Temporal and Spatio-temporal Graphs in Neo4j

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

Computer Science Master's Thesis
Year 2022-2023

Graduand Jaber Rahimifard (875545)

Supervisor Prof. Claudio Silvestri

Abstract

Graph databases have gained immense popularity as a leading choice for data representation and analysis, especially when it comes to modeling diverse types of networks. They are constructed using the property graph data model, which involves nodes and edges being valued with property-value combinations. Even though time is present in most real-world problems, the majority of prior research in this field revolves around graphs in which the temporal aspect is overlooked. This thesis describes the model presented in a paper published in The VLDB Journal 30[5] and offers an analysis of the problem of modeling, storing, and querying temporal property graphs, enabling the preservation of a graph database's historical data. Specifically, the thesis focuses on addressing a temporal graph data model, where nodes and relationships contain key-value pair attributes within a defined time interval. In this model, graphs may encompass different kinds of relationships.

Also, the paper introduces a high-level graph query language known as T-GQL, accompanied by a collection of algorithms for computing various types of temporal paths within a graph. These paths capture distinct temporal path semantics, including continuous paths, pairwise continuous paths, and consecutive paths. T-GQL proves to be particularly significant, capable of expressing queries like "Find paths between Anchorage and Los Angeles, taking into account flights where the arrival time precedes the departure time of the subsequent flight."

To validate the feasibility of the concept, a practical demonstration is provided through the utilization of Neo4j. Moreover, a user interface on the client side facilitates the submission of queries written in T-GQL to a Neo4j server.

In addressing the disparity between synthetic data set and real-world complexities, this thesis introduces the pivotal *Price* attribute, reflective of real-world aviation dynamics, into the analysis. It delves into how this variable influences algorithmic outcomes, shedding light on the intricate interplay between price and path selection. While recognizing that real-world aviation presents multifaceted attributes, the primary focus here remains on price and distance. This exploration aims to unravel the impact of price fluctuations on algorithmic results, offering insights into how practical considerations influence path selection. The study's outcomes encompass experiments on a synthetic data set, serving to both demonstrate method viability and assess the impact of variables, such as path length and graph dimensions.

This thesis expansion endeavors to bridge the theory-to-practice gap, offering valuable insights into the real-world dynamics of temporal property graphs. It culminates in a comprehensive set of experiments conducted on a synthetic data set, serving a dual purpose: firstly, to validate the method's feasibility, and secondly, to evaluate the variables influencing performance, including queried path lengths and graph dimensions.

Acknowledgments

I extend my sincere gratitude to my supervisor, Claudio Silvestri, whose remarkable proficiency, insightful perspective, and enduring patience have greatly contributed to the enrichment of my scholarly pursuit. I deeply value her expansive knowledge and multifaceted skills, and his guidance has been of immeasurable significance to my academic maturation.
I wish to express my recognition to the encouraging scholarly community at Ca' Foscari University of Venice. The conducive atmosphere they cultivate has played a pivotal role in the realization of this endeavor, providing an optimal amalgamation of mentorship, autonomy, and academic inspiration.
Last but not least, I want to thank my parents and my friends, who have allowed me to pursue the best education I could ever have gotten and have given me the possibility to stay here in Venice during my academic career. They comforted me with much patience, encouragement and also kind smiles which I will never forget.

Keywords Temporal graph databases , Neo4j , Query languages , Cypher query language , Graph databases

Contents

1 Introduction **1**
1.1 Problem Description . . . 1
1.2 Thesis Goal . . . 1
1.3 Contributions . . . 2
1.4 Outline . . . 3

2 Related Works **4**
2.1 Graph database models . . . 4
2.2 Data models for temporal graphs . . . 4
2.2.1 Duration-labeled temporal graphs . . . 5
2.2.2 Interval-labeled temporal graphs . . . 5
2.2.3 Snapshot-based temporal graphs . . . 6

3 Proposed Model **7**
3.1 Implement Graph . . . 8
3.1.1 Nodes . . . 9
3.1.2 Edges . . . 10
3.2 Queries to Return Paths . . . 10
3.2.1 Continuous Paths . . . 11
3.2.2 Consecutive Paths . . . 12
3.2.3 Consecutive Paths With Respect Price . . . 15
3.3 T-GQL Syntax . . . 18
3.3.1 Basic Statements . . . 18
3.3.2 Continuous Paths Queries . . . 19
3.3.3 Consecutive Paths Queries . . . 20
3.4 Implement Java Functions . . . 21
3.4.1 Creating a User-Defined Procedure . . . 21
3.4.2 Registering the Procedure . . . 21
3.4.3 Using the User-Defined Procedure . . . 21
3.4.4 Integration of Procedures into Neo4j Environment . . . 22

4 Implementation **24**
4.1 Define T-GQL Grammar . . . 24
4.1.1 Grammar Structure . . . 25
4.1.2 Statement Types . . . 25
4.1.3 Identifiers and Paths . . . 25
4.1.4 Matching Patterns . . . 25
4.1.5 Conditions and Boolean Expressions . . . 25
4.1.6 Keywords . . . 25
4.1.7 Whitespace and Comments . . . 25
4.2 Generate Lexers and Parsers . . . 25

4.2.1 Integrate with Neo4j 26
4.2.2 Parsing and query translation 26
4.3 Deploy Web Application 29

5 Experiments 31
5.1 Goal 31
5.2 Data set and Setup 32
5.3 Continuous path algorithms 32
5.4 Consecutive paths algorithms 35
5.5 Consecutive paths with respect price 40
5.6 Evaluation 42
5.6.1 Execution time evaluation for continuous paths 42
5.6.2 Execution time evaluation for consecutive paths 43
5.6.3 Assessment of consecutive paths in terms of pricing 45
5.7 Summary 46

6 Conclusion 48

List of Figures

2.1 a duration-labeled temporal graph (cf.[20]) 5
2.2 Interval-labeled temporal graphs (cf.[5]) 6

3.1 Diagram of the proposed framework 7
3.2 The graph visually represents the interconnected flights connecting cities, where these cities are joined through their respective airports linked by the "Located At" edges 8
3.3 The graph shows continuous paths from the related query 11
3.4 The graph shows pairwise continuous paths from the related query . 12
3.5 Earliest arrival path 13
3.6 Latest departure path 14
3.7 Fastest path 14
3.8 Shortest path 15
3.9 Earliest arrival time with respect the price 16
3.10 Latest departure path with respect the price 16
3.11 Fastest path with respect the price 17
3.12 Shortest path with respect the price 18

4.1 Select the city names "Los Angeles" 26
4.2 Neo4j itself provides and utilizes custom procedures. [12] 29
4.3 Web application to run the custom queries 29

5.1 Time Vs. length for continuous path 33
5.2 Time Vs. length for continuous path with different number of nodes 33
5.3 Time Vs. length for pairwise continuous path 34
5.4 Time Vs. length for pairwise continuous path with different number of nodes 34
5.5 Comparison CPath and PairCPath 35
5.6 Time Vs. length for earliest arrival path 36
5.7 Time Vs. length for latest departure path 37
5.8 Time Vs. length for shortest path 37
5.9 Time Vs. length for fastest path 38
5.10 Time Vs. length for 1000 Nodes 39
5.11 Time Vs. length for 10,000 Nodes 39
5.12 Time Vs. length for 100,000 Nodes 40
5.13 Shows how price impact on the length and distance 42

List of Tables

5.1 Earliest arrival path ordered by price 41
5.2 Earliest arrival path limited by price less than 1000 41
5.3 Latest departure path ordered by length 41
5.4 Shortest path ordered by distance 42

Chapter 1

Introduction

1.1 Problem Description

These days, there has been much use of property graphs[4, 11, 14], especially to analyze and model different kinds of networks. The property graph data model is the base of most graph databases, including Neo4j[1], Apache AGE[2], and RedisGraph[3]. The predominant focus of researchers and practitioners, thus far, has revolved around static graphs, whereby the temporal facet remains unexplored. Nevertheless, there are a lot of examples of applications and graphs in the real world that use time. When using time in property graphs, many changes occurred. Edges, nodes, and their properties can be changed, added, deleted, or updated. For instance, as described in the paper[5]:
In transportation schedules, each vertex in a graph represents a location, and an edge (u, v, t, λ) is a trip (flight, bus, etc.) from u to v departing at time t, whose duration is λ.

1.2 Thesis Goal

Neglecting the temporal aspect might yield inaccurate outcomes or hinder the exploration of intriguing analytical avenues. Therefore, the primary objective of this dissertation is to expand upon the approach outlined in the paper [5], which introduces incorporates time intervals as a fundamental component of the undertaking, enabling the establishment of connections and paths among nodes that lack a direct link. This component is accomplished through an assessment of edge properties and their comparison, thereby identifying conditions that facilitate the continuation of the path.
To facilitate the restoration process, three distinct approaches are adopted for path detection, following the methodologies introduced in the referenced paper. During the path detection process,

- The initial strategy employs continuous pathways, devoid of property consideration and conditions, resulting in the retrieval of all pathways connecting the two nodes.

[1] `https://neo4j.com/`
[2] `https://age.apache.org/`
[3] `https://redis.com/`

- The second approach which is pairwise continuous paths harnesses the capabilities of properties of the edges adept at identifying and precisely locating a diverse path within a graph.

- Ultimately, consecutive pathways encompass the shortest path, fastest path, earliest arrival path, and latest departure path. Similar to pairwise continuous pathways, these also incorporate edge properties, but with additional constraints to ascertain cumulative duration and distance, which are subsequently employed as determining factors.

In the context of evaluating algorithm performance, it becomes imperative to transition from surveying these algorithms' inherent capabilities to exploring how they respond when confronted with real-world features. In particular, the inclusion of dynamic elements from the aviation industry, such as pricing dynamics, represents a pivotal step in this analysis. This exploration aims to uncover the intricate dynamics at play when practical, real-world considerations are integrated into algorithmic decision-making processes, providing valuable insights into how these elements shape path selection and overall outcomes within the aviation domain.

1.3 Contributions

This research delves into the application of temporal database principles in graph databases, with a focus on enabling the modeling, storage, and querying of temporal graphs, preserving historical data. The research centers on the property graph data model[4, 11, 14], associating timestamps with nodes, relationships, and node properties to indicate their temporal validity intervals, all within heterogeneous graphs. These graphs are termed Interval-labeled Property Graphs, as defined in a paper [5]. The contributions of this thesis encompass the integration of the concepts from the referenced paper and the introduction of new elements to extend this work.

This includes a temporal data model specifically designed for property graphs, permitting the retention of historical records for nodes, edges, and associated properties. The development of T-GQL, a sophisticated graph query language built upon GQL, the established standard for property graph databases. A set of queries is introduced for computing diverse temporal paths within graphs, accommodating various interpretations of temporal path semantics. The implementation is realized using Neo4j, and a client interface is established to query Neo4j graphs. A series of experiments were conducted to evaluate the practical application of the semantics explored in this study. These experiments involved synthetic data sets simulating flights connecting different airports. Furthermore, this research delves into the pragmatic aspects of temporal graph databases by exploring the substantial impact of flight costs and edge expenses within the temporal graph data model. It uncovers how these real-world economic factors intricately influence the length and spatial dimensions of journeys. This practical examination serves to bridge the gap between theoretical modeling and real-world complexities, offering valuable insights into how the dynamics of cost and journey length play a pivotal role in shaping the characteristics of temporal graphs.

1.4 Outline

The initial section of this thesis, based on the paper [5], presents an overview of the research problem and its objectives. Section 2 offers an extensive examination of the existing literature and prior research in the realm of temporal relational databases, laying the groundwork for the present study. Proceeding from this foundation, Section 3 delves into the three-stage framework for temporal graph databases. The first phase involves the implementation of a graph structure and the execution of various queries within the Neo4j environment. In the second phase, the focus shifts to the integration of T-GQL syntax, enabling the transformation and translation of diverse query types into executable commands, extending the capabilities of Cypher queries. The third phase concentrates on the development of Java functions for each query, facilitating their migration to the server-side for execution.

In the subsequent section, *Implementation,* the procedural details encompass the comprehensive definition of each component of the T-GQL grammar, including the generation of Lexers and Parsers integrated with Neo4j. Ultimately, this section culminates in the deployment of a web application that enables the execution of extended Cypher queries within the Neo4j environment.

The concluding portion, *Experiments,* provides insights into the research's objectives, the utilized data set, the adopted evaluation metrics, and the experimental outcomes derived from the methodologies outlined in the "Proposed Model" chapter.

Finally, Section 6 encapsulates the thesis by summarizing its contributions and the broader implications of the research.

Chapter 2

Related Works

The field of temporal relational databases has seen significant research and study over the years[15, 19]. This study focuses on managing and querying data with temporal aspects, such as time-stamped records, historical data, and evolving relationships between entities. Researchers and practitioners in this field have explored various issues and challenges, leading to a rich body of literature.

2.1 Graph database models

A substantial collection of references concerning graph database models has been thoroughly explored in[18, 2]. In practical real-world applications, two types of graph database models are employed:

- Models based on RDF[1], oriented to the Semantic Web.
- Models based on Property Graphs.

In RDF[10, 9], data is represented in triple format, where each triple comprises three elements referred to as the subject, predicate, and object, delineating their characteristics and connections with other entities.
Within the *property graph* data model[1, 3], nodes and edges are tagged with sets of (attribute, value) pairs. Property graphs go beyond conventional graph structures and are the typical selection for contemporary graph databases employed in practical applications. As the research problem explored in the paper referenced as [5] is rooted in the property graph model, the forthcoming review exclusively focuses on this particular graph data model.

2.2 Data models for temporal graphs

Data models[5] in the temporal graphs literature can be classified in three groups:

1. Duration-labeled temporal graphs (DLTG)
2. Interval-labeled temporal graphs (ILTG)
3. Snapshot-based temporal graphs (SBTG)

[1] `https://www.w3.org/RDF/`

2.2.1 Duration-labeled temporal graphs

Wu et al.[20]. conducted research on these types of graphs. In these graphs, a node is depicted as a string, and the edges are tagged with a value denoting the length of the connection between two nodes. Definition 1 provides a formal elucidation of the previously mentioned concept.

Definition 1 (*Duration-labeled graphs (cf.[20])*)**.** Let $G_d = (V, E)$ be a temporal graph, where V is the set of vertices, and E is the set of edges in G.

- Each edge $e = (u, v, t, \lambda) \in E$ is a temporal edge representing a relationship from a vertex u to another vertex v starting at time t, with a duration λ. For any two temporal edges (u, v, t_1, λ_1) and (u, v, t_2, λ_2), $t_1 \leq t_2$.
- Each node $v \in V$ is active when there is a temporal edge that starts or ends at v.
- $d(u, v)$: the number of temporal edges from u to v in G_d.
- $E(u, v)$: the set of temporal edges from u to v in G, i.e., $E(u, v) = \{(u, v, t_1), (u, v, t_2), \ldots, (u, v, t_{d(u,v)})\}$.
- $N_{\text{out}}(v)$ or $N_{\text{in}}(v)$: the set of out-neighbors or in-neighbors of v in G_d, i.e., $N_{\text{out}}(v) = \{u : (v, u, t) \in E\}$ and $N_{\text{in}}(v) = \{u : (u, v, t) \in E\}$.
- $d_{\text{out}}(v)$ or $d_{\text{in}}(v)$: the temporal out-degree or in-degree of $v \in G_d$, $d_{\text{out}}(v) = \sum_{u \in N_{\text{out}}(v)} d(v, u)$ and $d_{\text{in}}(v) = \sum_{u \in N_{\text{in}}(v)} d(u, v)$.

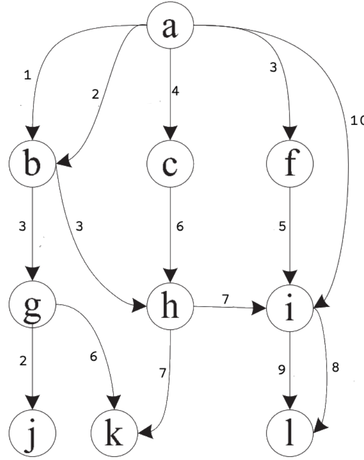

Figure 2.1: a duration-labeled temporal graph (cf.[20])

2.2.2 Interval-labeled temporal graphs

Definition 2 provided below defines the characteristics of interval-labeled temporal graphs (ILTG).

Definition 2 (*Interval-labeled temporal graphs (cf. [5])*)**.** Let $G_d = (V, E)$ be a temporal graph, where V is the set of vertices, and E is the set of edges in G.

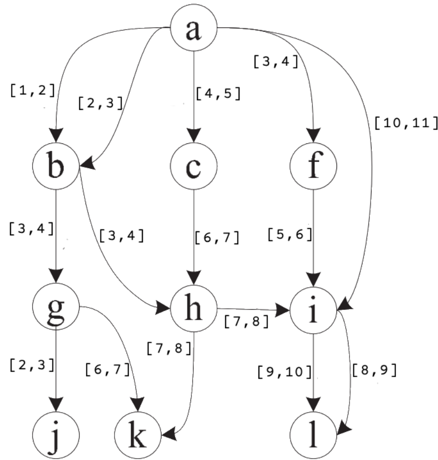

Figure 2.2: Interval-labeled temporal graphs (cf.[5])

The graph displays edges that are marked with their validity intervals, rather than using a timestamp to indicate a duration. As an instance, consider the edge connecting nodes a and i which is marked with the interval [10, 11]. This variation arises because, in the *Duration-labeled temporal graph*[2.2.1] depicted in the same figure, the same edge is labeled as 10, indicating the initial time of the edge, with a duration of 1. In practical terms, if the graph represents a flight schedule, it signifies that the flight departs from a at time instant 10, and the journey from a to i lasts for a single time unit.

2.2.3 Snapshot-based temporal graphs

Huo and Tsotras[13] investigated the efficient calculation of shortest paths in evolving social networks. They introduce the concept of a temporal graph, which starts as an initial snapshot and is then updated over time. To tackle this, they extend the conventional Dijkstra's algorithm[6] to calculate the shortest path distance(s) for a specific time point or time interval within the evolving history of a social network. Consequently, temporal queries are made by referencing particular historical snapshots of the graph.

Definition 3 (*Snapshot temporal graph (cf.[17])*)**.** A temporal graph $G[ti, tj]$ in a time interval $[ti, tj]$ is a sequence $\{G_{ti}, G_{ti+1}, \ldots, G_{tj}\}$ of graph snapshots.

For example, temporal shortest-path queries in a flight can discover how close two given cities were in the past and how their closeness evolved over time. Finally, several different kinds of path queries are defined. For example, *a time point shortest path query* returns the *shortest-path* p from a origin city v_s to a destination city v_t, such that both are temporally valid at query time t_q (all edges in p are valid at query time t_q).

Chapter 3

Proposed Model

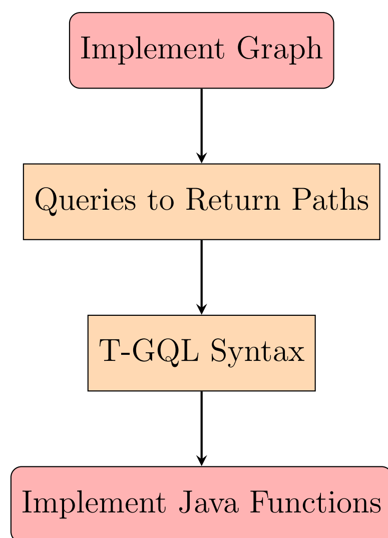

Figure 3.1: Diagram of the proposed framework

This thesis presents a method introduced in the paper [5], which primarily focuses on enhancing temporal graph databases and refining the precision of Cypher queries within Neo4j. In this thesis, we endeavor to implement and further extend the method outlined in the paper [5]. The entire procedure can be visualized as a sequence of stages within our framework. These elements collectively form a circumstance which can be further elaborated upon in the next section, defining a well-structured grammar. The figure 3.1 is a flow diagram illustration of our framework. The initial phase of framework involves the implementation of nodes and edges. This crucial step is fundamental in constructing a graph that can effectively manage and utilize various types of paths in the future. These path types encompass continuous paths, pairwise continuous paths, shortest paths, fastest paths, earliest arrival paths, and latest departure paths.

The second phase enables us to identify available paths between connections by formulating queries. These queries involve specifying the names of the initial and final nodes, along with the connecting edge. This phase is dedicated to path discovery. In the third phase, T-GQL, as introduced in the reference paper [5], is presented as a high-level query language for graph databases. This phase provides an in-depth understanding of T-GQL syntax and query structure through practical examples. In the Last phase, the model and language outlined were put into practice using the Neo4j graph database, an open-source Java-based solution. Neo4j[1] facilitates the expansion of its capabilities through user-defined procedures, which can be conveniently incorporated as plugins within a .jar file. These procedures can subsequently be employed in Cypher queries much like any of the other native functions provided by this language.

[1] https://neo4j.com/

3.1 Implement Graph

In the pursuit of achieving visual restoration, the initial and pivotal phase involves the implementation of the property graph by adding nodes and edges. The property graph is a graph that is implemented by the nodes and edges, where a collection of property-value pairs is held by them.

Definition 4 (*Temporal property graph (cf. [5])*)**.** A *temporal property graph* is a structure $G(N_o, N_a, N_v, E)$ where G is the name of the graph, E is a set of edges, and N_o, N_a, and N_v are sets of nodes, denoted as object nodes, attribute nodes, and value nodes, respectively.

The figure[3.2] shows the graph with *City* and *Airport* as nodes, with edges representing *Flight* and *LocatedAt*. The interconnections between airports are established using the *Flight* edge, while each airport is linked to its respective home city through the *LocatedAt* edge. As an illustration:

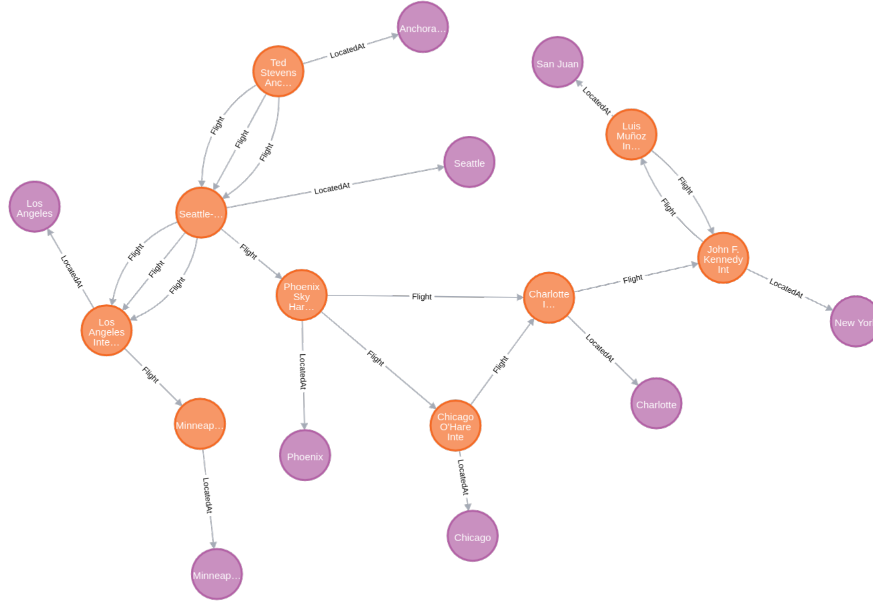

Figure 3.2: The graph visually represents the interconnected flights connecting cities, where these cities are joined through their respective airports linked by the "Located At" edges

Before introducing the constraints, it is essential to mention the concept of the *Lifespan of an edge*, which we will formally delineate in Definition 6.

Definition 5 (*Constraints* cf. [5]))**.** For the graph in Definition 4, the following constraints hold:

1. $\forall n, n' \in N_o, n = n' \vee n.\text{id} \neq n'.\text{id}$
2. $\forall n, n' \in N_a, n = n' \vee n.\text{id} \neq n'.\text{id}$
3. $\forall n, n' \in N_v, n = n' \vee n.\text{id} \neq n'.\text{id}$
4. $\forall n_v\{n_a\}, n'_v\{n_a\} \in N_v, n_v = n'_v \vee n_v.\text{value} \neq n'_v.\text{value}$
5. $\forall e_i\{n, n'\}, e_j\{n, n'\} \in E \wedge e_i.\text{name} = e_j.\text{name}, e_i = e_j \vee e_i.\text{name} \neq e_j.\text{name}$
6. $\forall n \in N_o, e\{n, n'\} \in E \Rightarrow n' \in N_o \cup N_a$

7. $\forall n \in N_a, e\{n, n'\} \in E \Rightarrow n' \in N_o \cup N_v$
8. $\forall n \in N_v, e\{n, n'\} \in E \Rightarrow n' \in N_v$
9. $\forall n \in N_a, (\exists n_o \in N_o, \exists e \in E(e(n_o, n) \wedge (\nexists n' \in (N_a \cup N_v \cup N_o) \wedge e' \in E \wedge e'\{n', n\})))$
10. $\forall n \in N_v, (e\{n', n\} \wedge n' \in N_a) \Rightarrow \nexists n'' \in (N_a \cup N_v \cup N_o)(e''\{n'', n\} \in E \vee e''\{n, n''\} \in E)$
11. $\forall n_e\{n, n'\} \in N_e, n_e.\text{interval} \subset n.\text{interval} \cap n'.\text{interval}$
12. $\forall n_a\{n\} \in N_a, n_a.\text{interval} \subset n.\text{interval}$
13. $\forall n_v\{n_a\} \in N_v, n_v.\text{interval} \subset n_v.\text{interval}$
14. $\forall n_v\{n_a\}, n'_v\{n'_a\}, n_v \neq n'_v, n_v.\text{interval} \cap n'_v.\text{interval} = \emptyset$

Constraints 1 to 3 ensure that each node within the graph possesses a unique identifier. Constraint 4 necessitates the consolidation of nodes sharing the same value linked to the same attribute node. Consequently, the interval transforms into a temporal entity encompassing all time periods during which the node held such a value. Similarly, Constraint 5 applies to edges: it consolidates all edges with the same name (i.e., representing identical relationship types) between identical node pairs.
Constraints 6 to 8 govern the connections between nodes as follows: (a) Object nodes can solely connect to attribute nodes or other object nodes; (b) Attribute nodes can exclusively connect to non-attribute nodes; and (c) Value nodes can only establish connections with attribute nodes. Constraints 9 and 10 specify the cardinalities of these connections. Specifically, attribute nodes must be linked to an object node by just one edge, while value nodes should have a single edge connecting them to an attribute node.
Lastly, Constraints 11 to 14 impose limitations on the values of the interval property.

3.1.1 Nodes

In Neo4j, nodes are fundamental entities that serve as the core building blocks of a graph database. Each node corresponds to a distinct entity or concept and is identified by a unique label. When creating nodes within Neo4j, it's imperative to approach them as objects, mirroring real-world entities, and meticulously assign attributes to encapsulate pertinent information.
For instance, nodes representing "City" entities should be endowed with attributes such as "name," "state," and "country," providing contextual details about the city's location. Similarly, nodes representing "Airport" entities must possess attributes like "name" and "iata code," offering essential information about the airport.
By adhering to this meticulous approach of assigning attributes, the database ensures that the nodes effectively capture the nuances of their real-world counterparts. This careful definition and skillful utilization of attributes guarantee the nodes' efficient integration and utilization within the graph, empowering data retrieval, analysis, and exploration. As an illustration, The nodes *City* and *Airport* are defined like below:

```
CREATE (:City {name: 'Anchorage', state: 'AK', country: 'USA'});
CREATE (:Airport {iata_code: 'LAX', name: 'Los Angeles Airport'});
```

3.1.2 Edges

Edges within the Neo4j graph database serve as pivotal bridges between nodes, forming the very essence of relationships and interactions.

Definition 6 (*Lifespan of an edge (cf. [5])*). Consider a node n, and a collection of k edges outgoing from n, E_{out_i}, $i = 1, \ldots, k$, such that Out_i.name is the same for all E_{out_i}. Also, let E_{in_j}, $j = 1, \ldots, m$, be the set of m edges with the same name incoming to node n. The union of the temporal labels of all these edges is called the *lifespan* of n, denoted $l(n)$.

Two specific edge types hold profound significance: the *Flight* edge and the *LocatedAt* edge. The *Flight* edge stands as a vital conduit, intricately binding two distinct *Airport* nodes. This connection masterfully illustrates a direct or indirect route linking the respective airports. Imbued within the *Flight* edge are essential attributes that enrich its meaning – *departure time, arrival time, distance, and price.* These four attributes collectively provide a comprehensive picture of air travel and costs, encompassing not only connectivity but also the temporal, spatial, and financial dimensions of flights. The *Flight* edge becomes a repository of flight-specific information, enriching the graph with insights into the durations, costs, and timings of these aerial journeys.
In contrast, the *LocatedAt* edge establishes a profound correlation between an *Airport* node and its corresponding *City* node. Through this vital linkage, a spatial narrative unfolds, revealing the precise geographic placement of the airport within the context of its home city. The *LocatedAt* edge fuses the urban and aeronautical realms, providing a vivid sense of place within the graph.
Both of these edge types operate as keystones in the architectural fabric of the graph, expertly molding the intricate tapestry of connections and interactions between nodes. These edges provide a rich foundation for diverse analyses and inquiries, empowering the database to unveil hidden relationships, pathways, and insights lying dormant within the data. The following queries, as an example, implements two different edges, namely, *Flight* and *LocatedAt,* to connect the Anchorage City to the corresponding Airport node and to create a flight from Anchorage to Seattle by matching their IATA codes, respectively.

```
MATCH (a:City) , (b:Airport)
WHERE a.name= 'Anchorage' AND b.iata_code='ANC'
CREATE (a)<-[:LocatedAt]-(b);

MATCH (a:Airport),(b:Airport)
WHERE a.iata_code='ANC' AND b.iata_code='SEA'
CREATE (a)-[e:Flight{departure_time:425, arrival_time:450, distance:
    1448}]->(b);
```

Therefore, when creating a graph with a substantial number of nodes and edges, we can either add them individually or import them from a data set.

3.2 Queries to Return Paths

During this phase, once the node and edge graph is set up, create queries to extract the paths needed. These queries should aid in effectively retrieving available paths.

In this specific scenario, the primary focus is on identifying continuous and consecutive paths. Within the consecutive path context, beside other crucial attributes like *arrival time*, *departure time*, and *distance*, *price* serves as a crucial condition that selection of these paths, potentially impacting their length. The following section illustrates the creation of these queries using an example where the origin city is "*Anchorage* and the destination is *Los Angeles*. You can formulate these queries using the fundamental grammatical structure in Neo4j.

3.2.1 Continuous Paths

Continuous Paths

In the context of ILTG, a continuous path [16], as introduced in Definition 7 , denotes a path that maintains its validity without interruption within the specified time interval.

Definition 7 (*Continuous path*)**.** A continuous path (cp) with interval T from node n_1 to node n_k in a temporal document graph is a sequence $(n_1, \ldots, n_k, T)$ of k nodes and an interval T such that there is a sequence of containment edges of the form $e_1(n_1, n_2, T_1), e_2(n_2, n_3, T_2), \ldots, e_k(n_{k-1}, n_k, T_k)$ such that $T = \bigcap_{i=1,k} T_i$.

As an example, The query retrieves all paths between the cities *Seattle* and *Charlotte* without considering any specific conditions or properties, as long as there is a connection between these two cities in the graph.

```
MATCH (c:City {name: 'Seattle'})<-[:LocatedAt*]-(a:Airport)
MATCH path = (a:Airport)-[e:Flight*]->(b:Airport)
MATCH (b:Airport)-[:LocatedAt*]->(c1:City {name: 'Charlotte'})
WITH c, a, b, c1, relationships(path) AS edges
RETURN edges
```

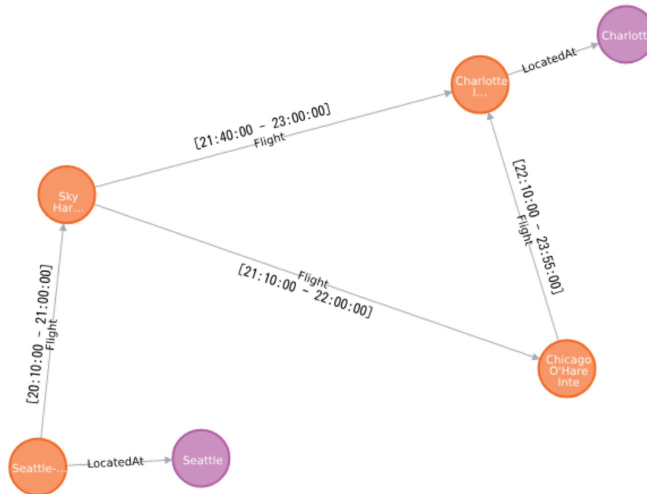

Figure 3.3: The graph shows continuous paths from the related query

Pairwise Continuous Path

In the context of temporal graphs, a pairwise continuous path refers to a sequence of edges or relationships between nodes, where each edge represents a temporal relationship with specified time intervals. The key characteristic of a pairwise

continuous path is that there is an intersection or overlap in the time intervals between consecutive edges in the sequence. This allows for a chain of pairwise temporal relationships between nodes, even if there isn't a continuous path between the nodes. Essentially, it represents a sequence of relationships where the time intervals of each relationship partially overlap with the time intervals of the next, enabling transitive connections through time.

Definition 8 (*Pairwise continuous path[5]*)**.** Given a temporal property graph G, a pairwise continuous path between two nodes n_1, n_k, through a relationship r, is a sequence of edges $e_1(n_1, n_2, r, [t_{s_1}, t_{f_1}]), \ldots, e_k(n_{k-1}, n_k, [ts_{k-1}, t_{f_k}])$, such that $(ts_1 \leq ts_2 \leq tf_1 \vee t_{s_2} \leq t_{f_1} \leq t_{f_2}) \wedge \ldots \wedge (t_{s_k-1} \leq t_{s_k} \leq t_{f_k-1} \vee t_{s_k} \leq t_{f_k-1} \leq t_{f_k})$.

For example, obtaining all paths between the cities of 'Anchorage' and 'Los Angeles' entails ensuring that the time interval between each pair connected cities in a sequence of cities is smaller than that of the next pair connected cities.

```
MATCH (c:City {name: 'Anchorage'})<-[:LocatedAt*]-(a:Airport)
MATCH path = (a:Airport)-[e:Flight*]->(b:Airport)
MATCH (b:Airport)-[:LocatedAt*]->(c1:City {name: 'Los Angeles'})
WITH c, a, b, c1, relationships(path) AS edges
WHERE all(i in range(0, size(edges)-1)
    WHERE edges[i-1].arrival_time < edges[i].departure_time)
RETURN edges
```

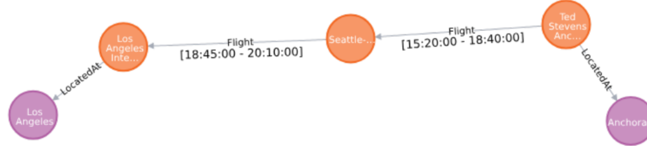

Figure 3.4: The graph shows pairwise continuous paths from the related query

3.2.2 Consecutive Paths

DLTG[2.2.1] can also be represented as ILTG[2.2.2]. Sometimes, queries ask for the earliest, latest, fastest, and shortest path in DLTG. It requires a different temporal graph than Sects 3.2.1 and 3.2.1. The following definition introduces the notion of a *consecutive path*.

Definition 9 (*Consecutive path*[5])**.** A *consecutive path* P_c traversing a relationship r in a temporal property graph G is a sequence of edges $P = (e_1, e_2, r, [t_1, t_2])$, $\ldots, (e_{k-1}, e_k, r, [t_{k-1}, t_k])$ where $(n_i, n_{i+1}, r, [t_i, t_{i+1}])$ is the ith temporal edge in P for $1 \leq i \leq k$, and $t_{i-1} < t_i$ for $1 \leq i \leq k$. Instant t_k is the ending time of P, denoted as $end(P)$, and t_1 is the starting time of P, denoted as $start(P)$. The duration of P is defined as $dura(P) = end(P) - start(P)$, and the distance of P as $dist(P) = k$.

Regarding consecutive paths, these paths resemble continuous paths due to their adherence to the condition that the arrival time at the origin city must exceed the departure time at the destination. Additionally, for each individual path, the interpretations of *first arrival path, last departure path, shortest path,* and *fastest path* vary. Additionally, we place significant importance on considering *price* as a key factor in our path selection process, as it enables us to better comprehend how *price* can impact the length and distance of the paths we identify. These distinctions are explained in the pertinent section.

Definition 10 (*Types of Consecutive Paths*[20]). Let G be a temporal property graph, r a relationship in G, a source node n_s, and a target node n_t, both in G; There is also a time interval $[t_s, t_e]$. Let $P(n_s, n_t, r, [t_s, t_e]) = \{P \mid P \text{ is a consecutive path from } x \text{ to } y \text{ such that } start(P) \geq t_s, end(P) \leq t_e\}$. The following paths can be defined:

- The earliest-arrival path (EAP) is the path that can be completed in a given interval such that the ending time of the path is minimum. Formally, $\textit{EAP: } P \in P(n_s, n_t, r, [t_s, t_e])$ such that $end(P) = \min\{end(P') : P' \in P(n_s, n_t, r, [t_s, t_e])\}$
- The latest-departure path (LDP) is the path that can be completed in a given interval such that the starting time of the path is maximum. Formally, $\textit{LDP: } P \in P(x, y, [t_s, t_e])$ such that $start(P) = \max\{start(P') : P' \in P(n_s, n_t, r, [t_s, t_e])\}$
- The fastest (FP) is the path that can be completed in a given interval such that its duration is minimum. Formally, $\textit{FP: } P \in P(n_s, n_t, r, [t_s, t_e])$ such that $dura(P) = \min\{dura(P') : P' \in P(n_s, n_t, r, [t_s, t_e])\}$
- The shortest path (SP) is the path that can be completed in a given interval such that its length is minimum. Formally, $\textit{SP: } P \in P(n_s, n_t, r, [t_s, t_e])$ such that $dist(P) = \min\{dist(P') : P' \in P(n_s, n_t, r, [t_s, t_e])\}$

Earliest Arrival Path

Earliest Arrival path or *EAP* Examining all paths connecting the cities *Anchorage* and *Los Angeles*, the analysis factors in the stipulation that among the possible paths, the earliest arrival path is determined by the one that reaches the destination with the minimum end time.

```
MATCH (c:City {name: 'Anchorage'})<-[:LocatedAt*]-(a:Airport)
MATCH path = (a:Airport)-[e:Flight*]->(b:Airport)
MATCH (b:Airport)-[:LocatedAt*]->(c1:City {name: 'Los Angeles'})
WITH c, a, b, c1, relationships(path) AS edges
WHERE all(i in range(0, size(edges)-1)
    WHERE edges[i-1].arrival_time > edges[i].departure_time)
RETURN edges, edges[-1].arrival_time AS earliestArrivalTime
ORDER BY earliestArrivalTime ASC
LIMIT 1
```

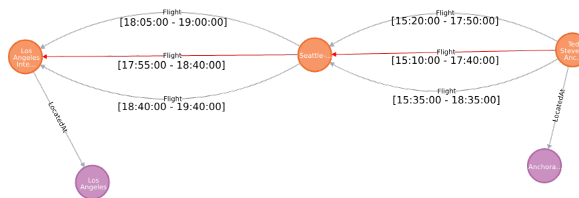

Figure 3.5: Earliest arrival path

Latest Departure Path

When we look at all the ways connecting *Anchorage* and *Los Angeles*, we find that the analysis considers a rule. This rule says that out of all the possible ways, the one where the starting city (Anchorage) has the latest time of leaving is called the *Latest Departure Path.*

```
MATCH (c:City {name: 'Anchorage'})<-[:LocatedAt*]-(a:Airport)
MATCH path = (a:Airport)-[e:Flight*]->(b:Airport)
MATCH (b:Airport)-[:LocatedAt*]->(c1:City {name: 'Los Angeles'})
WITH c, a, b, c1, relationships(path) AS edges
WHERE all(i in range(1, size(edges) - 1)
    WHERE edges[i-1].arrival_time > edges[i].departure_time)
WITH edges, edges[0].departure_time AS lastDepartureTime
ORDER BY lastDepartureTime DESC
RETURN edges
```

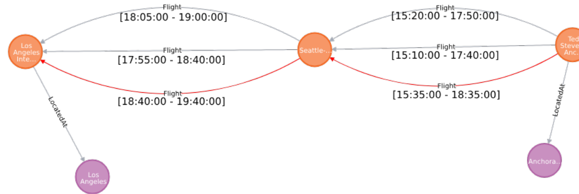

Figure 3.6: Latest departure path

Fastest Path

When we study all the paths connecting "Anchorage" and "Los Angeles," the analysis takes into account a rule. This rule states that among all the options for paths, the quickest path is the one that gets to the destination in the least amount of time.

```
MATCH (c:City {name: 'Anchorage'})<-[:LocatedAt*]-(a:Airport)
MATCH path = (a:Airport)-[e:Flight*]->(b:Airport)
MATCH (b:Airport)-[:LocatedAt*]->(c1:City {name: 'Los Angeles'})
WITH c, a, b, c1, relationships(path) AS edges
WHERE all(i in range(0, size(edges)-1)
    WHERE edges[i-1].arrival_time > edges[i].departure_time)
WITH edges, reduce(duration = 0, edge in edges | duration
    + (edge.arrival_time - edge.departure_time)) AS TotalDuration
ORDER BY TotalDuration ASC
RETURN edges, TotalDuration
```

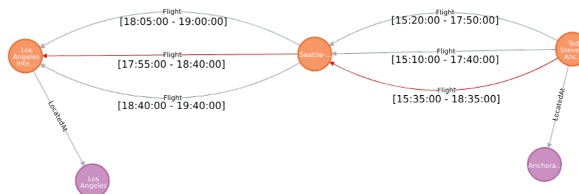

Figure 3.7: Fastest path

Shortest Path

Getting all the paths connecting "Anchorage" and "Los Angeles" considers the condition that the arrival time at the beginning city must be after the departure time of the following city within the path sequence. Additionally, the path's distance between the two cities is kept as small as possible, making it the shortest path.

```
MATCH (c:City {name: 'Anchorage'})<-[:LocatedAt*]-(a:Airport)
MATCH path = (a:Airport)-[e:Flight*]->(b:Airport)
```

```
MATCH (b:Airport)-[:LocatedAt*]->(c1:City {name: 'Los Angeles'})
WITH c, a, b, c1, relationships(path) AS edges
WHERE all(i in range(0, size(edges)-1)
    WHERE edges[i-1].arrival_time > edges[i].departure_time)
WITH edges, reduce(distance = 0, edge in edges | distance
    + (edge.distance)) AS totalDistance
ORDER BY totalDistance ASC
RETURN edges, totalDistance
```

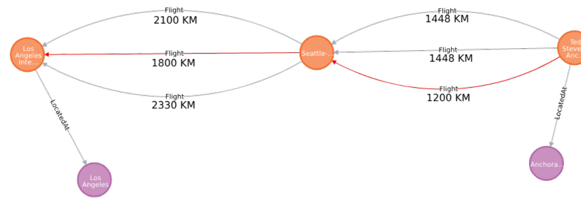

Figure 3.8: Shortest path

3.2.3 Consecutive Paths With Respect Price

In the context of examining all paths connecting, two specific cities, for example, *Anchorage* and *Los Angeles*, we introduce the element of *price* as a significant consideration, recognizing its paramount importance for travelers seeking the most suitable path. Let's explore a scenario where price becomes a pivotal factor in the path selection process:

Imagine a traveler who values not only reaching their destination quickly but also keeping travel expenses to a minimum. In the traditional analysis, the focus has primarily been on *arrival time, departure time, and distance*. However, in this updated scenario, the traveler's decision-making process now includes *price* as a crucial determinant.

The traveler is presented with multiple path options, each with varying *prices and corresponding travel duration*. One path offers a direct route with the *fastest path* but comes at a relatively high cost. Another path involves a brief layover, slightly extending the travel time but significantly reducing expenses.

Incorporating *price* into the path analysis means that the traveler's decision is no longer solely based on *arrival time* and *distance*. Now, they must strike a balance between reaching their destination in a reasonable time and keeping their budget in check. Therefore, they might opt for a path that allows them to save on expenses, even if it means a slightly longer journey. This scenario illustrates how the inclusion of *price* as a crucial factor can greatly influence the choice of path, providing a more comprehensive approach for travelers who prioritize both time and cost efficiency.

In the following scenario, I will illustrate how costs can significantly influence the choice of path when connecting the cities of Anchorage and Los Angeles. Let's consider a traveler who is planning a trip from Anchorage to Los Angeles. She's a budget-conscious traveler, so the cost of journey plays a vital role in her decision-making process. Traveler has different distinct options to choose:

Earliest arrival path with respect the price

If we consider price in the context of the *earliest arrival* path, we can observe that, although the red path is the earliest arrival option in the scenario, the blue route proves to be the more economical choice.

```
MATCH (c:City {name: 'Anchorage'})<-[:LocatedAt*]-(a:Airport)
MATCH path = (a:Airport)-[e:Flight*]->(b:Airport)
MATCH (b:Airport)-[:LocatedAt*]->(c1:City {name: 'Seattle'})
WITH c, a, b, c1, relationships(path) AS edges
WHERE all(i in range(0, size(edges)-1)
    WHERE edges[i-1].arrival_time > edges[i].departure_time)
WITH edges, edges[-1].arrival_time AS earliestArrivalTime,
    reduce(price = 0, edge in edges | price + edge.price)
    AS TotalPrice
ORDER BY TotalPrice ASC
RETURN edges, earliestArrivalTime, TotalPrice;
```

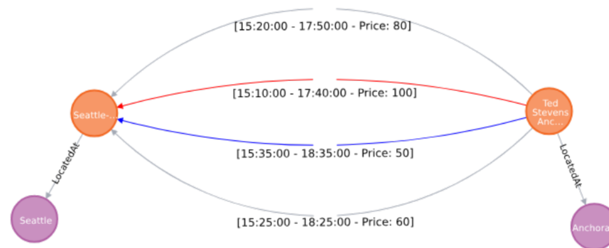

Figure 3.9: Earliest arrival time with respect the price

Latest departure path with respect the price

While the *latest departure path* equals budget-friendly travel, it's important to note that the duration of the flight in this path is longer compared to the other options. This trade-off between cost and time is a crucial consideration for travelers seeking the most suitable route.

```
MATCH (c:City {name: 'Anchorage'})<-[:LocatedAt*]-(a:Airport)
MATCH path = (a:Airport)-[e:Flight*]->(b:Airport)
MATCH (b:Airport)-[:LocatedAt*]->(c1:City {name: 'Seattle'})
WITH c, a, b, c1, relationships(path) AS edges
WHERE all(i in range(0, size(edges)-1)
    WHERE edges[i-1].arrival_time > edges[i].departure_time)
WITH edges, edges[0].departure_time AS latestDepartureTime,
    reduce(price = 0, edge in edges | price + edge.price)
    AS TotalPrice
ORDER BY TotalPrice ASC
RETURN edges, latestDepartureTime, TotalPrice;
```

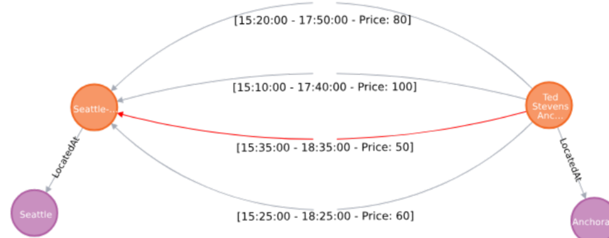

Figure 3.10: Latest departure path with respect the price

Fastest path with respect the price

In the context of selecting the *fastest path* when price falls within the regular, neither cheap nor expensive range, and when the primary focus is on optimizing

duration, travelers find themselves in a scenario where they prioritize efficiency over extreme budget considerations. In this situation, the chosen path may not necessarily align with the *latest departure path* or the *earliest arrival path*, as the primary objective is to reach the destination with the best time efficiency. Such a path could involve a direct flight, an efficient layover all aimed at minimizing travel time. This scenario underscores the significance of striking a balance between cost and speed, offering travelers a compelling alternative when neither the latest departure nor the earliest arrival time is the top priority.

```
MATCH (c:City {name: 'Anchorage'})<-[:LocatedAt*]-(a:Airport)
MATCH path = (a:Airport)-[e:Flight*]->(b:Airport)
MATCH (b:Airport)-[:LocatedAt*]->(c1:City {name: 'Seattle'})
WITH c, a, b, c1, relationships(path) AS edges
WHERE all(i in range(0, size(edges)-1)
    WHERE edges[i-1].arrival_time > edges[i].departure_time)
WITH edges, reduce(duration = 0, edge in edges | duration +
    (edge.arrival_time - edge.departure_time)) AS TotalDuration,
    reduce(price = 0, edge in edges | price + edge.price)
    AS TotalPrice
ORDER BY TotalPrice ASC
RETURN edges, TotalDuration, TotalPrice;
```

Figure 3.11: Fastest path with respect the price

Shortest path with respect the price

In our example, where we have four different paths connecting two specific cities, the concept of the shortest path introduces a unique perspective on path selection. The shortest path is typically determined by the number of edges, which reflects the fewest intermediate stops or connections between the two cities. In this scenario, the *shortest path* is depicted by the red line, signifying the most direct route with the fewest stops.

However, it's crucial to recognize that while the red path represents the shortest journey in terms of the number of edges, it may not always be the optimal choice for every traveler. The blue path, for instance, offers the best price, ensuring cost-effectiveness while also featuring the *latest departure time*. On the other hand, the green line guarantees the earliest arrival, although it comes at a higher price.

This scenario underscores that the selection of the shortest path is just one facet of the decision-making process. Travelers must weigh factors like *cost, departure time, and arrival time* to make a well-rounded choice that aligns with their specific priorities, whether they value speed, cost-efficiency, or a balance of both.

```
MATCH (c:City {name: 'Anchorage'})<-[:LocatedAt*]-(a:Airport)
MATCH path = (a:Airport)-[e:Flight*1..8]->(b:Airport)
MATCH (b:Airport)-[:LocatedAt*]->(c1:City {name: 'Seattle'})
WITH c, a, b, c1, relationships(path) AS edges
```

```
WHERE all(i in range(0, size(edges)-1) WHERE edges[i-1].arrival_time
    > edges[i].departure_time)
WITH edges, reduce(distance = 0, edge in edges | distance +
    (edge.distance)) AS totalDistance, reduce(price = 0, edge in
    edges | price + edge.price) AS TotalPrice
ORDER BY TotalPrice ASC
RETURN edges, totalDistance, TotalPrice;
```

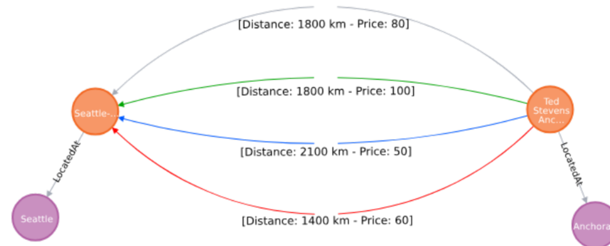

Figure 3.12: Shortest path with respect the price

3.3 T-GQL Syntax

This section introduces T-GQL, as denoted in the reference paper [5], as a high-level query language for graph databases. While it has some similarities to SQL, it is primarily built upon Cypher[2], which is Neo4j's high-level query language. The formal semantics of Cypher can be referenced in [7, 8]. In the section 4.2, provides instructions on how to convert T-GQL queries into Cypher and then transmit these converted queries to the server side.

3.3.1 Basic Statements

The language's syntax follows the familiar structure of SELECT-MATCH-WHERE. Within this structure, the SELECT clause enables the selection of variables defined within the MATCH clause, allowing for the use of aliases. The MATCH clause can include one or more path patterns, fixed or variable length, and function calls. The outcome of the query is a temporal graph.
Consider the query: *"List the cities that are connected to the Anchorage through intermediary city".*

```
SELECT c2
MATCH (c1:City) - [:Flight*2] -> (c2:City)
WHERE c1.name = 'Anchorage'
```

In this case, the query returns the object node *Cities.* To retrieve all possible paths, the asterisk wildcard operator '*' is employed. The following expression yields paths of a length of two starting from the node that represents Anchorage.

```
SELECT *
MATCH (c1:City) - [:Flight*2] -> (c2:City)
WHERE c1.name = 'Anchorage'
```

T-GQL includes support for the three path semantics described in earlier section 3.2: (a) Continuous path semantics, (b) Pairwise continuous path semantics, and (c) Consecutive path semantics. These semantics are realized through functions that are part of a Neo4j plugin library.

[2] https://neo4j.com/docs/cypher-manual/current/

3.3.2 Continuous Paths Queries

Continuous paths [3.2.1], as a graph traversal algorithm, operate with a unique simplicity. Unlike other path-finding algorithms that consider various attributes and conditions like time, cost, or specific constraints, continuous paths focus solely on establishing connections between nodes. Their primary objective is to uncover any viable path that links these nodes together. In the context of navigating a graph, continuous path algorithms effectively disregard any additional criteria, prioritizing the concept of connectedness.

For instance, when using a continuous path algorithm to explore a network or graph, it unearths all nodes that share a connection, forming a sequence of linked nodes without concern for other factors. This approach can be particularly useful when the primary goal is to identify the basic routes or establish connections between specific entities, such as cities. Continuous path algorithms excel at identifying any connection that can bridge these cities, offering a comprehensive view of the network's underlying structure. They are a valuable tool for situations where the focus is on mapping the fundamental relationships and connections within a graph. In our queries, the *cPath* function operates as a semantic identifier for finding continuous paths within the graph, emphasizing the importance of unbroken node sequences. Conversely, *pariCPath* represents the concept of pairwise continuous paths, highlighting connections between specific pairs of nodes.

For queries related to consecutive paths, such as *earliest arrival, latest departure, fastest path, and shortest path,* the semantic function aligns with the respective path's name, signifying the primary focus on attributes like time, cost, and distance in determining the optimal route. In the following, some query examples of continuous path exploration will be showcased.

Query 1 is about finding all the direct two-stop flight routes in the given flight network example.

Query 1 *List the cities that are connected to the Anchorage through intermediary city, and the period such that the relationship occurred through all the path.*

```
SELECT path
MATCH (n:City), path = cPath((n)-[:Flight*2]-> (c:City))
WHERE c.name = 'Anchorage'
```

Another example, searching for a continuous path between two particular cities.

Query 2 *Find the continuous paths between "Anchorage" and "Los Angeles" with a minimum length of two and a maximum length of three.*

```
SELECT paths
MATCH (c1:City), (c2:City),
paths = cPath((c1) - [:Flight*2..3] -> (c2))
WHERE c1.name = 'Anchorage' AND c2.name = 'Los Angeles'
```

The following query finds *pairwise continuous paths* (Definition 8) between two cities with a specific length. While continuous paths generally aim to uncover any unbroken sequences of connected nodes, *pairwise continuous paths* concentrate their efforts on identifying and mapping routes between two selected nodes.

Query 3 *Find pairwise continuous paths between "Anchorage" and "Los Angeles" with a minimum length of two and a maximum length of three.*

```
SELECT paths
MATCH (c1:City), (c2:City),
paths = pairCPath((c1)-[:Flight*2..3]->(c2))
WHERE c1.name = 'Anchorage' AND c2.name = 'Los Angeles'
```

3.3.3 Consecutive Paths Queries

The T-GQL implements consecutive path semantics (definition 9, 10). For experiments, *T-GQL* supports the *Fastest Path, Latest Departure Path, Earliest Arrival Path, Shortest Path*, and *Cheapest Path*. The first three ones receive two nodes as *City* and *Airport*, work with time intervals defined as attributes. *Shortest Path*, works with the distance attribute. And the last one, *Cheapest Path* feature in T-GQL introduces an intriguing dimension to the pathfinding capabilities of this query language. While the other path types such as *Fastest Path, Latest Departure Path, Earliest Arrival Path, and Shortest Path* focus on various attributes like time intervals and distance, the *Cheapest Path* extends its reach by specifically considering the *price* attribute in the path selection process.

The inclusion of the *price* attribute means that when searching for the *Cheapest Path,* the query takes into account the economic aspect of the journey. In other words, it aims to find the most cost-effective path between two nodes, such as a city and an airport. This is particularly beneficial for travelers who prioritize budget-conscious travel.

However, what makes the *Cheapest Path* even more compelling is its integrative potential. It can be seamlessly incorporated into the other path types, offering users the flexibility to balance various attributes. For example, you could look for the *Fastest Path* that is also the *Cheapest Path,* optimizing both time and cost efficiency. Alternatively, you could find the *Latest Departure Path* or *Earliest Arrival Path* that minimizes expenses, allowing for more control over your travel budget.

In essence, the *Cheapest Path* feature enhances the versatility of T-GQL, enabling users to make well-informed decisions based not only on *time, departure, arrival, or distance* but also on the *price*, offering a comprehensive approach to pathfinding and route optimization.

In the example depicted in Fig. [3.2], *Airport* and *City* are the object nodes, two temporal relationships, namely *Flight* and *LocatedAt*. Also, interval$[t_d, t_a]$ where t_d is departure time and t_a is arrival time and a distance attribute is used to calculate the distance for the *Shortest Path*.

The queries provided below serve as examples of both their syntax and semantics.

Query 4 *How can we go from "Anchorage" to "Los Angeles" as soon as possible?*

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport),(c2:City)-[:LocatedAt]->
    (a2:Airport), path = fastestPath((a1)-[:Flight*]->(a2))
WHERE c1.name = 'Anchorage' AND c2.name = 'Los Angeles'
```

Query 5 *How can we travel from 'Anchorage' to 'Los Angeles' while ensuring we reach our destination by the earliest arrival?*

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport),(c2:City)-[:LocatedAt]->
    (a2:Airport), path = earliestArrival((a1)-[:Flight*]->(a2))
WHERE c1.name = 'Anchorage' AND c2.name = 'Los Angeles'
```

Query 6 *How can we go from "Anchorage" to "Los Angeles", leaving as late as possible?*

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport), (c2:City)-[:LocatedAt]->
    (a2:Airport), path = latestDeparturePath((a1)-[:Flight*]-> (a2))
WHERE c1.name='Anchorage' AND c2.name='Los Angeles'
```

Query 7 *What is the shortest path between "Anchorage" and "Los Angeles"?*

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport), (c2:City)-[:LocatedAt]->
    (a2:Airport), path = shortestPath((a1)-[:Flight*]-> (a2))
WHERE c1.name='Anchorage' AND c2.name='Los Angeles'
```

3.4 Implement Java Functions

In reference to the last phase, Neo4j's potential to employ defined functions as user-defined procedures becomes evident. User-defined procedures in Neo4j enable you to extend the functionality of the database by creating custom operations and tasks in the form of procedures. These procedures can encapsulate complex workflows, queries, and data manipulation tasks, making them available for reuse and simplifying your interactions with the database. User-defined procedures are typically written in Java and are particularly useful for advanced users who need to implement custom logic within Neo4j. Here's an overview of how to work with user-defined procedures in Neo4j:

3.4.1 Creating a User-Defined Procedure

To create a user-defined procedure, you'll write custom Java code that implements the desired functionality. Neo4j provides a framework and a set of APIs for creating these procedures. The code should be compiled into a JAR (Java Archive) file.

3.4.2 Registering the Procedure

After developing your user-defined procedure, you need to register it with Neo4j. This involves placing the JAR file in the Neo4j plugins directory and configuring the procedure in the neo4j.conf file.
Example of registering a procedure in neo4j.conf:

```
dbms.security.procedures.my_procedure=earliestArrival
```

3.4.3 Using the User-Defined Procedure

Once registered, you can call the user-defined procedure in your Cypher queries and interact with it like any other built-in procedure. Procedures can take input parameters and return results, which can be used in subsequent parts of your Cypher query. Example of using a user-defined procedure in a Cypher query:

```
CALL earliestArrivale('Los Angeles','Flight*','New York')
    YIELD result
RETURN result
```

In summary, user-defined procedures in Neo4j allow you to implement custom, reusable functionality to meet specific database requirements. While they provide powerful capabilities, they should be used carefully, and security considerations should be a top priority. When employed effectively, user-defined procedures can greatly enhance your ability to work with Neo4j and implement custom logic within your graph database.

3.4.4 Integration of Procedures into Neo4j Environment

In this work, novel procedures can be established by crafting Java classes for distinct paths. The novel is on java version 8[3] or more ,and Apache Maven 3.9.3.[4]. *Apache Maven* is a widely used build automation and project management tool primarily used for Java projects, though it can be adapted for other languages and platforms as well. Maven helps streamline the build process, manage project dependencies, and standardize project structures. It was developed by the *Apache Software Foundation* [5] and is a popular choice among Java developers for managing their software projects. Maven uses a Project Object Model, represented in an XML file called *POM.xml,* to define project information, dependencies, build settings, and plugins. The POM file is at the core of Maven and serves as a blueprint for your project. To operationalize these user-defined procedures, integration of the Neo4j Java driver within the *POM.xml* file, a fundamental element within the Maven framework, is essential. To add the *Neo4j Java driver* dependency to your pom.xml file, you need to edit the pom.xml file of your Maven project. The pom.xml file is where you define the project configuration and dependencies.

```
<dependency>
    <groupId>org.neo4j.driver</groupId>
    <artifactId>neo4j-java-driver</artifactId>
    <version>5.12.0</version>
</dependency>
```

This approach allows us to establish connections between our Java classes and the Neo4j database.
To create user-defined procedures, the ”@Procedure” keyword is employed to annotate our functions, with an added mode attribute (Read, Write, DBMS). To illustrate this point, consider the following example:

```
package functions;

import org.neo4j.procedure.Name;
import org.neo4j.procedure.UserFunction;
import org.neo4j.procedure.UserProcedure;

public class ConsecutivePath {

    @Procedure(name = "earliestArrival", mode = Mode.READ)
    @Description("Returns earliest arrival path between two cities.")
    public void earliestArrival(@Name("startCity") String startCity,
        @Name("flight") String relationshipType, @Name("endCity")
        String endCity){

        // Your custom logic here
    }
}
```

These functions return Java 8 streams of simple objects with publicly accessible fields. After the Java classes have been implemented, generating a .jar package containing the functions and their dependencies can be achieved through *mvn clean package.* The mvn clean package command, used to build a Maven project,

[3]https://www.java.com/
[4]https://maven.apache.org/
[5]https://www.apache.org/

is executed from the root directory of the Maven project where the *POM.xml* file is located.

```
mvn clean package
```

By incorporating this .jar file into the plugin directory of Neo4j and subsequently restarting the Neo4j server, we can gain access to and list the functions and procedures. you can use the *SHOW PROCEDURES* command to display a list of user-defined procedures that have been registered with the database. This command is used to view the available custom procedures that can be called within your Cypher queries.
During this phase, we move our functions and pathways to the server, making them accessible and operational within the Neo4j environment. This shift involves transferring these functions to the server side, where they can be effectively employed within the Neo4j shell's capabilities.

Chapter 4

Implementation

Defining a custom grammar for Neo4j queries involves specifying the syntax and structure of the queries you want to support. In this context, custom queries are those that extend or modify the standard Cypher query language used with Neo4j. Here's an explanation of how to define a custom grammar to run these queries in Neo4j.

4.1 Define T-GQL Grammar

To define the grammar in a way that allows running custom queries to return continuous and consecutive paths, commence by specifying the grammar for the custom query language. This grammar should encompass rules and structures tailored to support the formulation of queries aimed at retrieving continuous and consecutive paths in your specific context. Various parser generator tools, including ANTLR v4 [1] or JavaCC [2], can be employed for creating the grammar. In this scenario, ANTLR v4 plugin in the IntelliJ IDEA is utilized for creating the grammar. ANTLR (ANother Tool for Language Recognition) is a powerful parser generator that allows you to create parsers and translators for various programming languages and file formats. ANTLR 4 provides a plugin for the IntelliJ IDEA integrated development environment (IDE) to facilitate the integration of ANTLR grammars and generated code into your Java projects. The definition of the grammar is typically located in a specific file with a .g4 extension, commonly used for ANTLR grammar files. Within this grammar file, the syntax rules for the language or parser are defined. A specific root is not assigned, as the root of a parse tree is determined by the starting rule specified when generating a parser for the grammar. This starting rule is commonly referred to as the "entry point" to the language or parser.

When the ANTLR tool is employed to generate parser code from the grammar, the starting rule is specified, and a parser class with a method corresponding to the starting rule is created by ANTLR. This method is where the parsing process commences and is typically named after the starting rule.

[1] https://plugins.jetbrains.com/plugin/7358-antlr-v4
[2] https://javacc.github.io/javacc/

4.1.1 Grammar Structure

The grammar begins with the TGQL rule, which is the starting point for parsing queries. A query consists of one or more statements separated by semicolons (;).

4.1.2 Statement Types

There are three main types of statements: *SELECT*, *MATCH*, and optional *WHERE and WHEN* clauses. *SELECT* statements are used to retrieve data based on specified paths. *MATCH* statements define patterns for matching data in the database. *WHERE* and *WHEN* clauses provide conditions to filter and refine data.

4.1.3 Identifiers and Paths

identifier represents user-defined names. *path* specifies a sequence of identifiers and relationships or other paths. Paths can include an optional alias using the AS keyword. *COMMA* and *DOT* are used for path concatenation and alias assignment. And, *ASTERISK* represents a wildcard path.

4.1.4 Matching Patterns

The *match_* rule defines complex patterns for matching data. These patterns can include conditions and filters. The *function* rule specifies supported functions for matching. *EQUALS* is used to compare values in patterns.

4.1.5 Conditions and Boolean Expressions

booleanExpression and *booleanExpression1* define conditions to filter results. Conditions include comparisons using *EQUALS*. *city* represents city names and identifiers in conditions. *range* specifies numeric ranges.

4.1.6 Keywords

Keywords like *SELECT*, *MATCH*, *WHERE*, and *WHEN* are case-insensitive.

4.1.7 Whitespace and Comments

White space is defined using the *WS* rule and is skipped. And, comments are not explicitly defined in the grammar.

4.2 Generate Lexers and Parsers

In this section, the process of converting T-GQL queries into Cypher queries will be discussed. The parsing steps and the transformation of T-GQL queries into their Cypher equivalents will be covered.

4.2.1 Integrate with Neo4j

Integrating the TGQL grammar with the Neo4j database necessitates the establishment of the requisite infrastructure for processing custom queries defined using the grammar and executing them within Neo4j.

4.2.2 Parsing and query translation

After implementing the grammar, testing can be conducted by composing queries and inspecting the query tree to ensure that everything is in order before proceeding. As an illustration, consider the parsing tree corresponding to the following query shown in the figure[4.1].

```
SELECT c
MATCH (c:City)
WHERE c.name = 'Los Angeles'
```

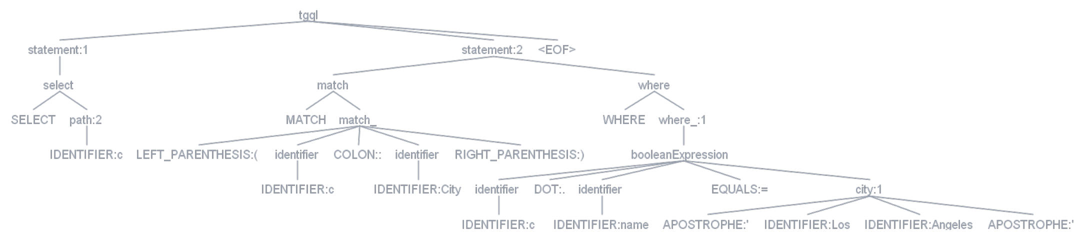

Figure 4.1: Select the city names "Los Angeles"

Once the grammar's correctness has been verified, and all possible types of paths have been taken into account as a customized query. Lexers and parsers are fundamental components of a compiler or interpreter, responsible for analyzing the syntax and structure of source code written in a programming language. They play a crucial role in converting human-readable code into an abstract representation that a computer can understand and execute. In the context of ANTLR4 and grammar definitions, the *visitor* and *listener* are two distinct mechanisms for traversing and processing the parse tree. The *visitor* is an object-oriented design pattern for navigating the parse tree and executing actions at specific nodes. On the other hand, the *listener* represents an alternative mechanism for traversing the parse tree within ANTLR4. The generation of lexers and parsers is achieved by executing the subsequent code.

```
antlr4 -visitor TGQL.g4
```

Java files (e.g., TGQLLexer.java and TGQLParser.java) are generated by this command. When generate a parser and lexer using ANTLR4, it creates Java classes that correspond to your grammar rules. These classes provide methods for traversing the parse tree and handling different parts of the input based on your grammar. To enhance the functions generated by the parser and lexer, you can seamlessly extend their capabilities. When parsing input utilizing the generated parser, you can associate it with your customized listener or visitor. This strategic association enables tree traversal while ensuring the invocation of your custom methods at specific points during the parsing process. This approach empowers you to tailor the parsing process to your specific requirements and enables the

execution of personalized logic, ultimately enhancing the parser's functionality to accommodate your unique needs.
It's important to understand that a query language conceals the underlying data structure. Within this structure, three types of nodes are identified: object nodes, attribute nodes, and value nodes. For instance, in the Fig.[3.2], we observe object nodes like City and Airport, connected by various types of relationships. In this scenario, these object nodes are linked to attributes and value nodes through a common type of connection called "Edge." Consequently, in the implementation, City and Airport become properties of objects (referred to as "Title"). The City's name serves as a property (also referred to as "Title") of an attribute node, and the actual City name is stored as a property of a value node, indicated as "value." All these structures are hides from users but interact with and stored in the Neo4j database. For example, the following queries:

Query 1 *Fastest path between Phoenix and New York*

```
SELECT path
MATCH (c1:City)<-[:LocatedAt]-(a1:Airport),(c2:City)<-[:LocatedAt]-
    (a2:Airport), path=fastestPath((a1)-[:Flight*]->(a2))
WHERE c1.name='Phoenix' AND c2.name='New York'
```

This is Translated to:

```
MATCH (c1:Object{title:'City'})<-[internal_l0:
    LocatedAt]-(a1:Object{title:'Airport'}),
    (c2:Object{title:'City'})<-[internal_l1:
    LocatedAt]-(a2:Object{title:'Airport'})
MATCH (c1)-->(internal_n0:Attribute{title:
    'name'})-->(internal_v0:Value)
MATCH (c2)-->(internal_n1:Attribute{title: 'name'})-->
    (internal_v1:Value)
WHERE internal_v0.value='Phoenix' AND
internal_v1.value='New York'
CALL consecutive.fastest(a1,a2,1,
{edgesLabel:'Flight',direction:'outgoing'})
YIELD path as internal_p0, interval as
internal_i0
WITH paths.intervals.fastest({path:internal_p0,
interval:internal_i0}) as path
RETURN path
```

Query 2 *Shortest path between Seattle and Chicago*

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport), (c2:City)-[:LocatedAt]->
    (a2:Airport), path = shortestPath((a1)-[:Flight*]-> (a2))
WHERE c1.name='Seattle' AND c2.name='Chicago'
```

This is Translated to:

```
MATCH (c1:Object{title:'City'})-[:LocatedAt]->
    (a1:Object{title:'Airport'}),
    (c2:Object{title:'City'})-[:LocatedAt]->
    (a2:Object{title:'Airport'})
MATCH (c1)-->(internal_n0:Attribute{title:'name'})-->
    (internal_v0:Value)
MATCH (c2)-->(internal_n1:Attribute{title:'name'})-->
    (internal_v1:Value)
WHERE internal_v0.value='Seattle' AND internal_v1.value='Chicago'
```

```
CALL consecutive.shortest(a1, a2, 1, {edgesLabel:'Flight',
    direction:'outgoing'})
YIELD path as internal_p0, interval as internal_i0
WITH paths.intervals.shortest({path:internal_p0, interval:internal_i0
    }) as path
RETURN path
```

Query 3 *Earliest arrival path between Charlotte and Los Angeles*

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport),(c2:City)-[:LocatedAt]->
    (a2:Airport), path = earliestArrival((a1)-[:Flight*]->(a2))
WHERE c1.name = 'Charlotte' AND c2.name ='Los Angeles'
```

This is Translated to:

```
MATCH (c1:Object{title:'City'})-[:LocatedAt]->(a1:Object{title:
    'Airport'}), (c2:Object{title:'City'})-[:LocatedAt]->(a2:
    Object{title:'Airport'})
MATCH (c1)-->(internal_n0:Attribute{title:'name'})-->(internal_v0:
    Value)
MATCH (c2)-->(internal_n1:Attribute{title:'name'})-->(internal_v1:
    Value)
WHERE internal_v0.value='Charlotte' AND internal_v1.value=
    'Los Angeles'
CALL consecutive.earliestArrival(a1, a2, 1, {edgesLabel:'Flight',
    direction:'outgoing'})
YIELD path as internal_p0, interval as internal_i0
WITH paths.intervals.earliestArrival({path:internal_p0,
    interval:internal_i0}) as path
RETURN path
```

Query 4 *Latest departure path between New York and Phoenix*

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport), (c2:City)-[:LocatedAt]->
(a2:Airport), path = latestDeparturePath((a1)-[:Flight*]-> (a2))
WHERE c1.name='New York' AND c2.name='Phoenix'
```

This is Translated to:

```
MATCH (c1:Object{title:'City'})-[:LocatedAt]->(a1:Object{title:
    'Airport'}), (c2:Object{title:'City'})-[:LocatedAt]->(a2:
    Object{title:'Airport'})
MATCH (c1)-->(internal_n0:Attribute{title:'name'})-->(internal_v0:
    Value)
MATCH (c2)-->(internal_n1:Attribute{title:'name'})-->(internal_v1:
    Value)
WHERE internal_v0.value='New York' AND internal_v1.value='Phoenix'
CALL consecutive.latestDeparturePath(a1, a2, 1, {edgesLabel:'Flight',
    direction:'outgoing'})
YIELD path as internal_p0, interval as internal_i0
WITH paths.intervals.latestDeparturePath({path:internal_p0,
    interval:internal_i0}) as path
RETURN path
```

4.3 Deploy Web Application

Incorporating the ability to create custom queries and thereby extend the functionality of the Cypher query language is a standard practice in web applications that engage with Neo4j. In the context of this study, a web page was meticulously designed, employing the Javalin framework[3], with a specific focus on version 5.6.1, to facilitate a seamless interaction with Neo4j. This amalgamation of technologies provides a potent and flexible approach, unlocking the potential of graph databases for diverse applications. Furthermore, it establishes a web-based interface for the execution of custom Cypher queries, with an emphasis on continuous and consecutive paths, measuring their execution times and presenting the results. The locally hosted web application, accessible at *localhost:7070*, fosters an efficient and intuitive platform for working with Neo4j, streamlining query execution and result visualization, which is particularly valuable in the context of complex temporal graph data models.

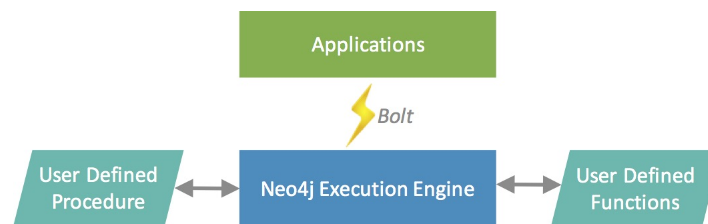

Figure 4.2: Neo4j itself provides and utilizes custom procedures. [12]

To effectively incorporate the Javalin framework into the project, it is crucial to add the appropriate dependency within the project's pom.xml file. The provided dependency snippet is meticulously tailored to ensure the seamless integration of Javalin into your project. It is as follows:

```
<dependency>
      <groupId>io.javalin</groupId>
      <artifactId>javalin</artifactId>
      <version>5.6.1</version>
</dependency>
```

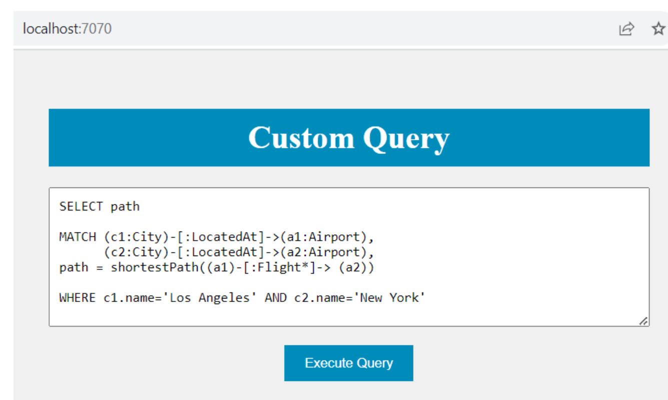

Figure 4.3: Web application to run the custom queries

In this scenario, it's imperative to implement functions for continuous and consecutive paths in a manner that results in data being appropriately formatted in

[3] https://javalin.io/

JSON (JavaScript Object Notation). This JSON formatting is a pivotal step to render the data suitable for transmission to the web application, ensuring seamless communication and understanding between the database and the application.
In essence, this setup constitutes a web-based interface that not only facilitates the execution of extended and custom Cypher queries but also measures the time required for their execution and presents the results. It harnesses the robust capabilities of the Javalin framework for web routing and managing requests and responses, offering a versatile solution for working with graph databases in real-world applications.
Overall, this sets up a web-based interface for executing extend and custom Cypher queries, measures the time it takes to execute them, and displays the results. It uses the Javalin framework for web routing and handling requests and responses.

Chapter 5

Experiments

This section aim to take into account the experiments conducted from a two-dimensional perspective. First, offers a comprehensive overview of the experiments carried out to evaluate the range of queries outlined in the initial paper [5], which were further developed and explored in the context of this study. In another dimension, in the real-world context of case study, which centers around the flight data set, we sought to investigate the extent to which factors such as price and duration can influence the length of the paths. In other words, the experiments not only encompassed the queries presented in the original paper but also delved into how practical factors, specifically ticket price and flight duration, could impact the results of these queries. For instance, we explored whether optimizing for the shortest path between two cities might lead to a trade-off between cost and travel time and how these variables interacted with the overall length of the paths discovered in the flight data set. This investigation was crucial in understanding the practical implications of the queries and their relevance in real-world scenarios, especially when making decisions related to travel or logistics based on the flight data. These experiments encompass two categories of path algorithms under investigation: continuous paths and consecutive paths. Given that the implementation serves as a proof-of-concept and Neo4j is not optimized for handling exceptionally large graphs, the primary objective of this evaluation is to identify the factors affecting performance rather than quantifying performance itself. Future research will focus on addressing performance concerns through the implementation of indexing schemes.

5.1 Goal

These experiments aim to examine how the length of paths and the size of the data set affect algorithm performance. In addition to the thesis's initial objective, for the second dimension, the experiments aim to delve into how algorithm performance is influenced not only by the length of paths and the size of the data set but also by two critical real-world factors: price and flight duration. Thus, the tests conducted encompass a dual perspective, examining the combined effects of path length, data set size, price, and flight duration on algorithm performance. As a result, various tests are being conducted, manipulating both variables.

5.2 Data set and Setup

In this section, we present a comprehensive overview of the flight data set [1] which served as the foundation for assessing the algorithms in the experiments. All experiments were meticulously conducted within a controlled and consistent environment, utilizing a Neo4j 5.9.0 server. This Neo4j server was operated on a 64-bit Windows 11 system, featuring an Intel Core i7-7700 HQ processor with a clock speed of 2.80GHz, equipped with 8 cores and 16 GB of RAM. This standardized setup ensured the reliability and repeatability of our experiments, allowing us to precisely measure and analyze algorithm performance.

For these experiments, the data set was extended by incorporating additional attributes, most notably the *Price* attribute, into the edges of the graph. This augmentation enabled us to explore the second dimension, wherein an exploration was undertaken to investigate the influence of *Price* and *Flight Duration* on the lengths of the paths. The introduction of the *Price* attribute provided valuable insights into the practical implications of the queries and algorithms, as it allows to examine not only the shortest or fastest paths but also those that were the most cost-effective. Moreover, by considering the impact of both *price* and *flight* duration on path lengths, Efforts were made to offer a more comprehensive perspective on route planning and optimization in the context of the flight data set. The foundation for a more nuanced understanding of the data set's potential applications and implications in real-world scenarios was established through this dual-dimensional analysis. Additionally, in the experiment, it should be considered that N represents the number of nodes, and R denotes the number of relationships in a sequence as the path.

5.3 Continuous path algorithms

In our analysis, we conducted continuous path queries on the flight transportation network, focusing on connecting two specific cities. These cities were identified using a property known as *City Name,* which was generated during the population of the data set. This property allowed us to precisely pinpoint cities within the network.

For instance, consider the following query scenario: we sought to discover all continuous paths within a specific range of lengths, spanning from 2 to 8, between nodes representing the cities of *Anchorage* and *New York*. It's worth emphasizing that there are precisely 8 direct city relationships connecting *Anchorage* and *New York*. Therefore, this particular query was applied to a range of city pairs, encompassing different path lengths, to ensure a comprehensive exploration of the flight transportation network. We meticulously examined all connections between the following city pairs: [*Anchorage, Seattle*], [*Anchorage, Los Angeles*], [*Anchorage, Minneapolis*], [*Anchorage, Phoenix*], [*Anchorage, Chicago*], [*Anchorage, Charlotte*], [*Anchorage, San Juan*], and [*Anchorage, New York*]. Each of these pairs presented a unique scenario, enabling us to assess how continuous paths of varying lengths could be established between cities in the flight transportation network. This extensive examination allowed us to gain valuable insights into the network's

[1] `https://www.kaggle.com/datasets/usdot/flight-delays?select=flights.csv`

structure, connectivity, and the feasibility of passenger routes between these cities. To investigate the scenario mentioned above, we can execute the following query:

```
SELECT path
MATCH (c1:City), (c2:City),
    path = CPath((c1)-[:Flight*8]->(c2))
WHERE c1.name='Anchorage' AND c2.name='New York'
```

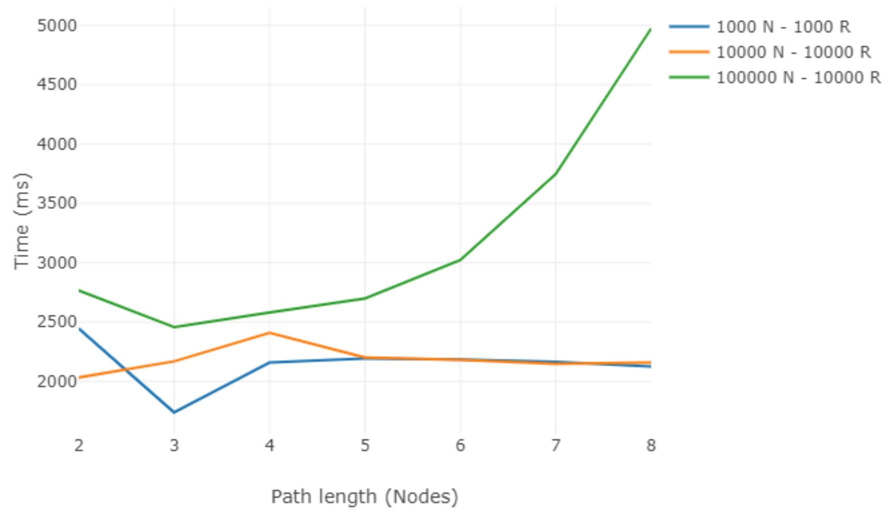

Figure 5.1: Time Vs. length for continuous path

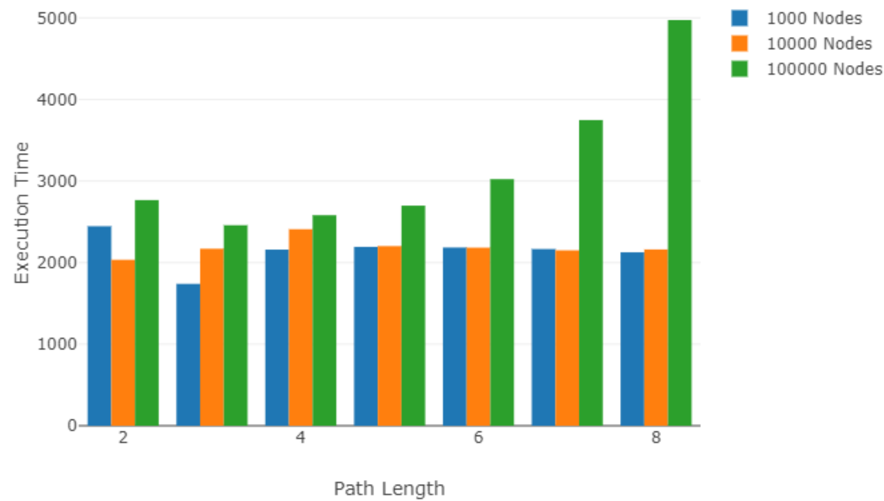

Figure 5.2: Time Vs. length for continuous path with different number of nodes

The same query type was executed to assess the pairwise continuous path algorithm, taking into account the definition of pairwise continuous paths as outlined in [8].

```
SELECT path
```

```
MATCH (c1:City), (c2:City),
    path = pairCPath((c1)-[:Flight*8]->(c2))
WHERE c1.name='Anchorage' AND c2.name='New York'
```

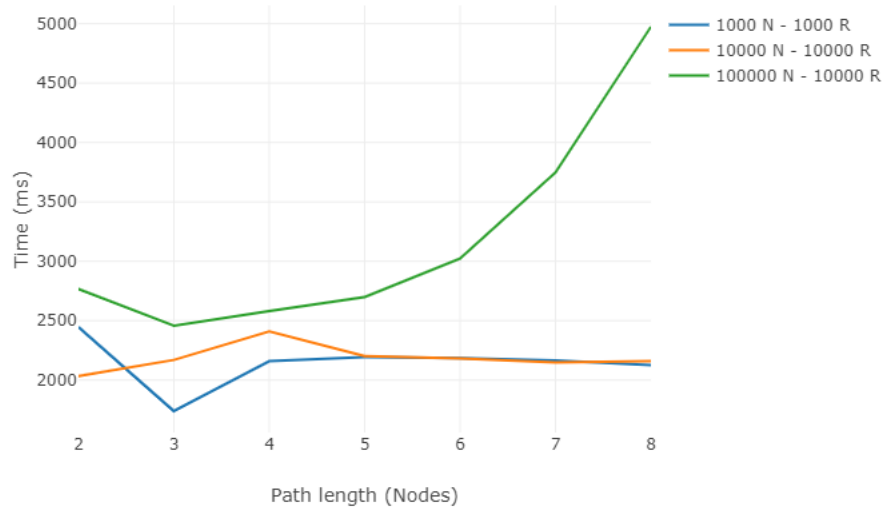

Figure 5.3: Time Vs. length for pairwise continuous path

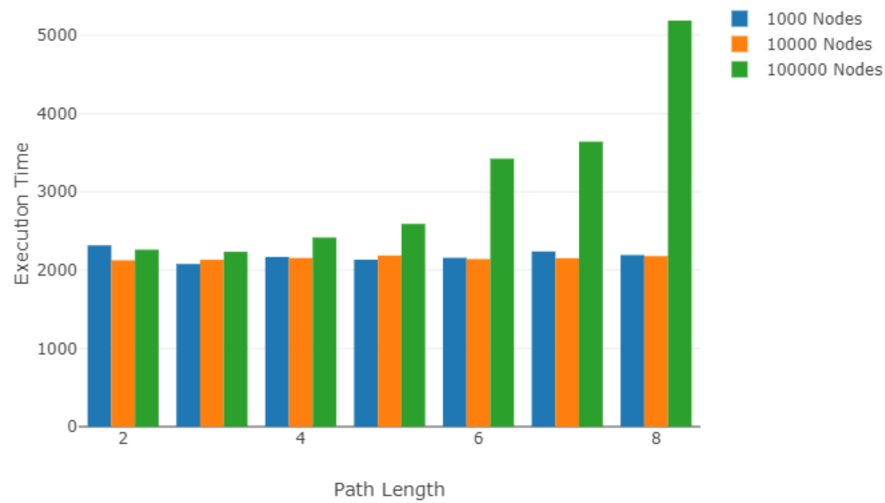

Figure 5.4: Time Vs. length for pairwise continuous path with different number of nodes

In Figure [5.5], a comparative analysis is presented, offering valuable insights into the performance differences between continuous and pairwise continuous paths across varying node counts. This visual representation allows for a more in-depth observation of how these two path types behave as the scale of the network, in terms of the number of nodes, is altered.

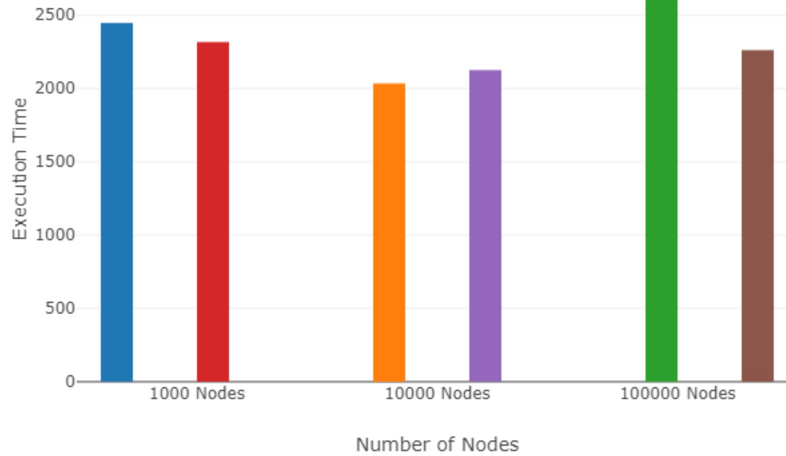

Figure 5.5: Comparison CPath and PairCPath

5.4 Consecutive paths algorithms

Continuing our exploration, we delve into consecutive path queries on the flight transportation network, mirroring the approach described in the preceding section. These queries maintain a primary objective: the discovery of consecutive paths between two specific cities. These cities are unambiguously identified by their respective names. In this intricate network, each city is intricately connected to its corresponding airport through the *locatedAt* edge, which serves as a pivotal link in the transportation chain. Furthermore, these airports are intricately interlinked via the *Flight* edge, forming a comprehensive web of air travel connections.
Notably, each airport possesses a unique property known as the IATA (International Air Transportation Association) code. This three-letter code serves as an exclusive identifier for airports worldwide, facilitating seamless recognition and differentiation between them. This code is vital for pinpointing and navigating the extensive network of airports.
The comprehensive range of queries conducted in this context encompasses all four distinct types of consecutive path algorithms. These algorithms have been structured with precision to ensure thorough exploration and efficient navigation of the flight transportation network. In essence, these queries are designed to uncover the various pathways and connections between cities, and they serve as a critical component in the analysis of the network's structure and functionality. By thoroughly examining these diverse algorithms, our goal is to acquire comprehensive insights into the operational dynamics of the flight transportation network as depicted within the data set. This effort lays the groundwork for a more nuanced understanding of the network's data-driven representation, shedding light on the intricacies of air travel and logistics as simulated within this complex data set.
The queries encompass all four types of consecutive path algorithms and have the following structure:
The first of the four consecutive path algorithms is the *Earliest Arrival* algorithm. This algorithm is designed to find paths between two points (in this case, cities) within the flight transportation network while prioritizing the earliest possible

arrival time. In other words, it focuses on identifying routes that minimize the travel time, ensuring that reach their destination as soon as possible.

To achieve this, the *Earliest Arrival* algorithm considers the flight schedule and its associated time constraints. It takes into account factors such as departure times, stops , and flight durations to determine the most time-efficient route.

```
SELECT path
MATCH (c1:City), (c2:City),
    path = earliestArrival((c1)-[:Flight*8]->(c2))
WHERE c1.name='Anchorage' AND c2.name='New York'
```

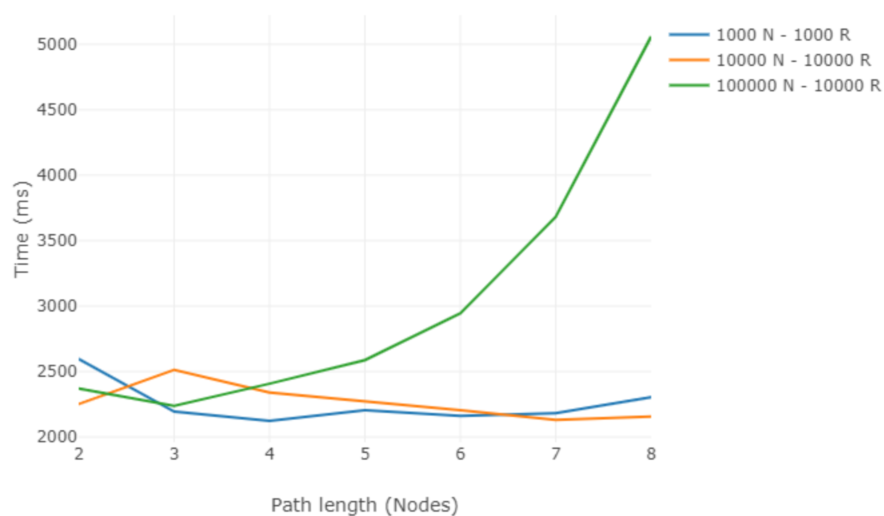

Figure 5.6: Time Vs. length for earliest arrival path

```
SELECT path
MATCH (c1:City), (c2:City),
    path = latestDeparture((c1)-[:Flight*8]->(c2))
WHERE c1.name='Anchorage' AND c2.name='New York'
```

The second query in the set employs the *Latest Departure* algorithm, searching for a path of exactly 8 consecutive flights between *Anchorage* and *New York.* This query focuses on pinpointing routes where the emphasis is placed on the departure time of each flight segment. It's valuable for those who prioritize departing as late as possible within the context of an 8-flight journey, offering flexibility in scheduling and planning

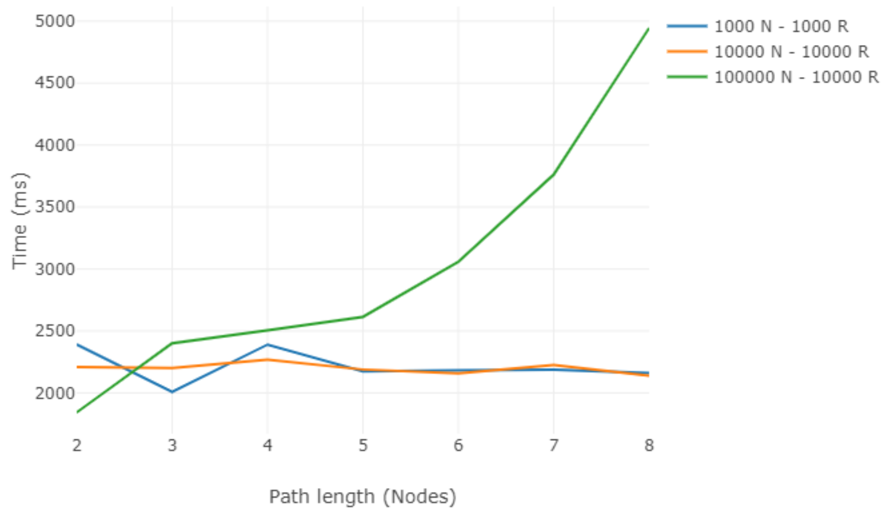

Figure 5.7: Time Vs. length for latest departure path

The third query within the set of consecutive path queries introduces the *Shortest Path* algorithm. This query is designed to uncover the most concise path between two cities, *Anchorage* and *New York*, with a strict requirement of traversing exactly 8 consecutive flight relationships. By applying the *Shortest Path* algorithm, the query emphasizes efficiency in terms of distance or travel time. It is particularly valuable in case of seeking to identify the quickest connection with a predefined number of intermediate flights, making it a crucial tool for route optimization within the flight data set.

```
SELECT path
MATCH (c1:City), (c2:City),
    path = shortestPath((c1)-[:Flight*8]->(c2))
WHERE c1.name='Anchorage' AND c2.name='New York'
```

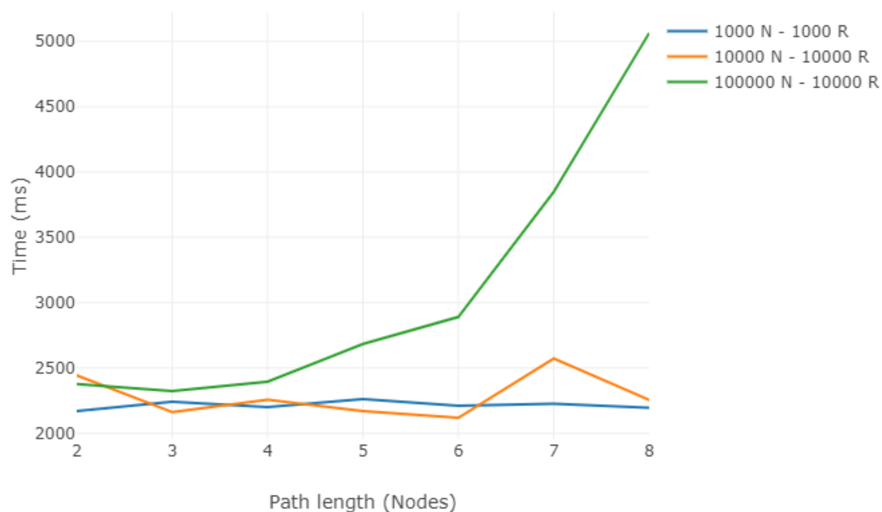

Figure 5.8: Time Vs. length for shortest path

The final query among the four consecutive path queries introduces the *Fastest Path* algorithm. This query is dedicated to discovering the speediest path between *Anchorage* and *New York,* with a strict requirement of traversing exactly 8 consecutive flight relationships. By employing the *Fastest Path* algorithm, the query emphasizes minimizing travel time, making it a valuable tool for prioritizing swift connections between these cities. This query is pivotal for planning scenarios where a specific number of stops and a focus on minimal travel duration are of utmost importance.

```
SELECT path
MATCH (c1:City), (c2:City),
    path = fastestPath((c1)-[:Flight*8]->(c2))
WHERE c1.name='Anchorage' AND c2.name='New York'
```

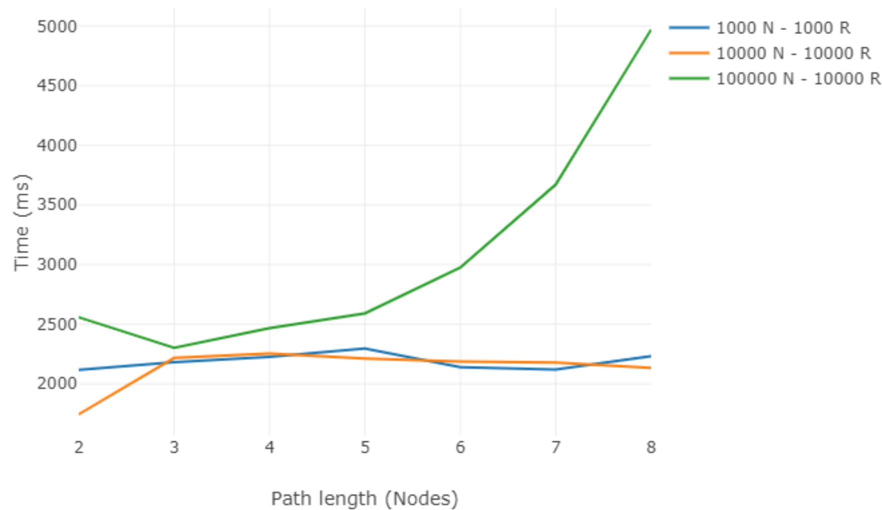

Figure 5.9: Time Vs. length for fastest path

Figure [5.10] provides an insightful comparison of various path types within the realm of continuous and consecutive paths when the number of nodes stands at 1000. This graphical representation aims to facilitate a more comprehensive observation of path behaviors in the context of a relatively moderate-sized network. The figure encompasses continuous paths, pairwise consecutive paths, earliest arrival paths, latest departure paths, fastest paths, and shortest paths. By juxtaposing these distinct path semantics under the same node count, it enables a nuanced analysis of how these paths perform and interact within a network of this scale, enhancing our understanding of their respective characteristics and suitability for different scenarios.

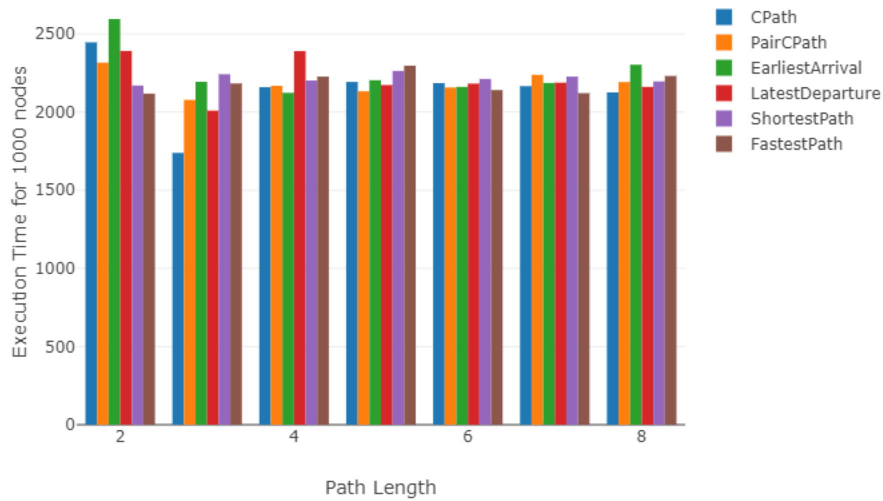

Figure 5.10: Time Vs. length for 1000 Nodes

In Figure [5.11], a comparative analysis of various path types, encompassing both continuous and consecutive paths, unfolds under the backdrop of a more extensive network, consisting of 10,000 nodes. This illustration offers a broader perspective, granting us a deeper understanding of how these path semantics behave when subjected to the complexities of a larger-scale network. The depicted path categories encompass continuous paths, pairwise consecutive paths, earliest arrival paths, latest departure paths, fastest paths, and shortest paths, all scrutinized in the context of 10,000 nodes. This extended node count allows us to draw nuanced insights into how these path types adapt to a more extensive network environment, shedding light on their performance characteristics and applicability within scenarios involving larger data set.

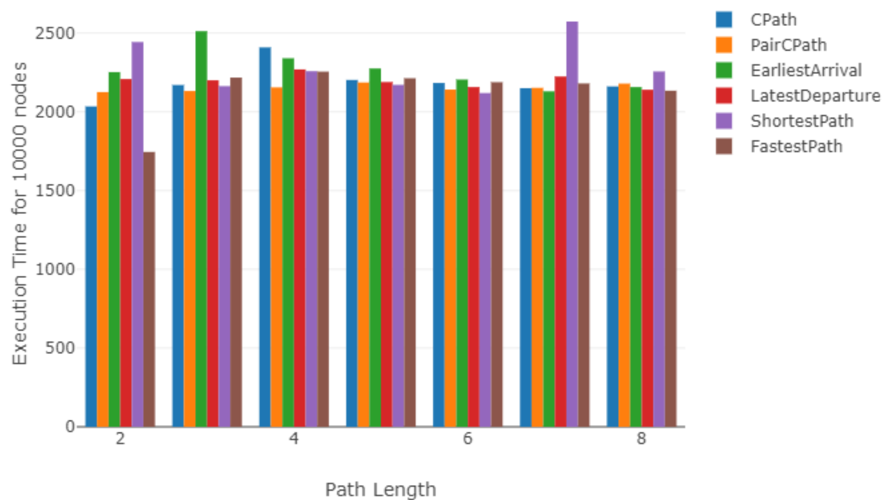

Figure 5.11: Time Vs. length for 10,000 Nodes

Figure [5.12] presents a comprehensive evaluation of diverse path semantics, examining the intricate dynamics of continuous paths and consecutive paths within

a significantly expanded network containing 100,000 nodes. This expansive data set allows for a detailed exploration of how these path types adapt to the demands of extensive networks. The visual representation showcases the same paths with previous figure, all within the context of 100,000 nodes. The larger node count provides a unique vantage point for understanding the scalability and behavior of these path semantics in the realm of substantial data set.

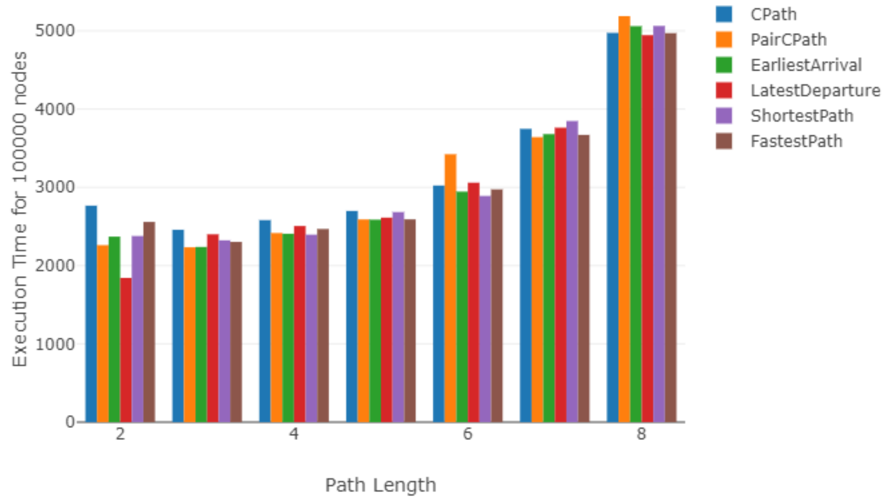

Figure 5.12: Time Vs. length for 100,000 Nodes

5.5 Consecutive paths with respect price

In the preceding sections, we have delved into the theoretical underpinnings of consecutive path queries within the context of a flight transportation network. However, the real-world application of such queries is subject to multifaceted considerations. Numerous factors can influence the decision-making process when selecting flights between cities, extending beyond theoretical constructs. One of the pivotal real-world elements is the economic aspect, specifically the price, which can significantly impact route choices. Additionally, the physical distance between cities is another crucial factor.

In the following section of this thesis, our focus shifts to the interplay between consecutive paths and the economic aspect, particularly the influence of price on the length of paths. We recognize that the real-world scenario is characterized by numerous attributes affecting travel decisions, but for the scope of this study, we primarily concentrate on the duality of price and distance. It's noteworthy to mention that, for the sake of simplicity, we adopt a straightforward model where shorter distances and fewer intermediate stops between the origin and destination cities are associated with higher prices.

We embark on an empirical exploration of consecutive path queries, considering price as a fundamental variable in the context of the four distinct types of consecutive path algorithms. This empirical analysis aims to shed light on the practical implications of price as a determinant in choosing flight routes between cities, thereby contributing to a more comprehensive understanding of air travel logistics in real-world applications. While the real-world encompasses a multitude of dy-

namic factors, this study serves as a focused examination of the specific interplay between price, distance, and path selection.

Query *Consider price in the earliest arrival path:*

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport), (c2:City)-[:LocatedAt]->
(a2:Airport), path = earliestArrivalPrice((a1)-[e:Flight*2..8]->(a2))
WHERE c1.name='Anchorage' AND c2.name='New York'
```

In the context of economic decision-making, this scenario [5.1] involves the pursuit of a cost-efficient approach to reach a destination, along with the enumeration of various strategies for achieving this goal.

Earliest Arrival Time	Total Price	Number of Flights
190	930	8
150	970	7
150	1000	6
150	1050	5
150	1100	4

Table 5.1: Earliest arrival path ordered by price

In the following scenario [5.2], we are constrained by a limited budget and looked for identifying viable pathways to reach a destination while adhering to our budgetary constraints.

Earliest Arrival Time	Total Price	Number of Flights
190	930	8
150	970	7

Table 5.2: Earliest arrival path limited by price less than 1000

Query *Consider length in the latest departure path:*

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport), (c2:City)-[:LocatedAt]->
(a2:Airport), path = latestDeparturePrice((a1)-[e:Flight*2..8]->(a2))
WHERE c1.name='Anchorage' AND c2.name='New York'
```

In the following table [5.3], the objective is to identify the latest departure path from an origin to a destination while minimizing the number of connections or transit points along the route. This aims to provide a travel solution that offers flexibility in departure times while ensuring a streamlined and efficient journey with fewer stops.

Latest Departure Time	Total Price	Number of Flights
100	1100	4
120	1000	4
122	970	5
100	1000	6

Table 5.3: Latest departure path ordered by length

Query *Consider distance in the shortest path:*

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport), (c2:City)-[:LocatedAt]->
(a2:Airport), path = shortestPathPrice((a1)-[e:Flight*2..8]->(a2))
WHERE c1.name='Anchorage' AND c2.name='New York'
```

The goal of the information in the table [5.4] is to reach a destination by following the shortest distance route. This entails finding the most direct and efficient path that minimizes travel distance, ideal for situations where time efficiency and resource optimization are paramount.

Shortest Path	Total Price	Number of Flights
58	1100	4
60	1100	4
80	1100	4
90	1050	5
95	1000	6
110	970	7
125	930	8

Table 5.4: Shortest path ordered by distance

The figure [5.13] provides a visual representation of the comparative analysis conducted on the attributes of distance, price, and path length in routes connecting the origin and destination cities.

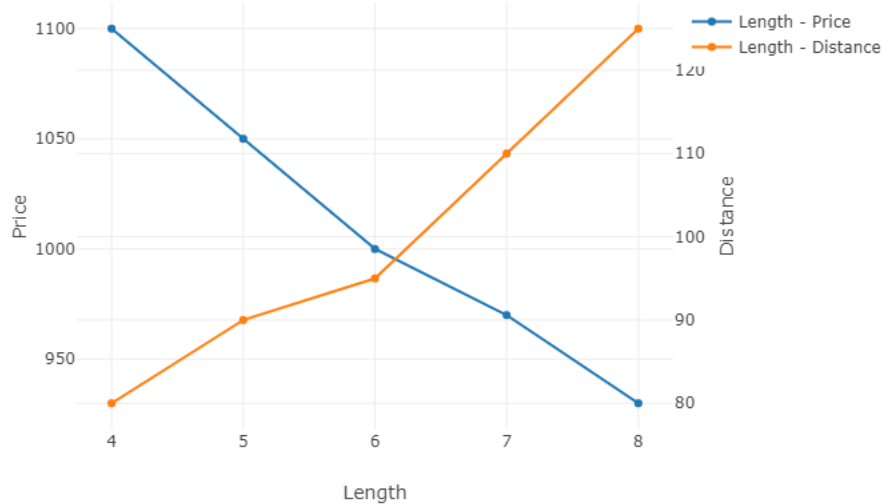

Figure 5.13: Shows how price impact on the length and distance

5.6 Evaluation

5.6.1 Execution time evaluation for continuous paths

In Figure [5.1, 5.2], illustrates the time it takes to execute different data set sizes and varying *continuous path* lengths. When N = 100,000, as the path length (L) increases, the execution times also increase, starting at approximately 2700 ms for L = 2 and reaching up to almost nearly 5000 ms for L = 8. In contrast, for N

= 1,000 and 10,000, the execution times remain relatively low. When N equals 1,000 and 10,000, the execution times exhibit fluctuations up to a path length 5, remaining consistently below 2,500 ms. However, beyond this point, their behavior becomes more similar, and they stabilize at around 2,200 ms.
On the figure [5.3, 5.4], The results for *pairwise continuous paths* are depicted. Similar to the continuous paths, as the number of nodes and path length increase, execution times also rise, but in this case, they are generally lower.

Figure [5.5] illustrates the divergence in execution times between continuous paths and pairwise continuous paths as the number of nodes and relationships in the data set increases. This graphical representation serves as a visual aid to understand how these path types adapt to more extensive networks. It sheds light on the influence of data set size on their respective execution times and provides valuable insights into the scalability of these path semantics. As evident from the data, it is observable that the execution time for continuous paths exceeds that of pairwise continuous paths in the same scenario. However, a notable observation is that when the data set comprises 10,000 nodes, the execution times for both path types become remarkably similar, demonstrating a convergence in their performance under these conditions.

5.6.2 Execution time evaluation for consecutive paths

Figures [5.6 5.7 5.8 5.9] present the findings of tests conducted on algorithms related to *earliest arrival, latest departure, shortest paths, and fastest paths.* In most cases, all the graphs exhibit a linear pattern. Due to significant variations in algorithm run times based on the sequences of paths and the system's inability to process this variability, the sequence was restricted to *Flight*8* instead of *Flight* for the purpose of analyzing specific paths. As expected, the execution time of the algorithm increases as the number of flights increases. The vertical axis represents time in milliseconds. Regarding the figures, the tests highlight a clear connection between the path's length and the execution time.

In the case of the *earliest arrival path* [5.6], the tests reveal that when the number of nodes (N) and relationships (R) is approximately 1000, starting the journey with a path length of 2 takes more than 2000 ms. Conversely, when N and R are set to 10000 or 100,000 and 10,000, respectively, they commence in less than 2000 ms.

In the case of the *latest departure algorithm* [5.7], when dealing with a data set consisting of 1000 nodes and 1000 relationships, the algorithm's execution time exhibits fluctuations before reaching a path length of 5, where it consistently hovers around 2400 milliseconds. Upon scaling the data set to 10000 nodes and 10000 relationships, the *Latest Departure* algorithm displays similar behavior, with execution times exceeding 2300 milliseconds for a path length of 2 and maintaining a similar pattern for longer path lengths. When applied to a larger data set comprising 100,000 nodes and 10,000 relationships, the algorithm's execution times continue to increase as the path length extends, consistently following this upward trajectory and reaching nearly 5000 milliseconds.

Analyzing the performance of the *Shortest Path* algorithm 5.8, notable trends in execution times across varying data sets are observed. When considering a data set with 1000 nodes and 1000 relationships, the *Shortest Path* algorithm demonstrates execution times spanning from 2170 milliseconds for a path length of 2 to 2263 milliseconds for a path length of 5. These execution times display minor fluctuations but consistently remain within this specific time frame. As the data set is expanded to encompass 10,000 nodes and 10,000 relationships, the algorithm's performance exhibits stability, with execution times ranging from 2119 milliseconds for a path length of 6 to 2573 milliseconds for a path length of 7. Although there are slight variations, the execution times generally adhere to this range. Upon further enlargement the data set to involve 100,000 nodes and 10,000 relationships, the *Shortest Path* algorithm mirrors a comparable pattern. The execution times progressively increase with the elongation of the path length, ultimately peaking at approximately 5062 milliseconds for a path length of 8.

Exploring the performance of the *Fastest Path* algorithm [5.9], distinct patterns in execution times across varied data set configurations are uncovered. In a data set featuring 1000 nodes and 1000 relationships, the *Fastest Path* algorithm yields execution times ranging from 2118 milliseconds for a path length of 2 to 2297 milliseconds for a path length of 5. Notably, the execution times exhibit minor fluctuations but consistently fall within this range. During the transition to a larger data set comprising 10,000 nodes and 10,000 relationships, dependable performance is upheld by the *Fastest Path* algorithm, with execution times spanning from 1745 milliseconds for a path length of 2 to 2255 milliseconds for a path length of 4. While slight variations exist, the execution times generally align within this interval. Elevating the data set to a grander scale of 100,000 nodes and 10,000 relationships, the "Fastest Path" algorithm showcases a parallel pattern. Execution times incrementally increase as the path length extends, culminating at approximately 4970 milliseconds for a path length of 8. The *Fastest Path* algorithm demonstrates a consistent relationship between the path length and execution time across varying data set sizes. As the path length elongates, execution times tend to rise, preserving this characteristic irrespective of the data set's magnitude.
Figures [5.10, 5.11, 5.12] offer a comprehensive comparison of various path types, both continuous and consecutive, including continuous path, pairwise continuous path, earliest arrival path, latest departure path, fastest path, and shortest paths. These comparisons are conducted for data set featuring different node quantities, specifically 1000, 10,000, and 100,000 nodes.

In Figure [5.10], with a node count of 1000 and a path length of 2, the execution times for all path types exceed 2000 milliseconds. This trend continues when examining the general pattern, with the execution times remaining consistently higher than in other scenarios with longer path lengths. In contrast, as the path length increases from 3 to 8, the execution times exhibit stability, remaining relatively constant across all path types.

In Figure [5.11], an insightful analysis of path execution times is provided for scenarios where the node count is set at 10,000. The execution times for all path types largely fall within the range of 2000 to 2500 milliseconds. Notably, the

Fastest Path stands out as an exception when the path length is 2, with execution times hovering around 1700 milliseconds. Conversely, the Earliest Arrival Path exhibits the longest execution time among the path types when the path length extends to 3, reaching approximately 2500 milliseconds. Similarly, the Shortest Path, when the path length is 7, surpasses the 2500 milliseconds threshold. These results unveil the varying performance of different path types under the same node quantity and distinct path lengths, emphasizing the intricate dynamics that influence their execution times.

Figure [5.12] extends the examination of path execution times to a significantly larger node count, with the number set at 100,000. In this scenario, the algorithms demonstrate a distinct pattern as the node count increases. The path length, in particular, plays a pivotal role as it significantly influences execution times. As the number of nodes expands, all path types exhibit an upward trajectory in their execution times, with noticeable increases as path lengths extend. These changes reflect the growing complexity of the graph as more nodes are added, impacting the efficiency of path computation. In the final stage, with a path length of 8, all path types converge, reaching an execution time of nearly 5000 milliseconds. This convergence underscores the consistent trend of rising execution times with larger node counts and longer path lengths, highlighting the interplay between these variables in shaping algorithm performance.

5.6.3 Assessment of consecutive paths in terms of pricing

The table [5.1] provides insights into selecting flight routes between Anchorage and New York, considering earliest arrival time, flight segments, and price. Price significantly impacts path choice, prioritizing cost-efficiency. For instance, at a price of 930, a 190 arrival time, not the earliest, becomes optimal, reflecting how price affects the number of flights, 8 in this case. At 970, a 150 arrival time, not the earliest, becomes optimal with 7 flights. This pattern continues at prices 1000 and 1050, highlighting price's influence on cost-efficient routes with fewer flights.

In table [5.2] , we encounter budget constraints while striving for efficient routes, with a primary focus on earliest arrival time, total price, and flight count. When our budget is limited to 930, we emphasize achieving an earliest arrival time of 190, which results in an 8-flight route, showcasing the pivotal influence of budget on travel decisions. Similarly, when adhering to a 970 budget, our selection prioritizes a 150 arrival time with 7 flights, underscoring the delicate balance between budget constraints and travel efficiency. In both instances, we are navigating the realm of budget-limited travel, seeking routes that align closely with our financial constraints.

Table [5.3] assesses latest departure paths ordered by length, examining the interplay of key factors: latest departure time, total price, and the number of flights.
In the first row, a departure time of 100 yields a 4-flight route with a total cost of 1100, showcasing the impact of budget constraints. In the second row, a 120 departure time maintains a 1000 cost and 4 flights, highlighting the subtle balance between departure time, price, and flight count. The third and fourth rows

present variations, with 122/970/5 and 100/1000/6 combinations, emphasizing the intricate relationships that influence path selection dynamics. This analysis offers profound insights into travelers' decision-making processes when navigating these complex variables.

Table [5.4] provides a comprehensive evaluation of shortest path options, organized based on travel distance. This assessment explores the intricate balance between path distance, total cost, and flight count. The primary objective is to determine the optimal route for reaching the destination while prioritizing minimal travel distance. In the initial row, a 58-unit path offers efficiency within a budget limit of 1100 and includes 4 flights. Subsequent rows exhibit variations in distance (60, 80), maintaining a total cost of 1100 and involving 4 flights, emphasizing the importance of cost-efficient travel. The following rows showcase different distance options (90 to 110 units) while gradually decreasing the total cost from 1050 to 930, which results in an increase in flight count from 5 to 8. These variations highlight the intricate dynamics of minimizing travel distance within various budget constraints, affecting the number of flights and the optimization of travel resources.

Ultimately, Figure [5.13] illustrates a contrast involving Path Length, Price, and Distance. Notably, as the path length extends, it has a direct impact on distance, while simultaneously resulting in a decrease in price.

5.7 Summary

The evaluation section provides an in-depth analysis of the execution times for continuous paths, pairwise continuous paths, and consecutive paths with varying attributes. In the continuous path execution time assessment, it is evident that execution times increase as the path length grows, particularly pronounced when the data set comprises 100,000 nodes. For pairwise continuous paths, the execution times are generally lower but still exhibit a similar pattern of growth with path length and data set size.
The evaluation of consecutive paths involving earliest arrival, latest departure, shortest path, and fastest path algorithms reveals a linear relationship between path length and execution time. This relationship is consistent across different data set sizes, with the execution times increasing as the path length extends. The analysis demonstrates how these algorithms perform under varying conditions, such as different numbers of nodes and relationships.

The assessment of consecutive paths concerning pricing illuminates the impact of price on route selection. Price significantly influences path choice, favoring cost-efficient routes with fewer flights. For instance, at a price of 930, an optimal route with a 190 arrival time, though not the earliest, is chosen, resulting in 8 flights. A similar pattern continues at prices 970, 1000, and 1050, highlighting how price considerations affect path selection.
In scenarios where budget constraints are imposed, the evaluation underscores the delicate balance between budget, earliest arrival time, and flight count. With a limited budget of 930, travelers prioritize routes that achieve an earliest arrival time of 190, leading to an 8-flight route. When adhering to a 970 budget, a 150

arrival time with 7 flights becomes the preferred choice, demonstrating the intricate interplay of budget and travel efficiency.
The analysis of latest departure paths ordered by length reveals how departure time, total price, and the number of flights influence route selection. Budget constraints become apparent in the choice of a 100 departure time with 4 flights and a total cost of 1100. The variations in departure times and costs in subsequent rows emphasize the intricate relationships that travelers navigate when selecting routes.
The assessment of shortest path options organized by travel distance provides insights into the balance between path distance, total cost, and flight count. Travelers seek to minimize distance while optimizing their budget. The analysis showcases how different distances, budgets, and flight counts impact path selection, highlighting the complexities involved in minimizing travel distance.

Overall, the evaluation section offers a comprehensive understanding of how various attributes, including path length, price, budget, and departure time, influence path selection within a flight transportation network. It sheds light on the practical implications of these factors and their impact on algorithm performance and route choices.

Chapter 6

Conclusion

In conclusion, this thesis embarks on an exploration of temporal property graphs, shedding light on their practical relevance in real-world scenarios. While graph databases have become a dominant choice for data representation and analysis, the temporal dimension has often been sidelined in prior research. Building upon a model introduced in The paper[5] published in the VLDB Journal, this study delves into the intricate task of modeling, storing, and querying temporal property graphs, introducing the T-GQL query language and an array of algorithms capable of computing various temporal path semantics, from continuous to pairwise continuous, and consecutive paths.

Recognizing the practical disparities between synthetic data set and real-world complexities, this thesis introduces the critical "Price" attribute, a reflection of real-world aviation dynamics, into the analysis. The exploration of how price influences algorithmic outcomes offers a glimpse into the complex interplay between cost considerations and path selection. Although real-world aviation entails multifaceted attributes, the primary focus remains on price and distance. This endeavor aims to unravel how price variations impact algorithmic results, providing insights into how practical considerations influence path selection.

The culmination of this thesis includes a comprehensive set of experiments conducted on a synthetic data set, encompassing three types of paths: continuous, pairwise continuous, and consecutive. These experiments served a dual purpose: firstly, to validate the viability of the methods proposed, and secondly, to evaluate the variables that influence performance, including queried path lengths and graph dimensions. Synthetic data set simulating flight transportation were generated, featuring sizes exceeding 100,000 rows, composed of 100,000 nodes, 60,000 edges, and more than 10,000 flights. By addressing the intricacies of temporal property graphs and integrating real-world attributes like price, this research bridges the gap between theory and practice, offering valuable insights into the practical dynamics of temporal graphs in real-world applications.

Bibliography

[1] Arenas M. Barceló P. Hogan-A. Reutter J.L. Vrgoc Angles, R. Foundations of modern query languages for graph databases. *ACM Comput. Surv. 50(5), 68:1–68:40*, 2017.

[2] Gutierrez C. Angles, R. Survey of graph database models. *ACM Comput. Surv. 40(1), 1:1–1:39*, 1995.

[3] R. Angles. The property graph database model. *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, Volume 2100 of CEUR Workshop Proceedings. CEUR-WS.org*, 2018.

[4] Renzo Angles. The property graph database model. *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia*, May 21–25, 2018.

[5] Matías Perazzo Valeria Soliani Alejandro Vaisman Ariel Debrouvier, Eliseo Parodi. A model and query language for temporal graph databases. *The VLDB Journal 30*, 2021.

[6] Dijkstra1959. A note on two problems in connexion with graphs. *Numer. Math. 1, 269–271*, 1959.

[7] Green A. Guagliardo P. Libkin L. Lindaaker T. Marsault V. Plantikow S. Rydberg M. Schuster M. Selmer P. Taylor A Francis, N. Formal semantics of the language cypher. *CoRR arXiv:1802.09984*, 2018.

[8] Green A. Guagliardo P. Libkin L. Lindaaker T. Marsault V. Plantikow S. Rydberg M. Selmer P. Taylor A. Francis, N. Cypher: an evolving query language for property graphs. *Das, G., Jermaine, C.M., Bernstein, P.A. (eds.) Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Hou*, 2018.

[9] Hurtado C.A. Vaisman A.A. Gutiérrez, C. Introducing time into rdf. ieee trans. *Knowl. Data Eng. 19(2), 207–218*, 2007.

[10] Hurtado C.A. Vaisman A.A.m Gómez-Pérez A. Euzenat J. Gutiérrez, C. Temporal rdf. *Gómez-Pérez, A., Euzenat, J. (eds.) The Semantic Web: Research and Applications, Second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece, May 29–June 1, 2005, Proceedings, Volume 3532 of Lecture Notes in Computer Science, pp. 93–107. Springer*, 2005.

[11] Olaf Hartig. Reconciliation of rdf* and property graphs. *CoRR arXiv:1409.3288`https://arxiv.org/abs/1409.3288`*, 2014.

[12] https://neo4j.com/. User Defined Procedure. `https://neo4j.com/docs/getting-started/cypher-intro/procedures-functions/`. Retrieved on 2023, 08, 31.

[13] Tsotras V.J. Huo, W. Efficient temporal shortest path queries on evolving social graphs. *Conference on Scientific and Statistical Database Management, SSDBM, Aalborg, Denmark, June 30–July 2, 2014, pp. 38:1–38:4*, 2019.

[14] Jim Webber Ian Robinson. Graph databases. *O'Reilly Media, Sebastopol*, 2013.

[15] Suryanarayana Sripada Opher Etzion, Sushil Jajodia. Temporal databases: Research and practice (the book grow out of a dagstuhl seminar. *Volume 1399 of Lecture Notes in Computer Science. Springer (1998)*, June 23-27 1997.

[16] Vaisman A. Rizzolo, F. Temporal xml: modeling, indexing, and query processing. *VLDB J. 1179–1212(5), 39–65*, 2008.

[17] Pitoura Semertzidis, K. Top-k durable graph pattern queries on temporal graphs. *IEEE Trans. Knowl. Data Eng. 31(1), 181–194*, 2019.

[18] R.T. Snodgrass. A comparison of current graph database models. *Proceedings of ICDE Workshops, Arlington, VA, USA, pp. 171–177*, 2012.

[19] Clifford J. Gadia S Tansel, A. Temporal databases: Theory, design and implementation. *Benjamin/Cummings, New York*, 1993.

[20] Cheng J. Huang S. Ke-Y. Yi L. Yanyan X. Wu, H. Path problems in temporal graphs. *PVLDB 7(9), 721–732*, 2014.