IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 26 May 2024, accepted 15 June 2024, date of publication 20 June 2024, date of current version 27 June 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3417352

RESEARCH ARTICLE

Optimising Queries for Pattern Detection Over Large Scale Temporally Evolving Graphs

HASSAN NAZEER CHAUDHRY AND MATTEO ROSSI
Department of Electronics, Information and Bioengineering, Politecnico di Milano, 20133 Milan, Italy
Corresponding author: Hassan Nazeer Chaudhry (hassannazeer.chaudhry@polimi.it)

ABSTRACT Large-scale graph processing and Stream processing are two distinct computational paradigms for big data processing. Graph processing deals with computation on graphs of billions of vertices and edges. However, large-scale graph processing frameworks mostly work on graphs that do not change over time, while on the other end of the spectrum, stream processing operates on a continuous stream of data in real-time. Modern-day graphs change very rapidly over time, and finding patterns in temporally evolving graphs could reveal a lot of insights that can not be unveiled using traditional graph computations. We have proposed a novel framework called FlowGraph which could find patterns in dynamic and temporally evolving graphs. Computations on large-scale graphs are iterative and take multiple steps before final results can be calculated, which is very different from stream processing which is one-shot computation. Therefore, the most critical bottleneck of such a system is the time required to process the query. In this work, we have proposed a query optimization technique that could reduce the time required to process the pattern. The proposed system has an optimization technique that could reduce the time required to process the pattern, especially those related to the temporal evolution of the graph. Our method shows for eight clauses the execution time is reduced by 75%, we also proved that this improvement is not affected by the scaling of the graph or the change of elements in given clauses.

INDEX TERMS Graph data structures, distributed computations, vertex-centric computations, temporal pattern recognition, query optimization, and dynamic graph computations.

I. INTRODUCTION

Large-scale graphs are complex data structures that consist of a set of vertices and edges that connect these vertices. These graphs represent complex systems or relationships between entities; some common use cases of large-scale graphs include social network analysis [1], transportation networks [2], [3], biological and genetics networks [4], [5], internet and web searches, recommendation systems [6], and cybersecurity. In social network analysis, large-scale graphs are commonly used to represent and analyze social networks, such as Facebook and Twitter, to understand social structures and relationships [7], [8]. Another use case is transportation networks where graphs are used to represent road networks and airline routes. Given graphs that represent the network, different algorithms are then used to optimize routes, reduce congestion, and improve efficiency [9], [10], [11]. In biology and genetics, Large-scale graphs are used to represent complex biological systems, such as protein-protein interaction networks and gene regulatory networks, to better understand biological processes and diseases [4], [12], [13]. One of the most used large-scale graph networks is the internet and web search. Graphs are used to represent the structure of the internet and web pages which is helpful in algorithms to improve search results. Similarly, graphs are employed in recommendation systems to represent user preferences and item similarities, such as those used by Amazon and Netflix [6], [14], [15].

The term ''large scale'' refers to graphs that are so large and complex that traditional methods for processing and analyzing graphs are no longer effective. These graphs can contain millions or even billions of vertices and edges, making them too large to be stored in memory on a single computer [16]. As a result, large-scale graphs require specialized techniques and tools for processing, analyzing, and visualizing the data. These techniques include distributed

The associate editor coordinating the review of this manuscript and approving it for publication was Giacomo Fiumara.

computing, parallel processing, graph partitioning, and compression algorithms [17]. In the last two decades, specialized software frameworks have been designed that can handle and process large-scale graphs efficiently. Several surveys [16], [17], [18], [19] are published in the field of large-scale graphs processing, which cover most of the state of the art in the realm of iterative computation of algorithms on large scale graphs. Pregel is one of the earliest efforts in 2010 in this direction, its central idea is based on the Think Like A Vertex (TLAV) paradigm [16]. In TLAV several distributed nodes also known as workers coordinate with the centralized master. The graph is divided into workers while going through several iterations known as super steps to perform computations on respective partitions of graphs. TLAV employs a bulk synchronous parallel programming model which is a message-passing and synchronous paradigm for computation. There are also graph processing systems that are asynchronous and use shared memory models instead of message passing [20], [21], [22]. The survey [16] provides a comprehensive review of different TLAV frameworks. The TLAV approach mostly works on static graphs, or on graphs that do not change very often. In practice, large-scale graphs change very often and their structure evolves. Some efforts to perform computations over large-scale temporally evolving graphs are Chronos [23], Immortalgraph [24], Tornado [25], Chronograph [26] and Graphbolt [27].

Large-scale graph processing and stream processing are two distinct paradigms that are used for different types of data processing tasks. Graph processing is used to analyze and manipulate large-scale graphs with billions of vertices and edges. In contrast, stream processing is used to process a continuous stream of data [28], [29]. However, there are some use cases where it may be necessary to combine both paradigms to achieve certain types of data processing tasks. **Firstly**, for real-time graph updates, it may be necessary to update the graph as new data arrive. For example, in a social media platform, it may be necessary to update the social graph as new users join or new relationships are formed. In this case, a stream processing system can be used to capture the incoming events and update the graph. **Secondly**, graph analytics can be applied to real-time data; in some applications, it may be necessary to analyze the graph as new data arrives. For example, in a financial fraud detection system, it may be necessary to detect suspicious transactions by analyzing the transaction graph. In this case, a graph processing system can be used to analyze the graph as new transactions arrive. **Thirdly**, graph queries may be necessary to query the graph in real-time as new data arrives. For example, in a logistics tracking system, it may be necessary to query the transportation network graph to find the optimal route for a package. In this case, a graph processing system can be used to store and query the graph data in real time. To mix large-scale graph processing and stream processing, it is necessary to have a system that can handle both types of data processing paradigms. Some popular open-source systems that can handle both paradigms include Apache Flink, Apache Beam, and Apache Spark. These systems provide a unified programming model for both batch and stream processing and can be used to process both graph and non-graph data. In our earlier research work, a framework called FlowGraph is proposed [30], [31]. FlowGraph is a distributed system that can process temporally evolving large-scale graphs as large-scale graph frameworks and detect patterns analogous to stream processing. TLAV-based large-scale graph processing frameworks are iterative and require several iterations before graph computation can be done; on the contrary, pattern detection works on streams of data and requires near to real-time flow of data. Both goals are distantly different from each other and require a reduction of time in iterative computation of graph processing. There are several aspects of the reduction in processing time including graph partitioning optimization [32], load balancing [33], graph compression [34], [35], caching and better locality optimization, speeding up graph traversal having better communication algorithms, hardware acceleration [36] and query optimization. This research work is based on efficient query structuring and optimising overall query processing time. In literature, query processing has been firstly achieved using heuristic-based techniques [37], [38], [39]. This involves reordering graph nodes [37], [39] and predicate pushdown [37]. The drawback of this group of techniques is that they need a very specific structure of the query. Secondly, it is done using operator fusion or a combination of multiple operators [40], [41], [42], [43]. However, this group of techniques require operators in which partial results can be stored between multiple operators. The third type of technique known as the interpretation-based query model, [42], [44], [45], translates the query into a data flow graph (DFG) and then tries to optimize DFGs. These techniques are slow in performance and need time to translate queries into DFGs [42], [43]. This work lies in the first category where the structure of the query is manipulated to reduce the operation time. The pattern of FlowGraph consists of N clauses where each clause ends up with the predicate evaluating to True or False [30], [31]. In FlowGraph some clauses take more computational time called heavier as compared to others called lighter. If lighter clauses evaluate to false, the entire pattern evaluates to false even though other clauses evaluate to True. The premise of the optimization technique lies in the fact that if lighter clauses are evaluated before heavier and checked for if they evaluate to False a computation the heavier clauses can be skipped. For all possible cases in which lighter are shifted before heavier computation time can be reduced by 75%. To summarize our key contributions are:

- The pattern detection capabilities of the FlowGraph framework are enriched by adding a construct, called *FollowedBy*, which allows users to relate properties of the graph at different points in time.
- A data structure to incrementally store the changes in the graph over time has been proposed, it also allows us to perform graph computations at different time points.

- A novel optimization technique is proposed for temporal patterns, that can reduce the total computation time by 75%.
- It is proved that the proposed optimization technique is independent of graph size, pattern structure and total number of clauses in the pattern.

Section II presents existing literature related to optimization; Section III introduces the architecture and language constructs of FlowGraph. Section IV describes the optimization technique. Section V, presents the experimental setup and discusses the results. Finally, Section VI concludes.

II. RELATED WORK

Optimization refers to a group of techniques which enhance the efficiency or performance of algorithms, data structure or software pipelines in general. In the context of this paper, it refers to Query Optimization(QO). QO is defined as an algorithm that changes a query in a way that the time required to process a query is reduced. One of the earliest use cases of QO is in databases, discussed in subsections II-A. The literature on databases is included since most of the optimization algorithms draw inspiration from DB optimization. The literature review of big data regimes is included such as stream processing and semantic graphs. Since they are the most related areas to the work and graph processing, query optimization in stream processing has been discussed in II-B and semantic graphs in II-C. Finally, large-scale graph processing optimization has been discussed II-D.

A. OPTIMIZATION IN DATABASES

Database (DB) is one of the earliest and most studied use cases for query optimization [46], [47], [48]. Optimization in DB can be broadly classified into indexing methods, cost-based techniques, join optimization, predicate push-down, and parallel execution. **Indexing methods** are one of the most widely applied query optimization techniques in DB [49], [50], [51]. Although the creation of indexes on columns is frequently referenced in queries for filtering or sorting purposes, DB can promptly pinpoint and fetch the required rows [52]. This approach reduces the need for exhaustive scans of entire tables, consequently speeding up query performance. **Cost-based optimization** entails assessing the cost associated with various query execution plans and opting for the plan with the lowest cost [53], [54], [55]. This approach takes into account factors such as disk I/O, CPU utilization [56], and memory usage to identify the most effective strategy for executing a query. Cost-based optimization draws upon statistics concerning the distribution of data within tables to make informed decisions, aiming to enhance query performance [57], [58]. **Join operations** merge rows from two or more tables based on a common column are bottleneck on the performance of database systems [59], [60]. Join optimization methods, including join reordering, selection of join types (such as nested loop joins, hash joins, or merge joins), and join elimination, are instrumental in streamlining join operations and reducing query execution duration [61], [62], [63], [64], [65], [66]. Simplifying join operations in databases can significantly improve the performance of DB systems [67]. **Predicate pushdown** refers to the practice of shifting filter conditions (predicates) near the data source [68], thereby diminishing the volume of data necessitating processing in subsequent stages of the query execution plan. By implementing filter conditions early in the query execution sequence, databases can curtail the volume of data transmitted and processed, ultimately enhancing query performance [69]. Several techniques in literature employ predicate pushdown methods for optimization [69], [70], [71], [72], [73]. **Parallel execution** methods encompass the division of a query into smaller tasks capable of simultaneous execution across multiple CPU cores or nodes within a distributed setting [74], [75]. These approaches harness the collective computational prowess of numerous resources concurrently [76], resulting in reduced query processing time, especially for CPU-intensive workloads. Parallel execution proves particularly advantageous for queries handling expansive datasets and intricate computations [77], [78], [79].

B. OPTIMIZATION IN STREAM PROCESSING

Achieving data parallelism in batch processing is very simple, in the case of stream processing data arrives in a single sequence, and extracting parallelism in Stream-processing systems (SPE) is challenging and requires a careful design. SPE optimisation techniques can be broadly divided into four groups. Firstly, **heuristic-based techniques** [37], [38], [39] form a graph from a query and perform graph transformation, for example, reordering of graph nodes, substituting certain graph nodes with others [37], [39]. In the **predicate pushdown** [37] technique, the filtering operation is shifted closer to data so that the remaining operations in the query are reduced. The major drawback of heuristic-based techniques is that they require a very specific shape of query such as filtering defined over events of an input stream. So, a push-down query can not be applied to filtering following a join operation. Secondly, in **queries based on operator fusion** [40], [41], the fusion techniques store immediate results between two operators in a register or cache hence reducing the unnecessary movement of data. The fusion technique can only work if a soft pipeline breaker (SPB) can be achieved [42], [43]. SPB is a technique that stores any operator's partial results before the next operator can be processed. Thirdly, SPE uses an **interpretation-based query model** [42], [44], [45] which translates queries into a data flow graph (DFG). However, despite being well-used optimisation regimes such architectures are slower in performance, the inefficiency is attributed to the conversion of a query into DFG and data transfer dependencies in DFG [42], [43].

C. OPTIMIZATION IN RESOURCE DESCRIPTION FRAMEWORK (RDF) AND SPARQL

Resource Description Framework (RDF) and SPARQL are the backbones of the Semantic Web. RDF is the standard

model for representing and exchanging data on the web and storing information in graph-based format. However, the graphs used in RDF are quite different from those used in our work. RDF mostly employ subject-predicate-object triples rather than standard vertex edge notation. SPARQL is a query language employed to query and manipulate RDF data. It provides a standardized way to retrieve, update, and manipulate RDF graphs stored in RDF databases or triple stores. **Query rewriting** involves transforming SPARQL queries into semantically equivalent forms that are more efficient to execute [80]. This process can include restructuring the query to leverage indexes, eliminating redundant operations, or reordering operations to minimize computational overhead [80], [81], [82], [83], [84]. For example, replacing certain complex graph patterns with simpler patterns or breaking down complex queries into smaller, more manageable subqueries can improve performance. The authors [85] returned top k queries sorted by the user-defined scoring function. The sort, split, and interleave scheme is employed on sorted k queries and speed up is achieved by selecting the best query.

Triple/Quad Store Indexing based on subject, predicate, object, and context enables efficient data retrieval and query processing. Various indexing structures, such as B-trees or hash tables, can be used to organize and store RDF data for fast lookup. Indexing allows the query engine to quickly locate relevant triples/quads based on query patterns, significantly reducing query execution time, especially for large datasets. **Materialized Views** are precomputed query results or intermediate graph patterns stored as persistent data structures [86]. By caching frequently accessed or computationally expensive query results, materialized views can accelerate query processing and improve overall system performance. Materialized views can be updated periodically to reflect changes in the underlying data or query patterns, ensuring query results remain up-to-date [87]. **Predicate and Resource Partitioning** The RDF dataset based on predicates or resources involves dividing the data into smaller subsets that can be processed independently [88]. By partitioning the data, query execution can be distributed across multiple processing units or nodes, reducing the size of individual queries and improving overall performance. Partitioning strategies may include horizontal partitioning (splitting data based on predicates) or vertical partitioning (splitting data based on resources), depending on the characteristics of the dataset and query workload [88].

D. OPTIMIZATION IN LARGE SCALE GRAPH PROCESSING

Large-scale graph processing can be achieved using several ways such as partitioning [32], load balancing [33], compression [34], [35], caching, fast graph traversal, inter-node communication optimization, hardware acceleration [36] and query optimization. **Graph Partitioning** means dividing the graph into smaller partitions based on certain criteria (e.g., node properties, connectivity) to distribute the workload across multiple processing units or machines [32], [89]. Effective partitioning reduces communication overhead and improves parallelism by ensuring that each processing unit handles a manageable portion of the graph [90]. References [32] and [91] provide a comprehensive survey on optimization using graph partitioning techniques. **Load Balancing** involves balancing computational workload evenly across all processing units or machines to prevent bottlenecks and maximize resource utilization [33]. Load balancing techniques ensure that each partition or processing unit receives a comparable amount of work, minimizing idle time and improving overall system performance [92], [93], [94]. Employing **graph compression** algorithms to reduce the storage space required for representing large graphs [34], [35]. Graph compression techniques aim to minimize memory usage and I/O overhead while preserving the essential structural properties and connectivity of the graph [35]. **Caching and Better Locality Optimization** store frequently accessed graph data or intermediate results, reducing the need for repeated computations and improving data locality [95]. By keeping relevant data closer to the processing units, caching enhances memory access patterns and reduces latency during graph traversal and processing [96], [96]. **Optimizing graph traversal** algorithms and data structures to minimize traversal time and improve overall performance [97]. Techniques such as parallel traversal, optimized data representations (e.g., adjacency lists, adjacency matrices), and efficient graph search algorithms (e.g., breadth-first search, depth-first search) can significantly speed up graph traversal operations [97], [98], [99]. Designing **efficient communication protocols** and algorithms for exchanging data and messages between processing units or nodes in a distributed graph processing system [100]. Optimized communication algorithms minimize overhead and latency, enabling fast and reliable data exchange during parallel graph processing tasks. **Hardware Acceleration** is leveraging specialized hardware, such as GPUs (Graphics Processing Units) or FPGAs (Field-Programmable Gate Arrays), to accelerate graph processing tasks. Hardware acceleration can significantly improve computational throughput and reduce processing time for graph analytics and traversal operations. **Optimizing query** execution plans and algorithms to minimize resource usage and execution time for graph queries and analytics tasks. Query optimization techniques aim to reduce the number of I/O operations, minimize intermediate data transfer, and exploit parallelism to improve query performance on large-scale graphs.

III. SYSTEM MODEL

This section is a summary of FlowGraph's architecture and its execution model, more details can be found in [30] and [31]. Section III-A provides an overview of the system's architecture and the different entities in FlowGraph. Section III-B explains the elements of FlowGraph patterns.

A. SYSTEM ARCHITECTURE

FlowGraph is a distributed large-scale graph processing framework. It has one master, a client, and N workers, and the graph of size K vertices is divided over N workers, as shown in Figure 1. The client acts as an interface for the external world, the graph changes, and the pattern is fed to the client. The following assumptions have been made on the changes arriving in the system: (i) the external source will time stamp the messages; (ii) the messages always arrive in order; and (iii) there is no clock source in FlowGraph, the timing is fed by messages coming from an external clock source. The pattern specifies what is to be extracted or obtained from the given graph or how to modify a certain graph. The client is connected to the master, which has several workers connected to it. The master is responsible for the orchestration of the entire operation. The graph changes over time, which means a new vertex or edge could be added, or removed or its attribute can be modified. Each time the change is fed into the system it distributes these changes to the appropriate worker node. Each vertex is assigned a unique identifier using a suitable hashing algorithm and based on that each worker gets a particular vertex. The corresponding edges are assigned to the worker containing the source vertex. Each worker keeps the data structure that maintains the key and vertex, along with the time at which the vertex was added or removed, as shown in Figure 1. FlowGraph performs graph computations and finds numerous patterns in the graph. After the client connects to the master node, the client sends graph changes as well as patterns to be evaluated on the graph. Section III-B explains different parts of patterns in FlowGraph. Each worker maintains a data structure for vertices, as shown below; it includes the vertex name, the time at which the vertex was added, and the vertex itself. A vertex can also store several properties as well as computational results in the key-value pair form. Computational results are stored after vertex-centric computation is performed. Similarly, edges are kept in key-value pair form, where the key contains the name of the source vertex, and the value contains the time at which the edge was added and the edge itself. Just like vertices, edges can also store properties—for example, in a social media graph, an edge might represent a relationship such as a ''friend-of'' between two vertices. In this case, the key would be ''relationship'' while the value would be ''friend-of''. Each vertex and edge is associated with a set of properties. For example, consider a graph for the social media use case: the vertex may have properties such as the name of a person, age, and location, while an edge would have properties such as ''friend-of''. As the graph evolves, each worker stores the current as well as the previous state of the graph. Since our focus is to develop a proof of concept of pattern evaluation on a temporally evolving graph, our system implements a simplistic graph partitioning policy based on hashing and does not have a fault tolerance or load balancing policy. In our system, it is assumed that all nodes are balanced; for the experimental setup, a balanced load is sent from the client side and uses a basic partitioning scheme. Our major focus is on how graph computation and stream processing alongside each other can be handled; and how the pattern evaluation paradigm of stream processing can be combined with an iterative graph computation model.

B. EXECUTION MODEL

This section provides an overview of the language offered by FlowGraph (see [30], [31] for more details). The client sends a certain set of commands (i.e., a pattern) to the master for graph computation to be performed and certain attributes to be detected. The master uses this pattern to send instructions to the workers step by step. The pattern has a certain set of operators that defines what workers have to do on the graph stored on the workers. The worker performs a given operation on the sub-graph and returns the aggregate information to the master. A pattern is a collection of certain clauses, where each clause can be subdivided into clause elements. All clause elements are evaluated sequentially. There are five types of clause elements: *compute*, *select*, *partition*, *aggregate*, *extract*, and *evaluate*. All clauses must end with *evaluate*, which is a predicate clause, and either be True or False. In the rest of this section, each clause's elements are discussed. A typical clause would look like the following:

```
graph.compute(...).selectV(...)
.partitionByV(...).extractV(...)
.aggregate(...).evaluate(...)
```

In the above snippet *compute()* is an iterative graph computation, explained further in Section III-B1. The next part is *selectV()*, which filters results based on given criteria, further explained in Section III-B2. Partitioning *partitionByV()* is analogous to a group used in databases (more details are provided in Section III-B3). During all aforementioned cases, the system is in graph domain; *extractV()* transforms the graph into a collection of tuples on which *aggregate()* could be applied (extraction is further explained in Section III-B4). *aggregate* applies to the tuples extracted by *extractV()* (see Section III-B5). Finally, a predicate can be applied (i.e., *evaluate*d), which asserts a certain condition and returns the result of the given clause, which can be True or False (see Section III-B6 for some details). Multiple clauses can be joined together using *followedBy()*, as shown below:

```
clause1. followedBy("time").clause2
```

The pattern above has two clauses sandwiched by *followedBy()*, which has a time value passed as an argument. As a result, clause 1 will be processed at the current time while clause 2 will be processed at the current time plus the time given inside *followedBy()* also known as the end time. In other words, the current time and the current time plus the end time define a window interval in which the computation is performed.

1) COMPUTE

Compute is the central part of graph processing, and it involves a vertex-centric paradigm. FlowGraph uses an

iterative computation model, where each step is known as a super step. After each step, workers wait for neighbouring workers to complete the task; this is known as the synchronization barrier, and during this waiting window messages are exchanged between the different workers. After the synchronization barrier messages are dispatched over the network without being sent to the master. After completion of all super steps, the final result is aggregated and sent to the master.

```
graph.compute(...)
```

Several graph computations can be performed on the graph; as a proof of concept, two computations are used in this paper as a test case. The first one is triangle identification, where a triangle is said to be formed if any given vertex A_i is connected to any other vertex A_j, and the vertex A_j is connected to A_k, such that the vertex A_k is connected to the A_i, thus forming a triangle of vertices.

```
graph.compute("triangle")
```

The above snippet takes a graph and returns the result as a list of vertices that form triangles. Similarly, another possible example could be counting the maximum number of outgoing edges of the given graph. In this case, the outdegree of each vertex is measured in the graph and the maximum degree of the graph is chosen.

```
graph.compute("maxOutDegree")
```

In the above piece of code, the input would be the graph and the output would be the number representing the maximum outdegree of the graph. Different types of computations take different amounts of time. For example, triangle identification is computationally heavy (i.e., it takes more time to compute) while maximum outdegree is light since it takes less time as compared to triangle identification. Indeed, triangle counting is performed by iterating over triplets of vertices and checking whether the triangle is formed or not, and its worst-case complexity is $O(V^3)$ (a time complexity of $O(E \times D)$, where E is the total number of edges and D is the maximum degree of vertex, can be obtained if an adjacency matrix is used with some optimizations). The time complexity for computing the maximum outdegree, instead, is $O(V^2)$.

2) SELECT

Select is the filtering-like operation in which certain vertices or edges are selected based on certain properties. For example, if vertices having an age greater than 30 are required, the selection would be given as follows:

```
graph.compute(...).selectV("age" > 30)
```

In the above code snippet, a computation is performed (where the ellipsis indicates that the kind of computation performed does not play any role). After getting the result the vertices whose age is greater than 30 are filtered. Similarly, edges can be selected based on certain properties. The select can be extended to multiple properties using conjunction; for example, if the age is greater than 30 and the country is Italy, select can be formulated as a conjunction of both attributes. As discussed before, each worker has data structures that store vertices and edges. Once the selection is done, the vertices and edges are reduced to only those which contain the given attributes.

3) PARTITION

The partition is analogous to the ''group by'' operator in databases. It takes a list of vertices or edges and groups the vertices or edges based on certain criteria. For example, consider a graph having a property named ''country'' on each vertex, and the partitioning by country is performed. Each group contains a single country such as Italy, Germany, the USA, etc. Each partition forms a subgraph, and any operations performed after partitioning are performed on each subgraph separately. For example, if *compute()* is performed after partitioning, the computation will be performed on each subgraph separately.

```
graph.compute(...).selectV("age" > 30)
.partitionByV("Country")
```

In the above code snippet, a computation is performed; after getting the results, the vertices are filtered whose age is greater than 30 and then grouped by country.

4) VALUES EXTRACTION

Extraction transforms the graph domain into a set of values, hence departing from graph processing. In other words, extraction is a bridge between graph processing and stream processing. The aforementioned *compute*(), *select*(), and *partition*() operate on graph data structures; however, once extraction is applied the graph becomes a set of values or a flat list. Now the *aggregate*() discussed in the next subsection can be applied to this set of values. An extraction can be considered as a conversion of a graph data structure into a tuple or flat list. For extraction on vertices and edges, there are two different operators *extractV*() and *extractE*(). A vertex or edge may have several properties, and *extractV* and *extractE* extract all properties stored on vertices and edges, respectively. However, both *extractV* and *extractE* can specify a subset of properties as an argument; in this case, only the specified properties will be extracted from edges or vertices. Let us consider the first example below, in which a computation is performed on the graph, then filtering is performed for all vertices whose age attribute is greater than 30, and finally *extractV()* is applied to get the flat list of vertices defined by the previous part of the clause.

```
graph.compute(...).selectV("age" > 30)
.extractV()
```

As another example consider the snippet below, in which *extractV()* is applied after partitioning; in this case, the graph is grouped by country using *partitionBy()*, then a flat list is

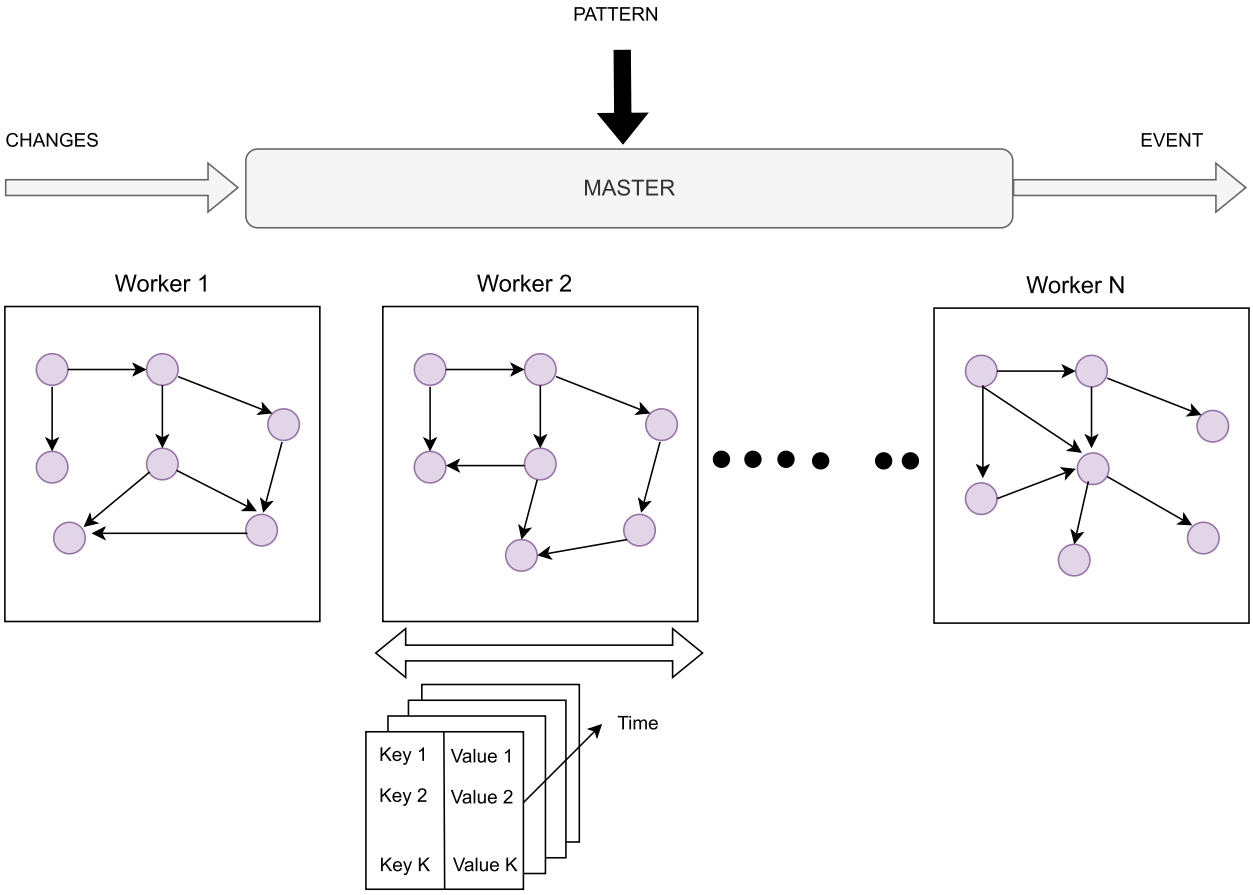

FIGURE 1. FlowGraph architecture.

extracted. However, once extraction is applied the grouping done by *partitionBy()* is unfolded and there are no groups anymore.

```
graph.compute(...).selectV("age" > 30)
.partitionBy("Country").extractV()
```

5) AGGREGATE

The *aggregate* operation in FlowGraph is similar to stream processing. *aAggregate* only works once the extraction has been performed on the data and the graph is transformed into tuples. FlowGraph has several aggregate operators such as *max*, *min*, *average*, *filtering*, *mapping*, *reduction*, and *flat map*. The operator might work on each worker or merge the data into the master. For example, *reduce* collects the data from each worker and brings it to the master. In other cases, data remains on the worker; for example, in the case of filtering each worker filters data on its corresponding nodes. The snippet below shows an example of *aggregate*; after the computation is performed on the graph, the graph is filtered by age greater than 30, and vertices are extracted. The flat list now contains a list of vertices with the ''age'' property included, and *aggregate()* applies the minimum operator on the list obtained through *extractV()*.

```
graph.compute(...).selectV("age" > 30)
.extractV("age").Aagregate("min")
```

6) VARIABLES AND EVALUATION

In the previous sections, several graphs and streaming transformations have been discussed. Once a clause has been extracted it is tested against certain conditions using the *evaluate* primitive. Each clause in FlowGraph must end with an evaluation. Evaluation tests the given clause against some condition and returns True or False. In other words, evaluation converts the given clause into a predicate. It is important to note that evaluation can only refer to a certain variable computed previously in the given clause. Let us consider the following structure:

```
graph.compute(...).<graph transformations>
. extractV()<Aggregate operators>
.evaluate()
```

In the above structure, a *compute()* is performed on the graph, and the graph is transformed using graph transformation operations such as *selectV()* (filtering) and *partitionBy()* (grouping). Once transformed *extractV()* converts the graph to a flat list, on which different Aggregate operators could be applied. Finally, *evaluate()* returns whether the predicate

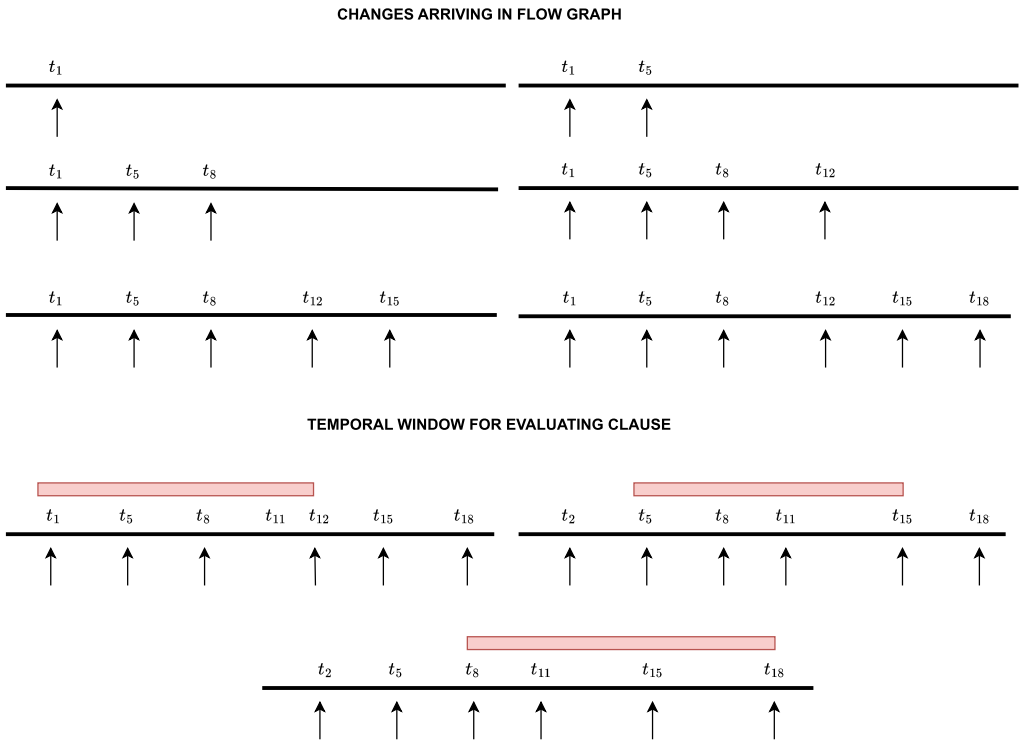

FIGURE 2. Temporal operator and time window in FlowGraph.

is True or False. The snippet below gives a more specific example:

```
graph.compute("triangle").
.selectV("age" > 30).extractV("age")
.aggregate("avg").evaluate("age" > 40)
```

In the above snippet, the triangle identification is done on the graph, which returns all triangles in the graph; the obtained triangles are then filtered using *selectV("age" > 30)*, thus reducing the vertices to only those who have an "age" property greater than 30. The obtained results are converted to a flat list using *extractV("age")*. The aggregation operator *aggregate("avg")* finds the average age on flat list extract by *extractV("age")*. Finally, the predicate is checked using *evaluate("age" > 40)*, which tests if the average age is greater than 40. It is important to note here that the attribute "age" is referred to repeatedly, so it could be replaced by a variable, as shown below.

```
graph.compute("triangle").
.selectV("age" > 30).extractV("age",X)
.Aggregate("avg").evaluate(X > 40)
```

The above snippet shows variable X, which is used in *extractV("age", X)*. This variable can referred to later in the clause as in *evaluate(X > 40)*.

7) TEMPORAL OPERATIONS

As discussed in Section III-A changes arrive at different points of time in FlowGraph. Any modification, insertion, or deletion of an edge or vertex is referred to as a change in the graph. For the sake of simplicity let us assume that the change is the addition of a vertex. Figure 2 shows that the first vertex is added at time t_1, and other changes occur at times t_5, t_8, t_{12}, t_{15} and t_{18}. Consider the pattern below, which contains *followedBy()* sandwiched by two clauses.

```
<clause1>.evaluate().followedBy("10")
<clause2>.evaluate()
```

FlowGraph does not evaluate clauses at a single point in time; instead, it uses a window in which evaluation is performed, as shown in the temporal window part of Figure 2. It is important to note that all computations are triggered only by the new change arriving in the system. In the given code snippet above the first part is a clause *followedBy()* by another clause. The time given in *followedBy()* provides the size of the window. The first computation should start from t_1 and adding the time in *followedBy()* it should go till t_{11}. In this window, t_1 is the start point while t_{11} is the end point of the computation. However, since no change has arrived at time t_{11}, the system will wait until the next change arrives, which is time t_{12} in this case. The first red part from t_1 to t_{12} indicates two points in time across which the computation will be performed. Since the original computation was to be performed at t_1 and t_{11}, at time t_{12} FlowGraph will go back at time t_{11}. As a result, clause 1 will be evaluated at time t_1, while clause 2 will be evaluated at time t_{11}. Once the computation is performed between time t_1 and t_{11}, the window slides to the next available point in time, t_5 in this

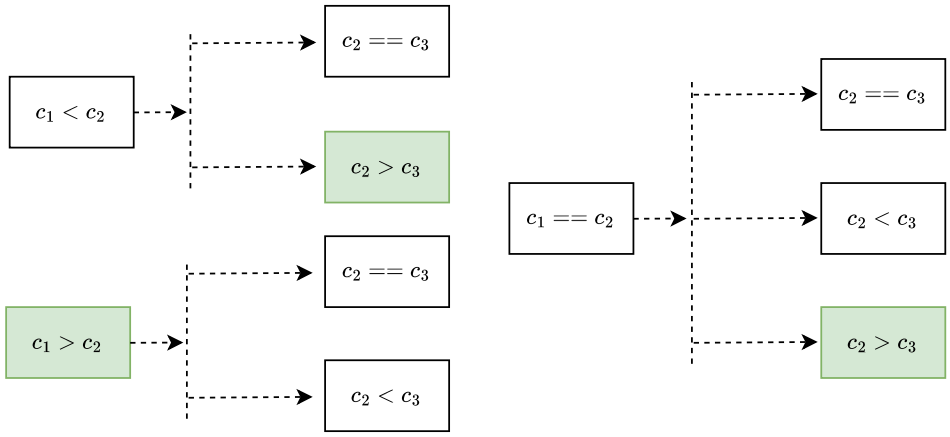

FIGURE 3. Optimization for depth of 3 clauses.

case, and the endpoint of the window is also updated from t_{12} to t_{15}—i.e., the window starts at t_5, and end after the window size specified by the *followedBy()*. In this case, Clause 1 will be evaluated at time t_5, while Clause 2 will be evaluated at time t_{15}; notice that in this case change has arrived exactly at the end of the window size—i.e. at t_{15}. Similarly, in the last case the window slides between t_8 to t_{18}. All workers store changes in vertices and edges along with the time at which the change was made. This means that each worker has different versions of the graph at any given point in time. The client sends a timestamp with each change in vertex and edge. To evaluate a clause at a certain time the worker extracts vertices and edges at a given point in time.

IV. PROPOSED TECHNIQUE

This section discusses the optimizer design, where the pattern is optimized at the master level before it is dispatched to different workers. Optimization refers to the process in which the given pattern is reformulated so that it can run in a more efficient form. In this section, the terms "pattern" and "query" are used interchangeably, since most optimisation problems are referred to as query. Section IV-A explains the basic idea behind the optimization, while Section IV-B presents an example of optimization. Section IV-B also discusses how shuffling the query for the sake of optimization does not affect the query result.

A. BACKGROUND

Section III-B1 describes the computation in FlowGraph. It is important to note that not all *compute()* are equal, some take more time than others. For example, a triangle identification operation in a large-scale graph takes a much longer time compared to computing how many outgoing edges there are in vertices. The amount of time required to perform a certain computing operation depends on how many iterations and computations are needed to get the final answer. In Section III-B1 triangle identification and max incoming edges are explained.

It can be seen that finding triangles requires several exchanges of messages to assert the formation of the triangle as compared to finding a maximum number of outgoing edges. The basic idea of optimization is based on the fact that if there is any heavy clause and it is executed first, then it would take a long time before it could be True or False. Let us assume that the heavy clause is *followedBy()* the lighter clause and this evaluates to False. The result of both clauses would be False, however, the time it took for a heavy clause to execute gets wasted. For an illustration of the concept, two types of *compute()* are chosen i.e. out-degree and triangle identification arranged in the order from lighter to heavier.

B. OPTIMIZATION OF QUERY

Section III-B introduced the concept of a pattern of two clauses. However, the pattern is not limited to two clauses; rather, in general, a pattern consists of N clauses, where all clauses are separated using *followedBy*(), as shown below.

```
<clause1>.evaluate(...).followedBy(...)
<clause2>.evaluate(...) .followedBy(...)
...
<clauseN>.evaluate(...)
```

Therefore, a single pattern is a set of N clauses $C = \{C_1, C_2, \ldots C_N\}$. Each clause has a *compute()* clause element. Every clause C_j consists of different components, which can be divided into graph-based clause elements i.e. *compute*(), *select*(), *partition*() aggregate-based clause elements i.e., *extract*() and *aggregate*() and predicate *evaluate*().

As an example consider the code snippet below, it has two clauses, where the first one has *compute("triangle")*, which is heavy, while the second one has the lighter computation *compute("maxOutDegree")*; both are evaluated with the window size of 10 since *followedBy("10")* has 10 as time value.

```
graph.compute("triangle").
.selectV("age" > 30).extractV("age",X)
```

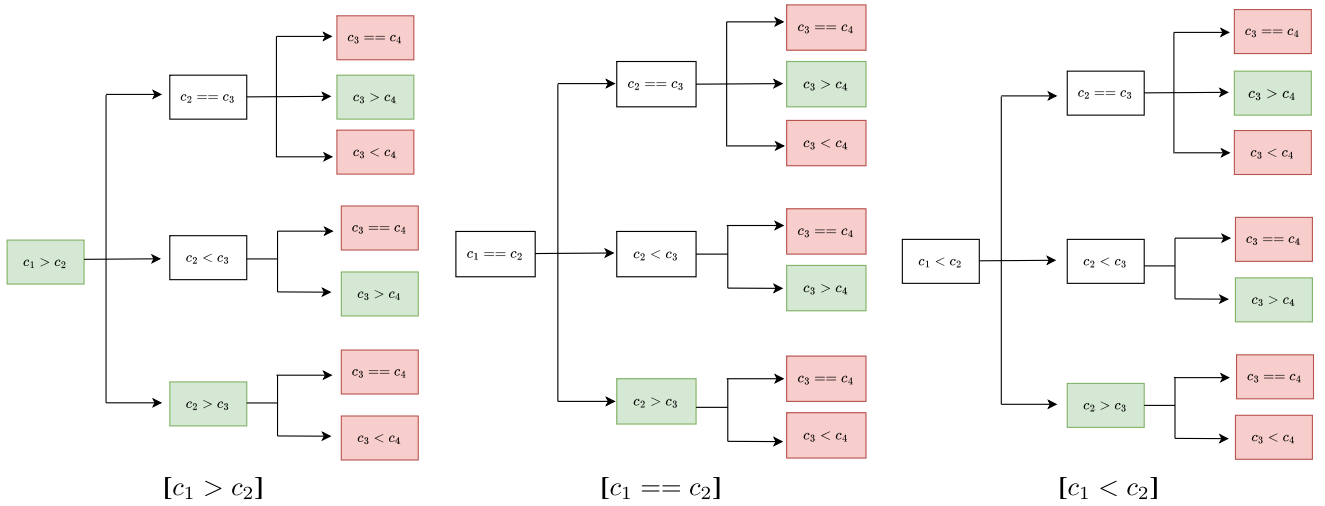

FIGURE 4. Optimization for depth of 4 clauses.

```
.Aggregate("avg").evaluate(X > 40).
.followedBy("10")
graph.compute("maxOutDegree","Y").
.extractV("Y").evaluate(Y <= 5)
```

The above pattern consists of two clauses separated by *followedBy()*. The first clause identifies the triangles and filters out triangles using *selectV()* whose age property is greater than 30. Converts the graph into a set of tuples using *extractV()*, it only gets age property from each vertex and gives it a variable name X. Finds the average age using *Aggregate()* and applies the predicate *evaluate()* to check if the average age is greater than 40. The second clause finds the maximum out-degree and stores the result in variable Y. This variable Y is stored on each vertex along with the other properties of vertices. The graph is converted to a set of tuples by using *extractV()* applied on variable Y. Finally, predicate *evaluate()* is applied to check if the maximum in-degree of the graph is less than equal to 5. Each j_{th} *compute*() has certain weights assigned to it, $C_j \rightarrow w_j$. For example, the weight for the first clause in the snippet above is higher (triangle identification) than the weight for the second clause, which is lighter (maximum out-degree). Optimization is defined as an out-of-order evaluation of N clauses based on their weights w_j, where the lighter clauses are evaluated before the heavier ones. If a lighter clause is evaluated as False, the heavier clauses are not evaluated. The major premise behind this scheme is that even though a heavier clause evaluates to True, if the following lighter clause is False, then the result will be False. The optimal scheme would be to check if the lighter clause is False, and then evaluation of the heavier clause is not required at all, resulting in less computational time.

Let us consider patterns made of three and four-clause as an example of query optimization. The clauses used for the proof of concept are shown in Figure 3 and Figure 4, respectively.

1) THE THREE-CLAUSE CASE

For the three-clause case $C = \{C_1, C_2, C_3\}$ shown in Figure 3, let *compute*() of these clauses be c_1, c_2 and c_3 respectively; there are two levels of clauses, i.e., $L = 2$. The optimization algorithm back-propagates from the last node c_3 to c_2, and it looks for nodes where $c_2 > c_3$. This means c_2 has a higher weight as compared to c_3, therefore, c_2 is heavy and c_3 is light. If $c_2 > c_3$ holds, the optimization algorithm executes c_3—the lighter clause—before c_2, which is a heavier clause. If c_3 evaluates to false there is no need to evaluate c_2, since even if c_2 is evaluated, the last clause c_3 becomes false and the entire pattern evaluates to false. If c_3 is pre-evaluated it would save the computation time needed to evaluate c_2.

If there is no such node at level $c_2 > c_3$, the algorithm goes to level $L = 0$ and checks if $c_1 > c_2$. In this case, c_2 is evaluated before c_1. This means c_2 is lighter than c_1, so the same principle applied for $c_2 > c_3$ can be used here.

2) THE FOUR-CLAUSE CASE

For the four-clause case $C = C_1, C_2, C_3, C_4$ shown in Figure 4, let *compute*() of C_1, C_2, C_3 and C_4 be c_1, c_2, c_3 and c_4 respectively; the number of levels is three, so $L = 3$. The optimization algorithm performs backpropagation going from the last compute c_4 backwards. It looks for nodes where $c_3 > c_4$ holds. This means c_3 has a higher weight as compared to c_4, therefore c_3 is heavy and c_4 is light. If $c_3 > c_4$ holds, the optimization algorithm executes c_4 (the lighter clause) before c_3, which is a heavier clause. If c_4 evaluates to false there is no need to evaluate c_4, since even if c_3 is evaluated, the last clause c_4 becomes false and the entire pattern evaluates to false. If c_4 is pre-evaluated it saves the computation time needed to evaluate c_3. If there is no such node at level $c_3 > c_4$, the algorithm goes at $L = 1$ and checks if $c_2 > c_3$, as shown in Figure 4. In this case, c_3 is evaluated before c_4 in a similar way as previously explained for $c_3 > c_4$. This means c_3 is

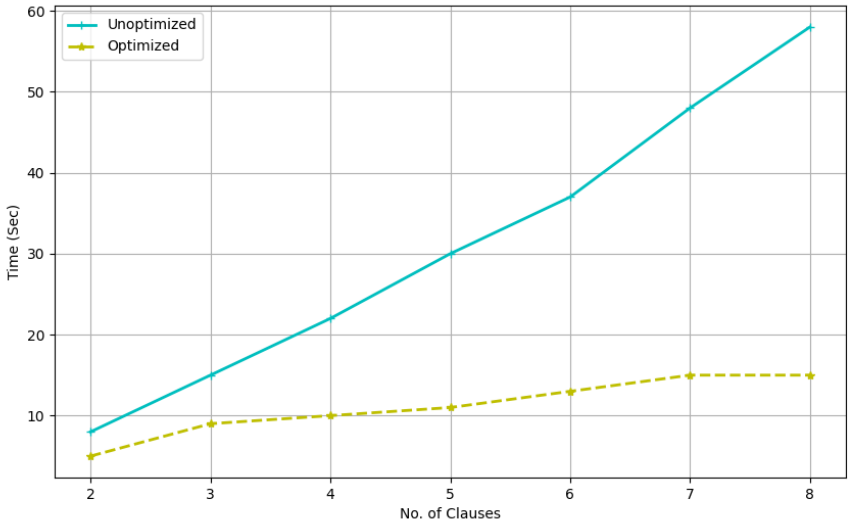

FIGURE 5. Increase of clauses versus times in second.

lighter than c_4 and the same principle applied for $c_3 > c_4$ can be used here.

3) THE *N*-CLAUSE CASE

The three and four-clause cases shown in Section IV-B1 and Section IV-B2, respectively, can be extended to N clauses. For such a case, the level of the tree will be $L = N - 1$. The tree will be parsed from the leaf node containing $c_{N-1} < c_N$. The process is repeated for each level j such that j varies from L to level 1, wherein each case condition $c_{N-(j-L+1)} < c_{N-(j-L)}$ is traversed backwards.

C. CORRECTNESS OF THE OPTIMIZATION

When the optimization is applied, the lighter clauses are evaluated before the heavier clauses, which modifies the order in which the pattern was intended to be executed. For example, let us consider that clause 2 is lighter than clause 1 in the snippet given below.

```
<clause1>.evaluate().followedBy()
<clause2>.evaluate()
```

With the optimization, clause 2 will be evaluated before clause 1, whereas in the original pattern clause 1 was intended to be executed before clause 2. But would this flipping modify the correctness of the pattern (in other words, are the two patterns equivalent)?

The best way to answer this question leads us to how workers store graphs with timestamps. Each vertex is stored with the time at which it was added, removed, or modified. This means that graph computation as well as pattern detection can be performed at any point in time without affecting the graph at other times. As a result, FlowGraph can move backwards or forward in history, irrespective of clause order in the graph. Consider the example of Figure 2, and assume that the first clause (e.g., triangle identification) is heavy and evaluated at time t_1, and the second is light (e.g., max incoming edges) and evaluated at time t_{11}. If the second clause is evaluated before the first clause at time t_{11}, it would be run on the graph at instance t_{11}. Since the graph at t_{11} is not affected by whether the triangle identification was run on time t_1 or not, evaluation at time t_{11} is order oblivious. In other words, changing the evaluation sequence of clauses does not alter the overall result of the clause. More precisely, the correctness of the optimization is ensured by the four principles of evaluation in FlowGraph, which make clauses position-oblivious.

1) GRAPH STATE INDEPENDENCE

The graph data structure maintains its state independently of the order in which computations are performed on it. Each vertex in the graph is timestamped, indicating when it was added, removed, or modified. As a result, the evaluation of each clause operates on the graph state at the time of evaluation, unaffected by the order in which other computations are performed. Let G denote the graph data structure, and V represent the set of vertices in G. Each vertex $v_i \in V$ is associated with a timestamp t_i, indicating its state at a given time. Consider a clause C_i in the pattern, evaluated at time t_i. Let $f(C_i, G_{t_i})$ represent the result of evaluating C_i on graph G at time t_i. The evaluation function f operates solely on the state of the graph at the time of evaluation. Therefore, for any clause C_i, its evaluation is independent of the state of the graph at any other time.

2) TEMPORAL EVALUATION

When a clause is evaluated, it operates on the state of the graph at that specific timestamp. This means that the result of evaluating a clause is determined solely by the state of the graph at the time of evaluation, regardless of when other computations were performed. Given the timestamped nature

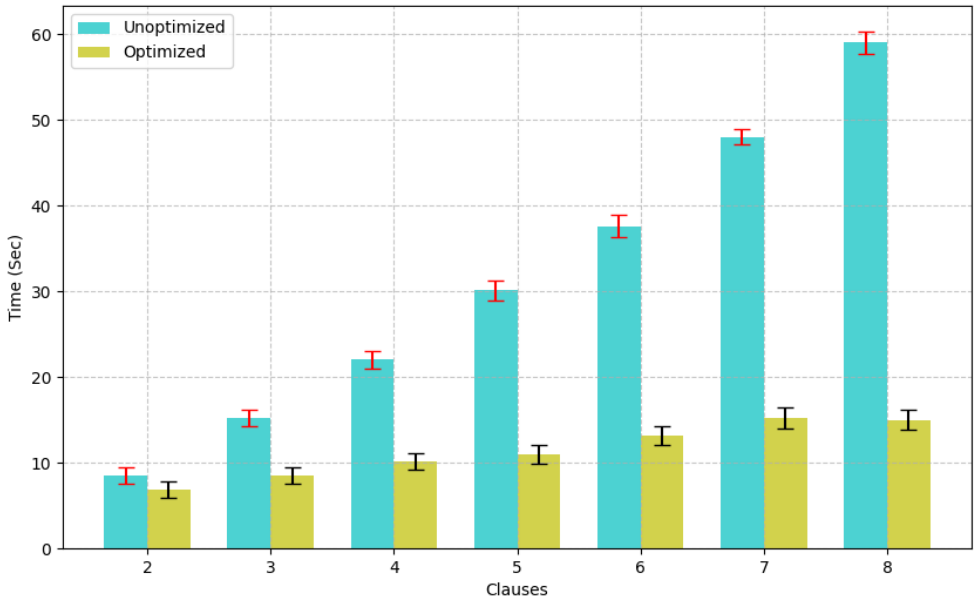

FIGURE 6. Impact of error variance on results.

TABLE 1. Abbreviation used in the simulations.

Abbreviation	Select Variation	Clause Elements
WOS-UEA	Without Select	Unoptimized Extract, Aggregate
WOS-OEA	Without Select	Optimized Extract, Aggregate
WOS-UPA	Without Select	Unoptimized Partition, Aggregate
WOS-OPA	Without Select	Optimized Partition, Aggregate
SA-USEA	Single Attribute	Unoptimized Select, Extract, Aggregate
SA-OSEA	Single Attribute	Optimized Select, Partition, Aggregate
SA-USPA	Single Attribute	Unoptimized Select, Partition, Aggregate
SA-OSPA	Single Attribute	Optimized Select, Partition, Aggregate
TA-USEA	Two Attribute	Unoptimized Select, Extract, Aggregate
TA-OSEA	Two Attribute	Optimized Select, Extract, Aggregate
TA-USPA	Two Attribute	Unoptimized Select, Partition, Aggregate
TA-OSPA	Two Attribute	Optimized Select, Partition, Aggregate

of the graph, each clause's evaluation is tied to a specific timestamp. Let t_i be the timestamp associated with clause C_i. For any clause C_i, its evaluation is solely determined by the state of the graph at time t_i.

3) PATTERN COMPOSITION

The pattern is composed of individual clauses, each representing a specific computation or condition. The overall result of the pattern is determined by the combined results of evaluating these clauses. Since each clause's evaluation is independent of the other clauses, the overall result of the pattern remains unchanged regardless of the evaluation order. The overall result of the pattern is determined by the combined results of evaluating individual clauses.

4) CONSISTENCY OF RESULTS

The consistency of results is maintained irrespective of the order in which clauses are evaluated. This is because the evaluation of each clause is based on the same underlying graph state, ensuring that the computed results are consistent regardless of the evaluation order.

V. RESULTS AND PERFORMANCE EVALUATION

This section presents the results obtained after performing various experiments. The main goal of our experiments is to validate that our optimization algorithm executes patterns in less time than the non-optimized algorithm. Section V-A describes the experimental setup. To validate our algorithm three experiments are being performed, with varying clause sizes (Section V-B), types of clauses (Section V-C and Section V-D), and graph sizes (Section V-E).

TABLE 2. Comparison of time of optimized versus non-optimized patterns.

No. Of clauses in the pattern	C1	C2	C3	C4	With Out Optimization	With Optimization	Reduction
Two clause Pattern	Heavy	Light	X	X	10 s (9.3 - 10.5)s	3.5 s (2.8 - 3.9)s	$\frac{1}{3rd}$
	Light	Heavy	X	X	10.2 s (9.8 - 10.8)s	10.2 s (9.7 - 10.9)s	No Optimisation
	Heavy	Heavy	X	X	10.2 s (9.5 - 10.6)s	10.1 s (9.8 - 10.7)s	No Optimisation
	Light	Light	X	X	3.5 s (3.3 - 4.0)s	3.5 s (3.2 - 3.9)s	No Optimisation
Three clause Pattern	Light	Light	Light	X	10.2 s (9.8 - 10.8)s	10.1 s (9.7 - 10.5)s	No Optimisation
	Light	Light	Heavy	X	13.1 s (12.9 - 13.3)s	13.2 s (12.8 - 13.4)s	No Optimisation
	Light^1	Heavy	Light^2	X	13.3s (12.6 - 13.7)s	3.5^1 s 6.1^2 s	$\frac{1}{4^{th}}$ $\frac{1}{2}$
	Heavy	Light^1	Light^2	X	13.1 s (12.7 - 13.5)s	3.3^1 s, 6.4^2 s	$\frac{1}{4^{th}}$, $\frac{1}{2}$
	Heavy	Heavy	Heavy	X	20.3 s (19.9 - 21.4)s	20.1 s (19.8 - 20.5)s	No Optimisation
	Heavy	Heavy	Light	X	17.2 s (17.1 - 17.9)s	3.4 s (3.2 - 4.1)s	$\frac{1}{4^{th}}$
	Heavy	Light	Heavy	X	17.1 s (16.5 - 17.4)s	3.4 s (3.1 - 3.9)s	$\frac{1}{4^{th}}$
	Light	Heavy	Heavy	X	17.3 s (16.5 - 17.6)s	17.2 s (16.4 - 17.6)s	No Optimisation
Four clause Pattern	Light	Light	Light	Light	12.3 s (12.1 - 13.1)s	12.2 s (11.8 - 12.7)s	No Optimisation
	Light	Light	Light	Heavy	18.1 s (17.7 - 19.0)s	18.2 s (17.8 - 18.7)s	No Optimisation
	Heavy	Light^1	Light^2	Light^3	18.4 s (17.9 - 18.6)s	3.4^1 s, 6.2^2, 9.3^3 s	$\frac{1}{6^{th}}$, $\frac{1}{3rd}$, 1/2
	Light^1	Heavy	Light^2	Light^3	18.2 s (17.7 - 19.0)s	3.2^1 s,6.2^2, 9.4^3 s	$\frac{1}{6^{th}}$, $\frac{1}{3rd}$, 1/2
	Light^1	Light^2	Heavy	Light^3	18.2 s (17.8 - 18.7)s	3.4^1 s, 6.2^2, 9.3^3 s	$\frac{1}{6^{th}}$, $\frac{1}{3rd}$, 1/2
	Light	Light	Heavy	Heavy	21.2 s (20.1 - 21.9)s	21.1 s (20.5 - 21.9)s	No Optimisation
	Heavy	Light^1	Light^2	Heavy	21.3 s (20.9 - 21.8)s	3.4^1 s, 6.4^2	$\frac{1}{7^{th}}$, $\frac{1}{4^{th}}$
	Heavy	Heavy	$Light^1$	$Light^2$	21.2 s (20.5- 21.5)s	3.31 s, 6.22	$\frac{1}{7^{th}}$, $\frac{1}{4^{th}}$
	Heavy	Light^1	Heavy	Light^2	21.4 s (20.2 - 21.8)s	3.3^1 s, 6.4^2	$\frac{1}{7^{th}}$, $\frac{1}{4^{th}}$
	Light^1	Heavy	Heavy	Light^2	21.3 s (20.6 - 21.5)s	3.4^1 s, 6.2^2	$\frac{1}{7^{th}}$, $\frac{1}{4^{th}}$
	Light^1	Heavy	Light^2	Heavy	21.3 s (20.3 - 22.0)s	3.4^1 s, 6.4^2	$\frac{1}{7^{th}}$, $\frac{1}{4^{th}}$
	Heavy	Heavy	Heavy	Light	24.8 s (23.8 - 25.5)s	3.3 s (2.7 - 3.8)s	$\frac{1}{8^{th}}$
	Heavy	Heavy	Light	Heavy	24.2 s (23.8 - 25.1)s	3.4 s (3.1 - 3.7)s	$\frac{1}{8^{th}}$
	Heavy	Light	Heavy	Heavy	24.7 s (23.1 - 24.9)s	3.2 s (2.9 - 3.7)s	$\frac{1}{8^{th}}$
	Light	Heavy	Heavy	Heavy	29.3 s (28.5 - 30.5)s	29.3 s (28.9 - 30.1)s	No Optimisation
	Heavy	Heavy	Heavy	Heavy	29.2 s (28.7 - 29.8)s	29.3 s (28.4 - 30.6)s	No Optimisation

A. EXPERIMENTAL SETUP

The original FlowGraph implementation, which is freely available [101], was evaluated on EC2 instances of the Amazon AWS cloud [30], [31]. The experiments presented in this paper were carried out on a server running Ubuntu Linux on an Intel i7 (12^{th} generation) processor with 128 GB of RAM.[1] The three-experiment setup proves that our optimization algorithm gives a much better time regardless of graph size, execution order, number of clauses, and clause elements. The first experiment, described in Section V-B, shows that the optimized queries take much less time as compared to non-optimized ones. The second experiment, presented in Section V-D, shows that varying clause elements do not affect the optimisation. Different possible combinations of clause elements have been used, and the results show that the optimization works regardless of the clause elements chosen. Finally, Section V-E shows that the optimization works on graphs of any size. The synthetic graph generator (SGG) was used to generate the graphs for the experiments. The SGG application in our framework can generate directed and undirected graphs of any size.

B. OPTIMIZING QUERIES WITH VARYING CLAUSE SIZES

The first experiment is designed to check the impact of the variation of the placement of heavy compute clauses in the pattern. As discussed in Section III-B1 a ''triangle identification'' operation is heavy, while ''Max Out-Degree'' is light. The number of clauses has been increased from two to eight. For this experiment, we have kept the injection speed to 5 changes per second (CPS), using graphs of size 1000k vertices, and 4 workers; also, the clause has all elements i.e. *compute()*, *selectV()*, *extractV()*, *partitionBy()*, *aggregate()*, and *evaluate()*. It can be seen in Figure 5 that in all cases the gap between optimized and non-optimized computations grows as the number of clauses increases.

Patterns consisting of multiple clauses, categorized as either heavy or light based on their computational requirements, can be optimized to improve execution time. However, certain patterns may be non-optimizable, meaning that the clauses within these patterns cannot be rearranged or swapped to achieve better performance, resulting in the execution time of the original and optimized clauses remaining the same. This can be checked by implementing the optimizer and comparing the execution times of the original and optimized patterns. By analyzing patterns that are categorized as non-optimizable, the conditions that prevent optimization can be determined. Table 2 shows the breakdown of clauses of Figure 5 from two clauses up to four. A scenario has been considered where the light clause is False to experiment with two clauses. For the first case ($\{Heavy, Light\}$), evaluating the pattern without optimization took 10 seconds, while with the optimization it took 3.5 seconds since in this case the light clauses are evaluated to False, so the heavy ones are not evaluated. This is a reduction of one-third of the total time. However, no optimization is done if both clauses are either light, heavy or end with heavy. For three clauses all eight combinations are shown in Table 2, and it can be observed that the optimization is possible for four out of eight clauses. For the $\{Light1, Heavy, Light2\}$ case two variations have been considered: in the first variation $Light1$ is False, in the second variation $Light1$ is True, while $Light2$ is False. In all cases, the optimized computation takes $\frac{1}{4}$ of the time of the non-optimized one. In the $\{Light1, Heavy, Light2\}$ and $\{Heavy, Light1, Light2\}$ cases the optimized computation takes $\frac{1}{4}$ and $\frac{1}{2}$ of the time of the non-optimized one, respectively. Similarly, for four clauses, there are 11 patterns out of a total of 16 that are optimizable.

Figure 5 shows the average of the total computation time in each N-clause pattern. The values given in Figure 5 and Table 2 are computed after repeating the experiment 20 times. The execution time of each iteration varies slightly depending on how busy the processor is during execution. Columns ''without optimization'' and ''with optimization'' in Table 2 show, between brackets, the minimum and maximum execution times for each repeated experiment. The times shown in the Table and figures are the averages over all iterations. As Figure 6 shows, the difference between minimum and maximum times is between 0.5s and 1.5s, independent of whether the pattern was optimized. Hence, the results given in Figure 7 are consistent, and the variance in execution time does not change the conclusion that the optimization reduces the time concerning non-optimized executions.

C. VARYING CLAUSE ELEMENTS

This section describes the method underlying the experiments shown in Section V-D, which concerns the impact of changing clause elements. Table 1 provides the abbreviations (explained later) used to label the experiments shown in Figure 7 and in Figure 8.

1) EXTRACT, AGGREGATE WITHOUT SELECT

In the first variation, the following are removed: *selectV()* (i.e., filtering) from the clause, abbreviated as ''without select'' (WOS). The clause contains *extractV()* and *aggregate()* operations abbreviated as ''extract, aggregate'' (EA), as shown below. The combination of ''without select'' and ''extract, aggregate'' is abbreviated as WOS-EA.

```
graph.compute(...)
.extractV(...).aggregate(...)
```

2) PARTITION, AGGREGATE WITHOUT SELECT

As a variation of Section V-C1, the following are being included: *partitionV()*. In this case, graph computation is performed, and the result is grouped (partition) and then converted to a list of tuples, abbreviated as ''partition,

[1] A single-threaded, monolithic version of the optimizer, which can be used to run FlowGraph on a single machine, is freely available at [102].

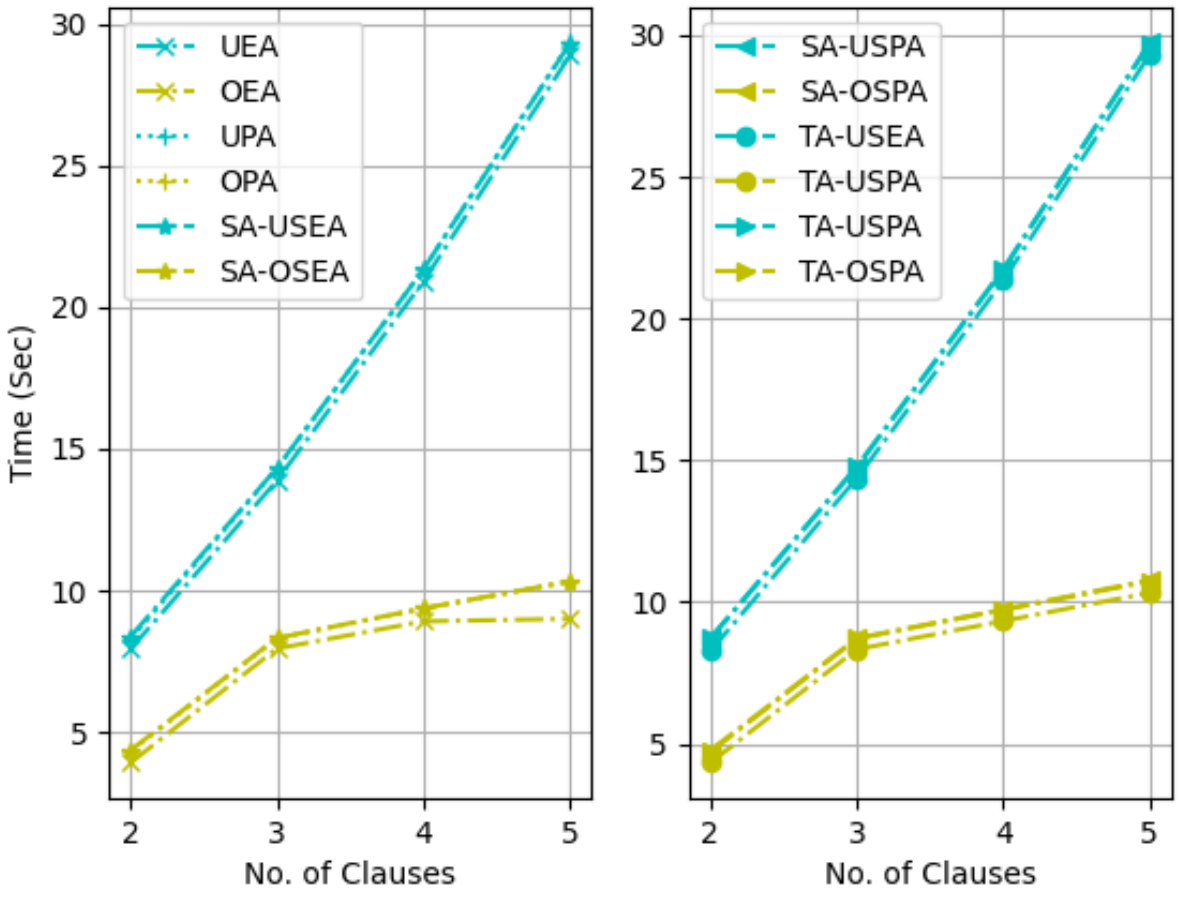

FIGURE 7. Varying number of clauses with different clause elements.

aggregate'' (PA). The combination of ''without select'' and ''partition, aggregate'' is abbreviated as WOS-PA.

```
graph.compute(...).partionBy(...)
.extractV(...).aggregate(...)
```

3) SINGLE ATTRIBUTE SELECT, EXTRACT, AGGREGATE

In sections V-C1 and V-C2 the clause without filtering is applied before tuples are made. In this new case, a single attribute filtering has been performed, without partitioning, as shown below and abbreviated as ''single attribute extract aggregate'' (SA-EA).

```
graph.compute(...).selectV("age" > 30)
.extractV(...).aggregate(...)
```

4) SINGLE ATTRIBUTE SELECT, PARTITION, AGGREGATE

The single attribute *select* in Section V-C3 is extended here to incorporate partition, abbreviated as ''single attribute partition select'' (SA-PS).

```
graph.compute(...).selectV("age">30)
.partitionBy(...).extractV(...)
.aggregate(...)
```

5) TWO ATTRIBUTE SELECT, EXTRACT, AGGREGATE

So far clauses without select have been discussed in sections V-C1 and V-C2, and single select attribute in sections V-C3 and V-C4. In these two new cases, the pattern is extended to two attributes (TA). In the first case below, the *selectV()* is applied with *partitionBy()*, abbreviated as ''two attributes select extract aggregate'' (TA-SEA), as shown below.

```
graph.compute(...)
.selectV("age">30 and"country"="Italy")
.extractV(...).aggregate(...)
```

In the second case, *partitionBy()* with two attributes *selectV()*, abbreviated as ''two attributes select partition aggregate'' (TA-SPA) has been included.

```
graph.compute(...)
.selectV("age">30 and "country"="Italy")
.partitionBy(...).extractV(...)
.aggregate(...)
```

D. IMPACT OF VARYING CLAUSE ELEMENTS

For the second experiment, the elements in each clause have been changed; notice that, in the figures, optimized

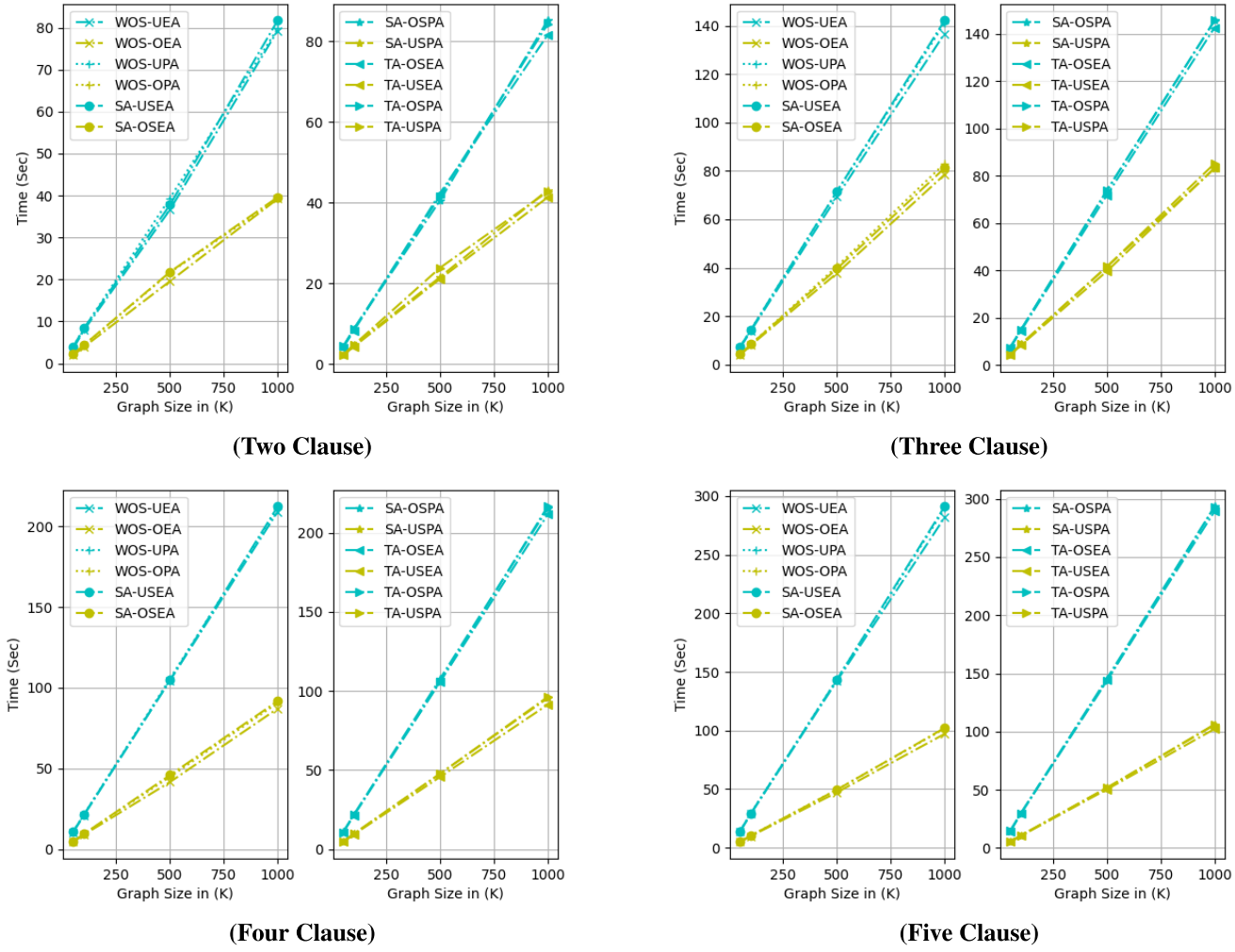

FIGURE 8. Varying graph sizes with different clause elements (For varying clauses).

executions are shown in yellow, while unoptimized one are shown in cyan. As shown in Figure 7, the first case does not have any *selectV()* (without filtering), the second one has a single *selectV()* attribute i.e. age, and the third one has two attributes *selectV()*, age and country. In each case, the number of clauses in the pattern is increased from two to six. The clause without *selectV()* has two variations, one with *extractV()* and one with *partitionBy()*. In all cases, a constant rate of change has been used, a graph size of 1000k vertices, and four workers. The number of clauses is being changed from two to five. Note that the experiments depicted in Figure 7 have slight variations of execution times and the figure shows the average duration of 20 executions per experiment. As shown in Figure 6, the execution time of each experiment varies between minimum and maximum values. Nevertheless, the time variance, which is in the order of the second, does not affect the shape of the graphs or the overall results of Figure 7. In each case, it has been considered that clauses do not have: all light, all heavy, or heavy at the end of clauses. In other words, the optimizable cases are being selected. For two clauses there is only one case where optimization is possible, i.e., heavy *followedBy()* light. For the three-clause pattern, there are four cases where optimization is possible, while for four clauses eleven optimizable scenarios are possible. Results obtained with varying clause sizes are shown in Table 2. The results presented in Figure 7 show that the trend for single and two attributes is the same if the number of clauses is changed from two to five.

E. IMPACT OF VARYING GRAPH SIZE

The third experiment is designed to check the impact of graph size on the optimization algorithm. The number of clauses is changed from two to four and the size of the graph is varied from 250k to 1000k vertices. For this experiment, 4 workers are used and the injection speed of 5 CPS is maintained. Along with graph size and the number of clauses, the number of clause elements is modified (the corresponding abbreviations are shown in Table 1). It can be seen in Figure 8 that in all cases the trend of the increase in optimization

time as the graph size increases are similar for two, three, four, and five clauses. Moreover, comparing different clause elements in the overlay plots shows that the trends also remain comparable for varying clause elements. Finally, the gap between the optimized and non-optimized curves shows an almost similar deviation for all clause elements. This leads us to argue that the proposed optimized algorithm is not affected by the graph size, clause elements or the number of clauses. As discussed in Section V-D, Figure 8 shows the average duration computed over 20 executions per experiment. Like in Figure 7, the slight variations (in the order of the second) of the execution time do not affect the overall results shown in Figure 8.

F. LIMITATIONS AND FUTURE WORK

In this paper, the optimizations for pattern detection frameworks over large-scale graphs are discussed. The results show a 75% reduction in time in query processing capabilities of the FlowGraph engine, though the same techniques could also be implemented for other platforms. However, the technique also has limitations, optimization is only possible if the clauses of the pattern occur in a certain order. For example, optimization is not possible if the pattern has all clauses of similar weight i.e., all light or all heavy. The second limitation of this work is that the results are tested over synthetic graphs, and software is used to generate graphs of different sizes. In this paper, testing for optimization algorithms is done for different graph sizes, varying clause types in a pattern, and clause sizes in a pattern. However, a constant injection rate for changes of 5 CPS is assumed. In future work, the impact of variations in the injection rate will be tested. FlowGraph can not handle deep learning algorithms, as future work is planned to include machine learning-based algorithms. In this case, the algorithm will be trained using offline training data, once trained the model will be deployed on a FlowGraph which could asses vertex or edge properties to take certain decisions.

VI. CONCLUSION

Modern-day graphs are dynamic and they evolve temporally. Current large-scale graph processing systems do not accommodate such temporal changes while evaluating large-scale graphs. Moreover, large-scale graph processing systems do not allow pattern detection which has a central place in stream processing systems. We presented FlowGraph [30], [31], a large-scale processing framework that can handle graph computations along with pattern detection for temporally evolving graphs. Large-scale graph processing uses an iterative computation paradigm known as TLAV; however, pattern detection requires a data pipeline that can process data in a very short time. In this paper, we presented a query optimization for FlowGraph which can reduce query processing time and enable more efficient pattern detection for large-scale graph processing. Our results proved that 75% of optimization can be achieved, independent of graph size, clause elements, and number of clauses in the pattern.

REFERENCES

[1] M. Gabielkov, A. Rao, and A. Legout, "Studying social networks at scale: Macroscopic anatomy of the Twitter social graph," in *Proc. ACM Int. Conf. Meas. Model. Comput. Syst.*, Jun. 2014, pp. 277–288.
[2] M. Kurant and P. Thiran, "Extraction and analysis of traffic and topologies of transportation networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 74, no. 3, Sep. 2006, Art. no. 036114.
[3] J. Lin and Y. Ban, "Complex network topology of transportation systems," *Transp. Rev.*, vol. 33, no. 6, pp. 658–685, Nov. 2013.
[4] G. A. Pavlopoulos, P. I. Kontou, A. Pavlopoulou, C. Bouyioukos, E. Markou, and P. G. Bagos, "Bipartite graphs in systems biology and medicine: A survey of methods and applications," *GigaScience*, vol. 7, no. 4, Apr. 2018, Art. no. giy014.
[5] G. A. Pavlopoulos, M. Secrier, C. N. Moschopoulos, T. G. Soldatos, S. Kossida, J. Aerts, R. Schneider, and P. G. Bagos, "Using graph theory to analyze biological networks," *BioData Mining*, vol. 4, no. 1, pp. 1–27, Dec. 2011.
[6] M. Eirinaki, J. Gao, I. Varlamis, and K. Tserpes, "Recommender systems for large-scale social networks: A review of challenges and solutions," *Future Gener. Comput. Syst.*, vol. 78, pp. 413–418, Jan. 2018.
[7] X. Zhao, J. Liang, and J. Wang, "A community detection algorithm based on graph compression for large-scale social networks," *Inf. Sci.*, vol. 551, pp. 358–372, Apr. 2021.
[8] L. Quick, P. Wilkinson, and D. Hardcastle, "Using pregel-like large scale graph processing frameworks for social network analysis," in *Proc. IEEE/ACM Int. Conf. Adv. Social Netw. Anal. Mining*, Aug. 2012, pp. 457–463.
[9] G. Mali et al., "A new dynamic graph structure for large-scale transportation networks," in *Int. Conf. Algorithms Complex.* Berlin, Germany: Springer, 2013.
[10] J. Cao, Q. Li, W. Tu, Q. Gao, R. Cao, and C. Zhong, "Resolving urban mobility networks from individual travel graphs using massive-scale mobile phone tracking data," *Cities*, vol. 110, Mar. 2021, Art. no. 103077.
[11] J. P. Pereira and D. Pavlov, "Route planning in large-scale transport networks: A multi-criteria approach using prefractal graphs with optimization of transportation costs," in *Trends and Applications in Information Systems and Technologies*, vol. 39. USA: Springer, 2021.
[12] M. Li, H. Cui, C. Zhou, and S. Xu, "GAP: Genetic algorithm based large-scale graph partition in heterogeneous cluster," *IEEE Access*, vol. 8, pp. 144197–144204, 2020.
[13] T. Aittokallio, "Graph-based methods for analysing networks in cell biology," *Briefings Bioinf.*, vol. 7, no. 3, pp. 243–255, May 2006.
[14] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2018, pp. 974–983.
[15] Z. Wu, C. Song, Y. Chen, and L. Li, "A review of recommendation system research based on bipartite graph," in *Proc. MATEC Web Conf.*, vol. 336, 2021, p. 05010.
[16] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 1–39, Nov. 2015.
[17] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–53, May 2019.
[18] M. E. Coimbra, A. P. Francisco, and L. Veiga, "An analysis of the graph processing landscape," *J. Big Data*, vol. 8, no. 1, pp. 1–41, Apr. 2021.
[19] V. Kalavri, V. Vlassov, and S. Haridi, "High-level programming abstractions for distributed graph processing," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 2, pp. 305–324, Feb. 2018.
[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Operating Syst. Design Implement.*, 2012, pp. 17–30.
[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning in the cloud," 2012, *arXiv:1204.6078*.
[22] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "GraphLab: A new framework for parallel machine learning," 2014, *arXiv:1408.2041*.
[23] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: A graph engine for temporal graph analysis," in *Proc. 9th Eur. Conf. Comput. Syst.*, Apr. 2014, pp. 1–14.

[24] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen, "ImmortalGraph: A system for storage and analysis of temporal graphs," *ACM Trans. Storage*, vol. 11, no. 3, pp. 1–34, Jul. 2015.

[25] X. Shi, B. Cui, Y. Shao, and Y. Tong, "Tornado: A system for real-time iterative analysis over evolving data," in *Proc. Int. Conf. Manag. Data*, Jun. 2016, pp. 417–430.

[26] B. Erb, D. Meißner, J. Pietron, and F. Kargl, "Chronograph: A distributed processing platform for online and batch computations on event-sourced graphs," in *Proc. 11th ACM Int. Conf. Distrib. Event-Based Syst.*, Jun. 2017, pp. 78–87.

[27] M. Mariappan and K. Vora, "GraphBolt: Dependency-driven synchronous processing of streaming graphs," in *Proc. 14th EuroSys Conf.*, Mar. 2019, pp. 1–16.

[28] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," *VLDB J.*, vol. 33, no. 2, pp. 507–541, Mar. 2024.

[29] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 1–62, 2012.

[30] H. N. Chaudhry, "FlowGraph: Distributed temporal pattern detection over dynamically evolving graphs," in *Proc. 13th ACM Int. Conf. Distrib. Event-Based Syst.*, Jun. 2019, pp. 272–275.

[31] P. Daverio, H. N. Chaudhry, A. Margara, and M. Rossi, "Temporal pattern recognition in graph data structures," in *Proc. IEEE Int. Conf. Big Data*, Dec. 2021, pp. 2753–2763.

[32] T. A. Ayall, H. Liu, C. Zhou, A. M. Seid, F. B. Gereme, H. N. Abishu, and Y. H. Yacob, "Graph computing systems and partitioning techniques: A survey," *IEEE Access*, vol. 10, pp. 118523–118550, 2022.

[33] M. V. der Boor, S. C. Borst, J. S. H. Van Leeuwaarden, and D. Mukherjee, "Scalable load balancing in networked systems: A survey of recent advances," *SIAM Rev.*, vol. 64, no. 3, pp. 554–622, Aug. 2022.

[34] S. Maneth and F. Peternek, "A survey on methods and systems for graph compression," 2015, *arXiv:1504.00616*.

[35] M. Besta and T. Hoefler, "Survey and taxonomy of lossless graph compression and space-efficient graph representations," 2018, *arXiv:1806.01799*.

[36] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on GPUs: A survey," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 1–35, Nov. 2018.

[37] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative API for real-time applications in apache spark," in *Proc. Int. Conf. Manag. Data*, May 2018, pp. 601–613.

[38] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing, "Trill: A high-performance incremental query processor for diverse analytics," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 401–412, Dec. 2014.

[39] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "StreamBox: Modern stream processing on a multicore machine," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 617–629.

[40] P. M. Grulich, B. Sebastian, S. Zeuch, J. Traub, J. V. Bleichert, Z. Chen, T. Rabl, and V. Markl, "Grizzly: Efficient stream processing through adaptive query compilation," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, Jun. 2020, pp. 2487–2503.

[41] G. Theodorakis, A. Koliousis, P. Pietzuch, and H. Pirk, "LightSaber: Efficient window aggregation on multi-core processors," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, Jun. 2020, pp. 2505–2521.

[42] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proc. VLDB Endowment*, vol. 4, no. 9, pp. 539–550, Jun. 2011.

[43] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl, "Analyzing efficient stream processing on modern hardware," *Proc. VLDB Endowment*, vol. 12, no. 5, pp. 516–530, Jan. 2019.

[44] P. Carbone et al., "Apache flink: Stream and batch processing in a single engine," *Bull. Tech. Committee Data Eng.*, vol. 4, p. 38, 2015.

[45] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "StreamBox-HBM: Stream analytics on high bandwidth hybrid memory," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 167–181.

[46] M. Khan and M. Khan, "Exploring query optimization techniques in relational databases," *Int. J. Database Theory Appl.*, vol. 6, no. 3, pp. 11–20, 2013.

[47] H. Herodotou, N. Borisov, and S. Babu, "Query optimization techniques for partitioned tables," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, Jun. 2011, pp. 49–60.

[48] J. Kossmann, T. Papenbrock, and F. Naumann, "Data dependencies for query optimization: A survey," *VLDB J.*, vol. 31, no. 1, pp. 1–22, Jan. 2022.

[49] S. Maesaroh, H. Gunawan, A. Lestari, M. S. A. Tsaurie, and M. Fauji, "Query optimization in MySQL database using index," *Int. J. Cyber IT Service Manag.*, vol. 2, no. 2, pp. 104–110, Mar. 2022.

[50] M. Ridani and M. Amnai, "Query optimization using indexation techniques in datawarehouse: Survey and use cases," in *Proc. Int. Conf. Artif. Intell. Smart Environ.* Cham, Switzerland: Springer, 2023.

[51] P. L. Bajaj, "A survey on query performance optimization by index recommendation," *Int. J. Comput. Appl.*, vol. 113, no. 19, pp. 36–40, Mar. 2015.

[52] D. Li, L. Han, and Y. Ding, "SQL query optimization methods of relational database system," in *Proc. 2nd Int. Conf. Comput. Eng. Appl.*, vol. 1, Mar. 2010, pp. 557–560.

[53] T. Shioi and K. Hatano, "Rule- and cost-based optimization of OLAP workloads on distributed RDBMS with column-oriented storage function," in *Proc. IEEE 4th Int. Conf. Future Internet Things Cloud Workshops (FiCloudW)*, Aug. 2016, pp. 165–170.

[54] H. Lan, Z. Bao, and Y. Peng, "A survey on advancing the DBMS query optimizer: Cardinality estimation, cost model, and plan enumeration," *Data Sci. Eng.*, vol. 6, no. 1, pp. 86–101, Mar. 2021.

[55] S. Kumar et al., "Cost-based query optimization with heuristics," *Int. J. Sci. Eng. Res.*, vol. 2, no. 9, p. 102, 2011.

[56] S. Breß et al., "A framework for cost based optimization of hybrid CPU/GPU query plans in database systems," *Control Cybern.*, vol. 41, no. 4, p. 941, 2012.

[57] C. Forresi, M. Francia, E. Gallinucci, and M. Golfarelli, "Cost-based optimization of multistore query plans," *Inf. Syst. Frontiers*, vol. 25, no. 5, pp. 1925–1951, Oct. 2023.

[58] V. K. Myalapalli and A. S. N. Chakravarthy, "Revamping SQL queries for cost based optimization," in *Proc. Int. Conf. Circuits, Controls, Commun. Comput. (IC)*, Oct. 2016, pp. 1–6.

[59] D. Khanna et al., "Performance analysis for select, project and join operations of Oracle, My-SQL and Microsoft access DBMSS," *Int. J. Comput. Eng. Technol. (IJCET)*, vol. 1, no. 1, p. 140, 2018.

[60] M. Sharma and G. Singh, "Analysis of joins and semi-joins in centralized and distributed database queries," in *Proc. Int. Conf. Comput. Sci.*, Sep. 2012, pp. 15–20.

[61] A. K. Z. Al Saedi, R. B. Ghazali, and M. B. M. Deris, "An efficient multi join query optimization for DBMS using swarm intelligent approach," in *Proc. 4th World Congr. Inf. Commun. Technol. (WICT)*, Dec. 2014, pp. 113–117.

[62] S. Sumathi and S. Esakkirajan, "Transaction processing and query optimization," in *Fundamentals of Relational Database Management Systems*, vol. 2, no. 3, Springer, 2017, pp. 319–352.

[63] Y. Xu and P. Kostamaa, "Efficient outer join data skew handling in parallel DBMS," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1390–1396, Aug. 2009.

[64] G. Mahajan, "Query optimization in DDBS," *Int. J. Comput. Appl. Inf. Technol. (IJCAIT)*, vol. 1, no. 1, p. 94, 2012.

[65] J. Duggan, O. Papaemmanouil, L. Battle, and M. Stonebraker, "Skew-aware join optimization for array databases," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, May 2015, pp. 123–135.

[66] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann, "Adopting worst-case optimal joins in relational database systems," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 1891–1904, Aug. 2020.

[67] C. Lee, C.-H. Ke, J.-B. Chang, and Y.-H. Chen, "Minimization of resource consumption for multidatabase query optimization," in *Proc. 3rd IFCIS Int. Conf. Cooperat. Inf. Syst.*, 1998, pp. 241–250.

[68] A. Y. Levy, I. S. Mumick, and Y. Sagiv, "Query optimization by predicate move-around," in *Proc. VLDB*, vol. 94, 1994, pp. 12–15.

[69] C. Yan, Y. Lin, and Y. He, "Predicate pushdown for data science pipelines," *Proc. ACM Manag. Data*, vol. 1, no. 2, pp. 1–28, Jun. 2023.

[70] Y. Yang, M. Youill, M. Woicik, Y. Liu, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker, "FlexPushdownDB: Hybrid pushdown and caching in a cloud DBMS," *Proc. VLDB Endowment*, vol. 14, no. 11, pp. 2101–2113, Jul. 2021.

[71] S. Kandula, L. Orr, and S. Chaudhuri, "Data-induced predicates for sideways information passing in query optimizers," *VLDB J.*, vol. 31, no. 6, pp. 1263–1290, Nov. 2022.
[72] F. Kastrati and G. Moerkotte, "Optimization of disjunctive predicates for main memory column stores," in *Proc. ACM Int. Conf. Manag. Data*, May 2017, pp. 731–744.
[73] S. Kandula, L. Orr, and S. Chaudhuri, "Pushing data-induced predicates through joins in big-data clusters," *Proc. VLDB Endowment*, vol. 13, no. 3, pp. 252–265, Nov. 2019.
[74] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi, "Query optimization for massively parallel data processing," in *Proc. 2nd ACM Symp. Cloud Comput.*, Oct. 2011, pp. 1–13.
[75] M. T. Özsu and P. Valduriez, "Distributed and parallel database systems," *ACM Comput. Surv. (CSUR)*, vol. 28, no. 1, pp. 125–128, 1996.
[76] D. Subramanian and K. Subramanian, "Query optimization in multi-database systems," *Distrib. Parallel Databases*, vol. 6, pp. 183–210, Apr. 1998.
[77] R. Nehme and N. Bruno, "Automated partitioning design in parallel database systems," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, Jun. 2011, pp. 1137–1148.
[78] T. Eavis and A. Taleb, "Query optimization and execution in a parallel analytics DBMS," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 897–908.
[79] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query optimization for parallel execution," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, Jun. 1992, pp. 9–18.
[80] J. Unbehauen, C. Stadler, and S. Auer, "Optimizing SPARQL-to-SQL rewriting," in *Proc. Int. Conf. Inf. Integr. Web-Based Appl. Services*, Dec. 2013, pp. 324–330.
[81] D. Hernández, L. Galárraga, and K. Hose, "Computing how-provenance for SPARQL queries via query rewriting," *Proc. VLDB Endowment*, vol. 14, no. 13, pp. 3389–3401, Sep. 2021.
[82] G. Correndo, M. Salvadores, I. Millard, H. Glaser, and N. Shadbolt, "SPARQL query rewriting for implementing data integration over linked data," in *Proc. EDBT/ICDT Workshops*, Mar. 2010, pp. 1–11.
[83] X. Jian, Y. Wang, X. Lei, L. Zheng, and L. Chen, "SPARQL rewriting: Towards desired results," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, Jun. 2020, pp. 1979–1993.
[84] W. Le, A. Kementsietsidis, S. Duan, and F. Li, "Scalable multi-query optimization for SPARQL," in *Proc. IEEE 28th Int. Conf. Data Eng.*, Apr. 2012, pp. 666–677.
[85] S. Magliacane, A. Bozzon, and E. Della Valle, "Efficient execution of top-k SPARQL queries," in *Proc. Int. Semantic Web Conf.* Berlin, Germany: Springer, 2012.
[86] D. Ibragimov, K. Hose, T. B. Pedersen, and E. Zimányi, "Optimizing aggregate SPARQL queries using materialized RDF views," in *Proc. 15th Int. Semantic Web Conf. (ISWC)*. Kobe, Japan: Springer, Oct. 2016.
[87] G. Troullinou, H. Kondylakis, M. Lissandrini, and D. Mottin, "SOFOS: Demonstrating the challenges of materialized view selection on knowledge graphs," in *Proc. Int. Conf. Manag. Data*, Jun. 2021, pp. 2789–2793.
[88] M. Schmidt, M. Meier, and G. Lausen, "Foundations of SPARQL query optimization," in *Proc. 13th Int. Conf. Database Theory*, Mar. 2010, pp. 4–33.
[89] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi, "A distributed algorithm for large-scale graph partitioning," *ACM Trans. Auto. Adapt. Syst.*, vol. 10, no. 2, pp. 1–24, Jun. 2015.
[90] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent Advances in Graph Partitioning*. Berlin, Germany: Springer, 2016.
[91] H. W. Y. Adoni, T. Nahhal, M. Krichen, B. Aghezzaf, and A. Elbyed, "A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems," *Distrib. Parallel Databases*, vol. 38, no. 2, pp. 495–530, Jun. 2020.
[92] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, Apr. 2013, pp. 169–182.
[93] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proc. 24th Int. Conf. World Wide Web*, May 2015, pp. 1307–1317.
[94] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "GraphGrind: Addressing load imbalance of graph partitioning," in *Proc. Int. Conf. Supercomputing*, Jun. 2017, pp. 1–10.
[95] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "PaGraph: Scaling GNN training on large graphs via computation-aware caching," in *Proc. 11th ACM Symp. Cloud Comput.*, Oct. 2020, pp. 401–415.
[96] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *Proc. IEEE Int. Conf. Big Data*, Dec. 2017, pp. 293–302.
[97] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo, "The more the merrier: Efficient multi-source graph traversal," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 449–460, Dec. 2014.
[98] H. Cao, Y. Wang, H. Wang, H. Lin, Z. Ma, W. Yin, and W. Chen, "Scaling graph traversal to 281 trillion edges with 40 million cores," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Apr. 2022, pp. 234–245.
[99] F. Checconi and F. Petrini, "Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 425–434.
[100] O. Batarfi, R. E. Shawi, A. G. Fayoumi, R. Nouri, S.-M.-R. Beheshti, A. Barnawi, and S. Sakr, "Large scale graph processing systems: Survey and an experimental evaluation," *Cluster Comput.*, vol. 18, no. 3, pp. 1189–1213, Sep. 2015.
[101] H. N. Chaudhry. (2022). *Flowgraph Distributed Engine*. [Online]. Available: https://github.com/HassanNazeerChaudhry/flowgraphnew
[102] H. N. Chaudhry. (2023). *Flowgraph Optimizer*. [Online]. Available: https://github.com/HassanNazeerChaudhry/FlowGraphOptimizer

HASSAN NAZEER CHAUDHRY received the B.Sc. degree in computer engineering and the M.Sc. degree in electrical and ICE from the University Engineering and Technology at Taxila, Pakistan, in 2006 and 2009, respectively, and the M.Sc. degree in electrical and ICE from Technische Universität Darmstadt, Germany, in 2015. He is currently pursuing the Ph.D. degree with Politecnico di Milano, Italy. Before this, he worked at several international companies, including the Center of Excellence for ASIC Design and DSP, UET Taxila, Pakistan, Siemens, Germany, MIMO On GmbH, Germany, Fiat, Italy, Basic Net, Italy, Algo Systems S.a.S., Italy. His current topic of research is large-scale graph processing.

MATTEO ROSSI is currently an Associate Professor with Politecnico di Milano. His research interests include formal methods for safety-critical and real-time systems, architectures for real-time distributed systems, and transportation systems both from their design and application in urban mobility scenarios.

Open Access funding provided by 'Politecnico di Milano' within the CRUI CARE Agreement