Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Web Engineering Group

A Study of Execution Strategies for openCypher on Apache Flink

Master Thesis

Mengqi Yang

Supervisor:
dr. George Fletcher, Eindhoven University of Technology

External Supervisors:
Alex Averbuch, Neo Technology
Vasiliki Kalavri, Royal Institute of Technology
prof. Alexandra Poulovassilis, Birkbeck University of London

Committee Members:
dr. George Fletcher, Eindhoven University of Technology
dr. ir. Joaquin Vanschoren, Eindhoven University of Technology
dr. Michel Westenberg, Eindhoven University of Technology

Eindhoven, July 2016

Abstract

The concept of big data has become popular in recent years due to the growing demand of handling datasets of large sizes. A lot of new frameworks have been proposed to deal with the problem of processing, analysis and storage of big data. As one of them, Apache Flink is an open source platform allowing for distributed stream and batch data processing. Cypher, a declarative query language developed by Neo4j, shows its significant strengths when it encounters graph database compared to traditional relational query languages. However, by now the newest released version of Cypher still has limitations on its data parallelism and scalability.

In this thesis, we first formulate an algebra for expressing Cypher queries. Then we describe the implementation of basic operators of Cypher algebra on Flink that allows the representation of a subset of Cypher queries. To our knowledge, there is no previous work in this field. Based on our observations, a cost-based query optimizer and a rule-based query optimizer are proposed to build efficient query plans. We compare the performance of these two query optimizers and finally analyze the advantages and disadvantages of these two optimization strategies.

Preface

This thesis is about my master project of six months' work at the Eindhoven University of Technology in collaboration with Neo Technology, under the supervision of dr. George Fletcher.

Firstly, I would like to thank my supervisor dr. George Fletcher, for providing me with the fascinating topic when I first talked about my intentions about my master thesis. More importantly, the patient guidance, the new ideas and directions offered by dr. George Fletcher during the project all have inspired me a lot and become my invaluable experience.

I would also like to thank all my external supervisors, dr. Alex Averbuch from Neo Technology, Vasiliki Kalavri from Royal Institute of Technology and prof. Alexandra Poulovassilis from Birkbeck University of London. Dr. Alex Averbuch assisted me in diving into Cypher and invited me to a nice trip to the HQ of Neo Technology in Malmö. Dr. Vasiliki Kalavri, who is a contributor of Apache Flink, offered me a lot of help with Apache Flink. The immediate feedback and valuable advice from prof. Alexandra Poulovassilis always helped me refine my work.

Throughout the period of my thesis, all my supervisors have been giving remarkable insights and suggestions both to this project as well as the thesis. Your devotions to this thesis are indispensable.

Finally, I am grateful to my parents, Tao Yang and Yongxin Dai, for their support and encouragement to all my decisions. Also, a special thanks to all my friends for their company and help, especially to Qi Xiao for his technical and emotional support, and to Xi Yu for her long lasting friendship.

I have suffered a lot as well as gained a lot in these six months. I hope you enjoy reading my thesis.

Mengqi Yang

Eindhoven, July 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Definition

Cypher is a declarative query language developed by Neo4j [1], which provides powerful and expressive pattern matching for a graph database. Tailored for the property graph data model, it shows significant strengths over traditional query languages when handling graph databases.

However, by now the newest released version of Cypher is still limited in terms of its data parallelism and scalability. An implementation of Cypher on top of a distributed execution engine could be one of the possible solutions to overcome these limits. Apache Flink [2] is an open source platform allowing for distributed computation and batch data processing, aiming to support data processing with all types of data.

In this thesis, we describe our implementation of Cypher on top of Apache Flink. To implement this system, we first design the logical query algebra to present logical representation of basic operators. Afterwards, we develop an implementation of the Cypher language on top of Flink, which involves researching at which abstraction level to integrate Cypher on Flink, which existing Flink operators can be mapped to Cypher constructs, and whether or not new Cypher or Flink operators need to be created. Additionally, we investigate the data models of Flink and Cypher. The differences between them are identified and mappings between them are proposed. Also, the following problems need to be researched and addressed:

1. Which basic operators and clauses of Cypher should be contained in this implementation?

2. By mapping Cypher on Apache Flink, what kind of optimization strategies could we apply according to properties of Apache Flink? Also, what are the potential strengths and shortcomings of these optimization strategies?

3. How could we evaluate the performance of our implementation of Cypher? How should we conduct these experiments? For example, What kind of database should we use to proceed our evaluations?

1.2 Contributions

Our contributions of this thesis consist of the following aspects: we proposed our own algebra to present Cypher queries comprised of graph-specific operators. All Cypher queries could be easily translated to basic operators in Cypher algebra. We implemented a subset of Cypher on Apache Flink system. We also proposed two query optimizers, a cost-based query optimizer and a rule-based query optimizer to build query plans by using the basic operators implemented. Finally, we conducted experiments to compare the performance of these two optimization strategies.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 describes the background of our work. We first make a brief survey on related works, including different graph query languages with the feature of pattern matching, algebras formulated by others for Cypher and other implementations of graph query languages on big data frameworks. We then describe the technologies on which our project is based on. We introduce Cypher, the query language that we are interested in investigating in this thesis, Flink, the big data framework on which our implementation is based, and LDBC-SNB and gMark, the two benchmarks utilized in our experiments.

Then in Chapter 3 we propose our own Cypher algebra (CA) and then describe the approach to translate the pattern matching part of all Cypher queries into basic operators in CA.

Chapter 4 begins with detailed descriptions about graph representation and the algorithms of our basic operators in our implementation on Flink. This chapter also details the procedure of constructing a logical query plan for a query. Based on our understandings of Cypher and Flink and observations of the datasets, we propose several hypotheses about the construction of query plans. Then we introduce two different query optimization strategies basing on such hypotheses that are implemented in our system.

Chapter 5 presents the experimental design for our implementation, showing the queries and the datasets we are going to use. Chapter 6 details the experiments conducted by us. We show the experimental results from different queries, datasets and levels of parallelism of the Flink system, and then evaluate and analyze these results.

At last, Chapter 7 summarizes the contributions and the limitations of this thesis.

Chapter 2

Background

This chapter describes research background, relevant concepts and definitions on which this thesis is based. Section 2.1 introduces up-to-date graph and graph-like query languages applied widely for management, processing and analysis of graph data. Section 2.2 begins with an introduction to graph processing systems based on existing Big Data frameworks. Then a brief survey of related work about implementations, analysis and various optimization strategies of graph and graph-like query languages on top of Big Data execution engines is presented. Section 2.3 introduces the data model that we will investigate in this project. Section 2.4 describes Cypher, including its syntax and interesting features. Section 2.5 focuses on Apache Flink, the Big Data framework used in our project. Section 2.7 and Section 2.8 give a brief introduction to LDBC benchmark and gMark respectively, both of which are used to generate the datasets used for the experiments in this work.

2.1 Query Languages for Graph Databases

Graphs are expressive and powerful representations of connected data [15]. A graph consists of a set of vertices and a set of edges. Traditional relational databases can also be used to represent connected data, but since they need to utilize foreign keys and join operations to connect tables, the computation complexity may rise remarkably in connection-intensive databases compared to graph databases. In recent decades, the popularity of graph database has given rise to the appearances of various graph or graph-like query languages.

2.1.1 SPARQL

RDF, namely Resource Description Framework, is a standard model for data interchange on the Web [3]. The building block of RDF is "triple". A triple denotes an expression in the form of subject-predicate-object, which is used to describe the relationship between two resources (subject and object). A collection of triples form a edge-labeled, directed graph.

Due to the similarity between a RDF data model and a graph data model, a semantic query language named SPARQL is proposed for retrieving and manipulating the data stored in RDF format. SPARQL is based on graph pattern matching [4]. Generally, SPARQL consists of three parts, namely the pattern matching part together with various features of pattern matching of graphs, the solution modifiers, and the output of different types [14]. An example of SPARQL is shown as follows:

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT  ?name
WHERE   { ?x foaf:name ?name }
ORDER BY ?name
```

The triple (?x, foaf: name, ?$name$) is considered as the graph pattern of the example query, and the clause "ORDER BY" functions as a solution modifier to reorganize the output order of the results. Besides, selected values of the variables which match the patterns constitute the output part of the SPARQL query. Particularly, SPARQL is suitable to handle situations where the relationships are crucial part to answer queries.

However, SPARQL also has limitations on presenting property graphs since additional nodes are needed to represent the properties of vertices and edges [11].

2.1.2 Cypher

Cypher is a declarative graph query language developed by Neo4j, which provides powerful and expressive pattern matching for graph database [1]. Unlike SPARQL, Cypher language is tailored for the labeled property graph data model, which will be introduced later. A query concerning the properties of a edge or a vertex could be more easily phrased in Cypher than in SPARQL. An example of Cypher query looks like:

```
MATCH (a: {name: "Bob"}) - [:follows] -> (b)
RETURN a
```

The statement in the "MATCH" clause defines a pattern consisting of two vertices a and b linked by an edge labeled "follows". A filtering criteria on the property of a, "the name of vertex a should be Bob", could be naturally contained in the pattern.

Cypher also has some common features with SPARQL. Similarly, the syntax of Cypher provides a convenient approach to match patterns of vertices and edges in the graph. A detailed description of Cypher is presented in Chapter 3.

2.1.3 Others

Other query languages also exist for handling queries applied to graph databases. UnQL, Unstructured Data Query Language, is a functional query language for NoSQL databases in order to process semi-structured data based on structural recursion. SoSQL, BiQL and SNQL are query languages for social networks. A more detailed introduction with analysis of query languages for graph databases could be found in [22].

2.2 Literature Review

2.2.1 Algebra for Cypher

An algebra for Cypher has been developed in [11]. This algebra builds upon the data model of a *property graph* where each node and edge may be associated with a label and a set of properties. The concept of *graph relations* is introduced and each row in a graph relation represents a subgraph. The inputs and outputs of the algebra are all graph relations; this has the advantage that operators on relations can be reused directly instead of being duplicated in the algebra.

The algebra defines two operators:

1. GetNodes returns a relation containing all nodes in a sole column x, written as $\bigcirc_x$;
2. Expand finds all neighbors of a given column x and stores them in a new column y, written as either $\uparrow_x^y$ or $\downarrow_x^y$, depending on whether in-neighbors are out-neighbors are to be found.

Graph queries are implemented in a step-to-step manner. For instance, the following Cypher query:

```
MATCH (x) -[]-> (y)
WHERE x.name = "foo"
RETURN y
```

may be expressed in the algebra as $\pi_y(\uparrow_x^y (\sigma_{x.name="foo"}(\bigcirc_x)))$. As seen in the example, since the algebra uses normal relations for input and output, ordinary relation operators like σ and π may be reused, where σ represents the selection operator and π represents the projection operator.

Another algebra, developed in [10], uses different notations but is essentially equivalent to the one introduced above.

2.2.2 Existing Graph Query Systems on Big Data Frameworks

Since graph databases are capable of expressing very large and complex relationships, it is sometimes necessary to store and process the data in a distributed fashion. There have been several efforts of supporting graph queries using big data frameworks.

In [12], a framework for storing and retrieving large RDF graph using Hadoop and MapReduce is developed. It also contains an algorithm for answering SPARQL queries. It shows the feasibility of using distributed big data frameworks to store and query graph databases and the scalability of such a system.

Another system called Gradoop is described to support graph queries on Flink in [13]. It is designed around the so-called Extended Property Graph Data Model (EPGM) supporting semantically rich, schema-free graph data within many distinct graphs. A domain-specific language to define analytical workflows is developed.

In [17], a system called S2RDF is developed to facilitate RDF querying with SPARQL on Spark. The major challenge is that SPARQL queries don't perform well on existing big data infrastructures; existing approaches often favor certain query pattern shape while performance drops significantly for other shapes. To address this problem, a novel relational partitioning schema for RDF data called ExtVP that uses a semi-join based preprocessing, akin to the concept of Join Indices in relational databases, was developed, to efficiently minimize query input size regardless of its pattern shape and diameter.

A system called Lighthouse is described in [9], which supports graph queries on the Apache Giraph framework. The system supported a subset of the Cypher query language and is able to scale to data sets of 16 million entities. The Lighthouse system is extended to support shortest path queries in [16]. It proposes an extension to the Cypher query language and evaluates several different algorithms for such queries.

The S2X system, described in [7], also supports SPARQL queries on Spark, and is the first approach to combine graph-parallel and data-parallel computation for SPARQL querying of RDF data based on Hadoop.

Whilst other systems utilize either the Cypher or SPARQL query language, there is also interest for the Datalog query language. A system called BigDatalog is described in [18], which utilizes Spark as the big data framework, and proves scalable and effective in answering Datalog queries.

To our knowledge, there has been no study of Cypher on Apache Flink.

2.3 Data Model

2.3.1 Labeled Property Graph Data Model

Labeled property graph data model is used to represent our graph data. Basically, a labeled property graph data model is comprised of nodes, relationships, properties and labels. An example is shown in Figure 2.1. Basic components and features of the labeled property graph data model are explained below:

Vertex A vertex could contain zero or more properties (key-value pairs) and be tagged with one or multiple labels. In Figure 2.1, all ellipses represent vertices in the graph instance.

Edge Edges are used to connect nodes in this data model. An edge can also have zero or more properties and be tagged with one label. In most cases, an edge has a direction. All directed arrows in Figure 2.1 indicate edges in the model.

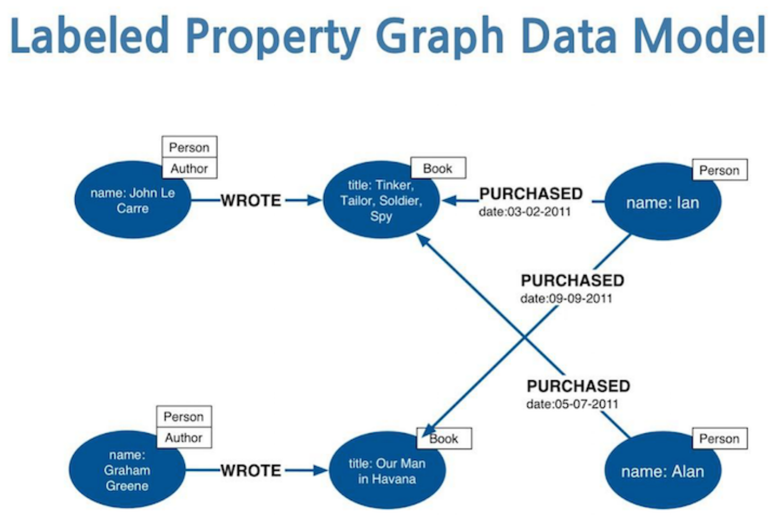


Figure 2.1: A labeled property graph data model [3]

Property Properties are usually expressed as key-value pairs in the property graph data model, which describe attributes of vertices and edges. The key here refers to a feature owned by a vertex or an edge, and the corresponding value in the pair shows the status of the feature. For example, a vertex which represents a user might have the property "age = 18", of which the key of this property is "age", and the value is 18. In the Cypher language, the keys are always strings, while the values can be strings or other primitive data types of Java. In Figure 2.1, the vertex in the upper left corner has a property with "name" as the key and "John Le Carre" as the value.

Label A label marks a specific type of a vertex or an edge. Labels are also represented by strings in Cypher. In this example, "Person" and "Author" are both labels of the vertex in the top left corner.

Labeled property graphs are often stored as sets of vertices and edges, with properties and labels attached to individual vertices and edges.

2.4 Cypher

Neo4j is a world leading graph database, which focuses on retrieving, processing and managing graph data from graph databases and supports the declarative graph query language named Cypher. The syntax of Cypher is simple, expressive and easy to use.

As we have already indicated, the most useful technique of this query language is to employ pattern matching between a query and similar patterns existing in graphs. Here we introduce the syntaxes and corresponding properties of Cypher.

2.4.1 Cypher Clauses and Operators

Table 2.1 and 2.2 show widely used clauses and operators in the Cypher language. Note that the clauses contained in Cypher are quite similar to those of SPARQL. Both of them are straightforward and understandable.

Generally, there are three types of clauses, namely *general clauses*, *reading clauses*, and *writing clauses*. It is noticeable that instead of using clause "SELECT" as in SPARQL, Cypher uses the clause named "MATCH" to emphasize its pattern matching feature. The most common clauses in a Cypher query are "MATCH", "WHERE" and "RETURN". A "MATCH" clause usually specifies

Clause Type	**Clauses**
General Clauses	RETURN, ORDER BY, LIMIT, SKIP, WITH, UNWIND, UNION, USING
Reading Clauses	MATCH, OPTIONAL MATCH, WHERE, LOAD CSV
Writing Clauses	CREATE, CREATE UNIQUE, MERGE, SET, DELETE, REMOVE, FOREACH

Table 2.1: Cypher clauses

Operator Type	**Operators**
Mathematical operators	$+$, $-$, $\star$, $/$, $\%$
Boolean operators	AND, OR, NOT, XOR
Comparison operators	$=$, $<$, $>$, $<=$, $<=$, $<>$
String operators	$+$

Table 2.2: Cypher operators

a basic query graph pattern. A "WHERE" clause can not only be utilized to further expresses the query graph pattern, but also to indicates the properties or labels of vertices and edges. A "RETURN" clause is always required in a Cypher query to return the demanded results. The operators shown in Table 2.2 can be used in a Cypher query to impose requirements on properties or labels of graph components.

2.4.2 Pattern Matching

In this project, we will mainly focus on the "pattern matching" syntax of Cypher since it is the most important and distinctive property of Cypher. Table 2.3 shows the symbols and corresponding explanations around graph components in Cypher. Here the user-defined variables are denoted as v, the label names as lb, the property keys as pk, and the property values as pv. Also, note that $*$ symbols contribute a lot to the simplicity of Cypher. The presence of $*$ with values or variables assigned with values could allow multi-hops between vertices, by which not only are the lengths of repeated edges reduced, but also more paths can be easily determined.

By combining all of the patterns we could generate all the graph query patterns for matching. The procedure of pattern matching is to identify and collect all the subgraphs in the graph database which match specific patterns expressed in queries. Here is an example of Cypher query:

```
MATCH (a: {name: "John"}) - [:follows] -> (b)
RETURN b
```

The results of this query would be a union of all of the vertices which has one or more followers whose names are John.

Symbol	**Explanation**
(v)	Get all vertices in a graph
(v: lb)	Get all vertices tagged with label lb
({ pk: "pv"})	Get all vertices whose property of key pk has value pv
--	Connect related vertices, with no specific directions indicated
--> or <--	Connect related vertices, with specific direction indicated
[v]	Indicate an edge
[:lb]	Indicate an edge tagged with label lb
$*$	Traverse multiple edges

Table 2.3: Cypher patterns

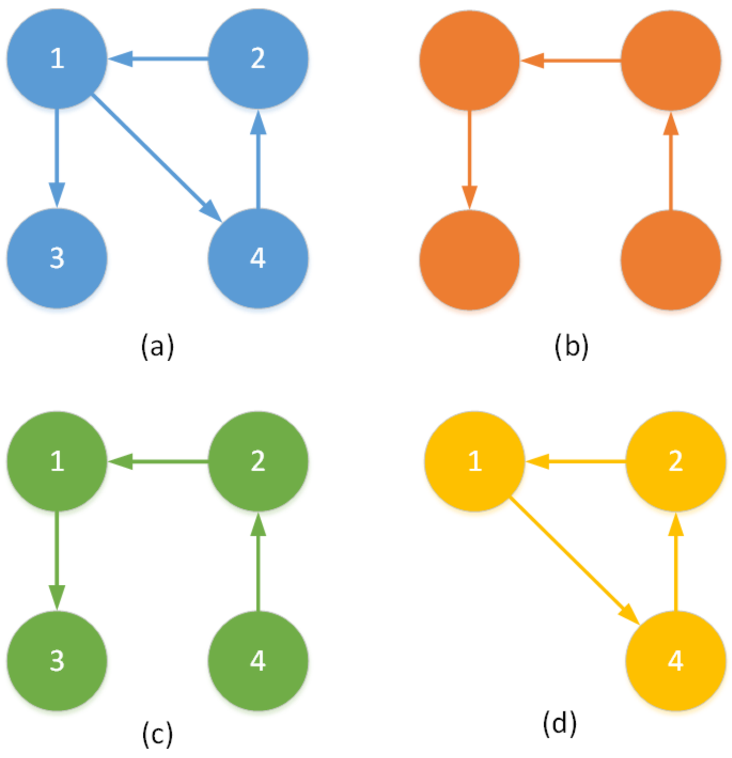


Figure 2.2: (a) A graph instance (b) A query instance
(c) subgraph that satisfies both homomorphism and isomorphism
(d) subgraph that only satisfies homomorphism

2.4.3 Implementation Properties

Subgraph Isomorphisms and Subgraph Homomorphisms

When matching graph patterns with subgraphs, we can use the semantics of either *homomorphism* or *isomorphism*. We first give the definition of isomorphism according to [9], where both the query graph Q and the data graph G are formulated as quadruples of vertices (V and V'), edges (E and E'), labels (L and L') and properties (P and P'):

Definition 2.1 *Given a query graph (pattern graph)* $Q = (V, E, L, P)$ *and a data graph* $G = (V', E', L', P')$ *we define a subgraph isomorphism as being an injective function* $f : V \to V'$ *such that:*

1. $\forall u \in V$, $L(u) \subseteq L'(f(u))$ *and* $P(u) \subseteq P'(f(u))$

2. $\forall (u_i, u_j) \in E$, $(f(u_i), f(u_j)) \in E'$, $L(u_i, u_j) = L'(f(u_i), f(u_j))$ *and* $P(u_i, u_j) \subseteq P'(f(u_i), f(u_j))$

By not requiring the mapping function to be injective, we obtain the definition of graph homomorphism, which is the semantics used in our implementation.

We use an example to illustrate the difference between isomorphism and homomorphism.

In Figure 2.2, (a) is a graph instance, and (b) is the query graph. According to the semantics of both isomorphism and homomorphism, the subgraph in (c) matches the query graph. However, a subgraph homomorphism algorithm will also take Figure 2.2 (d) as a matching subgraph since the matched vertices are not required to be unique. In our implementation, we will assume that all the matching patterns only needs to satisfy criteria of homomorphism, and will consider both (c) and (d) to be valid matches.

Execution Plan

The execution planner is responsible for tuning each query into a execution plan. And the execution plan would choose corresponding operators for Neo4j. Generally, two execution planning strategies are included [1]:

Rule The planner offers rules to generate all the operators. But it does not involve statistical information to facilitate the query compilation.

Cost The planner utilizes statistical information to evaluate all alternative execution plans and select a lowest-cost plan.

2.5 Apache Flink

The concept of Big Data has become popular in recent years for the traditional approaches of processing and storing data has been proved insufficient for handling large datasets. A lot of new Big Data frameworks have been proposed to handle the problem of processing, analysis and storage of big data. As one of the big data engines, Apache Flink is an open source platform allowing for distributed stream and batch data processing [2]. Moreover, traditional databases commonly only store data in well-defined format, which naturally claim a pre-defined schema in the first place. In contrast, the new proposed big data execution engines mostly aim to take all types of data into consideration, including well-structured data, semi-structured data and no-structured data. Apache Flink is also one of them, targeting to support all data types flexibly defined by users.

2.5.1 Architecture

The stack of Apache Flink with three levels is shown in Figure 2.3. The first level consists of all APIs and currently most used libraries, supporting a diversity of data-related programs, e.g. the Table API being a SQL-like expression language for relational stream and batch processing. The second level is the runtime layer which is the critical part of the execution of a program. The lowest level deals with specific execution platforms, including local servers, cluster servers and clouds servers.

A Flink program is usually written against the DataSet API or DataStream API in Flink together with provided libraries. A *JobManager*, which serves as the coordinator during the execution procedure, and several *TaskManagers*, which serve as workers, constitute a Flink system. The program code will be sent to an optimizer to generate a directed acyclic graph (DAG), indicating the dataflow of the program. This graph is called *JobGraph* in Apache Flink, of which the vertices represents the Flink operators for data transformations such as **join** and **map**. Each operator contains relevant properties for the execution, e.g. the defined parallelism.

After receiving a JobGraph, the JobManager will transform it into an *ExecutionGraph* and send it to each TaskManager. The ExecutionGraph can be considered as a parallel version of a JobGraph. Each vertex in the ExecutionGraph denotes a parallel sub-task which will be assigned to a task slot in each TaskManager for execution. Task slots are the parallel resources in each TaskManager, each of which is responsible for executing a pipeline of parallel subtask. The level of parallelism is defined as the number of TaskManagers times the number of task slots in each TaskManager.

The whole execution process of a Flink system is shown in Figure 2.4.

2.5.2 Properties

Apache Flink also has some important properties to support its functionality. The properties are listed below.

1. **Execute programs as data streams**. The Flink system will extract and analyze the operators, data types, and parameters about runtime or execution environment of a program,

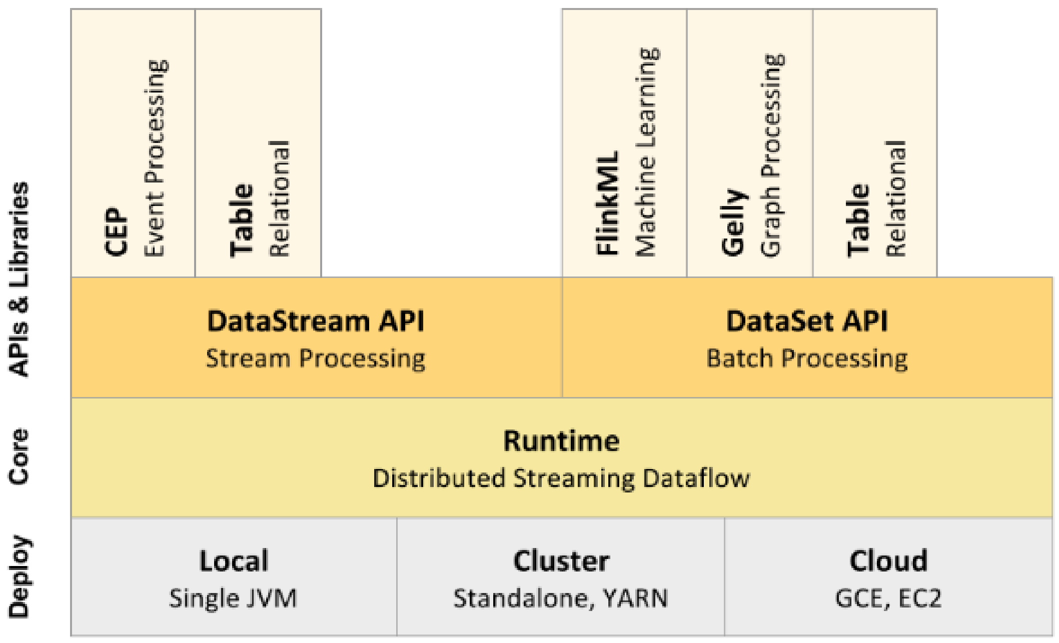


Figure 2.3: Stack of Apache Flink [4]

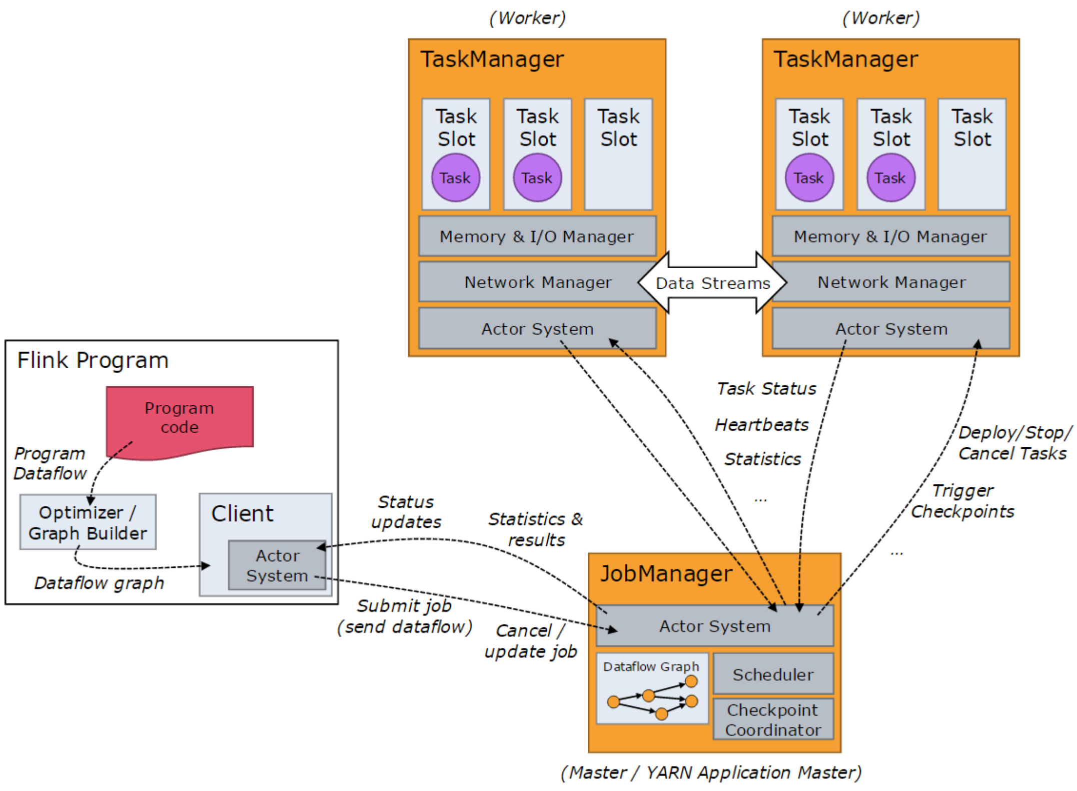


Figure 2.4: Execution process of a Flink system [4]

then build a execution plan to execute it. These data partitioning strategies and efficient pipelines during the execution guarantee the performance of batch data processing programs and distributed computations.

2. **Iterative dataflows are allowed**. By allowing iterative data flows, complicated and continued Map-Reduce functions could be applied to solve problems. This property extend the utilization of Apache Flink.

3. **Alternative execution plans are offered**. For one program, the stream builder in DataStream API or the optimizer in DataSet API of Flink will propose several alternative execution plans. The engine will then select the lowest-cost one according to its analysis, but the choice can also be affected by the user.

2.6 LDBC - SNB

The Linked Data Benchmark Council (LDBC) is an EU authority established to develop benchmarks for the management of graph and RDF data [5]. Nowadays LDBC develops two benchmark, namely the Social Network Benchmark (SNB) and the Semantic Publishing Benchmark (SPB). The main purpose of this benchmark framework is to fairly test and compare graph-like database technologies, including graph query functionality and their performances [8].

LDBC-SNB provides a data generator that can generate a synthetic social network. In general, the data generator simulates all activities of a user in a social network during a period of time. It is noticeable that the design principle of LDBC-SNB is not completely duplicating a real-life social network database, but concentrating on making the benchmark queries exhibit certain desired effects [8].

2.6.1 Data Schema

The datasets generated by LDBC-SNB are based on a data schema, which specifies different entities, their properties and their connected relations of a social network. Each dataset forms a graph that is a fully connected component of persons connected by different types of relations. Some common entities of the LDBC-SNB datasets are *person*, *comment*, *post* and *forum*. The schema also specifies the various relations between entities, which could be considered as the labels of edges in a property graph data model, e.g. *isLocatedIn* and *workAt*. Besides, the properties of entities or the relations are also be integrated in the schema.

We have chosen LDBC-SNB as one of the two benchmarks for generating the datasets to conduct our experiments for two reasons. On one hand, due to its data schema, a LDBC-SNB dataset can be easily transformed into a property graph data model. On the other hand, a social network dataset is a representative application of a graph database, which allows us to evaluate the value of our research in the real world.

2.7 gMark

gMark is another dataset generator used in the experiments of our project. Based on the design principles of gMark, gMark provides schema-driven generation of graphs and queries by utilizing a graph configuration. In term of generated graph instances, node types, edge types, both including proportions to the whole instance, and in- and out-degree distributions could all be defined by the users [6].

Chapter 3

Algebra for Cypher

This chapter describes the Cypher algebra (CA), which consists of nullary, unary and binary operators. In Section 3.1, we introduce the data model which the operators are built on in this algebra. In Section 3.2, we describe the basic operators in detail. Section 3.3 describes how Cypher queries can be formulated in terms of our CA operators.

3.1 Data Model

This query algebra is based on labeled property graph data model. Let the set of all vertex identifiers and the set of edge identifiers be denoted as $\mathcal{V}$ and $\mathcal{E}$ respectively. Also, we denote the set of all labels as $\mathcal{L}$, the set of all property names as $\mathcal{P}$, the set of all property values as Val. The sets $\mathcal{V}$, $\mathcal{E}$, $\mathcal{L}$, $\mathcal{P}$ and Val should be pairwise disjoint. A labeled property graph then consists of the following three datasets:

Definition 3.1 *$G(s,t,e)$ denotes ternary relations of all triplets consisting of identifiers of vertex pairs, a source vertex and a target vertex, with the identifier of the connected edge in between in the input graph, where $s \in \mathcal{V}$, $t \in \mathcal{V}$ and $e \in \mathcal{E}$.*

Definition 3.2 *$L(id,l)$ denotes binary relation of identifiers with related labels, where $id \in \mathcal{V} \cup \mathcal{E}$ and $l \in \mathcal{L}$.*

Definition 3.3 *$P(id,k,v)$ denotes the set that consists of all identifiers and related property keys and values. Here $id \in \mathcal{V} \cup \mathcal{E}$, $k \in \mathcal{P}$ and $v \in Val$.*

In L and P, a given id could be associated with multiple labels and multiple values for any given key k. Figure 3.1 shows an example of a graph database which is also used as the input graph in later examples.

In this graph, $\mathcal{V}$ = {1, 2, 3}, $\mathcal{E}$ = {e1, e2}, $\mathcal{L}$ = {User, Administrator, Likes, Follows}, $\mathcal{P}$ = {name, age} and Val= {John, 34, Bob, 45, Amy, 20}. The three data sets G, L and P are shown in Table 3.1, 3.2 and 3.3 respectively.

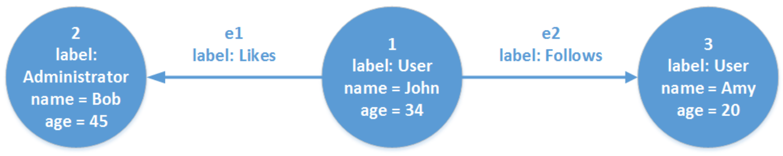


Figure 3.1: An example of the input graph database

Src	*Trg*	*Edge*
1	2	e1
1	3	e2

Table 3.1: Graph table G

ID	*Label*
1	User
2	Administrator
3	User
e1	Likes
e2	Follows

Table 3.2: Label table L

ID	*Key*	*Value*
1	name	John
1	age	34
2	name	Bob
2	age	45
3	name	Amy
3	age	20

Table 3.3: Property table P

3.2 Cypher Algebra

3.2.1 Nullary Operators

Label Matching

First we define a infinite set of graph variables $\mathcal{G}$. Semantically, each graph variable $G_i \in \mathcal{G}$ denotes an instance of G in the preceding data model, which will be used in further examples. Also, $G_i \in CA$, namely Cypher algebra.

The label matching operator, is one operator that selects paths of a variable number of edges through specified labels in the data graph.

Definition 3.4 *If $G_i \in \mathcal{G}$ and $\ell \subseteq \mathcal{L}$,*

$$G_i[\ell]^{lb,ub} \in CA$$

where $0 \leqslant lb \leqslant ub$ or

$$G_i[\ell]^* \in CA.$$

The difference between $G_i[\ell]^{lb,ub}$ and $G_i[\ell]^*$ is that for $G_i[\ell]^{lb,ub}$, the lower bound and upper bound of the variable number of edges could be set explicitly, while for $G_i[\ell]^*$, this variable number will be left unbounded. The schema of $G_i[\ell]^{lb,ub}$ or $G_i[\ell]^*$ is a table containing all pairs in forms of $(G_i.s, G_i.t)$, where $G_i.s$ and $G_i.t$ represent the identifiers of the source vertex and the target vertex respectively on one selected path. A path denotes a sequence of edges which connect a sequence of vertices. Note that we could also include all the edges in a path in this schema but now we first focus on the simpler one.

Example 3.0 First we will give an example of a special query in the form of Q_0.

$$Q_0 \equiv G_1$$

Q_0 indicates that in the absence of transitive closure of G, then each graph variable is semantically a copy of the entire input graph set G.

Example 3.1 Assume the query posed on the previous data example is “find all paths labeled by ‘Likes’ within 1 or 2 steps”, then the query would be

$$Q_1 \equiv G_1[\text{Likes}]^{1,2}$$

According to the semantics of the label matching operator and the schema mentioned above, the results returned by Q_1 are shown in Table 3.4.

$G_1.src$	$G_1.trg$
1	2

Table 3.4: Results returned by Q_1

$G_2.src$	$G_2.trg$	$G_2.edge$
1	3	e2

Table 3.5: Results returned by Q_2

$G_3.src$	$G_3.trg$	$G_3.edge$
1	2	e1

Table 3.6: Results returned by Q_3

3.2.2 Unary Operators

Selection

The selection operator, $\sigma_\theta(Expr)$, is an unary operator that selects either edges or verticess through conditions.

Definition 3.5 *Suppose that there exists an expression Expr ∈ CA, then a selection operator is defined as*

$$\sigma_\theta(\mathit{Expr}) \in \mathit{CA}$$

where θ is a boolean combination of basic predicates of the following type.
*θ ::= pos **COMP** pos*
*| pos **COMP** val*
*| labels(pos) **SETCOMP** labels(pos)*
*| labels(pos) **SETCOMP** ℓ for ℓ ⊆ L′*
*| pos.prop **COMP** pos.prop*
*| pos.prop **COMP** v ∈ Val*

COMP and **SETCOMP** indicate normal comparisons and set comparisons. $labels(pos)$ indicates the set of all labels associated with *pos* in L from our data model. Besides, *pos.prop* is one of the property values associated with positions on properties in P from our data model.

Here $pos \in \{G_i.s, G_i.t, G_i.e\}$, and the types of *val* are restricted to all primitive data types in a programming language such as Int type or Float type.

Example 3.2 Assume another query posed on the previous data example is that “select all edges of users who follow another user whose age is larger than 18”, then the query algebra would be

$$Q_2 \equiv \sigma_{\theta_1}(G_2)$$

where

$$\begin{aligned}\theta_1 \equiv & labels(G_2.e) \supseteq \{\text{Follows}\} \\ & \wedge labels(G_2.s) \supseteq \{\text{User}\} \\ & \wedge labels(G_2.t) \supseteq \{\text{User}\} \\ & \wedge G_2.t.age > 18\end{aligned}$$

The results returned by Q_2 are shown in Table 3.5. Or assume the query is “select all edges from users who likes an administrator whose name is Bob”, the query algebra becomes:

$$Q_3 \equiv \sigma_{\theta_2}(G_3)$$

where

$$\begin{aligned}\theta_2 \equiv & labels(G_3.e) \supseteq \{\text{Likes}\} \\ & \wedge labels(G_3.s) \supseteq \{\text{User}\} \\ & \wedge labels(G_3.t) \supseteq \{\text{Administrator}\} \\ & \wedge G_3.t.name = \text{Bob}\end{aligned}$$

$G_3.src$
1

Table 3.7: Results returned by Q_4

$G_2.src$	$G_2.trg$	$G_2.edge$	$G_3.trg$	$G_3.edge$
1	3	e2	2	e1

Table 3.8: Results returned by Q_5

And the returned results are displayed in Table 3.6.

Projection

The operator Projection $\Pi_{\overline{pos}}(Expr)$ is another unary operator in Cypher algebra that restricts *Expr*to a subset of identifiers.

Definition 3.6 *If there exists an expression Expr $\in$ CA, then a projection operator is defined as*

$$\Pi_{\overline{pos}}(Expr) \in CA$$

where $\overline{pos}$ is a vector of positions which could be empty.

It is noticeable that the projection operator here is different from the one in relational algebra which may cause confusions since the results returned by projection in Cypher query algebra are either vertex identifiers or edge identifiers while the projection in relational algebra could return specific properties of selected tuples in a database. Hence, we could also define a sub-operator value projection though this operator is not part of our Cypher algebra since that the value projection operator is not applied directly on graph instances compared to other operators:

Definition 3.7 *If an expression Expr $\in$ CA, then we can perform a sub-operator value projection*

$$\Pi_{\overline{vp}}(Expr) \notin CA$$

where $\overline{VP}$ is a vector of pos, pos.prop and labels(pos), and here $|\overline{VP}| \geqslant 0$.

Example 3.3 Suppose another query imposed on the results of Q_3 is "collect all the source vertex IDs from Q_3", then we have

$$Q_4 \equiv \Pi_{G_3.s}(\sigma_{\theta_2}(G_3))$$

The results are shown in Table 3.7 .

3.2.3 Binary Operators

Join

Join operator $\bowtie_\theta$ is a binary operator which joins two Cypher algebra expressions by position variables and return the combined results. Here we define the join operator as follows.

Definition 3.8 *Suppose $Expr1, Expr2 \in CA$, then a join operator is defined as*

$$Expr1 \bowtie_\theta Expr2 \in CA$$

where $\theta \equiv \wedge_{i=0}^{n} pos_i^1 = pos_i^2$, and pos_i^1 and pos_i^2 are positions in Expr1 and Expr2 respectively.

Example 3.4 If we apply a query that joins the source vertices returned by Q_2 and Q_3, namely the query is "Join the users who follows another user whose age large than 18 and the users who like an administrator named Bob" then we have

$$Q_5 \equiv \sigma_{\theta_1}(G_2) \bowtie_{\theta_3} \sigma_{\theta_2}(G_3)$$

where

$$\theta_3 \equiv G_2.s = G_3.s$$

Return results of this query are displayed in Table 3.8

$G_2.src$	$G_2.trg$	$G_2.edge$	$G_3.src$	$G_3.trg$	$G_3.edge$
1	3	e2	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	1	2	e1

Table 3.9: Results returned by Q_6

Union

Union operator $\cup$ is another binary operator in Cypher algebra. The operator is defined below.

Definition 3.9 *If* $Expr1, Expr2 \in CA$*, then a union operator is defined as*

$$Expr1 \cup Expr2 \in CA$$

Similarly, other set operators could also be simply defined in this way.

Example 3.5 Suppose the query posed on the input graph is to unite the results of Q_2 and Q_3, namely "get an union of the users who follows another user whose age large than 18 and the users who like an administrator named Bob", hence the query looks like below:

$$Q_6 \equiv \sigma_{\theta_1}(G_2) \cup \sigma_{\theta_2}(G_3)$$

Returned results are shown in Table 3.9. Here $\perp$ is a reserved symbol denoting undefined values, such that
$\perp$ **COMP** $\perp$
$=$ $\perp$ **COMP** v, for $v \in \mathcal{V} \cup \mathcal{E} \cup \mathcal{L} \cup \mathcal{P} \cup Val$
$=$ v **COMP** $\perp$
$=$ **FALSE**

3.3 Query Transformation

All the queries above are still not expressed in the form of Cypher query language. Since the basic operators of Cypher algebra are already defined, here we will present how the Cypher queries could be easily transformed into our proposed Cypher algebra. Though constructions of all the basic operators in Cypher algebra inherits fundamental operators in relational algebra, Cypher algebra still shows its strengths in expressing pattern matching based graph queries for the similarity to relational algebra adds to the simplicity and understandability of Cypher algebra.

It is quite straightforward that one of the basic graph patterns of $CA$is a triple component, consisting of a source vertex, a target vertex and an edge connecting this pair of vertices. While the Cypher queries of more complex graph patterns are involved, this basic graph pattern will be treated as the fundamental component and used to connect with other components to transform a complex graph pattern by utilizing the join operator defined before. The selection operator will be responsible for filtering graph components based on the requests on labels or properties of them. And a projection operator facilitates to retrieve all the returned resulting data by the queries. Besides, the label matching operator helps transform the syntax of "$\star$" symbols in Cypher query language into CA. Followed is a complex Cypher query and then we show how to transform it into an expression in CA:

```
MATCH (a: {name: "John"}) - [:Follows*1...2] -> (b: User) - [:Likes] -> (c: City)
RETURN a
```

Since a type of basic graph pattern in $CA$is the triplet in the form of (src, tar, edge), the first step of transformation is to decompose this query into sub queries until each sub query be regarded as a basic query graph pattern. The graph pattern appeared in "MATCH" clause could be considered as the intersection of the following two graph patterns:

```
Pattern 1 (a: {name: "John"}) - [:Follows*1...2] -> (b: User)
Pattern 2 (b: User) - [:Likes] -> (c: City)
```

After decomposition step of the query, a problem arises: how should we deal with the edge labeled “Follows” since a “⋆” symbol appears there? The problem could be easily solved if we also regard the graph instances, which also contain a source vertex and a target vertex while linked via a variable length of edges tagged with specified labels, as another type of our basic graph pattern. Then we name the graph instance of pattern 1 as G_1, of pattern 2 as G_2. Two sub queries could be stated respectively as:

$$Q_1 \equiv \sigma_{\theta_1}(G_1[\text{Follows}]^{1,2})$$

where

$$\begin{aligned}\theta_1 \equiv & labels(G_1.t) \supseteq \{\text{User}\} \\ & \wedge G_1.s.name = \text{John}\end{aligned}$$

and

$$Q_2 \equiv \sigma_{\theta_2}(G_2)$$

where

$$\begin{aligned}\theta_2 \equiv & labels(G_2.s) \supseteq \{\text{User}\} \\ & \wedge labels(G_2.t) \supseteq \{\text{City}\} \\ & \wedge labels(G_2.e) \supseteq \{\text{Likes}\}\end{aligned}$$

To incorporate these two sub-queries, a join operator is added, then the query becomes:

$$Q_3 \equiv \sigma_{\theta_1}(G_1[\text{Follows}]^{1,2}) \bowtie_\theta \sigma_{\theta_2}(G_2)$$

where

$$\theta \equiv G_1.t = G_2.s$$

Finally we consider the “RETURN” statement. A projection operator is applied correspondingly as follows and the query transformation is finished then.

$$Q \equiv \Pi_{G_1.s}(\sigma_{\theta_1}(G_1[\text{Follows}]^{1,2}) \bowtie_\theta \sigma_{\theta_2}(G_2))$$

Chapter 4

Implementation of Cypher Algebra on Flink

This chapter describes our implementation of Cypher algebra on Flink and query optimization strategies. Section 4.1 begins with the introduction to our design criteria of the implementation of Cypher operators. Then Section 4.2 introduces the implementation of the labeled property graph data model. Section 4.3 details the physical implementation of Cypher algebra. Section 4.4 describes how to translate Cypher queries into basic operators. In Section 4.5, two optimization strategies are proposed for the query plan generation.

4.1 Design Criteria

In general, when we consider the implementation of Cypher Algebra on top of Flink, there are several questions that we need to answer:

- How should we design and implement the schema to represent the labeled property graph data model in Flink system in a efficient way?
- How could we implement the basic CA operators in our framework?
- How could our implementation both be aligned to Cypher Algebra as well as take advantage of the Flink engine?

Based on the above questions, our main design criteria are shown as follows. The whole implementation of Cypher Algebra contains an implementation of graph representations and the implementations of CA operators.

Flink oriented. The features and the advantages of Flink engine should be utilized in our design since the whole implementation framework is built on it.

Simplicity and efficiency. The implementations of both the graph representation and operators should exhibit the simplicity of themselves as well as provides efficient approaches to retrieve, process and store graph data.

Aligned to CA. This whole implementation is based on our CA, so the design also targets at keeping pace with the CA as much as possible when there are no contradictions with the above criteria.

id	*labels*	*properties*
1L	{User}	{name: "John", age: "34"}
2L	{Administrator}	{name: "Bob", age: "45"}
3L	{User}	{name: "Amy", age: "20"}

Table 4.1: The vertex set in Figure 1

id	*srcId*	*tgtId*	*label*	*properties*
e1	1L	2L	{Likes}	{}
e2	1L	3L	{Follows}	{}

Table 4.2: The edge set in Figure 1

0	1	2
1L	e1	2L
1L	e2	3L

Table 4.3: All paths in Figure 1

4.2 Implementation of Graph Representations

A labeled property graph instance consists of an edge set and a vertex set. The labels and properties are stored internally, within edges and vertices. Here we denote the graph instance as G, vertex set as V, edge set as E. Thus, $G = (V, E)$.

In our design, vertices and edges of a graph are stored in a basic data type of Apache Flink, DataSet. Each vertex is a 3-tuple which is comprised of a vertex identifier (Long), a label list (ArrayList) and a property hash map (HashMap). Similarly, each edge is a 5-tuple which is comprised of a edge identifier (String), a source vertex identifier (Long), a target vertex identifier (Long), a label (String) and a property hash map (HashMap). Namely,

$$e = (id, srcId, tgtId, label, properties)$$

$$v = (id, labels, properties)$$

Compared to a traditional graph representation, like the adjacent linked lists, this representation allows for easier data partitions and stream processing. When allowing the parallelism in Flink system, a input dataset might be partitioned and assigned to different task slots or different *TaskManagers*. Each edge and each vertex in our graph representation can be processed and stored independently by the Flink engine such that the data partitions of a graph dataset is flexible. While for adjacent linked lists, the workload of each may vary greatly, which restricts the way that datasets could be partitioned.

Accordingly, the vertex set and the edge set in Figure 3.1 are presented in Table 4.1 and Table 4.2 respectively. Each row of the table represents an element in the set.

The basic data store schema is defined as above. Since we mainly focuses on pattern matching graph queries, the schema of intermediate results generated during the execution process is also considered as part of our graph representation. Here we use the *paths*, a sequence of edges which connect a sequence of vertices, to record our intermediate results which matches the graph query patterns. All resulting paths are stored in a DataSet. Each path here, is represented as an ArrayList comprised of all edge identifiers and vertex identifiers. The reason why we use ArrayList to store the identifiers is that by simply specifying the index of a list the corresponding edge or vertex identifiers could be retrieved. Two paths in Figure 3.1 could be expressed in Table 4.3. Specification 4.1 presents all data types of CA.

Specification 4.1 Data types of Cypher algebra

Graph ⇒ {*vertices*: DataSet<Vertex>, *edges*: DataSet<Edge>}

Vertex ⇒ {*id*: Long, *labels*: HashSet<String>, *properties*: HashMap<String, String>}

Edge ⇒ {*id*: Long, *srcId*: Long, *tgtId*: Long, *label*: String, *properties*: HashMap<String, String>}

Path ⇒ ArrayList<Long>

4.3 Implementation of Operators

4.3.1 Selection Operator: $\sigma_\theta(Expr)$

The selection operator is used to find edges or vertices from earlier results ($Expr$) which satisfy specific requirements of properties or labels (θ). Predicates of a selection operator are combinations of various conditions, each of which is constructed by utilizing comparison or set comparison operators. The structure of condition expressions is detailed in Specification 4.2.

Specification 4.2 Expressions used in selection operator

```
Condition ⇒
        =(CondArgument1 : label, CondArgument2 : label) |
        <(CondArgument1 : prop.value, CondArgument2 : constant) |
        >(CondArgument1 : prop.value, CondArgument2 : constant) |
        =(CondArgument1 : prop.value, CondArgument2 : constant) |
        <=(CondArgument1 : prop.value, CondArgument2 : constant) |
        >=(CondArgument1 : prop.value, CondArgument2 : constant) |
        <>(CondArgument1 : prop.value, CondArgument2 : constant) |
        eq(CondArgument1 : prop.value, CondArgument2 : prop.value)
```

The results of selecting vertices and edges are different. For vertices, the selection operator only filters the vertices specified by the index in each path according to the given conditions. While for edges, the selection operator first will obtain source vertex identifiers, filter all out-coming edges of these vertices by the input conditions and then extract all target vertices of every qualified edge. As a result, when filtering edges, the identifiers of edges themselves as well as the identifiers of target vertices are added to the result, so that further selection on target vertices could be done.

Scan Operator

A scan operator is used to find all the vertices by certain conditions. Generally speaking, a scan operator will scan the vertex set of the entire graph instance and return a collection of ArrayList instances, where each ArrayList only includes one element indicating the selected vertex identifier fulfilling specific filtering conditions. Algorithm 1 shows the implementation of the scan operator. Note that the processing procedure of each vertex in the vertex set would be handled by the Flink engine. Then Flink will partition all the vertex data first, process each partition separately and then combine all results together if the level of parallelism in Flink is determined.

Algorithm 1: Scan Operator for Vertices

```
Input  : cond: Condition, graph: Graph
Output: paths: DataSet<ArrayList>
1 $paths \leftarrow \emptyset$, $vertexIds \leftarrow \emptyset$
2 $startingVertexIds \leftarrow graph.vertices$
3 process $v$ in $startingVertexIds$:
4     if $v$ satisfies $cond$:
5         $vertexIds \leftarrow vertexIds \cup \{\{v.id\}\}$
6 $paths \leftarrow vertexIds$
7 return $paths$
```

Edge-join Operator

An edge-join operator in our implementation is a basic operator to expand the lengths of all resulting paths by two. By taking the preceding paths and the graph vertices as inputs as well as a specific index number in the paths, an edge-join operator would first extract all the vertex IDs to be joined with the edge set in a graph instance. A join operator in Flink then would be applied to combine the previous selected vertices with filtered edges. Meanwhile all target vertices of filtered edges would also be kept in the paths for further selections. The size of the collection of paths could be reduced, extended or remain the same due to this operator as well. It is quite obvious that two more elements would be added to each ArrayList dynamically. A join operator in Flink system is employed here to join source vertex identifiers with edges as well as to filter edges by specified conditions. Algorithm 2 illustrates how the edge-join operator is implemented.

Algorithm 2: Edge-join Operator for Edges

```
Input  : cond: Condition, index: Integer, paths: DataSet<ArrayList>, graph: Graph
Output: updatedPaths: DataSet<ArrayList>
1 startingVertexIds ← paths(index)
2 pattern(srcId, edgeId, tarId) ← ∅
3 process v in startingVertexIds:
4     join e in graph.edges with v:
5         if e satisfies cond:
6             pattern ← pattern ∪ {(v.id, e.id, e.tarId)}
7 upadatedPaths ← paths.update(pattern)
8 return updatedPaths
```

4.3.2 Projection Operator: $\Pi_{\overline{pos}}(Expr)$

A projection operator is used to implement the return statement in Cypher. Through a given index number in resulting paths, a projection operator would match edge identifiers or vertex identifiers with corresponding edges or vertices from the graph and then collect all these components. The implementation of a projection operator is shown in Algorithm 3. The look-up operation in a projection operator is implemented by the join operator in Flink system. Note that *index* in Algorithm 3 corresponds to $\overline{pos}$ in $\Pi_{\overline{pos}}(Expr)$.

Algorithm 3: Projection Operator

```
Input  : index: Integer, paths: DataSet<ArrayList>, graph: Graph
Output: output: DataSet<Vertex> or DataSet<Edge>
1  output ← ∅
2  outputIds ← paths(index)
3  process o in outputIds:
4      if paths(index).type == vertex:
5          join v in graph.vertices with o:
6              output ← output ∪ {{v}}
7      else:
8          join e in graph.edges with o:
9              output ← output ∪ {{e}}
10 return output
```

4.3.3 Join Operator: $Expr1 \bowtie_{\theta} Expr2$

The function of a join operator is to join two paths on a vertex shared by them. Two sets of input paths are given as well as two index numbers for determining the vertex positions in paths, respectively. For each path in one set of the input paths, if the selected vertex is the same as its counterpart in another path in the other set of the input paths, then a merged path will be added to the set of output paths. This join operator is quite similar to the join operator in relational algebra, which binds two tables based on the foreign keys. Algorithm 4 describes how a join operator works.

Algorithm 4: Join Operator

```
Input  : $index1$, $index2$: Integer, $paths1$, $paths2$: DataSet<ArrayList>
Output: $updatedPaths$: DataSet<ArrayList>
1 $updatedPaths \leftarrow \emptyset$
2 process $p1$ in $paths1$:
3     process $p2$ in $paths2$:
4         join $p1$ with $p2$:
5             if $p1(index1) == p2(index2)$:
6                 $p1 \leftarrow p1.append(p2)$
7                 $p1 \leftarrow p1.delete(p2(index2))$
8                 $updatedPaths \leftarrow updatedPaths \cup \{\{p1\}\}$
9 return $updatedPaths$
```

4.3.4 Label Matching Operator: $G_i[\ell]^{lb,ub}$ or $G_i[\ell]^*$

The label matching operator selects paths of a variable number of edges with the specified labels in the data graph. Two types of label matching operators are implemented in our system. One specifies the upper bound and/or the lower bound between the starting vertex and the end vertex. Another type does not specify any bounds but limits the iteration number to a certain value. The basic implementation is shown in Algorithm 5. The implementation of the label matching operator is facilitated by the iteration operators in Flink system. The *ub* variable in Algorithm 5 could be a null value, representing $G_i[\ell]^*$; for $G_i[\ell]^{lb,ub}$, both *lb* and *ub* need to be specified.

4.3.5 Union Operator: $Expr1 \cup Expr2$

The union operator combines two paths of the same arity. The operator takes two sets of paths as input and output the union of these two sets of paths. Note that the judgment of whether the two input paths are of same arity or not should be completed during the procedure of reading queries, which would not be considered in the implementation algorithm. Algorithm 6 shows the implementation of the union operator.

Algorithm 6: Union Operator

```
Input  : $paths1$, $paths2$: DataSet<ArrayList>
Output: $updatedPaths$: DataSet<ArrayList>
1 $updatedPaths \leftarrow path1$.union($path2$)
2 return $upadatedPaths$
```

Algorithm 5: Label Matching Operator

```
Input  : $label$: String, $lb$, $max$: Integer, $ub$: Null | Integer, $index$: Integer, $graph$: Graph,
         $path$: DataSet<ArrayList>
Output: $updatedPaths$: DataSet<ArrayList>
1  $vertexIds \leftarrow paths(index)$, $updatedPaths \leftarrow \emptyset$
2  iterate $lb$ times:
3      process $v$ in $vertexIds$:
4          join $v$ with $e$ in $graph.edges$:
5              if $e.label == label$:
6                  $vertexIds \leftarrow vertexIds.update(v.id)$
7              else:
8                  $vertexIds \leftarrow vertexIds.delete(v.id)$
9  if ub $\neq$ Null:
10     iterate $ub$ - $lb$ times:
11         process $v$ in $vertexIds$:
12             join $v$ with $e$ in $graph.edges$:
13                 if $e.label == label$:
14                     $vertexIds \leftarrow vertexIds.update(v.id)$
15                 else:
16                     $vertexIds \leftarrow vertexIds.delete(v.id)$
17 else:
18     iterate until $max$ times:
19         $vertexIds \leftarrow vertexIds.update(v.id)$
20 $upadatedPaths \leftarrow paths.update(vertexIds)$
21 return $upadatedPaths$
```

4.4 Logical Query Plan Constructions

Basic operators that we have implemented so far have already been able to support the construction of logical query plans from a Cypher query. Different approaches to combine basic operators could lead to different query plans of the same query. As an example, here a Cypher query applied on the graph instance in Figure 3.1 is listed:

```
MATCH (a: Administrator) <- [:Likes] - (b: User) - [:Follows] -> (c: User)
RETURN b
```

The most straightforward query plan of the query above is quite easy to form. By utilizing scan operators, edge-join operators and join operators in our system, we could start the matching procedure from the left side of the graph query pattern in the "MATCH" clause then expand to the right side. This query plan is shown in Figure 4.1. Also, the matching procedure could also be executed in a reverse order, of which the query plan is shown in Figure 4.2.

Here we present the execution procedure of query plan 1. First scan operators will retrieve all the vertex IDs with specific labels, of that the results are shown in Table 4.4 and 4.5. Then, an edge-join operator which selects edges labeled by "Likes" would take the datasets shown in Table 4.4 and the edge set shown in Table 4.2 as inputs to execute a Flink join operator. The results after the edge-join operator execution are displayed in Table 4.6. Here the resulting paths would be extended by two columns, one column of edge identifiers and one of target vertex identifiers. Next, a join operator in our system would be applied on the results in Table 4.6 and the results returned by the operator "Scan (b)" in Table 4.5. The join results would be the same in Table 4.6. The other edge-join operator in Figure 4.1 would also be applied, the results of which are shown in Table 4.7. Then similar steps as preceding ones would be taken. The final results after the execution of the query above are also presented in Table 4.7.

Alternative query plan 3 and 4 are shown in Figure 4.3 and 4.4. The only difference here is that

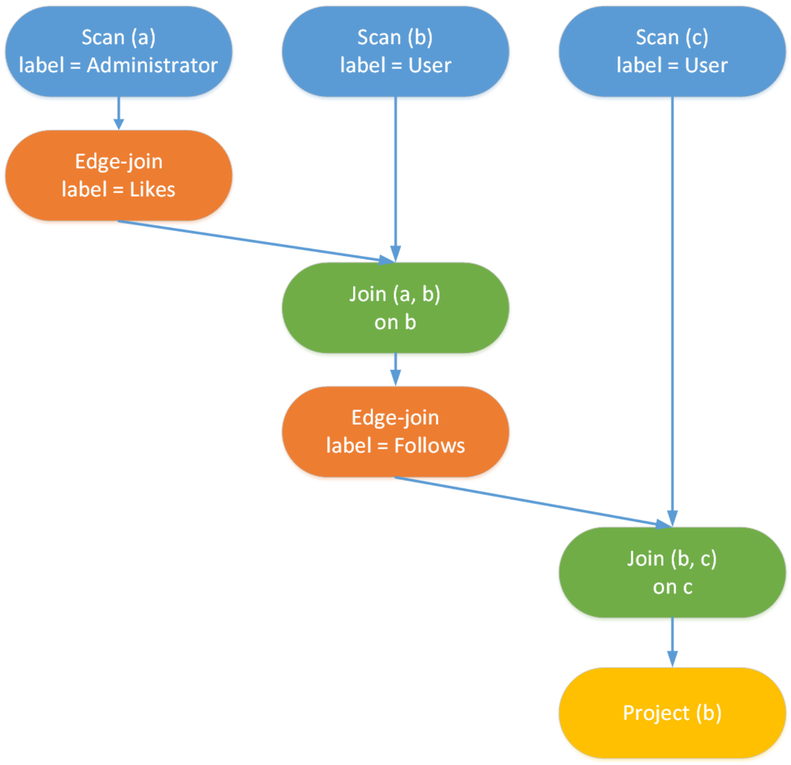


Figure 4.1: Query plan 1 for the example query

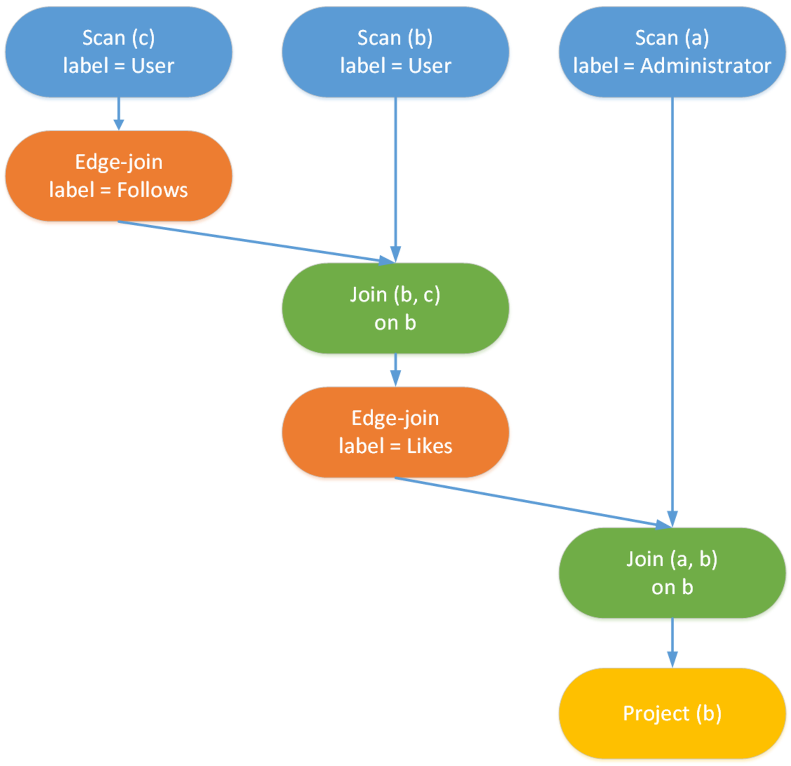


Figure 4.2: Query plan 2 for the example query

id
2L

Table 4.4: Results returned by Scan (a)

id
1L
3L

Table 4.5: Results returned by Scan (b) and Scan (c)

srcv_id	*e_id*	*tarv_id*
2L	e1	1L

Table 4.6: Results returned by Edge-join with label "Likes"

v_id	*e_id*	*v_id*	*e_id*	*v_id*
2L	e1	1L	e2	3L

Table 4.7: Results returned by Edge-join with label "Follows"

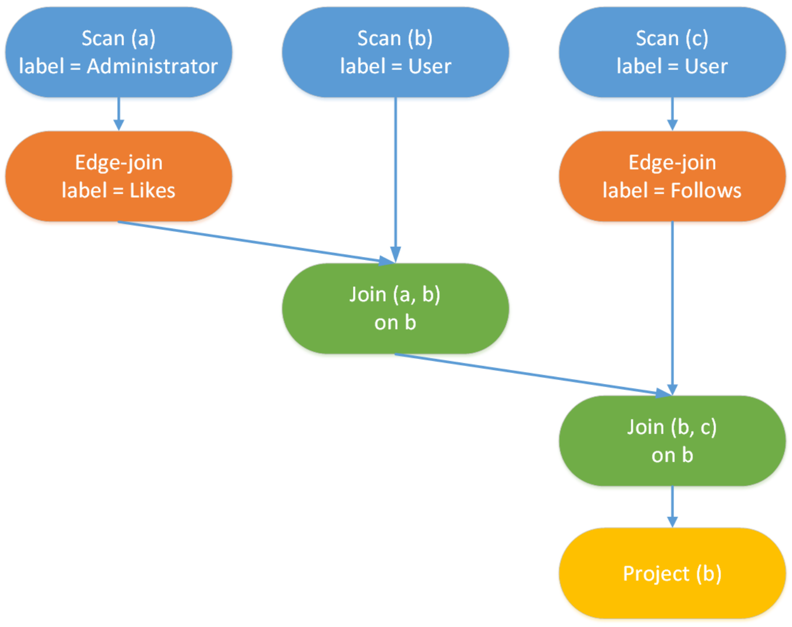


Figure 4.3: Query plan 3 for the example query

the two edge-join operators could be executed concurrently. Figure 4.5 and 4.6 show the execution plans of query plan 1 and query plan 3 in the Flink system respectively. We could see that the concurrent execution of two edge-join operators does provide Flink engine more flexibility since the execution plan of query plan 1 has a longer pipeline than that of query plan 3. The algorithms for generating query plans will be illustrated in later chapters. But here we could already have a general impression about how to generate a query plan for a graph pattern match query: each vertex requires a scan operator to be applied on; And each edge demands an edge-join operator to be applied on to extend the resulting paths. Hence, generally there are two decisions needed to be made in a query plan: The first is about whether a join operator should be applied. The second is about the order of the join operators, including both edge-join operators and join operators.

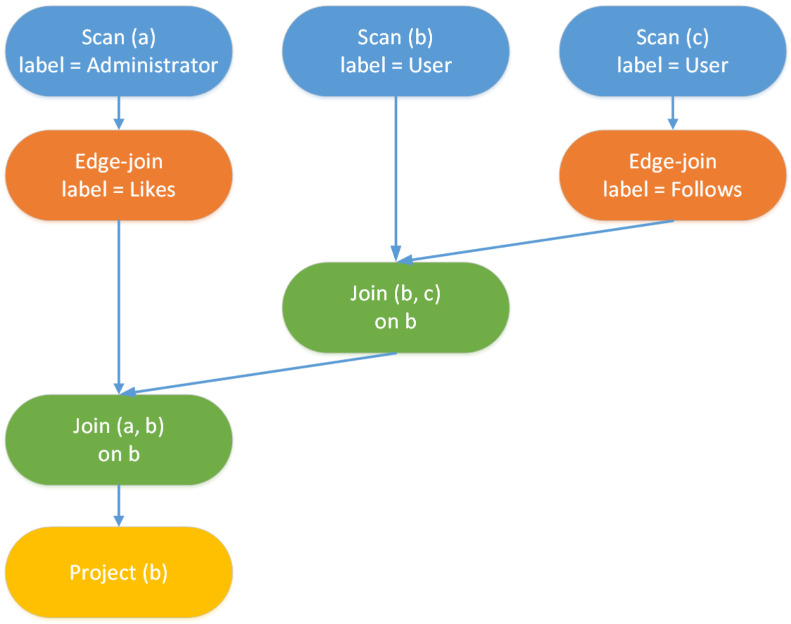


Figure 4.4: Query plan 4 for the example query

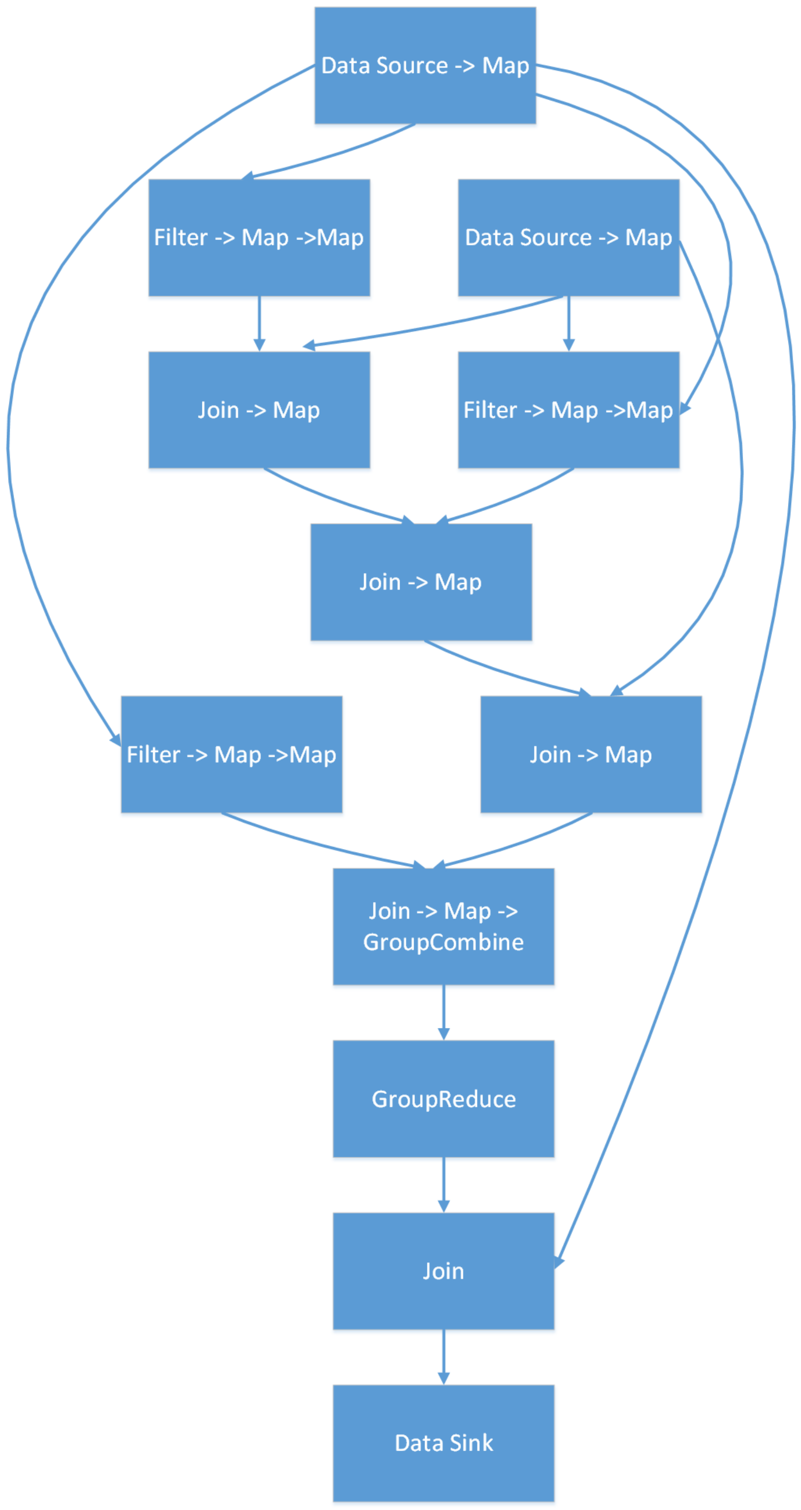


Figure 4.5: Flink execution plan for query plan 1

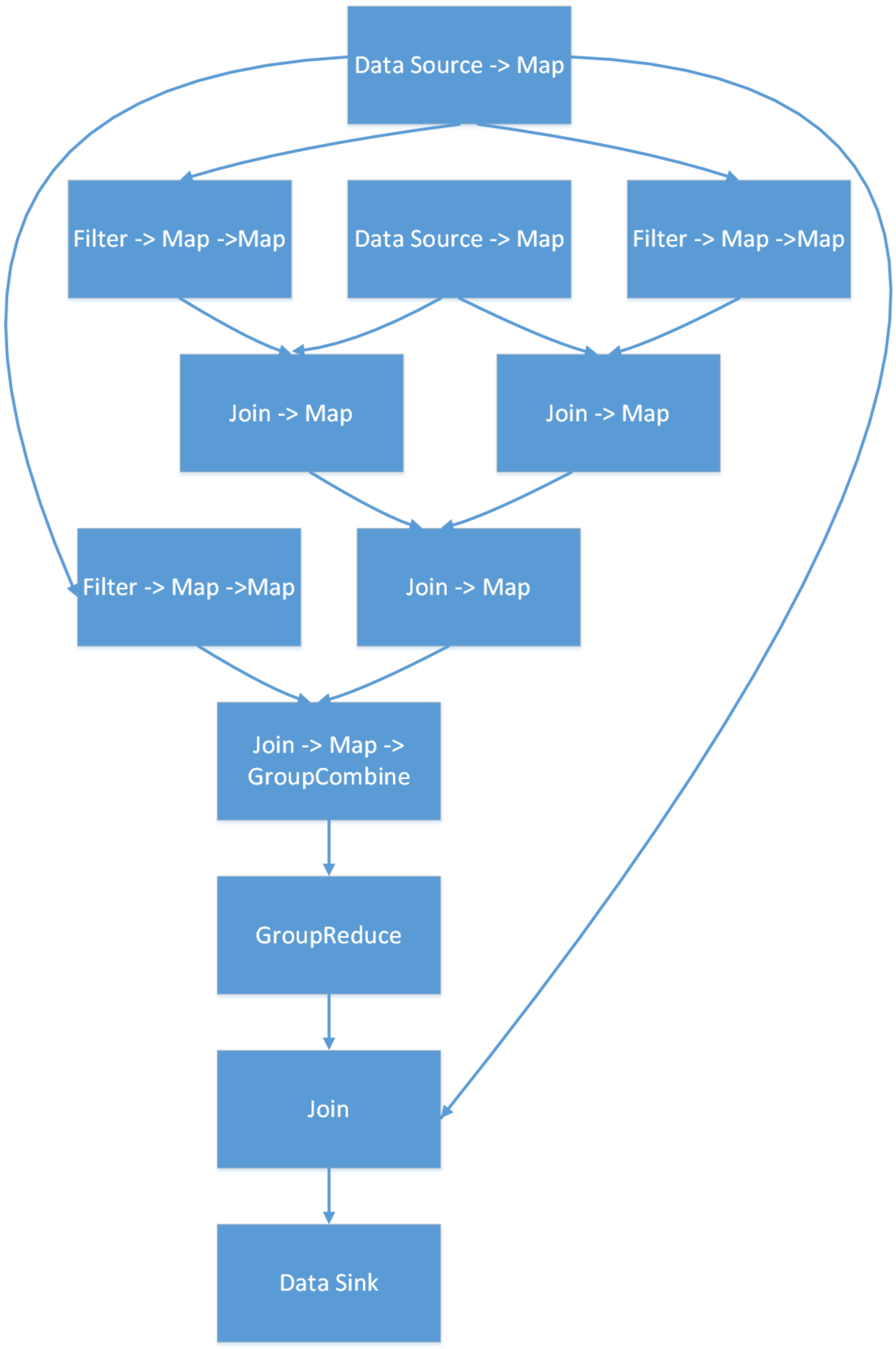


Figure 4.6: Flink execution plan for query plan 3

4.5 Optimization Strategies

4.5.1 Observations and Hypotheses

Based on our implementation, we have the following observations as our hypotheses for further constructions of the query plan generator and query optimization strategies. These hypotheses will be tested during our experiments.

Expanding Directions of Edge-join Operators

According to the fundamental implementation above, the expanding direction of edge-join operators would affect the performance of a query since it influences the sizes of intermediate results returned by queries. Consider a basic query graph pattern (v_1, e, v_2), after a scan operator applied on v_1 or v_2, the intermediate results would be sent to an edge-join operator as an input to do a join operation. For a join operator in the Flink system, on which our edge-join operator is based, the reduction on cardinality would decrease the execution time of this Flink join operator. Besides, if multiple workers are included or even a cluster of servers are involved, the results after scan operators may be even transferred among multiple workers or even servers. Then the intermediate results of larger sizes could incur higher overhead on the communication costs of a Flink system, which is considered to be a potential bottleneck. Hence, the expanding direction of edge-join operators should depend on the data sizes of its left side and right side. For the same example (v_1, e, v_2), we could either calculate or estimate the data sizes of v_1 and v_2 and then select the one with smaller size to start the edge-join operator. Note that the Flink system has different join strategies about how to partition two inputs of a join operator, and this feature would be considered later.

Choosing Vertices With Most Selective Conditions

As well as the expanding direction of edge-join operators, the selectivity of a vertex or an edge should also matter. Same explanations of expanding direction of edge-join operators could also be applied here, namely reduction on the data size could not only decrease the execution time of a Flink operator, but also lower the possible communication costs among workers or servers. Moreover, considering the selectivity of a query edge is quite important since choosing one with high selectivity in a query graph pattern in earlier steps would introduce an intermediate dataset with a lower size and greatly affect the execution time of this query.

Join Order Selection

For a basic graph pattern (v_1, e, v_2), the selectivity about two types of vertices and the expanding direction of an edge-join are relatively easy to be determined. But for more complicated graph patterns, the selection of join orders is another important aspect that needs to be treated seriously. Figure 4.1, 4.2, 4.3 and 4.4 have already exhibited different query plans constructed by different join orders for one same query. It is noticeable that there exists one edge-join operator in Figure 4.1 and 4.2 can only be executed until one preceding join operator finishes and passes the results to an edge-join operator. While in Figure 4.3 and 4.4, the two edge-join operators can be executed simultaneously. Though basic operators in our implementation are built on Flink operators and the Flink optimizer may reorder the Flink operators to form a new job graph, a proper selection of the join order could still provide the Flink optimizer with more flexibility.

4.5.2 Query Optimization

Based on preceding hypothesis, we have proposed two algorithms to generate efficient query plans for different queries, which are cost-based query optimization and rule-based query optimization correspondingly. The query plans shown in Figure 4.1, 4.2, 4.3 and 4.4 have already revealed that for a pattern matching graph query, each vertex requires a scan operator to filter the vertex set by properties and labels and each edge needs an edge-join operator to expand the paths from the previous source vertices by adding edges and target vertices. We treat a query graph pattern in the form of (v_1, e, v_2) as a basic query graph pattern. Thus a query graph could be regarded as a set of connected basic graph patterns by join operators in our system. Note that here we also introduce the concept of a query graph component: a query graph component could be either a single vertex in a query graph or a collection of connected basic query graph patterns. Then there leaves several decisions for designing a query plan generator:

1. Within a query graph component, each edge-join operator should start expanding from which side? Based on our hypothesis, an edge-join operator should start from the side with lower data size.

2. How to determine the selectivity of a query graph component since the selectivity would affect the query execution?

3. In which order the basic query graph patterns should be joined?

Our fundamental idea for generating a query plan here is to treat a query graph as a set of query graph components. The query plan generator would exert a scan operator on each vertex. For each step during the generation of the query plan, the generator would select one edge from the query graph. After that, the generator would compare the estimation cardinality of both sides of this edge and pick one side to start the expanding of an edge-join operator. As indicated before, the target vertex IDs of this edge would also be appended to the paths by this edge-join operator. Then a join operator would join the target vertex IDs with the paths on the other side of the edge. An example about how to generate a query plan for Query 9 (in Appendix A) is shown in Figure 4.7. The entire generation procedure could be described as follows. Here the estimation cardinality of a basic query graph pattern (v_1, e, v_2) is denoted as $card(e)$, and the estimation cardinality of a graph component g is denoted as $card(g)$. The query graph is denoted as qg.

1. Selects an edge e in the query graph where $card(e, e.gc_1, ,e.gc_2)$ has the minimum cardinality;

2. Check the two query graph components gc_1 and gc_2 connected by e, compare $card(gc_1)$ and $card(gc_2)$;

3. If $card(gc_1) <= card(gc_2)$, the edge-join operator would be set to start expanding from gc_1 side and leave gc_2 untouched. Otherwise it would be set to start expanding from gc_2 side;

4. Exert a join operator on the results obtained from the previous edge-join operator and the dataset on untouched side;

5. Regard the query graph component named gc, namely gc_1 and gc_2 connected by e, as a new query graph component in the query graph, re-estimate $card(gc)$; Also delete e from the edge set in the query graph;

6. Check if there are still any edges left in the edge set of query graph. If so, repeat the whole procedure. Otherwise the generation procedure is finished.

As mentioned above, we regard each vertex v in the query graph also as a graph component. This algorithm is presented in Algorithm 7.

4.5.3 Cost-based Query Optimization

The cost-based query optimization algorithm is mainly based on pre-computed statistical information about the datasets. The general idea here is to first collect statistical information, that the number of vertices and edges with a specific label and then utilize these statistics to estimate the cardinality of query graph components in the query graph. From the previous stated generation procedure we can find that the pattern matching Cypher queries could execute a series of join

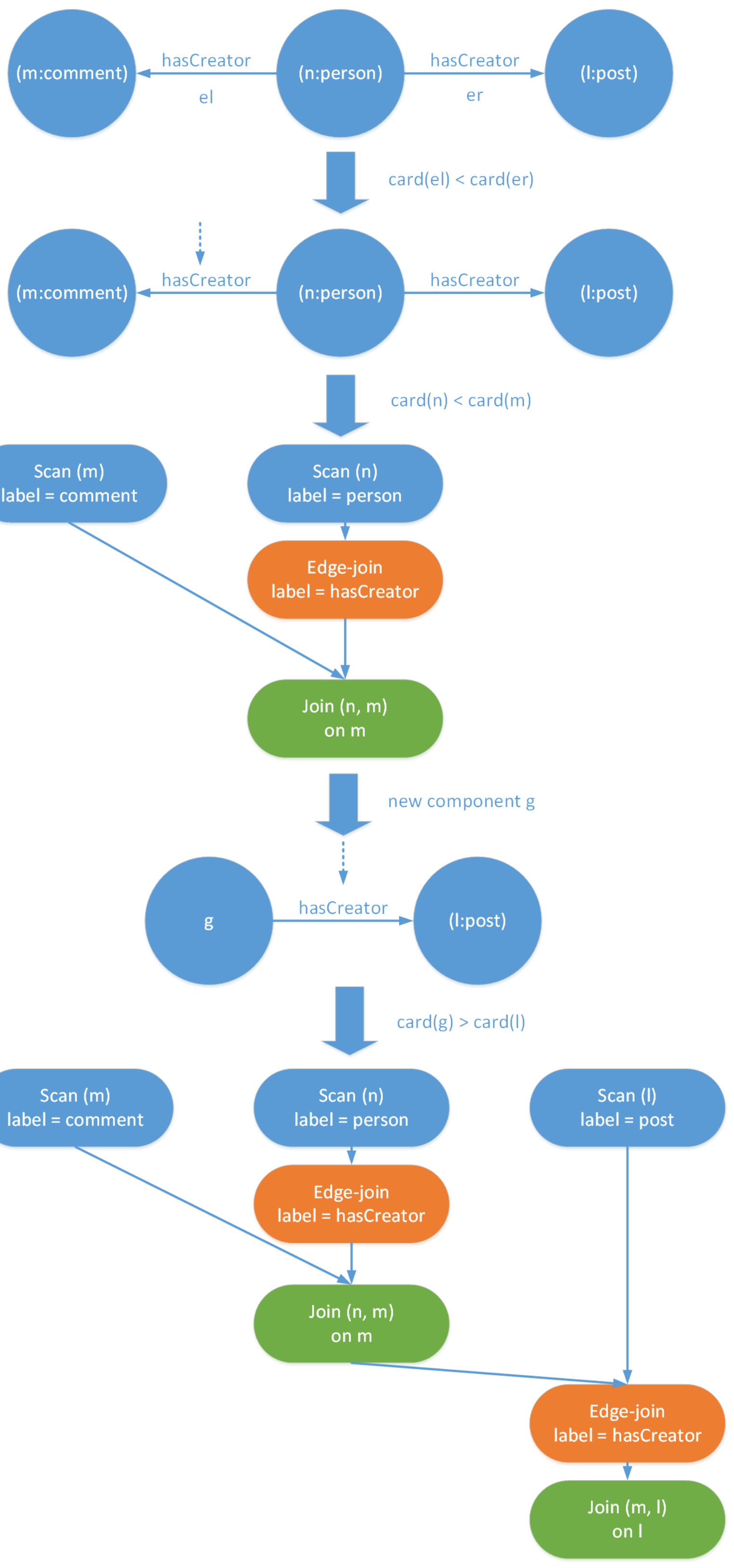


Figure 4.7: Query plan generation procedure

Algorithm 7: Query Plan Generation Algorithm

```
Input  : $qg$: QueryGraph
Output: $plan$: Queue<($e$, $gc$, $gc$)>
1  $edges \leftarrow qg.edges$
2  $gcs \leftarrow qg.vertices$
3  while $edges \neq \emptyset$:
4      select $e$ in $edges$ with $min\ card(e, e.gc_1, e.gc_2)$:
5          $gc_1 \leftarrow e.gc_1 \in gcs$
6          $gc_2 \leftarrow e.gc_2 \in gcs$
7          $v_1 \leftarrow e.v_1 \in gc_1$
8          $v_2 \leftarrow e.v_2 \in gc_2$
9          if $card(gc_1) \leqslant card(gc_2)$:
10             edge-join from $gc_1$
11             join on $scan(v_2)$
12             $plan \leftarrow plan.add(e, v_1, v_2)$
13         else:
14             edge-join from $gc_2$
15             join on $scan(v_1)$
16             $plan \leftarrow plan.add(e, v_2, v_1)$
17         $gc \leftarrow (e, gc_1, gc_2)$
18         $gcs \leftarrow gcs$ - $\{gc_1, gc_2\} \cup \{gc\}$
19         $edges \leftarrow edges$ - $\{e\}$
20 return $plan$
```

operations, either a join operator or an edge-join operator in our implementation. Also the time cost of a scan operator could always be ignored compared to the time cost of a join operator.

From the general algorithm shown in Algorithm 7, we use basic approaches in [19] to estimate the cardinality of a dataset after a join operation is applied since edge-join operators and join operators in our implementation behave the same as join operators in relational algebra. Let A and B denote the inputs of an edge-join operator, $est(A)$ and $est(B)$ denote the estimation size of A and B where A and B share a column (exactly one common attribute of two inputs), Then we have:

$$est(A \bowtie B) = \frac{est(A) * est(B)}{max(est(A), est(B))}$$

It is noticeable that the comparisons are involved in two places in our algorithm. The first comparison is to select an edge with minimum cardinality from the edge set in the query graph. $card(gc)$ denotes the estimation cardinality of a new generated query graph component, namely gc, of which the resulting dataset would be obtained by two join operators. Hence the estimation cardinality of gc is :

$$\begin{aligned} est(gc) =& min(est(gc_1 \bowtie e \bowtie gc_2), est(gc_2 \bowtie e \bowtie gc_1)) \\ =& min(\frac{\frac{est(gc_1)*est(e)}{max(est(gc_1),est(e))} * est(gc_2)}{max(\frac{est(gc_1)*est(e)}{max(est(gc_1),est(e))}, est(gc_2))}, \frac{\frac{est(gc_2)*est(e)}{max(est(gc_2),est(e))} * est(gc_1)}{max(\frac{est(gc_2)*est(e)}{max(est(gc_2),est(e))}, est(gc_1))}) \end{aligned}$$

Note that $est(e)$ indicates the number of edges with a specific label which is pre-computed before the generation of a query plan. If in the data graph there exists a e without specifying any labels, then $est(e)$ should be the number of total edges in the database. Here gc_1 and gc_2 represent two query graph components connected by edge e. Initially, gc_1 and gc_2 are two vertices linked by e when there is no join operators applied on gc_1 and gc_2. Hence, initial $est(gc_1)$ and $est(gc_2)$ both indicate the pre-computed number of vertices with a specific type of label. After e e, gc_1 and gc_2

are bound as a new graph component gc, the $est(gc)$ would be added to the set comprised of query graph components as a new estimation cardinality of gc, computed by using the formula above.

4.5.4 Rule-based Query Optimization

Another algorithm to generate a query plan is to use heuristic rules to estimate the cardinality of query graph components. First we consider using the selectivity of a basic query graph pattern to estimate the cardinality of graph components since one of our hypothesis has claimed that the selectivity of a query graph component would affect the performance of a query. Besides, the features offered by Flink are also utilized to facilitate the query optimization. Inspired by [21] and [20], we propose the selectivity hierarchy below.

Selectivity Hierarchy

The selectivity of a basic graph pattern, namely a graph component in the form of (v_1, e, v_2), is shown below, where p represents a selective property-related condition is included in this position, l represents a selective label-related condition is included in this position and n represents no conditions are involved in this position:

$$\begin{aligned}(p,p,p) \succ (p,l,p) \succ (l,p,p) = (p,p,l) \succ (l,p,l) \succ (p,l,l) = (l,l,p) \succ (l,l,l)\\ \succ (l,n,l) \succ (l,l,n) = (n,l,l) \succ (l,n,n) = (n,n,l) \succ (n,n,n)\end{aligned}$$

The ranking order above is based on the following observations: usually within a query, a condition related with properties on a certain position of the triplet (v_1, e, v_2) owns higher selectivity than a label related condition; And within the property-related conditions, usually edges in a graph database has less properties than vertices, so a feasible assumption here is that property related filtering conditions of edges have higher selectivity than that of vertices; On the other hand, when it comes to label-related cases, the filtering conditions of edges have lower selectivity than that of vertices. The reason is that in a graph database, the number of edges is greatly larger than that of vertices in most cases. Thus we make the assumption that the cardinality of a specific type of edges is usually less than that of vertices with a specific label. Besides, it is quite obvious that the selectivity of a query vertex or a query edge with a label is higher than one without any filtering conditions on it. Based on these observations, we have constructed the selectivity hierarchy above.

Besides, the selectivity of a query vertex or a query edge also depends on the number of filtering conditions on it. Combined all previous observations, we assign different weights to filtering conditions according to the positions on which these conditions would be applied and types of conditions, namely whether the filtering conditions are label-related ones or property-related ones. Let w_{vl}, w_{vp}, w_{el} and w_{ep} denote the weight of a condition which is related to a label of a query vertex, a property of a query vertex, a label of a query edge and a property of a query edge, respectively. Based on the selectivity hierarchy above, the relationship of these four weights should be:

$$w_{ep} > w_{vp} > w_{vl} > w_{el}$$

In this algorithm, we combine these weights with the number of filtering conditions of a query graph component to estimate and predicate the cardinality. For a query edge e and a query vertex v, the estimations of e and v look as follows, where n_{ep}, n_{el} respectively denotes the number of filtering conditions on the properties and the label of e and n_{vp}, n_{vl} denotes the number of filtering conditions on the properties and the labels of v:

$$est(e) = n_{ep} * w_{ep} + n_{el} * w_{el}$$

$$est(v) = n_{vp} * w_{vp} + n_{vl} * w_{vl}$$

While a new query graph component forms, the estimation of gc based on the maximum value of estimations of $est(e)$, $est(gc_1)$, $est(gc_2)$, namely

$$est(gc) = max(est(e), est(gc_1), est(gc_2))$$

We set $w_e p$ to 1.5, $w_v p$ to 1.0, $w_v l$ to 0.7 and $w_e l$ to 0.5, in order to distinguish the selectivity of the filtering conditions from different positions and types. Note that the weights could also be tuned. While in our experiments we used the previous values above.

Join Strategy

Also different join strategies provided by Flink engine could also be utilized to do the query optimization. There are two types of join strategies used in our query plan generation:

BROADCAST_HASH_FIRST For a join operator with broadcast-hash-first strategy, Flink engine will broadcast the first input and build a hash table from it. Then the second input then will probe the hash table. This strategy works well if the first input is smaller than the second one.

REPARTITION_HASH_FIRST For a join operator with repartition-hash-first strategy, Flink engine will shuffle each input and build a hash table from the first input. This strategy works when the first input is smaller than the second, but both inputs are still large. And this strategy is the default one if no size estimates can be made.

Since during each step of the query plan generation a query graph component with highest selectivity would be selected, so we set a threshold for the selectivity estimation. Once the estimation selectivity of a query graph component is larger than the threshold, it could be assumed that the cardinality of this input is remarkably smaller than the other one, then the broadcast-hash-first strategy will be applied.

Chapter 5

Experimental Design

In this chapter, we introduce our experimental design. Section 5.1 briefly introduces the datasets used in our experiments. Section 5.2 describes the queries which we are interested in for the experiments. In Section 5.3, the experimental environment is described. To investigate the performance of our implementation and query optimization strategies, we designed relevant experiments for further investigations on the behaviours of our query optimizers.

5.1 Datasets

5.1.1 LDBC-SNB

For experimental measurements about our implementations of Cypher operators, we used datasets of social network benchmark from LDBC and other datasets generated by gMark. The number of main entities, vertices and edges in the social network graph are shown in Table 5.1. Here the number of vertices labeled "person" is specified in the generation setting of datasets from LDBC benchmark to stem the rest of the datasets. For each step this number is increased by approximately 5,000.

5.1.2 gMark

gMark is also used in our experiments to generate graph instances and query workloads. Several aspects of the generated graph instances can be defined by the user, including node types, edge types, their proportions to the whole instance, and in- and out-degree distributions. Note that here we used the schema of a protein network provided by gMark. For each dataset, all edges only has one kind of edge distributions no matter the directions or the labels of the edges. Three

Dataset	Size (:KB)	#Person	#Comment	#Post	#Vertex	#Edge
LDBC - 0	60,541	903	73,233	77,884	185,647	768,113
LDBC - 1	495,160	4,541	742,178	435,125	1,248,450	6,360,291
LDBC - 2	1,223,218	9,060	1,856,798	916,140	2,890,528	15,603,101
LDBC - 3	2,068,671	13,591	3,149,492	1,423,781	4,737,897	26,333,130
LDBC - 4	2,912,022	18,121	4,470,236	1,910,876	6,590,828	37,061,882
LDBC - 5	3,860,499	22,644	5,988,355	2,446,454	8,691,978	49,104,907
LDBC - 6	4,930,864	27,172	7,578,568	2,988,543	10,872,560	62,223,412
LDBC - 7	5,841,973	31,697	8,982,889	3,478,134	12,810,366	73,589,177
LDBC - 8	6,941,161	36,174	10,673,088	4,004,446	15,072,041	86,931,309

Table 5.1: LDBC-SNB datasets

Dataset	Size (:KB)	Edge Dist	#Vertex	#Edge
gMark - 1	295,589	normal	4,601,165	5,448,459
gMark - 2	582,723	uniform	9,874,534	10,576,065
gMark - 3	271,797	zipfian	3,004,434	5,789,069

Table 5.2: gMark datasets

types of edge distributions are included here, namely normal distribution, uniform distribution and zipfian distribution. Table 5.2 shows the datasets generated using gMark.

5.2 Queries

The queries used in our experiments could be categorized into three types, namely the linear queries, the quadratic queries and the complex queries. All numbered queries mentioned below in the experimental steps are all presented in Appendix A and B.

Linear Queries

Linear queries refer to the queries of which the number of returned results grows approximately linearly to the rate of the growth of the number of vertices or edges. In our implementation, a scan operator which selects the vertices labeled "person" in LDBC-SNB datasets of various data sizes is expected to exhibit this characteristics.

Quadratic Queries

Similar to the concept of linear queries, quadratic queries refer to the queries of which the number of returned results grows quadratically to the rate of the growth of the number of vertices or edges. In our implementation, an edge-join operator which selects the edges labeled "likes" is expected to behave as a quadratic query when applied on LDBC-SNB datasets.

Complex Queries

Complex queries includes those more complicated than linear queries and quadratic queries. Usually the workload of a complex query can not be evaluated straightforwardly. Thus an algorithm is necessary to generate a feasible query plan to avoid the inefficiency brought by an improper query plan. In general, our research interests are around the complex queries from the following different shapes, which are shown in Figure 5.1. Each blue dot in Figure 5.1 represents a query vertex in a query graph, and each line indicates a query edge in a query graph.

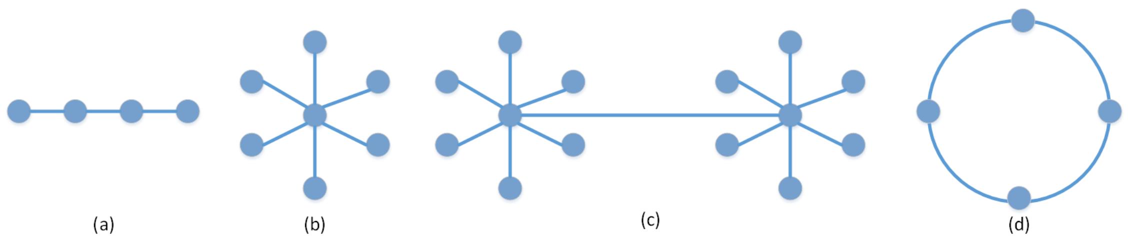


Figure 5.1: (a) A chain query (b) A star query (c) A chain-star query (d) A cycle query

5.3 Environment

The experiments were run on a single machine using the Google compute engine. The machine is equipped with 8 Intel (R) Xeon(R) CPU cores @2.30GHz (Haswell platform) and 30 GB memory.

5.3.1 Settings for Flink

The main settings of Flink on a single machine are listed below:

- The memory heap size of the *jobmanager* is 2048 MB (2 GB);
- The memory heap size of the *taskmanager* is 20480 MB (20 GB). Usually the memory heap size of a *taskmanager* should be larger than the data size used for experiments;
- The number of task slots in the *taskmanager* is set to 8, since our server has 8 CPU cores.

We leave the parameter *parallelism.default* with its default value 1, since we will change the level of parallelism with our program. The temporary directory also needs to be set to a suitable value in the configuration file, but we do not show it here since it depends on the disk configuration of the server.

Chapter 6

Experimental Results and Evaluations

—–This chapter details the basic measurements, experimental results and corresponding evaluations based on the previous chapter. Section 6.1 shows some basic measurements on simple queries for testifying our hypotheses. Section 6.2 describes our experiments and evaluations on more complicated queries, which investigate the behaviours on data sizes, parallelism of Flink system. Results from the cost-based optimizer and the rule-based optimizer are compared here. We then discuss and analyze the comparison results.

6.1 Basic Measurements

First we evaluated basic cases of some simple queries without allowing any parallelism in Flink system to prove the proposed hypotheses for query optimization in Chapter 4 and the query-related hypothesis in Chapter 5. It is necessary to prove the hypotheses about simple queries since only if the behaviours of basic operators satisfy our expectations, could the scalability of more complicated queries be guaranteed. Besides, the hypotheses for query optimization also need to be proved correctly since these are the fundamentals of our optimization strategies.

6.1.1 Linear Query

The first tested query, Query 1, is a simple selection query which filters all vertices with the label "person". The relationship between the execution time and the number of vertices is shown in Figure 3.1. Since selecting starting vertex IDs only requires a scan operation applied on the whole vertex set in graph database, the running time of Query 1 is expected to grow linearly as the number of vertices increases. The x-axis in Figure 6.1 represents the number of vertices from the datasets. The y-axis represents the running time of the query when it is applied to datasets of various sizes. The results in Figure 6.1 proves this hypothesis.

6.1.2 Quadratic Query

We then evaluate the relationship between the number of edges and the running time of Query 2 which is expected to behave like a quadratic query. The selected vertex IDs first retrieved by Query 2 will be joined with all edges in database. Hence, the running time is expected to grow quadratically as the number of edges increases. The x-axis in Figure 6.2 represents the number of edges and the y-axis shows the running time. From the results shown in Figure 6.2, our hypothesis about quadratic queries is also proved correct.

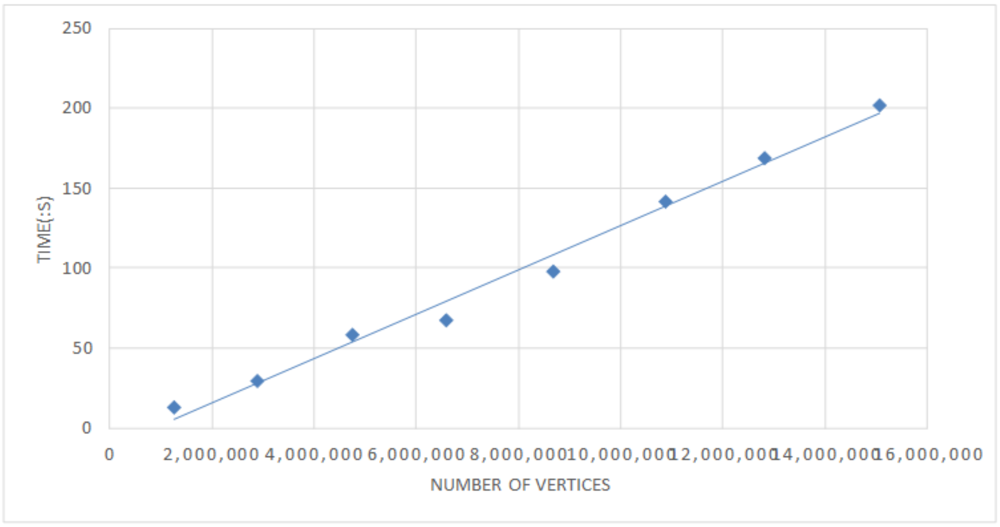


Figure 6.1: Execution time vs number of vertices: Query 1

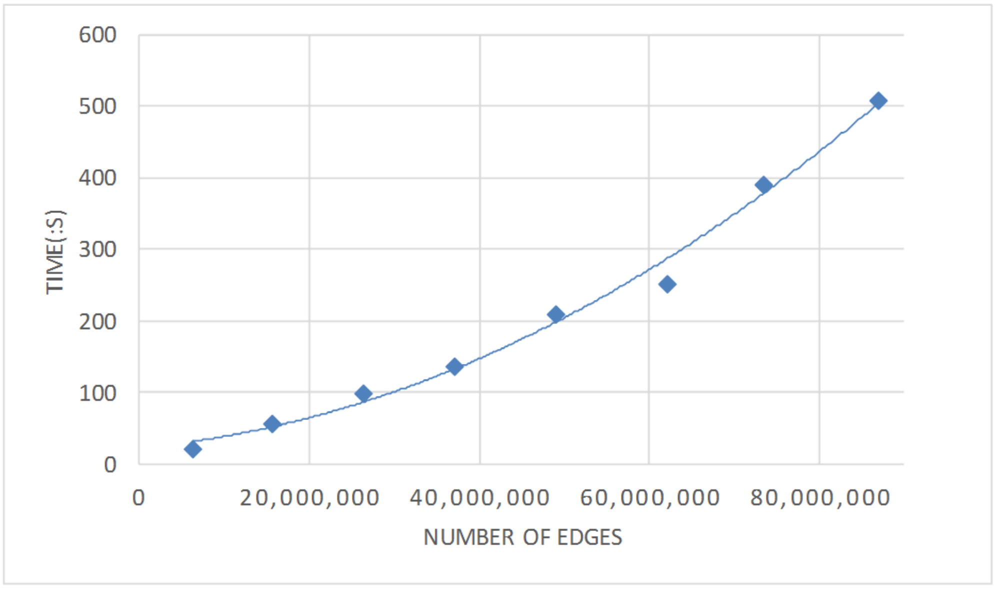


Figure 6.2: Execution time vs number of edges: Query 2

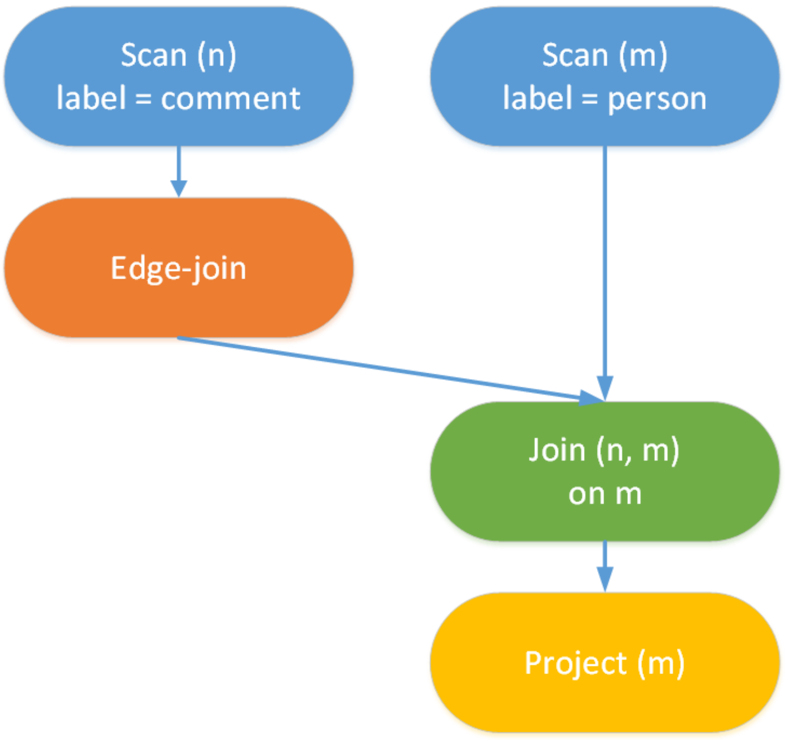


Figure 6.3: Query plan for Query 3

6.1.3 Complex Queries

Next, more simple queries are used to investigate the relationship between the running time and the sizes of data (KB) to test the correctness of our hypotheses for query optimization.

Experiments On Basic Graph Patterns

Among all complex queries, it is considered helpful to figure out the behaviors of a query in the form of a basic graph pattern at first. Thus we introduce the Query 3 and 4. Query 3 and Query 4 are logically equivalent queries which differs in selecting starting vertices, which could be regarded as two query plans of a simple query, shown in Figure 6.3 and Figure 6.4. Query 3 chooses vertices with label "comment" first then associates with edges in the corresponding graph. Instead, Query 4 chooses vertices with label "person" first. According to Table 5.1, the number of vertices labeled with "person" is smaller than that of vertices labeled with "comment".

Figure 6.5 displays the relationship between the execution time of two queries and the data sizes. The x-axis in Figure 6.5 stands for the raw data sizes and the y-axis indicates the execution time in seconds. Here the blue dashed curve represents the results of Query 3, and the red dashed curve, respectively, represents the results of Query 4. It is noticeable that the results obtained from the first scan operator, no matter this scan operator is used to select vertices labeled by "comment" or "person", would be joined with the whole edge set in the graph dataset. Hence, a reduction on the size of a data set, which is one of the inputs of a join operator, will efficiently decrease the execution time of a query.

Similar experiments were also done on Query 6 and 8. For Query 6, the starting vertices are still the vertices labeled with "person" while in Query 8 the order is reversed. However, the results shown in Figure 6.6 are quite distinct from that shown in Figure 6.5. In Figure 6.6, the blue bars represent the running times of Query 6, while the red bars represents those of Query 8. Except for the data of size 495,160 KB (the smallest data), the execution time of Query 6 is less than that of Query 8. The results do not mean that our previous observations are wrong. It is noticeable that

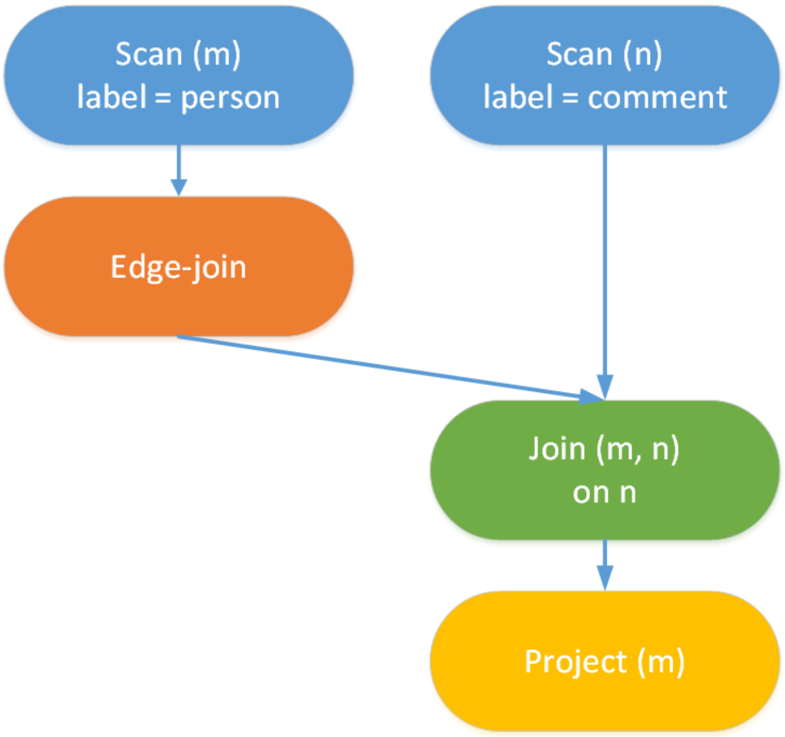


Figure 6.4: Query plan for Query 4

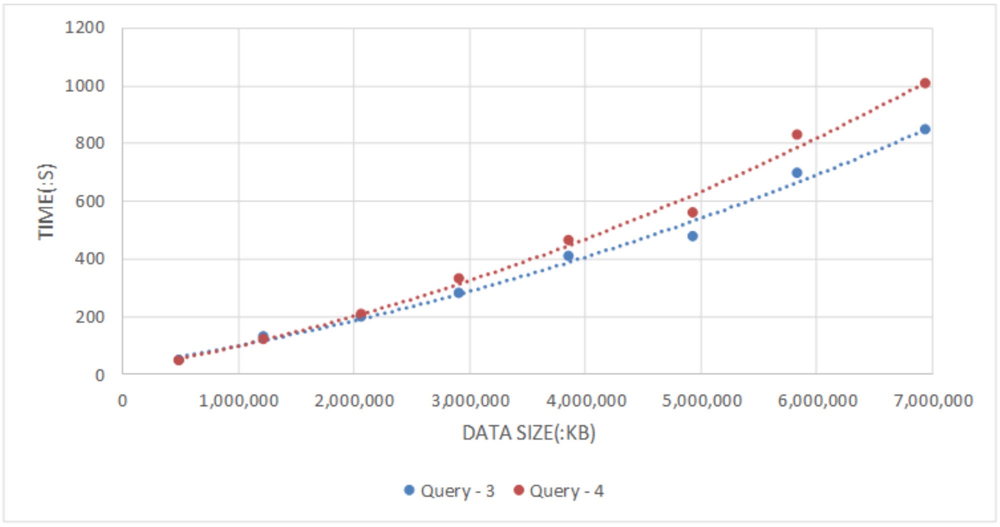


Figure 6.5: Execution time vs data size: Query 3 and Query 4

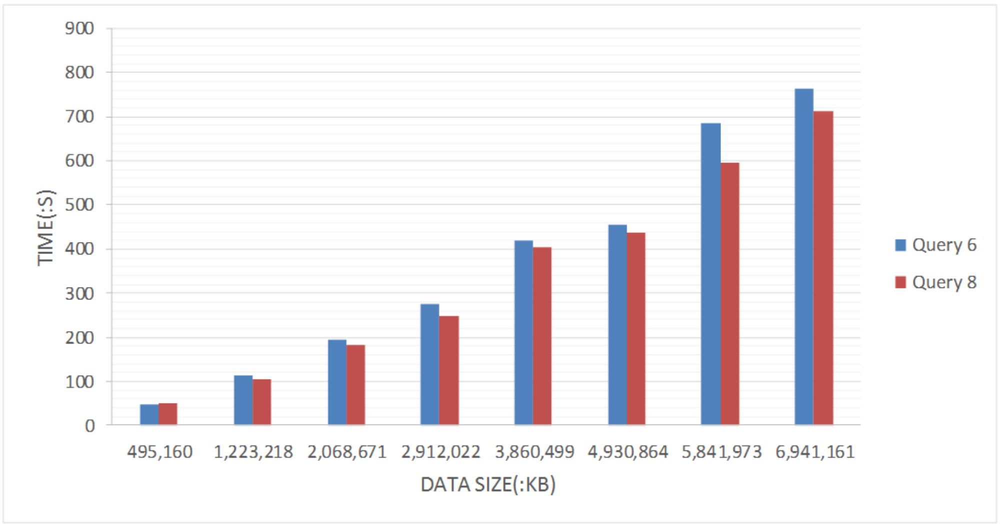


Figure 6.6: Execution time vs data size: Query 6 and Query 8

Query 6 and 8 add more filtering conditions to the vertices with label "comment" by specifying the lengths of the comments and which browser is used. Particularly, we calculated the number of vertices in each side of these queries, it proves that the number of vertices which indicates the filtered comments is less than the number of vertices labeled by "person". Hence, the previous hypothesis in Chapter 5 is proved to be correct, namely expanding an edge-join operator from the side, of which the data size is smaller, could be an efficient approach for query optimization.

Selectivity

Our next consideration of query optimization is the selectivity of a query graph element. This element could be a query vertex, a query edge or a query graph component which consists of connected edges and vertices. Hence, we also conducted an experiment to test the influences of selectivity and compared the results of Query 6 and 7, of which the relationship between the execution time and the data size is shown in Figure 6.7. The blue bars show the running times of Query 6, and the red bars show those of Query 7. On each dataset, the execution time of Query 7 is equal or larger than that of Query 6. Both Query 6 and 7 have similar query plans, which starts from the vertices labeled "person" and then the edge-join operator extends the paths from these vertices. The amount of operators and the execution order of all operators are the same for Query 6 and 7. The only difference between these two queries is that the filtering conditions on vertices with label "comment" of Query 6 is more constrained than that of Query 7. It shows that the selectivity of the vertices is also an influential factor on the performance of a query.

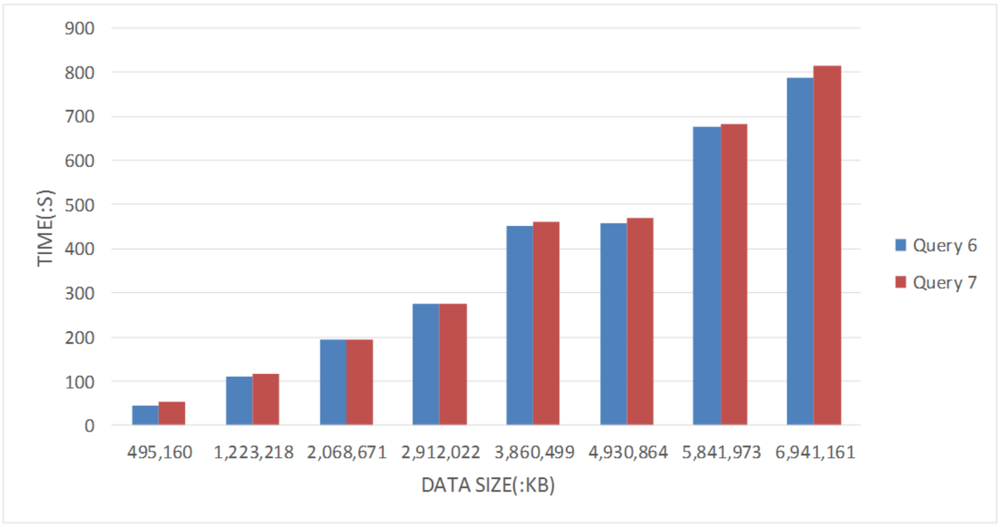


Figure 6.7: Execution time vs data size: Query 6 and Query 7

6.2 Experiments and Evaluations

For all the experiments we use Query 10 to Query 18 on LDBC-SNB datasets. These queries could be categorized into three types: Query 10 to Query 12 are chain queries; Query 13 to Query 15 are star queries and Query 16 to Query 18 are chain-star queries. We do not consider cycle queries here since they could be regarded as a special case of chain queries.

In each shape type, different queries own different levels of selectivity, which is shown in Table 6.1. All the results of the execution time from two query plan generators are shown in Table 6.2 and 6.3, where P1, P2, P4 and P8 on each table on the left top corner represents the level of parallelism during the query executions.

6.2.1 Experiments on Data Sizes

We tested the execution time versus data sizes without allowing any parallelism (P1) and the results are shown in Figure 6.8, 6.9, 6.10, 6.9, 6.12 and 6.13. Regardless of query shapes, the results show that the selectivity of a query could influence the execution performance of a query while using the rule-based query plan generator. Meanwhile, the query selectivity has less effects on the results when using a cost-based query plan generator. In Figure 6.8, we see that as the data size increases to 817.9% from LDBC - 0 to LDBC - 1 and then increases to 779.6% from LDBC - 1 TO LDBC - 5, the execution time first increases to 1423.1% and then to 1398.9% for the query with relevant low selectivity, Query 10. Even though in Figure 6.12 the blue curve which indicates that the relationship between execution time and the data size has the sharpest turning point among these figures, the execution time still only increases to 1154.5% of the first piecewise and to 1279.5% of the second piecewise. Looking at the execution results from high selectivity queries of which the execution plans were generated also by this rule-based query plan generator, the green curves in Figure 6.8, 6.10 and 6.12 are relevantly flat compared to the blue curves.

	Chain	Star	Chain-star
low selectivity	Q10	Q13	Q16
medium selectivity	Q11	Q14	Q17
high selectivity	Q12	Q15	Q18

Table 6.1: Query types

P1(:S)	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
LDBC-0	10	9	8	8	9	8	10	9	9
LDBC-1	82	62	53	61	64	62	85	80	75
LDBC-5	976	914	724	914	941	893	1398	1128	1062
P2(:S)	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
LDBC-0	6	6	6	7	7	6	8	7	7
LDBC-1	42	37	35	41	42	41	56	52	51
LDBC-5	799	798	695	887	927	883	1226	1039	1048
P4(:S)	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
LDBC-0	7	7	7	7	7	7	8	7	7
LDBC-1	33	32	31	39	42	42	57	56	52
LDBC-5	756	739	696	861	878	874	1138	1028	1046
P8(:S)	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
LDBC-0	9	10	10	10	10	11	12	11	11
LDBC-1	38	36	36	42	48	43	62	54	55
LDBC-5	747	729	697	859	880	875	1128	1033	1050

Table 6.2: Results of cost-based query plan generator

P1(:S)	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
LDBC-0	13	12	9	10	10	8	21	11	9
LDBC-1	185	82	55	121	77	66	149	127	76
LDBC-5	2588	1172	667	1559	981	826	2306	1625	980
P2(:S)	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
LDBC-0	9	10	7	7	7	6	18	8	7
LDBC-1	138	62	27	80	47	41	114	81	49
LDBC-5	1927	1064	657	1320	875	828	2158	1365	990
P4(:S)	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
LDBC-0	10	10	7	7	8	6	18	8	6
LDBC-1	118	52	29	61	35	30	94	75	36
LDBC-5	1832	1023	675	1149	882	856	2138	1330	1016
P8(:S)	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
LDBC-0	14	13	9	11	11	9	21	11	10
LDBC-1	119	50	29	69	37	32	100	84	39
LDBC-5	1840	1164	678	1142	879	858	2536	1315	1002

Table 6.3: Results of rule-based query plan generator

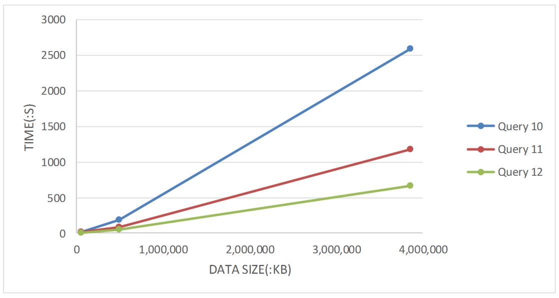


Figure 6.8: Data Size vs Time : Chain Queries, rule-based

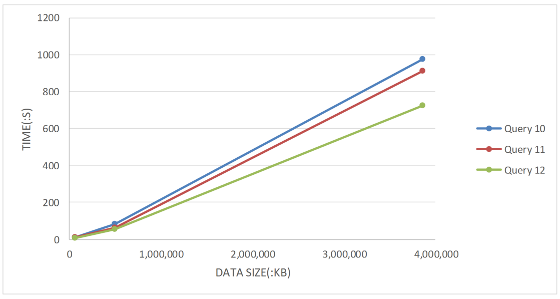


Figure 6.9: Data Size vs Time : Chain Queries, cost-based

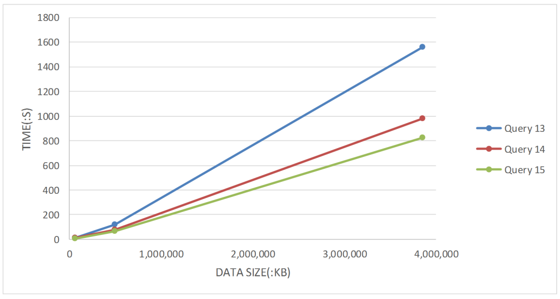


Figure 6.10: Data Size vs Time : Star Queries, rule-based

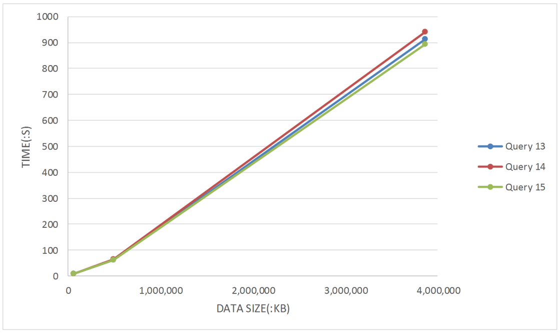


Figure 6.11: Data Size vs Time : Star Queries, cost-based

On the other hand, the selectivity of a query does not affect the execution time of a query much, of which the query plan is generated by a cost-based query generator. In Figure 6.10, the lines of the execution results from star queries are almost overlapped, that indicates the little impact of the selectivity of queries. The reason here is that the cost-based query plan generator only makes use of the statistics related to a label type of a vertex or an edge, not like the rule-based query plan generator.

Note that here we only did tests on a single machine by only using a single worker, so that all the intermediate data sets were restricted to one process and there was no communication overhead. It excludes the possibility that the different behaviors of the results returned by utilizing rule-based optimization strategy were caused by the reduction on communication overhead due to the smaller data sizes of intermediate results. Despite the differences between two types of optimization strategies, our implementation of queries still scale well regardless of query types or query complexity.

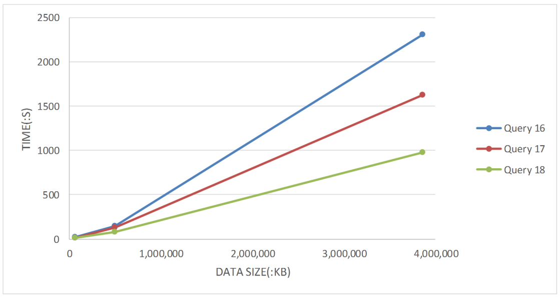


Figure 6.12: Data Size vs Time : Chain-star Queries, rule-based

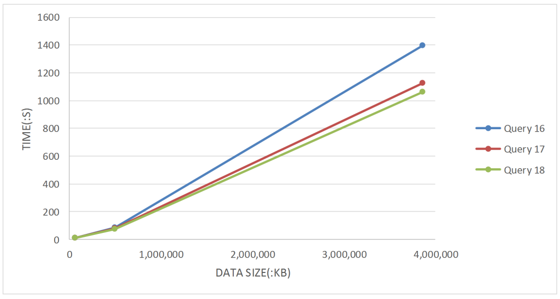


Figure 6.13: Data Size vs Time : Chain-star Queries, cost-based

6.3 Experiments on Edge Distributions

Here we used the queries from gMark in Appendix B for the experiments related to edge distributions of datasets. Query 19 and Query 20 are chain-shaped. Query 21 and 22 are queries of the star shape. And Query 23 and 24 are of chain-star shapes. Table 6.4 and 6.5 show the execution results from Query 19 to Query 24 on the datasets with different edge distributions.

P1(:S)	Q19	Q20	Q21	Q22	Q23	Q24
gMark-1	85	75	89	100	103	118
gMark-2	140	165	199	234	232	260
gMark-3	53	58	2322	2335	2194	2201

Table 6.4: Execution time of the rule-based query plan generator

P1(:S)	Q19	Q20	Q21	Q22	Q23	Q24
gMark-1	62	77	85	97	99	111
gMark-2	139	164	198	225	231	267
gMark-3	51	57	66	50	78	95

Table 6.5: Execution time of the cost-based query plan generator

The comparisons of these three datasets are shown in Figure 6.14, 6.15 and 6.16. Since the datasets generated by gMark do not include properties, these queries could all be considered as low selectivity queries. The consequences show that for the datasets of which the edge distribution is either normal distribution or uniform distribution, there are no significant differences between the rule-based optimization strategy and the cost-based optimization strategy.

But if the edges are in zipfian distribution, then the rule-based optimization strategy shows great disadvantage compared to the cost-based optimization strategy for star queries and chain-star queries. The results indicate that the query execution efficiency could be highly influenced if there is no efficient optimization strategies.

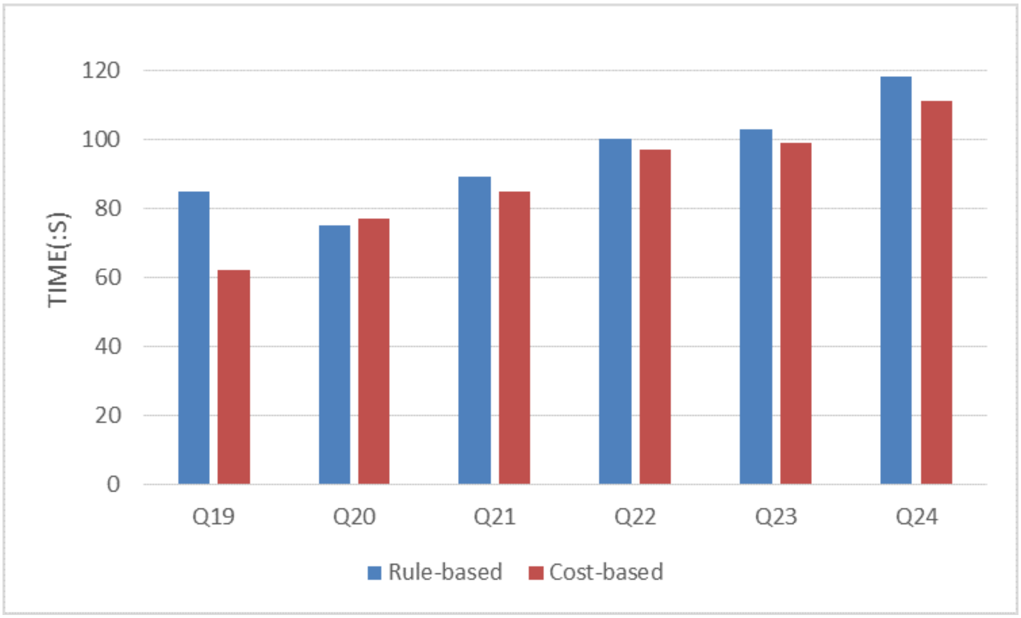


Figure 6.14: Execution time on gMark-1, normal dist.

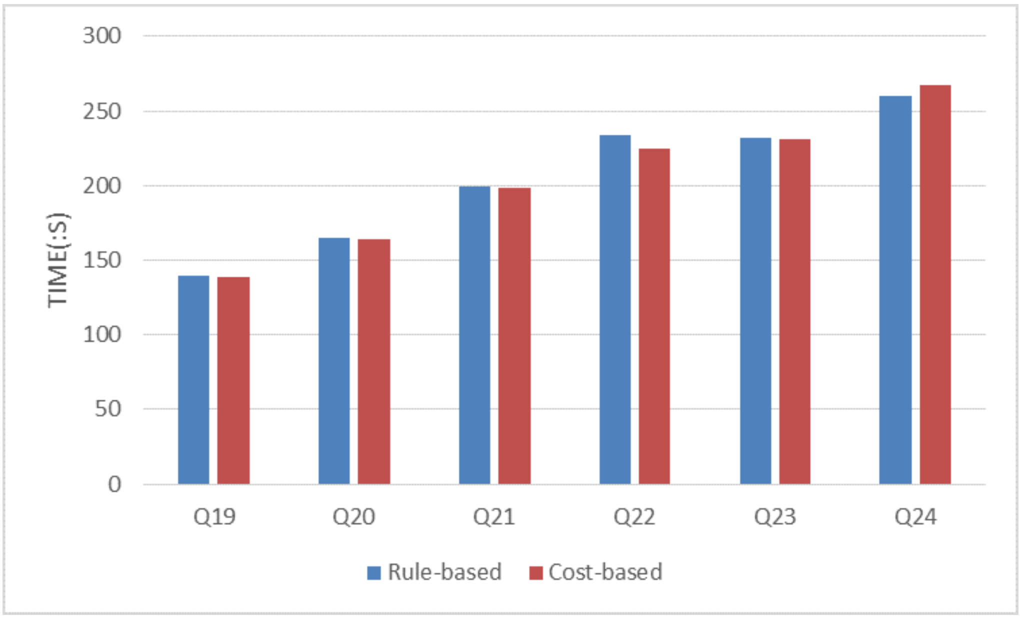


Figure 6.15: Execution time on gMark-2, uniform dist.

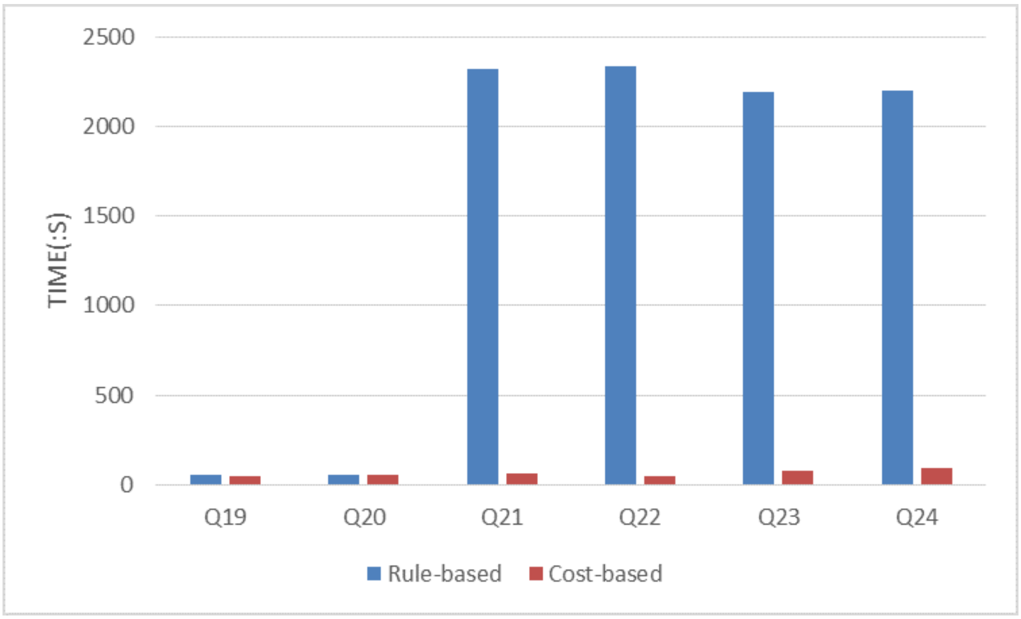


Figure 6.16: Execution time on gMark-3, zipfian dist.

6.4 Experiments on Parallelism

We have also conducted experiments on the level of parallelism of Flink. Thus we set the number of task slots of the *taskmanager* to the same number as CPU cores for full parallelism. The input data will be assigned to different task slots and executed separately until the reduce step occurs to collect the output data.

Figure 6.17, 6.18 and Figure 6.19, display the relationship between the execution time versus the parallelism of Flink on chain queries, of LDBC - 0, LDBC - 1 and LDBC - 5 respectively. As the level of parallelism is increased from 1 to 2, a remarkable speed-up could be obtained on the query execution time no matter which dataset is used in this situation. And in most cases, especially for larger datasets, an enhancement of parallelism from 2 to 4 would also speed up the query executions. But once the parallelism is set to 8, the execution time of a query might increase instead. Two reasons related to Flink could explain these consequences. For smaller datasets, both the execution time of programs or the communication overhead could be relevantly lower than those of larger datasets. Then a query which is executed on a single task slot or on two task slots with quick data delivery or synchronization will not take too much time. However, if more cores are involved, the communication cost among cores would trade off the benefits obtained by using multiple cores to speed up the execution procedure by running duplicated instances on multiple cores.

As for larger datasets, processing the whole dataset with a single task slot would be a time-consuming task. Thus at first, the bottleneck of the graph query execution could be the computation resources, in our case, the CPU cores. But if we keep increasing the level of parallelism, the benefits obtained before will be traded off again.

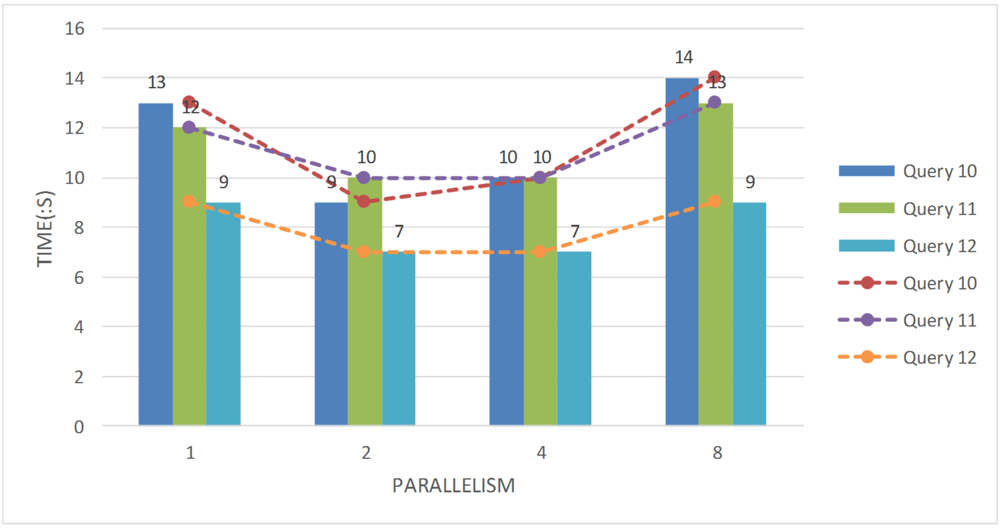


Figure 6.17: Parallelism vs Time : LDBC - 0

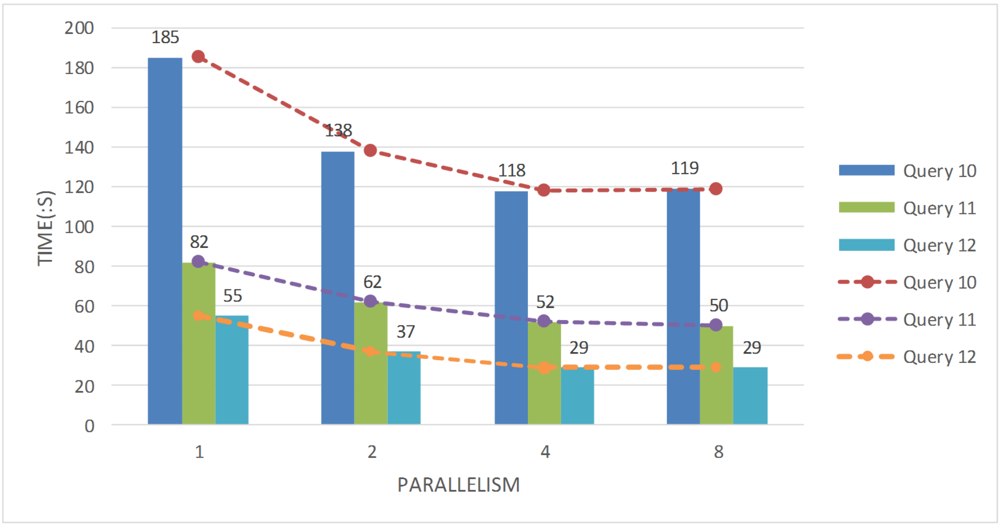


Figure 6.18: Parallelism vs Time : LDBC - 1

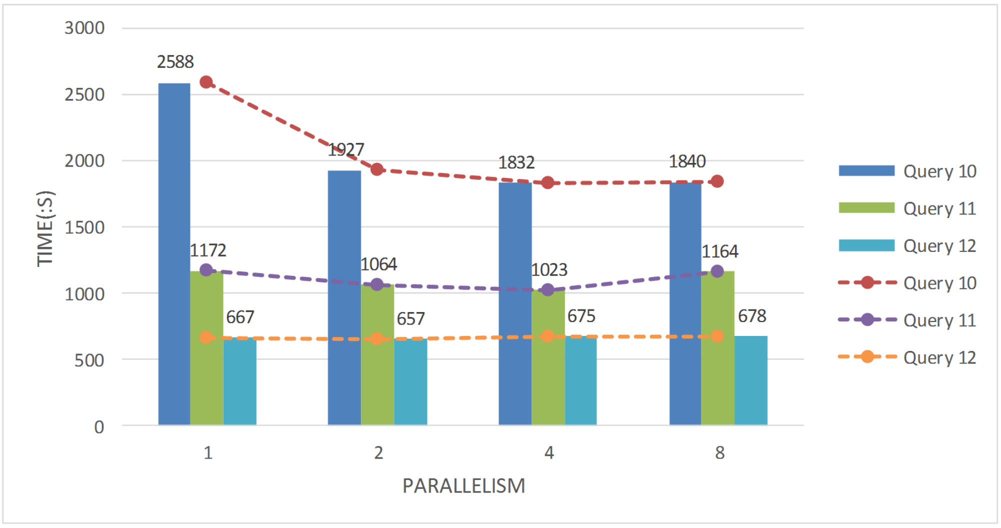


Figure 6.19: Parallelism vs Time : LDBC - 5

6.5 Discussions

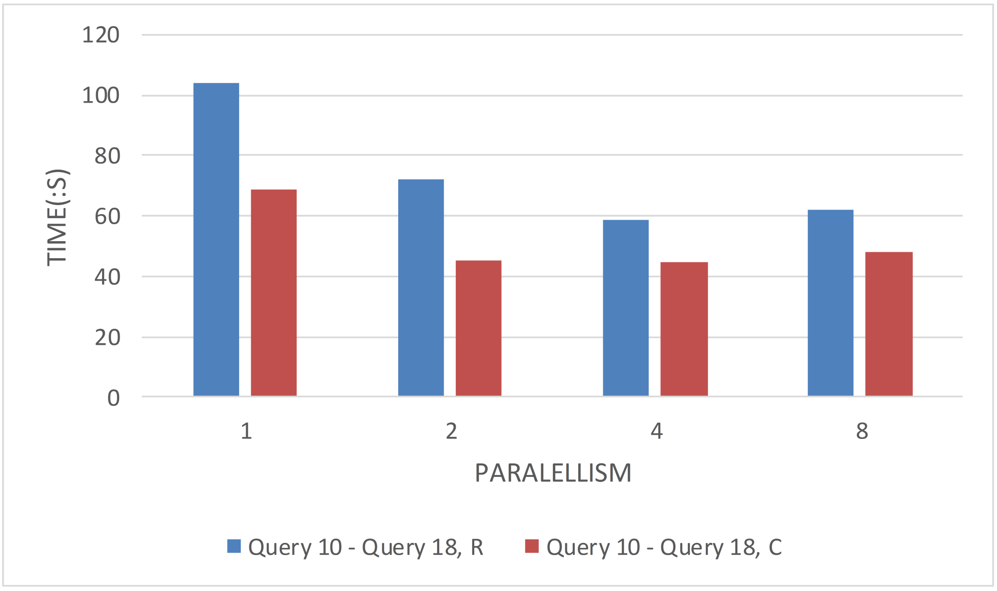
From the previous experiments, we have developed an algorithm and used two different approaches to design a cost-based query optimizer and a rule-based query optimizer. These two optimizers treat a query by applying different estimation strategies. For the same queries, the results are quite different, which are displayed in Figure 6.20. Figure 6.20 exhibits all the query execution results with all tested parallelism on LDBC - 1, and Figure 6.21 exhibits all results with all tested parallelism on LDBC - 5.

As Figure 6.20 and 6.21 show, we can conclude that in most cases, the performance of queries which utilize cost-based query optimization is better that of queries which apply rule-based query optimization, except Query 12, 15 and 18. From Table 6.1, it is known that these three queries are all queries with high selectivity. Also since these experiments were done with different shapes of queries, it can be said that the rule-based query optimizer works well for queries with high selectivity while the cost-based query optimizer is more suitable for processing queries with relevantly low selectivity.

According to the experiments on parallelism in Section 6.2, we do not observe significant speedups with small datasets. This is expected, since with small datasets the networking and other overheads are comparable to the actual time processing the data. The speed-ups of LDBC - 1 and LDBC - 5 are shown in Table 6.6, 6.7, 6.8 and 6.9. For LDBC-1, as the query selectivity varies from low to high, the speed-ups of queries which use the rule-based optimizer increase in general. As for the queries utilizing the cost-based optimizer, the speed-ups of these queries decrease as the query selectivity varies from low to high. For LDBC-5, the speed-ups of both these two kinds of queries all decrease in general as the query selectivity from low to high.

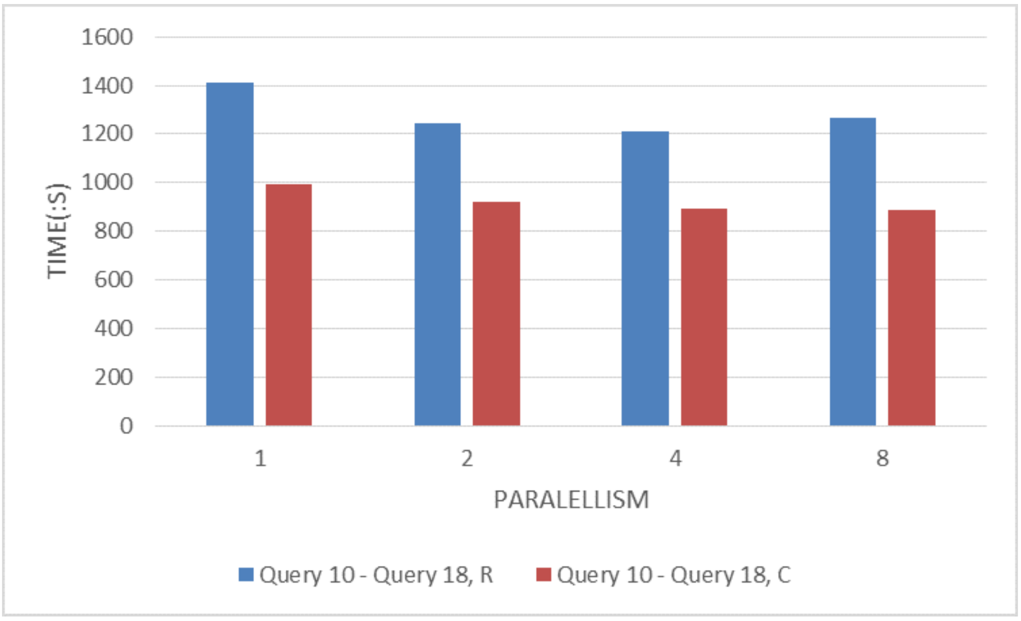
P	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	1.341	1.323	1.486	1.513	1.638	1.610	1.307	1.586	1.551
4	1.568	1.577	1.897	1.984	2.200	2.200	1.585	1.693	2.111
8	1.555	1.640	1.897	1.754	2.081	2.063	1.490	1.512	1.949

Table 6.6: Speed-ups on LDBC - 1, rule-based optimizer



.

Figure 6.20: Parallelism vs Time : LDBC - 1 (average time)



.

Figure 6.21: Parallelism vs Time : LDBC - 5 (average time)

P	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	1.952	1.676	1.514	1.488	1.524	1.512	1.518	1.538	1.471
4	2.485	1.938	1.710	1.564	1.524	1.476	1.491	1.429	1.442
8	2.158	1.722	1.472	1.452	1.333	1.442	1.371	1.481	1.364

Table 6.7: Speed-ups on LDBC - 1, cost-based optimizer

Parallelism	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	1.343	1.102	1.015	1.181	1.121	0.998	1.069	1.190	0.990
4	1.413	1.146	0.988	1.357	1.112	0.965	1.079	1.222	0.965
8	1.407	1.007	0.983	1.365	1.116	0.963	0.909	1.236	0.978

Table 6.8: Speed-ups on LDBC - 5, rule-based optimizer

Parallelism	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	1.222	1.145	1.041	1.030	1.015	1.011	1.140	1.086	1.013
4	1.291	1.237	1.040	1.062	1.072	1.022	1.228	1.097	1.015
8	1.307	1.254	1.039	1.064	1.069	1.021	1.239	1.092	1.011

Table 6.9: Speed-ups on LDBC - 5, cost-based optimizer

6.5.1 Rule-based Optimizer vs Cost-based Optimizer

From the previous experiments, we have already shown that both of these two optimizers have their strengths and weaknesses. For queries with high selectivity, the rule-based optimizer works better than the cost-based one. For other queries, there are different results.

One of the strengths of the rule-based optimizer is that it is concerned about the constitution of a query. The filtering conditions embedded in a specific query will be treated differently according to the heuristic rule presented above. Also, this rule-based query optimizer is quite flexible since the weights assigned to filtering conditions of various types and positions could be tuned to suit different types of graph databases. This optimizer also has the limitation that it ignores the real workload of a graph, e.g. the number of a specific type of vertices, though.

As for the cost-based optimizer, it considers the real workload of a graph dataset. The estimation workloads could be quite precise for simple queries. The bad aspect about it is that a cost-based optimizer will not consider the workload of a specific query, which may lead to an inefficient query plan. Besides, the estimation of a query graph component could be quite tricky if the edge labels are closely related to some certain types of vertices since the estimate presented above depends on the assumption that each value appears with equal probability.

Chapter 7

Conclusions

In this chapter, we summarize our contributions, recognize the limitations of this project and propose topics for future work.

7.1 Our Contributions

In this thesis, we proposed an algebra for Cypher, comprised of graph-specific operators. All Cypher queries could be easily translated to our Cypher algebra. We presented how these basic operators in Cypher algebra were implemented on Apache Flink engine using Flink operators. We also proposed two query optimizers, a cost-based query optimizer and a rule-based query optimizer. Then we conducted experiments to compare the performance of these two optimization strategies.

Here are the conclusions:

1. In most cases, the performance of queries which utilize cost-based query optimization is better that of queries which apply rule-based query optimization;

2. For high selectivity queries, the rule-based query optimizer works better than the cost-based query optimizer;

3. An inefficient query optimization strategy may lead to significant performance degradation for the datasets of a specific edge distribution and the queries of certain shapes;

4. In general, the level of parallelism of Flink would also affect the execution time of all the queries. The speed-ups increase as the parallelism first increases, but then drop down since the communication cost among workers becomes the bottleneck.

7.2 Limitations and Future Work

So far the system that we have implemented just allows a subset of Cypher queries. Especially these two query optimizers could only generate the query plans for queries which are comprised of conjunctive conditions. For those queries with disjunctive conditions or even more complicated queries, the query plans need to be generated manually. Hence, we could in future extend the optimization range of our query optimizers.

As we have said in the discussions in the previous chapter, there are several advantages and disadvantages existing in both query optimization strategies. For a rule-based query optimizer, we assigned different weights to query edges consisting the conditions of various selectivity and types. The results show that for high selectivity queries, the rule-based query optimizer works better than the cost-based one. To some extent, it indicates that the weights assigned do not reflect the real selectivity of conditions embedded in a query vertex or a query edge accurately.

Hence, one of the approaches to improve the performance of this rule-based query optimizer is to tune the weights and make it aligned to the real selectivity of queries.

We also recognize the limitations of the cost-based query optimizer. The cost-based query optimization algorithm does not take complicated conditions into account. Instead, it only makes use of the statistical information related to the number of edges or vertices of a specific label. In other words, the cost-based query optimizer ignores the practical information about the filtering conditions on vertices and edges. Since both two query optimizers have their own strengths and weaknesses, a combination of both estimation approaches can also be a promising execution strategy to investigate.

Also, so far the experiments that we have done are all on a single machine. Executing all the queries in a distributed system will help us understand the behaviours of our implementation and the execution strategies better. Hence, investigating the features of our system in a distributed system could also be our future work.

Bibliography

[1] http://neo4j.com/docs/stable/cypher-introduction.html. [Online; accessed 14-January-2016]. 1, 4, 9

[2] https://ci.apache.org/projects/flink/flink-docs-master/. [Online; accessed 15-January-2016]. 1, 9

[3] https://www.w3.org/RDF/. [Online; accessed 25-June-2016]. 3

[4] https://www.w3.org/TR/rdf-sparql-query/. [Online, accessed 25-June-2016]. 3

[5] http://www.ldbcouncil.org/benchmarks. [Online; accessed 25-June-2016]. 11

[6] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George HL Fletcher, Aurélien Lemay, and Nicky Advokaat. Controlling diversity in benchmarking graph databases. *arXiv preprint arXiv:1511.08386*, 2015. 11

[7] Thorsten Berberich and Georg Lausen. S2x: Graph-parallel querying of rdf with graphx. In *Biomedical Data Management and Graph Online Querying: VLDB 2015 Workshops, Big-O (Q) and DMAH, Waikoloa, HI, USA, August 31–September 4, 2015, Revised Selected Papers*, volume 9579, page 155. Springer, 2016. 5

[8] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630. ACM, 2015. 11

[9] Siziana Maria Filip. A scalable graph pattern matching engine on top of apache giraph. Master's thesis, Vrije Universiteit, 2014. http://homepages.cwi.nl/ boncz/msc/2014-SinzianaFilip.pdf. 5, 8

[10] Andrey Gubichev. *Query Processing and Optimization in Graph Databases.* PhD thesis, München, Technische Universität München, Diss., 2015, 2015. 5

[11] Jürgen Hölsch and Michael Grossniklaus. An algebra and equivalences to transform graph patterns in neo4j. In *EDBT/ICDT 2016 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ)*, 2016. 4

[12] Mohammad Farhan Husain, Pankil Doshi, Latifur Khan, and Bhavani Thuraisingham. Storage and retrieval of large rdf graph using hadoop and mapreduce. In *IEEE International Conference on Cloud Computing*, pages 680–686. Springer, 2009. 5

[13] Martin Junghanns, André Petermann, Kevin Gómez, and Erhard Rahm. Gradoop: Scalable graph data management and analytics with hadoop. *arXiv preprint arXiv:1506.00548*, 2015. 5

[14] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009. 3

[15] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases.* " O'Reilly Media, Inc.", 2013. 3

[16] P. Rutgers. Extending the lighthouse graph engine for shortest path queries. Master's thesis, Vrije Universiteit, 2015. http://homepages.cwi.nl/ boncz/msc/2015-Rutgers.pdf. 5

[17] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2rdf: Rdf querying with sparql on spark. *arXiv preprint arXiv:1512.07021*, 2015. 5

[18] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *SIGMOD. ACM*, 2016. 5

[19] Korth H. F. Silberschatz, A. and S Sudarshan. *Database system concepts (6e ed.).* McGraw-Hill, 2010. 33

[20] B. G. J. Wolff. A framework for query optimization on value-based rdf indexes. Master's thesis, Technische Universiteit Eindhoven, 2013. http://alexandria.tue.nl/extra1/afstversl/wsk-i/wolff2013.pdf. 34

[21] Bart GJ Wolff, George HL Fletcher, and James J Lu. An extensible framework for query optimization on triplet-based rdf stores. In *EDBT/ICDT Workshops*, pages 190–196, 2015. 34

[22] Peter T Wood. Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50–60, 2012. 4

Appendix A

Queries for LDBC-SNB

```
Query 1
MATCH (m:person)
RETURN m

Query 2
MATCH (m) - [:hasCreator] -> (n)
RETURN m

Query 3
MATCH (n:comment) - [] -> (m:person)
RETURN m

Query 4
MATCH (m:person) <- [] - (n:comment)
RETURN m

Query 5
MATCH (m:person) - [:likes] -> (n:comment)
WHERE n.length > 50
RETURN m

Query 6
MATCH (m:person) - [:likes] -> (n:comment)
WHERE n.length > 50 AND n.browserUsed = ‘Chrome’
RETURN m

Query 7
MATCH (m:person) - [:likes] -> (n:comment)
WHERE n.length > 50 OR n.browserUsed = ‘Chrome’
RETURN m

Query 8
MATCH (n:comment) <- [:likes] - (m:person)
WHERE n.length > 50 AND n.browserUsed = ‘Chrome’
RETURN m

Query 9
MATCH (n:comment) - [:hasCreator] -> (m:person) <- [:hasCreator] -> (l:post)
WHERE m.
```

```
RETURN m

Query 10
MATCH (m:post) - [:hasCreator] -> (n:person) <- [:hasCreator] -  (l:comment)
               - [:hasTag] -> (k:Tag)
WHERE l.length >= 150
RETURN n

Query 11
MATCH (m:post) - [:hasCreator] -> (n:person) <- [:hasCreator] - (l:comment)
               - [:hasTag] -> (k:Tag)
WHERE n.lastName = ‘Yang’
RETURN n

Query 12
MATCH (m:post) - [:hasCreator] -> (n:person) <- [:hasCreator] - (l:comment)
               - [:hasTag] -> (k:Tag)
WHERE n.lastName = ‘Yang’ AND n.browserUsed = ‘Safari’
RETURN n

Query 13
MATCH (m:post) - [:hasCreator] -> (n:person) <- [:hasCreator] - (l:comment)
WHERE (n) - [:studyAt] -> (o:organisation) AND
      (n) - [:isLocatedIn] -> (p:place) AND
      l.length >= 150
RETURN n

Query 14
MATCH (m:post) - [:hasCreator] -> (n:person) <- [:hasCreator] - (l:comment)
WHERE (n) - [:studyAt] -> (o:organisation) AND
      (n) - [:isLocatedIn] -> (p:place) AND
      n.lastName = ‘Yang’
RETURN n

Query 15
MATCH (m:post) - [:hasCreator] -> (n:person) <- [:hasCreator] - (l:comment)
WHERE (n) - [:studyAt] -> (o:organisation) AND
      (n) - [:isLocatedIn] -> (p:place) AND
      o.type = ‘company’
RETURN n

Query 16
MATCH (m:post) - [:hasCreator] -> (n:person) <- [:hasCreator] - (l:comment)
WHERE (n) - [:studyAt] -> (o:organisation) AND
      (l) - [:hasTag] -> (t:tag) AND
      (l) - [:isLocatedIn] -> (p:place) AND
  l.length >= 150
RETURN n

Query 17
MATCH (m:post) - [:hasCreator] -> (n:person) <- [:hasCreator] - (l:comment)
WHERE (n) - [:studyAt] -> (o:organisation) AND
      (l) - [:hasTag] -> (t:tag) AND
      (l) - [:isLocatedIn] -> (p:place) AND
      n.lastName = 'Yang'
RETURN n

Query 18
MATCH (m:post) - [:hasCreator] -> (n:person) <- [:hasCreator] - (l:comment)
WHERE (n) - [:studyAt] -> (o:organisation) AND
      (l) - [:hasTag] -> (t:tag) AND
      (l) - [:isLocatedIn] -> (p:place) AND
      o.type = 'company'
RETURN n
```

Appendix B

Queries for gMark

```
Query 19
MATCH (a:Protein) - [:Interacts] -> (b:Protein) - [:EncodedOn] -> (c:Gene)
RETURN a

Query 20
MATCH (a:Protein) - [:Interacts] -> (b:Protein) - [:Reference] ->
      (c:Article) - [:PublishedIn] -> (d:Journal)
RETURN a

Query 21
MATCH (a:Protein) - [:Interacts] -> (b:Protein)
WHERE (a) - [:Reference] -> (c:Article) AND
      (a) - [:EncodedOn] -> (d:Gene) AND
      (a) - [:HasKeyword] -> (e:Keyword)
RETURN a

Query 22
MATCH (a:Protein) - [:Interacts] -> (b:Protein)
WHERE (a) - [:Reference] -> (c:Article) AND
      (a) - [:EncodedOn] -> (d:Gene) AND
      (a) - [:HasKeyword] -> (e:Keyword) AND
      (a) - [:PublishedIn] -> (f:Journal)
RETURN a

Query 23
MATCH (a:Protein) - [:Interacts] -> (b:Protein)
WHERE (a) - [:Reference] -> (c:Article) AND
      (a) - [:EncodedOn] -> (d:Gene) AND
      (b) - [:HasKeyword] -> (e:Keyword) AND
      (b) - [:PublishedIn] -> (f:Journal)
RETURN a

Query 24
MATCH (a:Protein) - [:Interacts] -> (b:Protein)
WHERE (a) - [:Reference] -> (c:Article) AND
      (a) - [:EncodedOn] -> (d:Gene) AND
      (b) - [:HasKeyword] -> (e:Keyword) AND
      (b) - [:PublishedIn] -> (f:Journal) AND
      (b) - [:Interacts] -> (h:Protein)
```

```
RETURN a
```