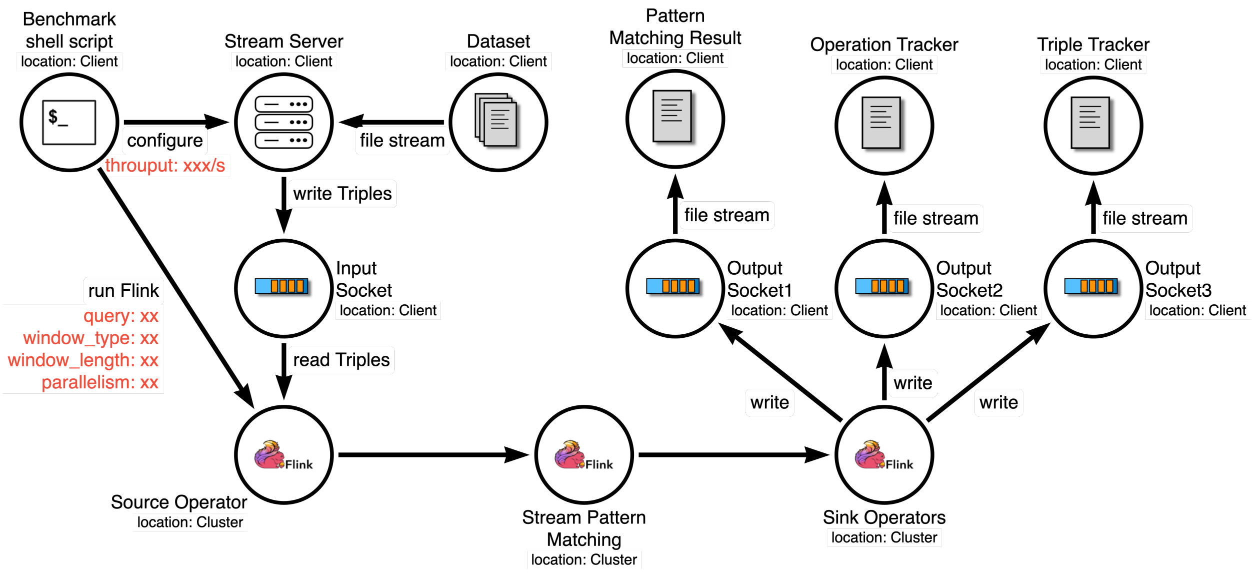


Figure 6.1.: The workflow of the prototypical implementation

6.3. Input and Output

To read the text stream input from the client's local Stream Server through clients socket, the implementation utilizes the `socketTextStream` method described in the Listing 6.1 in the StreamExecutionEnvironment provided by Flink[10].

```
public DataStreamSource<String> socketTextStream(String hostname, int port)
```

Listing 6.1: socketTextStream method

Below Listing 6.2 is the code snippet of the input and the `socketTextStream` method. The Stream Server runs on the client's local JVM and its throughput configuration will be informed by the benchmark program through the port 9998. The Stream Server will write a stream of triples as text with the given throughput configuration.

```
//Benchmark shell script
//configure the Stream Server with throughput of 1000/s for example
echo 1000 | nc localhost 9998

//Stream Sever side
acceptThroughputConfig(); //accept the throughput configuration from port 9998
printWriter.println(tripleAsTextLine); //write triples as text stream to socket

```

[10]socketTextStream method supports only TCP socket. WebSocket is, however, not supported.

Sung Geun Yun
3735467

```
//Flink side
DataStreamSource<String> inputStream
    = socketTextStream(hostName, 9999); //read triple stream from socket
DataStream<Triple> tripleStream
    = inputStream.map(new TripleParser()); //parse triple text to a Triple object
```

Listing 6.2: Stream input and socketTextStream method

The Listing in 6.3 demonstrates an example output of the pattern matching result from the prototypical implementation. Each `null` in the adjacency matrix indicates that there exists an edge between the matching vertices in the windowed graph stream, however, the edge does not have a corresponding edge in the query graph.

```
[2023-01-09T22:25:20Z - 2023-01-09T22:25:29.999Z]
Adjacency-Matrix(4x4) with 12 elements
         v3   v1   v2   v4
  v3 [   e3   e7 null   e4 ]
  v1 [   e6 null   e1    . ]
  v2 [   e2    . null    . ]
  v4 [ null   e5    . null ]
//ID and property of each element will be printed additionally.
```

Listing 6.3: Example of Pattern Matching Output

All the outputs are finally processed in the `Sink` transformation operators. The prototypical implementation will sink its output to the sockets with `writeToSocket` method in the Listing 6.4.

```
public DataStreamSink<T> writeToSocket(String hostName, int port
                                       , SerializationSchema<T> schema)
```

Listing 6.4: writeToSocket method

The client can access the output stream as described in the code snippet in the Listing 6.5.

```
//Flink side
matchResultStream.writeToSocket(hostName, 9991, new SimpleStringSchema());

//Client side. Client will append the stream to a local file.
//In terminal
nc -lk 9991 >> /result/result_file1
```

Listing 6.5: Stream output and writeToSocket method

6.4. Program Setup

The table 6.2 lists the arguments available for the implemented program. For Sliding Windows, the half of the window length is set as the length of slide, and the successor windows are automatically set to Tumbling Windows accordingly (see section 4.7). The IngestionTime for timestamp argument is an alternative for the processing time window. If processing time window is used, the group of elements of one window can have different timestamp than the elements in the successor windows. And their timestamp will change in the course of going through several windowing operations. To avoid this, the evaluation uses time windows not with the embedded timestamp but by assigning the ingestion time of each triple. By assigning them, the elements out of the first window and the successor windows will have the same timestamp. This consideration is made only for the purpose of evaluation.

The window_runtime serves to restrict the maximum processing time for each window with the help of the Tracker. When a Tracker identifies that it exceeded the maximum runtime in a window, which is set to five minutes by default, the Tracker stops the Flink Job. This function is designed for more efficient evaluation because it enables to identify the bottleneck during the runtime and to dismiss the evaluation earlier.

Arguments	Options	Purpose
jar	name of jar file	to select different implementations
query_file	name query file	to select different query
parallelism	a number	to set the parallelism and evaluate its impact
window_type	Tumbling and Sliding	to configure window type
window_size	length in seconds	to evaluate impact of window size
throughput	input throughput	to evaluate impact of throughput
input_port	port number	to specify input port
output_port1	port number	to specify output port for pattern matching result
output_port2	port number	to specify output port for identifying bottleneck
output_port3	port number	to specify output port for observing the triples in the stream
server_config_port	port number	to update the server configuration during the evaluation
timestamp	EventTime or IngestionTime	to configure timestamp
time_unit	milliseconds, seconds, minutes, hours, days	to configure the time unit
window_runtime	milliseconds	to configure maximum running time for a window(default 5 minutes)
number_of_window	the number of window	to configure the number of windows to evaluate(default 30 windows)

Table 6.2.: Arguments for the program

The code snippet in Listing 6.6 is an example running command with the arguments in the table 6.2.

```
#In terminal
/opt/flink-1.14.5/bin/flink run -p 576 -c edu.leipzig.streaming.graphs.
    patternmatching.subgraphisomorphism.app.BikePatternMatchApp /local/d1/users/
    userxxx/jar/pattern_match/StreamGraphPatternMatch-1.0.jar -queryFile /local/d1/
    users/userxxx/query/query1 -testKey test01 -host localhost -inputPort 9999 -
    outputPort1 9991 -outputPort2 9992 -outputPort3 9993 -timestamp IngestionTime -
```

```
        timeUnit seconds -parallelism 576 -windowType TumblingWindow -windowSize 30 -
        maxRuntimePerWindowMilliSec 300000
```

Listing 6.6: Flink running command with program arguments

7. Evaluation

This section dedicates to the evaluation of the *Windowed Graph Stream Pattern Matching using Subgraph-Isomorphism*. The first object is to verify if the distributed Ullmann's Subgraph-Isomorphism can process the graph stream pattern matching with a low latency as emphasized in the paper [7]. To achieve this, the prototypical implementation measures the *runtime* of the Flink Job for each *matching task* in various scenarios, and the resulting runtime to complete the 30 cycles of windows will be compared against their *allowed latencies*.

$$\begin{aligned} Allowed\ latency &= Total\ duration\ of\ a\ matching\ task \\ &= length\ of\ the\ window \times 30 \end{aligned}$$

The second object is to determine whether increasing the degree of parallelism can improve the processing of large graphs that could not be effectively processed with a lower degree of parallelism. Confirming the necessity of the distribution of M^0 (see section 4.9) is the next object. The last object is to identify which operations are the most time-consuming within the dataflow.

To achieve the objects above, the benchmark program sequentially runs each *matching task* scenario with the combination of the parameters in the table 7.1. To confirm the expected advantage from the distribution of M^0, a specific version of Flink application has been implemented, which performs the pattern matching without the distribution of M^0. The timestamp records in the `Tracker`(see Table 4.8) can be analyzed to identify the time-consuming operations.

Parameters	values
parallelism	36, 72, 144, 288, 576 (task slots)
window_type	TumblingWindow, SlidingWindow
window_size	10, 20, 30 (seconds)
throughput	300, 500, 1000, 2000, 3000 (per second)

Table 7.1.: Parameters for Evaluation

7.1. Evaluation Environment

The evaluation is conducted on a Flink cluster consisting of 1 JobManager and 16 TaskManagers. Flink version 1.14.5 executes the program on 16 workers, connected by 1 Gbit/s Ethernet without sharing their resources. Each JobManager has 376 GB RAM, CPU of Intel(R) Xeon(R) Gold 6240 @ 2.60GHz with 72 Cores, and 11TB SSD.

7.2. Dataset

Citibike dataset[11] is frequently used for data analysis, as it offers a large public bike-sharing usage in New York City with valuable information that can be interpreted as a graph. The dataset

[11]https://citibikenyc.com/system-data, accessed February 2023

contains trip data between bike stations with the information about the trip such as the station IDs, locations, start and end times, user types, etc.

This thesis uses the `SF10 Citibike dataset`[12], containing 10% of sampling data from 06/2013 to 05/2020. The dataset has about 20 million records of trips and the records are stored in each edge CSV files and vertices CSV files respectively.

To represent the Citibike dataset as a set of triples in the form of `(Source Station, Trip, Target Station)`, a preprocessing step was carried out specifically for this evaluation. The preprocessing integrated the two types of CSV files into the files of triples retaining all the original information in the vertices and edges. This enables the Stream Server to easily read the file and emit triples without further processes. Moreover, this enhances the comparability of the scenarios by guaranteeing the same order of triples in the input stream in each scenario.

7.3. Query

The query used for the evaluation is presented in a GDL in Listing 7.1 and visualized in Figure 7.1. This query seeks for the pattern of rides with the specified location boundary and a time sequence for some of the rides.

```
MATCH (v1:Station{})-[e1:Trip ]->(v2:Station {})-[e2:Trip ]->(v3:Station{})-[e3]->(v
    3)-[e4]->(v4)-[e5]->(v1), (v1)-[e6]->(v3), (v3)-[e7]->(v1)
WHERE e1.stopTime < e2.startTime AND e2.stopTime < e3.startTime AND e3.stopTime < e4.
    startTime AND e4.stopTime < e5.startTime
    AND v1.latitude < 40.725273d AND v2.latitude < 40.725273d AND v3.latitude <
        40.725273d AND v4.latitude < 40.725273d
    AND v1.longitude < -73.976019d AND v2.longitude < -73.976019d AND v3.longitude <
        -73.976019d AND v4.longitude < -73.976019d
```

Listing 7.1: GDL Query for the Evaluation

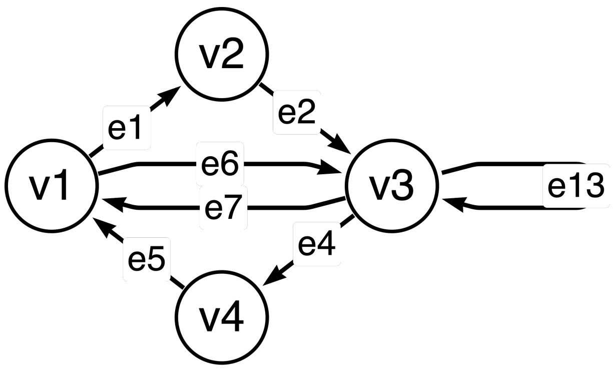


Figure 7.1.: Illustration of the Query

[12]The dataset is imported by Gradoop and transformed into Gradoop's IndexedCSV format.

7.4. Results

Based on the preprocessed dataset and the query, test scenarios have been carried out with the following results. A missing result of a scenario occurs when the `Tracker` stops the stream job due to a bottleneck of a windowed operation longer than 5 minutes. Therefore, a missing result can be considered as failure of keeping the low latency.

Verifying the Low Latency Processing

The Figure 7.2 verifies that keeping the allowed latency was possible with a lower throughput ranging from 300 up to 500 per seconds. However, it is noticeable that the scenarios with high parallelism result in higher latency. This implies that some of the operations related to distributed computation takes larger amount of time as the degree of parallelism increases.

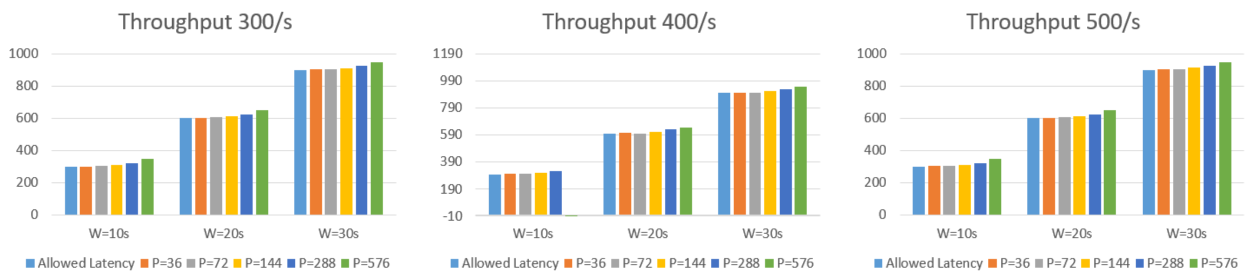


Figure 7.2.: Scenarios with low throughput of input

Effect of Increasing the Parallelism

The Figure 7.3 shows the results of scenarios with higher throughput. Increasing the degree of parallelism also does not improve the processing of larger graphs.

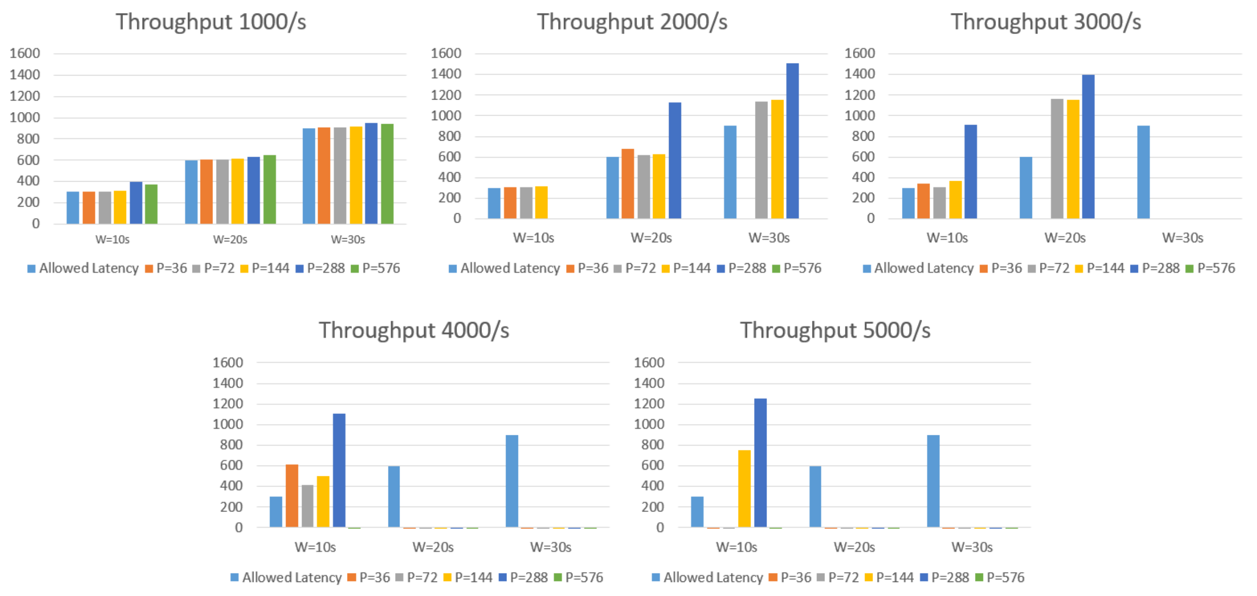


Figure 7.3.: Scenarios with high throughput of input

Necessity of the Distribution of M^0

These scenarios show less missing results than the previous scenarios in 7.3. Along with the previous observation, the scenarios in the Figure 7.4 confirm that it is not necessary to compute the $NeighborsMap$ with distributing M^0. Despite this confirmation, low-latency processing is still not possible in large graphs with high throughput even without distributing M^0 for the computation of $NeighborsMap$.

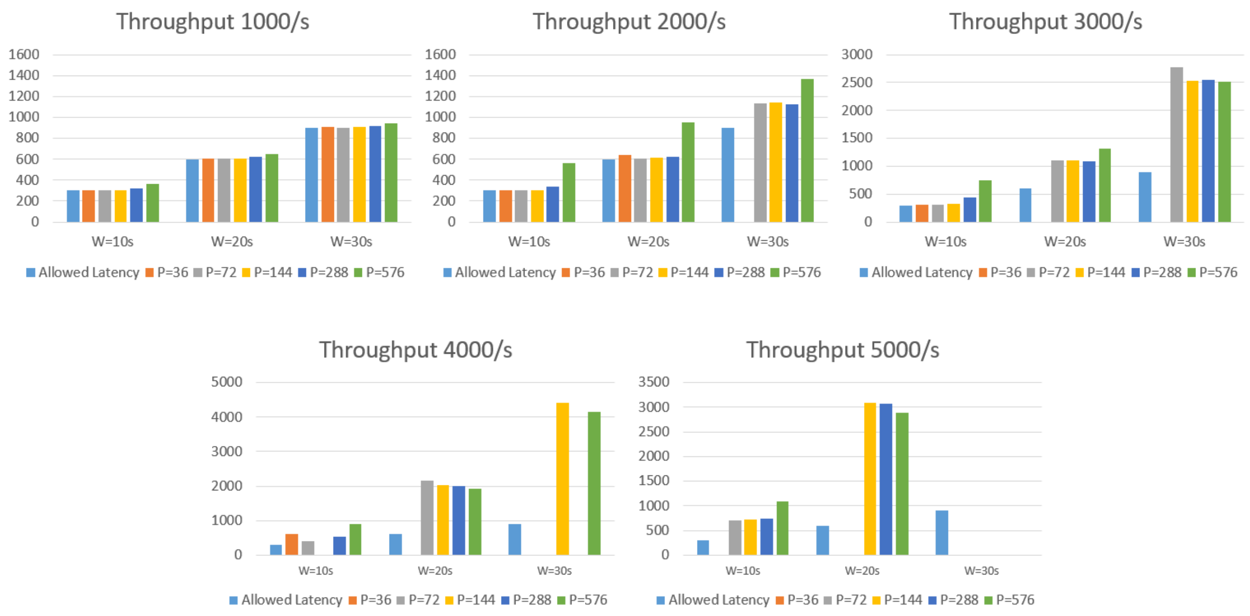


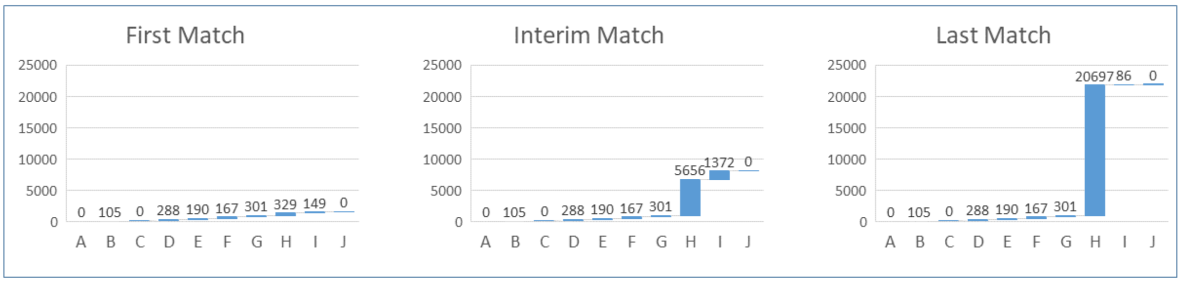
Figure 7.4.: Scenarios without distribution of M^0

Identifying the Time-Consuming Operation

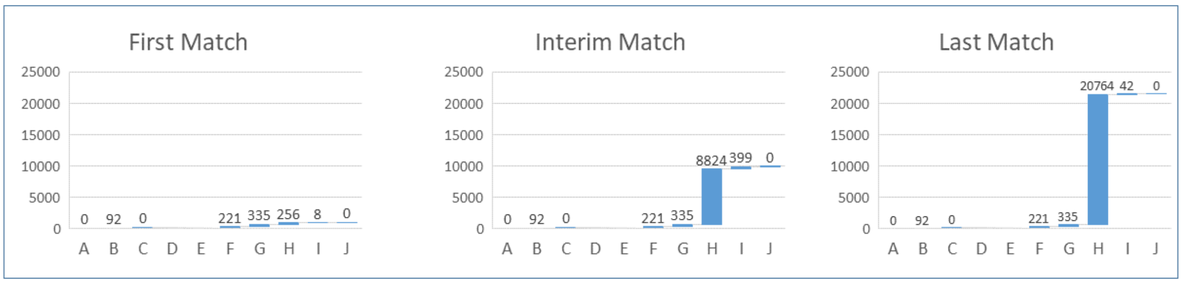
The waterfall charts in Figure 7.5 are from the scenario with the window length of 30 seconds, parallelism of 576 and throughput of 1000 triples per seconds. Each chart illustrates the cumulative progress of windowed operations until outputting the pattern matching result. The alphabets on x-axis correspond to the entries in the Table 4.1 of the Tracker example in timely order. By analyzing the charts, the most time-consuming operations have been identified: the entry H(the distributed search step). While the first matching results are computed very quickly, the last matching results takes significantly longer amount of time until finishing the pattern matching. The last matching candidate simply waits in the window for longer than 20 seconds. And there are no considerable difference between the version with the distribution of M^0 and the other version without the distribution of M^0. This means that the problems identified from the previous figures in the 7.2, 7.3 and 7.4 are from the distributed search step.

Limitation of Evaluation

Although the prototypical implementation was evaluated using a variety of window sizes, degrees of parallelism, and input throughputs, it is important to note that the evaluation scenarios on



(a) With M0-Distribution



(b) Without M0-Distribution

Figure 7.5.: Waterfall chart of the records in Trackers

the Sliding Windows did not provide comprehensible results. As a result, the evaluation results presented in this chapter are only from the Tumbling Windows.

8. Conclusion

In this thesis, Ullmann's Subgraph-Isomorphism has been implemented in the distributed streaming system Flink by modifying the algorithm to be able to perform the property graph stream pattern matching. Ullmann's algorithm has been extensively analyzed in this thesis and various new aspects have been considered to enhance its performance. Notably, adopting the AdjacencyMap as a graph representation has facilitated better indexing and reduced repetitive iterations. The distributed computation of the NeighborsMap and the distributed tree search strategies without backtracking leveraged the distributed system Flink.

By integrating all the considerations into the pipelines of the Flink operations, *Windowed Graph Stream Pattern Matching using Subgraph-Isomorphism* has been designed. To evaluate the design, a prototypical implementation has been developed and tested with a set of stream pattern matching scenarios with the combination of various sizes of the windows, the degrees of parallelism and the throughput of the input stream.

The evaluation results demonstrate that the modified Ullmann's Subgraph-Isomorphism algorithm can perform low-latency stream pattern matching in small graphs at a throughput of up to 1000 events per second and with a maximum degree of parallelism of 144. However, the implementation's performance in producing results with low latency is limited when the graph stream has a large number of elements in a window. Increasing degree of parallelism presents its limitation on reducing the latency. High parallelism can cause problem even with small graphs.

As a future work, it might be considerable to employ the vertex-ordering step during the initialization of the search space as suggested in Ullmann's paper [4], which was not implemented in this prototypical implementation. This could enhance the search performance by enabling earlier termination of tree search. Additionally, testing and examining the implementation on Sliding Windows could provide further insights, widening the scope of its application.

Bibliography

[1] Stephen A Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing.* 1971, pp. 151–158.

[2] Martin Junghanns et al. "Cypher-based graph pattern matching in Gradoop". In: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems.* 2017, pp. 1–8.

[3] Abdalrahman Alkamel. "„Distributed Pattern Matching on Graph Streams "". In: *Magisterarb. Leipzig University* (2020).

[4] Julian R Ullmann. "An algorithm for subgraph isomorphism". In: *Journal of the ACM (JACM)* 23.1 (1976), pp. 31–42.

[5] Robert A Beezer. "Review of: Graph Theory by JA Bondy and USR Murty". In: (2008).

[6] The Apache Flink Project. *Stream Window Operators.* 2023. URL: `https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/windows/` (visited on 02/15/2023).

[7] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. "The 8 requirements of real-time stream processing". In: *ACM Sigmod Record* 34.4 (2005), pp. 42–47.

[8] Luigi P Cordella et al. "A (sub) graph isomorphism algorithm for matching large graphs". In: *IEEE transactions on pattern analysis and machine intelligence* 26.10 (2004), pp. 1367–1372.

[9] Haichuan Shang et al. "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 364–375.

[10] Huahai He and Ambuj K Singh. "Query language and access methods for graph databases". In: *Managing and mining graph data* (2010), pp. 125–160.

[11] Shijie Zhang, Shirong Li, and Jiong Yang. "GADDI: distance index based subgraph matching in biological networks". In: *Proceedings of the 12th international conference on extending database technology: advances in database technology.* 2009, pp. 192–203.

[12] Peixiang Zhao and Jiawei Han. "On graph query optimization in large networks". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 340–351.

[13] Jinsoo Lee et al. "An in-depth comparison of subgraph isomorphism algorithms in graph databases". In: *Proceedings of the VLDB Endowment* 6.2 (2012), pp. 133–144.

[14] SAM Makki and George Havas. "An Efficient Method for Constructing a Distributed Depth-First Search Tree." In: *PDPTA.* 1997, pp. 660–666.

[15] The Apache Software Foundation. *Apache Flink documentation: Windowing Operators.* Accessed: February 15, 2023. 2021. URL: `nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/windows/#consecutive-windowed-operations`.

Statement Of Authorship

Ich versichere, dass ich die vorliegende Arbeit mit dem Thema:

„Windowed Graph Stream Pattern Matching using Subgraph-Isomorphism“

selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann. Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.

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

„Windowed Graph Stream Pattern Matching using Subgraph-Isomorphism“

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, den 20.02.2023

Sung Geun Yun