Graph Data Processing and Analysis: From Algorithms to System Development

Shunyang Li

A thesis in fulfilment of the requirements for the degree of

Master of Philosophy

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

19/06/2022

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: **Shunyang**

First name: **Li** Other name/s:

Abbreviation for degree as given in the University calendar: **MPhil**

School: **School of Computer Science and Engineering** Faculty: **Faculty of Engineering**

Title: Graph Data Processing and Analysis: From Algorithms to System Development

Abstract

There are many real-world application domains where data can be naturally modelled as graphs, such as social networks and computer networks. The amount of data generated and published is rapidly increasing with the explosion of information. Effective storage of graph data and querying has become a significant challenge; hence the graph database is emerging to address this challenge. Graph databases have the unique advantages of modelling and querying complex relationships, capturing and navigating complex data relationships and recursive path querying when handling graph data. In this thesis, we enhance graph databases from both system and algorithm perspectives.

Firstly, we propose two systems, SQL2Cypher and FSPS, to improve the usability and efficiency of graph databases. SQL2Cypher automatically migrates data from a relational database to a graph database. This system also supports translating SQL queries into Cypher queries. FSPS is the first FPGA-based system for accelerating graph queries on the massive graph. FSPS has the following features 1) a CPU-FPGA co-designed framework, 2) a fully pipelined FPGA execution, and 3) reduced data transfer from FPGA's external memory. FSPS supports the two most fundamental types of graph queries, namely subgraph and path queries. Performance evaluation shows that FSPS outperforms the most popular graph database Neo4j by up to three orders of magnitude. All the draft demo videos can be found at https://www.youtube.com/watch?v=oSpHtJ8iVio and https://www.youtube.com/watch?v=eGaeBrVTJws.

Secondly, the graph database does not widely support the cohesive subgraph models (i.e., Neo4j and PatMat). Many real-world relationships can be naturally represented as a bipartite graphs such as customer-product, user-item, and author-paper. Therefore, we use efficient construct algorithms to investigate the *bipartite hierarchy* model. The *bipartite hierarchy* is the first model to discover the hierarchical structure of bipartite graphs based on the concept of (α, β)-core and graph connectivity. These algorithms can effectively identify the affected regions to limit computation scope and avoid re-building the bipartite hierarchy from scratch. Extensive experiments on 10 real-world graphs demonstrate the effectiveness of the proposed bipartite hierarchy and validate the efficiency of our hierarchy constructions algorithms.

Originality Statement

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

19/06/2022

Copyright Statement

19/06/2022

Authenticity Statement

I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis.

19/06/2022

Thesis submission for the degree of Master of Philosophy

Thesis Title and Abstract | **Declarations** | **Inclusion of Publications Statement**

Corrected Thesis and Responses

ORIGINALITY STATEMENT

☑ I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

COPYRIGHT STATEMENT

☑ I hereby grant the University of New South Wales or its agents a non-exclusive licence to archive and to make available (including to members of the public) my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known. I acknowledge that I retain all intellectual property rights which subsist in my thesis or dissertation, such as copyright and patent rights, subject to applicable law. I also retain the right to use all or part of my thesis or dissertation in future works (such as articles or books).

For any substantial portions of copyright material used in this thesis, written permission for use has been obtained, or the copyright material is removed from the final public version of the thesis.

AUTHENTICITY STATEMENT

☑ I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis.

Thesis submission for the degree of Master of Philosophy

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in the candidate's thesis in lieu of a Chapter provided:

- The candidate contributed **greater than 50%** of the content in the publication and are the "primary author", i.e. they were responsible primarily for the planning, execution and preparation of the work for publication.
- The candidate has obtained approval to include the publication in their thesis in lieu of a Chapter from their Supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis.

☑ The candidate has declared that **some of the work described in their thesis has been published and has been documented in the relevant Chapters with acknowledgement**.

A short statement on where this work appears in the thesis and how this work is acknowledged within chapter/s:

> The introduction, motivation, system overall and demonstration sections from "SQL2Cypher: Automated Data and Query Migration from RDBMS to GDBMS" in the International Conference on Web Information Systems Engineering (published) is contained in parts in Chapters 1 and 3.

Candidate's Declaration

I declare that I have complied with the Thesis Examination Procedure.

Abstract

There are many real-world application domains where data can be naturally modelled as graphs, such as social networks and computer networks. The amount of data generated and published is rapidly increasing with the explosion of information. Effective storage of graph data and querying has become a significant challenge; hence the graph database is emerging to address this challenge. Graph databases have the unique advantages of modelling and querying complex relationships, capturing and navigating complex data relationships and recursive path querying when handling graph data. In this thesis, we enhance graph databases from both system and algorithm perspectives.

Firstly, we propose two systems, SQL2Cypher and FSPS, to improve the usability and efficiency of graph databases. SQL2Cypher automatically migrates data from a relational database to a graph database. This system also supports translating SQL queries into Cypher queries. FSPS is the first FPGA-based system for accelerating graph queries on the massive graph. FSPS has the following features 1) a CPU-FPGA co-designed framework, 2) a fully pipelined FPGA execution, and 3) reduced data transfer from FPGA's external memory. FSPS supports the two most fundamental types of graph queries, namely subgraph and path queries. Performance evaluation shows that FSPS outperforms the most popular graph database Neo4j by up to three orders of magnitude. All the draft demo videos can be found at https://www.youtube.com/watch?v=oSpHtJ8iVio and https://www.youtube.com/watch?v=eGaeBrVTJws.

Secondly, the graph database does not widely support the cohesive subgraph models (i.e., Neo4j and PatMat). Many real-world relationships can be naturally represented as a bipartite graphs such as customer-product, user-item, and author-paper. Therefore, we use efficient construct algorithms to investigate the *bipartite hierarchy* model. The *bipartite hierarchy* is the first model to discover the hierarchical structure of bipartite graphs based on the concept of (α, β)-core and graph connectivity. These algorithms can effectively identify the affected regions to limit computation scope and avoid re-building the bipartite hierarchy from scratch. Extensive experiments on 10 real-world graphs demonstrate the effectiveness of the proposed bipartite hierarchy and validate the efficiency of our hierarchy constructions algorithms.

Acknowledgement

Firstly, I am very grateful to my supervisors Prof. Xuemin Lin and Prof. Wenjie Zhang. They helped me a lot and gave me a lot of guidance for selecting my topic. I learned from them to have high expectations of myself; then, I can do better. And I learned from them the complete process of conducting research and what kind of spirit I should use to approach research.

Secondly, my colleagues Dr. Kai Wang and Dr. Zhengyi Yang gave me a lot of help when designing the algorithms, experimenting, and writing papers. They gave me a lot of guidance when doing my research and helped me avoid some research mistakes.

Besides, special thanks go to the group members: Mr. Kongzhang Hao, Mr. Yizhang He, Mr. Gengda Zhao, Ms. Yuting Zhang, Mr. Han Zhang, Dr. Xiaoshuang Chen, Dr. Chenji Huang, Dr. Hanchen Wang, Dr. Michael Ruisi Yu and Mr. Deming Chu. The time we spent together will be memorized forever.

Finally, I once again thank the people who taught and helped me in this work.

Publications and Presentations

List of Publications

- Li, S., Yang, Z., Zhang, X., Zhang, W., & Lin, X. (2021, October). SQL2Cypher: Automated Data and Query Migration from RDBMS to GDBMS. In International Conference on Web Information Systems Engineering (pp. 510-517). Springer, Cham.
- Wang, K., Zhang, W., Lin, X., Zhang, Y., & Li, S. (2021, November). Discovering Hierarchy of Bipartite Graphs with Cohesive Subgraphs. IEEE International Conference on Data Engineering (has been accepted).

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

BFS	Breadth-first search
BRAM	Small sizes of on-chip memory of FPGAs
CST	candidate search tree
DRAM	FPGA's external memory
FPGAs	Field-programmable gate arrays
FSPS	FPGA-based Subgraph and Path querying System
GDBMS	Graph database management system
RDBMS	Relational database management system

Chapter 1

Introduction

Graph has been playing an increasingly important role in data management with the prevalence of graph data in different application domains in recent years, such as social networks [1, 2], road networks [3–5] and protein-protein interaction networks [6, 7]. With the increased data, storing and querying highly connected data is a significant problem. Since it is hard for relational database management systems (RDBMS) to capture the relationships and inherent graph structure of data and are inappropriate for storing highly connected data, graph databases have emerged to address the challenges of high data connectivity. Graph database management systems (GDBMS) are among the most fundamental infrastructure when managing graph data and have received a lot of attention from researchers and programmers globally [8]. GDBMS have the unique advantages of modelling and querying complex relationships, capturing and navigating complex data relationships and recursive path querying when handling graph data.

It has been more than 50 years since E.F. Codd introduced the concept of relational databases in 1970 [9]. Therefore, for legacy reasons, relational databases are still the majority in the market, even when storing highly connected data. However, in relational databases, there is a significant system overhead for highly connected data or complex relational join operations, which can lead to long execution times and excessive consumption of computer resources [10]. Therefore, the need for converting relational databases to graph databases emerged. Most of the current algorithms for graph databases are based on CPU design. Compared with CPUs, field-programmable gate

arrays (FPGAs) have significant advantages in parallelism and energy efficiency over CPUs and GPUs. Furthermore, FPGAs are widely deployed by many enterprises and cloud service providers nowadays. Currently, the graph database does not support many algorithms, graph types and models (i.e., bipartite graph and (α, β)-core model). This thesis investigates graph databases from both system and algorithm perspectives.

1.1 Motivations

1.1.1 SQL2Cypher: Automated Data and Query Migration from RDBMS to GDBMS

However, relational database management systems (RDBMS) still comprise the majority share of the database market for legacy reasons, even when storing highly connected data [10]. Querying highly connected data in an RDBMS usually requires complex join operations and significant system overhead, which can lead to a long execution time [11]. Hence, there naturally emerges the demand for migrating from RBDMS to GDBMS. In this work, we demonstrate *SQL2Cypher*, an automated tool for migrating data from RBDMS to GDBMS.

Migrating data from RDBMS to GDBMS involves redefining data schema, mapping relations and rewriting queries. The migration process is often time-consuming and labour-intensive. The high time and labour costs are significant reasons why companies choose to keep their legacy RDBMS. To address this problem, several automated tools [12–14] have been proposed to migrate data from RDBMS to GDBMS. However, we find that they are either outdated or incomplete. For example, [12] focuses on XQuery and [14] focuses on RDF data, while ignoring the nowadays more widely adopted property graph model [8]. The open-source tool Neo4j-ETL [13] allows the user to import data from relational databases to the popular graph database Neo4j. However, it does not provide automatic query translation, and users have to rewrite all previous SQL queries to Cypher queries manually [15] (the graph querying language used by Neo4j). More critically, Neo4j-ETL is not well maintained and has many issues at present[1] (e.g., error when loading large dataset and error when mapping relations).

[1]https://github.com/neo4j-contrib/neo4j-etl/issues

Challenges. To convert RDBMS to GDBMS, a basic approach is to covert all the tables in RDBMS to nodes in GDBMS. (1) However, this approach may ignore the relationship between tables and take up a lot of storage space. And occasionally, the tables may contain very similar data and need large storage space. Our system needs to convert RDBMS to GDBMS with less memory usage to store the relationships between tables. (2) Another challenge is to convert the SQL queries to Cypher queries because we need to consider all the relationships between tables when converting the queries.

Our Approaches. For the first challenge, we implement an optimization strategy based on [16]. The strategy is to convert the *join table* to an edge and the attributes in the *join table* as properties of the edge. This strategy will save memory usage and make the data more adaptable to the nature of the graph database. And, we implement duplication detection [17] further to improve the speed and the quality of the migration. For the second challenge, we store all the relationships between converted tables as a graph structure in the file. Then we can translate the queries based on the relationship graph.

1.1.2 FSPS: Accelerating Subgraph and Path Queries Using FPGA

Considerable efforts are made in industry and academia to develop efficient systems for subgraph and path queries [18–22]. However, almost all solutions are developed on CPUs which have the following limitations when handling graph data: 1) CPUs do not offer flexible high-degree parallelism, and 2) CPU caches do not work effectively for irregular graph processing with the limited locality.

With the recent advance of field-programmable gate arrays (FPGAs), people are provided with a new alternative to accelerate graph computations at the hardware level. FPGAs have shown significant advantages over multi-core CPUs in parallelism due to their pipelining design and highly efficient hardware circuit. Furthermore, compared with GPUs, FPGAs are more energy-efficient [23]. FPGAs are now widely deployed by enterprises and cloud service providers such as Microsoft, Alibaba, Tencent, Huawei, and Amazon Web Services (AWS).

Challenges. To design a graph database system, we have the following challenges: (1) storing the data of nodes and edges for quick access, and all the nodes and edges have properties. (2) Design a query approach that is easy to express for the user. (3) Design the most basic two algorithms (subgraph matching and path queries) based on FPGA for the graph database.

Our Approaches. For the first challenge, we use the RocksDB[2] (a key-value database) to store the data graph on disk and build an additional inverted index of user-defined properties to accelerate the query speed. We design a new query approach for the second challenge that lets the user draw the query pattern with properties in the frontend. This approach could save the user time in learning a new Cypher[3] language. For the last challenge, we adopt the subgraph matching algorithm in [24] and a simple path enumeration algorithm in [25] for path queries for fast and efficient subgraph and path queries on FPGA.

1.1.3 Discovering Hierarchy of Bipartite Graphs with Cohesive Subgraphs

Finding the hierarchy of graphs is a popular research topic in the field of graph analysis. Existing studies mainly focus on finding the hierarchy of general (unipartite) graphs based on the models of k-core [26, 27], k-truss [28, 29], k-ECC [30, 31], and nucleus [32, 33]. However, these models are unsuitable for bipartite graphs since they do not consider the special structure of bipartite graphs (i.e., formed by two different vertex layers). Note that the two vertex layers represent two different types of entities and are usually of different scales. Alternatively, one may consider using graph projection [34] to first project the bipartite graph into a unipartite graph and then build a hierarchy based on the projected graph. However, it is not practical in many real-world cases since graph projection can usually cause the explosion of edges/triangles and information loss [35, 36].

Only a few studies [35, 37, 38] focus on finding the hierarchy of bipartite graphs directly, which are all based on the butterfly structure (i.e., the 2×2-biclique). Specifically, they study the k-bitruss (or called k-wing) and k-tip decomposition problems. Here k-bitruss and k-tip are the subgraphs that

[2]https://rocksdb.org/

[3]https://neo4j.com/developer/cypher/

require each edge/vertex is contained in at least k number of butterflies, respectively. However, these butterfly-based models are prone to include "loosely connected" vertices in a k-bitruss/k-tip with a high k value since they do not consider two vertex layers separately. Such circumstances can usually happen if some vertices are linked to two hub vertices (i.e., high degree vertices). For instance, consider H as a complete bipartite graph with 2 upper vertices and 1,001 lower vertices. Then, a "loosely connected" lower vertex $v \in H$ with only two incident edges is contained in a k-bitruss with k = 1000. The butterfly-based decomposition approaches can generate misleading information since real-world graphs are usually skewed and contain many hub vertices. The following challenges need to be addressed to make our idea practically applicable.

1. How to bound the *bipartite hierarchy* size and the constructing time.
2. How to efficiently maintain the *bipartite hierarchy*.

Our Approaches. To address the first challenge, we propose the *bipartite hierarchy* model, which is the first to reveal the hierarchy of bipartite graphs based on (α, β)-cores and graph connectivity. The bipartite hierarchy has a two-dimensional structure to analyze bipartite graphs with different granularity levels. Notably, it only has a linear space usage and can depict the hierarchical tree structure of bipartite graphs. Because for every bipartite hierarchy the vertex only appeared in one tree node. We will explain the implementation details in Chapter 5. By utilizing the nested property of (α, β)-core and exploring possible cost-sharing, we also propose efficient algorithms HC-BU to construct the bipartite hierarchy. For the second challenge, we present algorithms HM-Ins and HM-Del to maintain the bipartite hierarchy incrementally regarding the edge insertion/deletion cases. The proposed algorithms can effectively identify the affected regions to limit the computation scope and achieve high efficiency.

1.2 Contributions

In this section, we will summarize our contributions to this thesis.

- We propose SQL2Cypher that can convert the RDBMS to GDBMS. SQL2Cypher could represent the relationships between tables and translate the query language based on these migrated tables.
- We develop and demonstrate the prototype of the first FPGA-based subgraph and path querying system, called FSPS. Specifically, FSPS has the following features. (1) A CPU-FPGA co-designed architecture. (2) Fully pipelined execution on FPGA. (3) Reduced data transfer from FPGA's external memory.
- Based on the nested property of (α, β)-core and exploring possible cost-sharing, we propose an algorithm HC-BU to build the *bipartite hierarchy* with linear space usage. We also present an efficient algorithm HM-Ins and HM-Del for maintaining the *bipartite hierarchy*. And we conduct comprehensive empirical studies on 10 real-world bipartite graphs.

1.3 Organization

This dissertation is organized as follows.

- Chapter 2 reviews the existing works related to our work in this thesis.
- Chapter 3 presents our approaches for implementing SQL2Cypher.
- Chapter 4 presents the implementing detail of FSPS.
- Chapter 5 presents our algorithms for *bipartite hierarchy*.
- Chapter 6 concludes our research and provides several possible future directions.

Chapter 2

Literature Review

This section will revisit some existing solutions of subgraph pattern matching and bipartite graph (k-core, (α, β)-core). Specifically, (1) we will review some graph data storage implementations, (2) we will revisit some distributed subgraph pattern matching and (3) we will revisit the models on the bipartite graph, such as (α, β)-core and k-core. Before revisiting articles, we will introduce some fundamental definitions.

2.1 Definitions

Definition 1. ***(Vertex Degree)*** *The degree of a vertex of a graph is the number of edges that are incident to the vertex.*

Definition 2. ***(Unlabeled Graph)*** *A unlabeled graph g can be defined as a 2-tuple, g = (V, E), where V is the vertex set and $E \subseteq V \times V$ is the edge set of g.*

Definition 3. ***(Subgraph)*** *A graph $g^{'}$ is a subgraph of g if and only if $\forall v \in V(g^{'}), v \in V(g)$ and $\forall (v_i, v_j) \in E(g^{'}, (v_i, v_j) \in E(g)$.*

Definition 4. ***(Bipartite Graph)*** *A bipartite graph $G = (U, V, E)$ is a graph that nodes set U and nodes set V are disjoint such that every edge in E is $E \subseteq U \times V$.*

Definition 5. ***(α, β)-core)*** *A bipartite graph G with two integer α and β, (α, β)-core can be denoted as $C_{\alpha,\beta}$. $C_{\alpha,\beta}$ has two node set $U' \subseteq U(G)$ and $V' \subseteq V(G)$. For all nodes in U' have degree at least α and for all nodes in V' have degree at least β.*

Definition 6. ***(k-core)*** *Given a graph G and an integer k. k-core is the largest subgraph of G in which all vertices have degree at least k.*

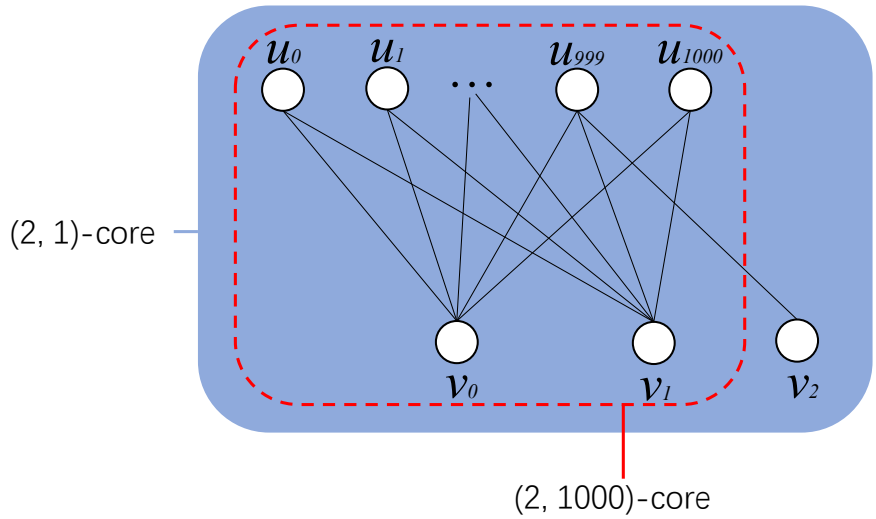


Figure 2.1: Definition examples

Example 1. *According to Figure 2.1, we present several examples of definitions. Firstly, the graph in Figure 2.1 is an unlabelled bipartite graph. The degree of u_0 is 2, since u_0 connects v_0 and v_1. The blue area in the figure represents the (2,1)-core, and the square formed by the red dotted line represents the (2,1000)-core.*

2.2 Graph Storage

Efficient storage and data management are critical in graph databases and relational database files. Existing databases can be divided into two categories: row-based and column-based. The row-based database has the advantage of reading and writing efficiently, and the column-based database has the advantage of querying data. However, none of the existing storage solutions can effectively handle both the querying and column aggravation. Huang D et al. [39] presented an HTAP (hybrid

transactional and analytical processing) system based on a consensus algorithm and implemented a Raft-based HTAP database called TiDB.

The author [39] highlights the limitations of existing HTAP systems before explaining how the HTAP system they developed guarantees data freshness and isolation and efficiently handles transnational and analytical processing. Their HTAP system provided the learners (dedicated nodes) based on the Raft consensus algorithm. The learners will transform the data in row format into column format to provide high performance when handling analytical queries. In addition, the learners provide real-time OLAP (online analytical processing) queries by generating a columnar store. The system reduces the latency of replication to ensure data freshness. They also optimize HTAP requests based on row and column format data replicas.

They evaluated the system's OLAP and OLTP (online transactional processing) separately by using CH-benCHmark, consisting of TPC-H and TPC-C. For OLTP, OLTP and HTAP models, the result shows that in most cases, both throughput and latency are better than the existing distributed databases. Additionally, the evaluation of log replication delay shows that log replication delays no more than 300ms on ten warehouses. Most cases are less than 100ms. The benchmark result indicates that the system can achieve real-time analytical processing and efficiently execute OLAP and OLTP queries.

The HTAP system provides a new approach to solving OLAP and OLTP quires in a database compared to existing databases. Meanwhile, the system also contributed to real-time processing, consistency, freshness, and isolation. Chang F et al. [40] and Sivasubramanian s. [41] presented NoSQL systems which are Google Bigtable and DynamoDB. The features of these two systems are a flexible data structure and high scalability. However, data consistency has not been implemented well for these systems. Weak consistency may result in different thread processes that access the same data and display different results.

However, the weakness of the system is that it is a relational database. In the past, relational databases have been the mainstay of the database industry. However, relational databases have difficulty capturing relationships between highly connected data. De Virgilio R et al. [12] describe that traversal over highly connected data with a relational database requires a large number of

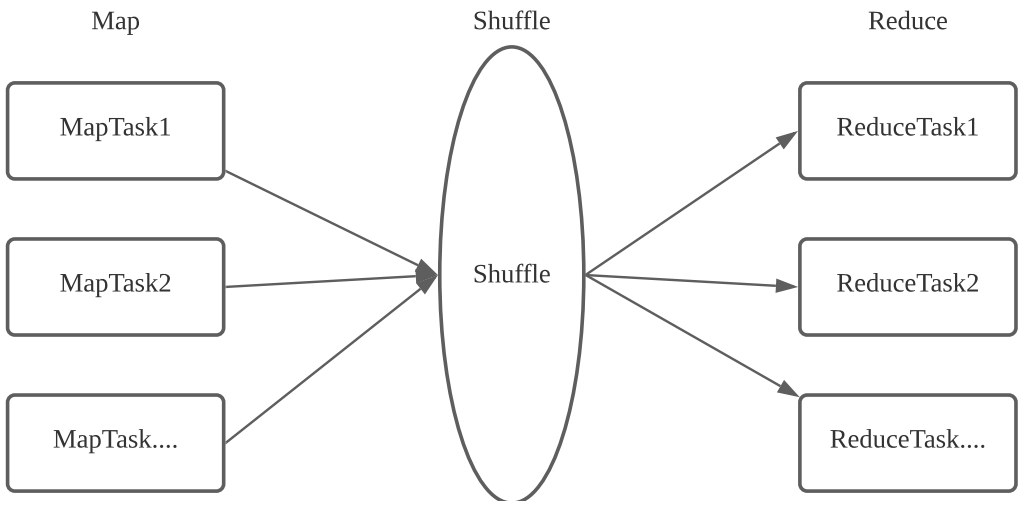


Figure 2.2: MapReduce Architecture

complex join operations, which is time-consuming.

In this paper [39], they proposed a new database design concept as a hybrid OLAP and OLTP model and provided comparative experiments in the database field. The experiment results offered a new idea to design the graph databases storage system.

2.3 Distributed Computing Frameworks

Distributed computing is a framework for dividing data that requires multiple calculations into small pieces of work calculated by various computers. After uploading the results, the results are unified and combined to produce the final result. There are many distributed frameworks such as MapReduce [42], Pregel [43], Spark [44], Flink [45], and so on.

MapReduce. MapReduce is a framework proposed by Google [42]. It greatly facilitates programmers to run their programs on distributed systems when they do not know distributed parallel programming. In addition, it also supports high scalability and high fault tolerance. The current software implementation specifies a Map function to map a set of key-value pairs into a new set of key-value pairs, and a concurrent Reduce function ensures that all mapped key-value pairs share the same set of keys. A MapReduce algorithm executes in a few rounds, for each round involves

three phases: map, $shuffle$ and $reduce$ as shown in Figure 2.2. Assume that our data is stored as a key-value in The Hadoop Distributed File System (HDFS). The main three steps are as follows.

- Map : At this stage, the map function processes the data row by row, encapsulating each row into a key-value pair ($(key : value)$). When the data has been processed, it is sent to the shuffle function.
- $Shuffle$: The key-value pairs $(key : value)$ comes from in Map stage are shuffled across all machines. The shuffle function will partition the data firstly. When the amount of data written reaches a pre-set queue, the spill thread is started to spill the data in the buffer to a temporary file on disk and sort and combine (optional) according to the key before writing. Finally, Shuffle allocates the data to the Reduce task function and makes sure that each reduced task gets the same key-value pairs (key: val_1), (key: val_2), ...
- $Reduce$: The Reduce function sorts all the data in a single merge. The Reduce function then sequentially processes the datasets with the same key as ($key : val_1, val_2, ...$) and writes the results to HDFS.

However, the weakness of the MapReduce framework is that it is hard to process the complex architecture. When constructing more complex processing architectures, you often need to coordinate multiple Map and Reduce tasks. However, each MapReduce step has the potential to go wrong. Many people have started designing their orchestration systems (orchestration) to handle these exceptions, which is a time waste. And MapReduce cannot do real-time processing because MapReduce saves intermediate results to disk.

The author [42] proposes a new distributed MapReduce framework. MapReduce has the advantage of processing large amounts of data offline and is also easy to develop. The drawback of MapReduce is that it cannot perform real-time streaming computation.

Pregel. Pregel is designed based on BSP (Bulk Synchronized Parallel) [43]. In BSP, a computation process consists of a series of global supersteps composed of three steps: concurrent computation, communication, and synchronization. The completion of synchronization marks the completion

of this superstep and the beginning of the next superstep. The criterion of the BSP model is bulk synchrony, which is unique in that it introduces the concept of superstep. A BSP program has both horizontal and vertical structures. Vertically, a BSP program consists of a series of serial supersteps.

BSP has the following features: (1) divides the computation into supersteps, effectively avoiding deadlocks, (2) separating processors and routers emphasizes the separation of computing and communication tasks, while routers accomplish only point-to-point messaging and do not provide functions such as combining, replication, and broadcasting, which obscures the specific interconnect network topology and simplifies communication protocols and (3) global synchronization using barrier synchronization, implemented in hardware, is a controlled coarse-grained level and provides an efficient way to perform tightly coupled synchronous parallel algorithms.

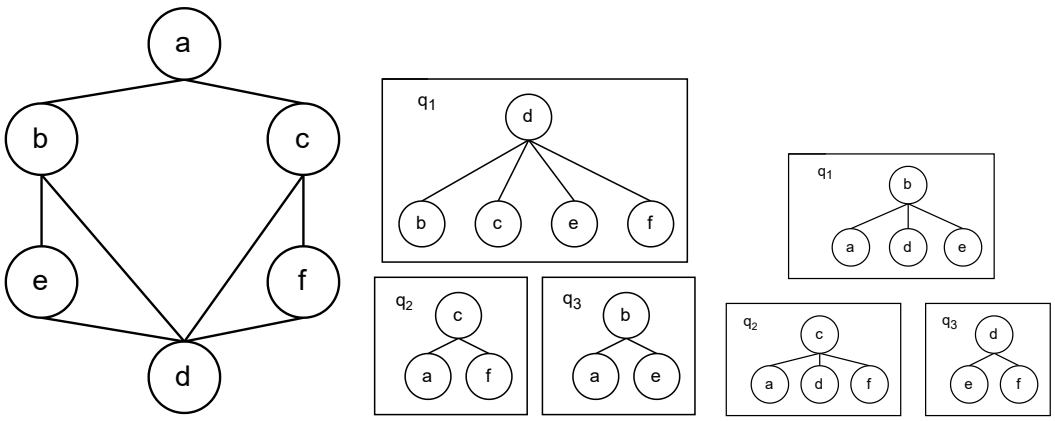
Pregel chooses a pure message-passing model, ignoring remote data reading and other shared-memory methods. Because (1) message passing is efficient enough to express itself without remote reading and (2) reading a value from a remote machine is subject to high latency. But the obvious drawback is that for vertices with many neighbours, the messages it needs to process are substantial and cannot be processed concurrently in this model. So, it is easy to crash under this computational model for natural graphs that conform to a power-law distribution.

[43] proposes a new distributed graph processing framework, Pregel, based on bulk synchronized parallel. The Pregel framework uses the Master-Worker cluster model. The typical edge-cutting method is used to store the graph, which divides the graph into many partitions, each containing some vertices and edges (outgoing edges) starting from these vertices. However, with a large number of vertices, it needs to process a massive number of messages which may cause downtime.

In summary, distributed computing frameworks and strategies provide a new and efficient way to compute. Therefore, we can use design distributed frameworks to accurate the graph computation. Distributed computing frameworks can also improve the computational performance of graph databases.

2.4 Join-based Subgraph Matching

The state-of-the-art subgraph matching algorithm is a join-based strategy [46]. The main categories are (1) Binary-join-based (BINJOIN), and (2) worst-case optimal [46]. This section analyses the binary-join-based algorithm (BINJOIN) and worst-case optimal join (GenericJoin). BINJOIN strategy computes subgraph matching through a series of binary joins. It first decomposes the original query graph into a set of concatenated units whose matching terms can satisfy the basic relations of the concatenation. Then, the basic relations are joined based on a predefined join order. For example, StarJoin [47], TwinTwigJoin [48] and CliqueJoin [49].



(a) A query q (b) A decomposition of query q (c) Another decomposition of query q

Figure 2.3: Decomposition And Order Selection

Definition 7. ***(Star Decomposition)*** *Given a pattern graph G and a node $v \in V(G)$, $star(v)$ denotes the star rooted at v with $N(v)$ as the child nodes. A start decomposition is p_0, p_1, ... p_k of a pattern graph G, such that there exists $v_{k_0}, v_{k_1}, ..., v_{k_t} \subseteq V(G)$ with p_0 = $star(v_{k_0}$ and p_x = $star(v_{k_x})$ $E(G_{x-1})$ for $i \leq x \leq t$).*

In the following section, this paper will be reviewed [47]. In this article, a new join algorithm is proposed, which is StarJoin. StarJoin applies to star as join a unit, a tree with depth 1, following left-deep join order. The query will be decomposed into a set of STwig. The following example will illustrate query partitioning. To process the query in Figure 2.3(a), it will be decomposed into STwig, as shown in Figure 2.3(b) and 2.3(c). Figure 2.3(c) shows another possible decomposition

which contains only 3 STwig. different decomposition will result in different query processing costs. To ensure the optimal solution of STwig Order Selection, two rules of edge selection are added to the algorithm (1) selects the edge that has been attached to the previously selected edge and (2) selects the edges incident to the nodes with high selectivity.

Example 2. *In Figure 2.3(b), the execution order can be: <q_1,q_2,q_3> and <q_2, q_1, q_3>. It is obvious that the first order is the optimal solution. Because, the root node of q_2 and q_3 will be processed in q_1. However, for the second order, the root node of q_1 is not bound by the result of q_2 (repeat computation).*

Compared to other join-based algorithms, the StarJoin algorithm generates fewer intermediate results during execution, thus avoiding a large amount of computing time due to decomposition optimization and execution order optimization. However, StarJoin still encounters scalability problems because many matches are generated when evaluating stars with many edges.

Ngo et al. proposed a worst-case optimal join algorithm GenericJoin [50]. Before we review this algorithm, we will introduce what the worst-case optimal join is. Given a query vertex set $\{v_0, v_1, ..., v_n\}$, the worst-case optimal strategy first computes all matches for $\{v_0\}$ that can appear in the result, then matches for $\{v_0, v_1\}$, after that matches for $\{v_0, v_1, v_2\}$, and so on, until the result is constructed. Given a query Q and the data graph G, the maximum possible result set can be denoted as $R_G(Q)$. The algorithm is worst-case optimal if all intermediate result aggregation can be bounded by $O(|R_G(Q)|)$. The algorithm uses the optimal join in the worst case so that the size of the intermediate result does not exceed the final result.

In conclusion, the subgraph matching algorithm provides operations commonly used in graph databases. Subgraph matching algorithms are supported by most graph databases, such as PatMat [51], Neo4j [19], etc. The subgraph matching algorithm provides an effective computational operation for our graph database.

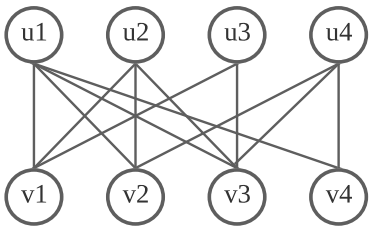


Figure 2.4: An example of (α, β)-core

2.5 Core like Graph Processing

(α, β)-core decomposition. The (α, β)-core is a maximal subgraph of the bipartite graph G such that the vertices on the upper or lower level have at least α or β neighbours, respectively, and is initially recommended for the fault-tolerant group [52]. For example, as shown in Figure 2.4, the graph is a bipartite graph shown is (2, 2)-core, because the minimal degree on the top layer is 2 and the minimal degree on the lower layer is 2. An online algorithm for (α, β)-core computation is proposed by Ding D et al [53]. The naive approach starts from the input graph and iteratively vertices without sufficient neighbourhoods and incident edges until all remaining vertices in the subgraph satisfy the degree constraint. However, the online algorithm is inefficient when huge data graph (more than 1 billion edges). To avoid long processing time, Liu [52] proposes a new approach to compute (α, β)-core and build a $BiCore - Index$ with a linear size of the input graph. The time complexity of the index is $O(\delta \cdot m)$, where m is the number of edges and δ is the maximum value such that (δ, δ)-core in the data graph is nonempty and is bounded by $\sqrt{m}$. Experiments show that query processing with index support is several orders of magnitude better than the state-of-the-art online computation algorithms.

The naive of $BiCore - index$ is that first calculate all the combinations of α and β. Then store these α and β values as two-dimensional arrays. The values of the two-dimensional arrays are the nodes corresponding to the (α, β)-core nodes. But there are a lot of duplicate values and some cases of no data in the double array. The author then proposes an optimization strategy that restricts the loops of α and β by using δ. Moreover, the empty and duplicate values will be removed from the index. In addition, the user also proposed index maintenance on dynamic graphs.

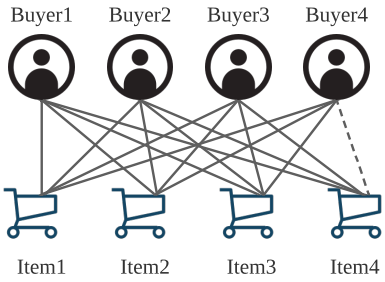


Figure 2.5: An example of (α, β)-core application

Compared with other core computation algorithms [53–55], $BiCore - index$ query time and efficiency have been significantly improved, and the index design is a great space saver. However, as the data graph continues to grow beyond the capacity of a single machine, the author did not consider a distributed framework, so this is a relatively weak point.

Applications of (α, β)-core (α, β)-core can be used to build online recommendation systems. In user-product networks, collaborative filtering techniques are often used to build recommendation systems. In this process, it is important to group similar users. Fault-tolerant panel recommendations are proposed to handle missing values in incomplete data and have proven their effectiveness in panel recommendations [56, 57]. An example will be given to help to understand as shown in Figure 2.5.

Example 3. *As shown in Figure 2.5, "Buyer 1-4" have all purchased "Item 1-3" recently, while only "Buyer 1-3" have bought "Item 4". It can be reasonably inferred that "Buyer 4" may also be interested in "Item 4" and suggested accordingly [51]. The graph can be presented as (3,4)-core.*

k-core decomposition. In this section, a k-core decomposition paper will be described, which is "k-core decomposition of large networks on a single PC" [58]. Currently, it is possible to compute and process graphs with scales up to hundreds of millions of edges through distributed systems. However, the shortcoming is that although distributed resources are now readily available, there are few distributed algorithms for graph computation. Developing distributed algorithms for graph computation can be difficult because, based on existing frameworks, the first thing to face is an effective graph partitioning method. So it is challenging to develop a distributed graph processing system to solve significant graph computation problems. In this paper, the authors propose to use

a personal computer (PC) with external memory technology to implement k-core decomposition.

The schemes mentioned above are inefficient or infeasibility led the author to implement the k-core algorithm based on GraphChi. GraphChi uses a parallel sliding window technique (BSW). This technique stores the graph in external memory (SSD or disk) and requires only a small number of non-sequential reads and writes to the secondary memory to compute graph updates up to a million times per second and supports an asynchronous computation model. The processing of graphs in BSW is divided into three steps: (1) transfer a subgraph of the entire diagram from the secondary storage into the memory, (2) update the data graph based on the user-defined update function and (3) write the updated subgraph into the disk.

In summary, in the bipartite graph and unbipartite graph, cohesive subgraph models have been extensively studied. Based on these cohesive subgraph models, finding the hierarchical structure of graphs is a popular research topic in the field of graph analysis. Moreover, cohesive subgraph mining is also a potential trend for graph databases.

Chapter 3

SQL2Cypher: Automated Data and Query Migration from RDBMS to GDBMS

3.1 Introduction

Graphs have played an increasingly important role in data management with the prevalence of graph data in different application domains such as social networks, road networks and protein-protein interaction networks. Graph database management systems (GDBMS) are among the most fundamental infrastructure when managing graph data and have received a lot of attention from researchers and programmers globally [8]. GDBMS have the unique advantages of modelling and querying complex relationships, capturing and navigating complex data relationships and recursive path querying when handling graph data.

However, relational database management systems (RDBMS) still comprise the majority share of the database market for legacy reasons, even when storing highly connected data [10]. Querying highly connected data in an RDBMS usually requires complex join operations and significant system overhead, which can lead to a long execution time [11]. Hence, there naturally emerges

the demand for migrating from RBDMS to GDBMS. In this thesis, we demonstrate *SQL2Cypher*, an automated tool for migrating data from RBDMS to GDBMS.

Migrating data from RDBMS to GDBMS involves redefining data schema, mapping relations and rewriting queries. The migration process is often time-consuming and labour-intensive. The high time and labour costs are one of the major reasons why companies choose to keep their legacy RDBMS. To address this problem, several automated tools [12–14] have been proposed to migrate data from RDBMS to GDBMS. However, we find that they are either outdated or incomplete. For example, [12] focuses on XQuery and [14] focuses on RDF data, while ignoring the nowadays more widely adopted property graph model [8]. The open-source tool Neo4j-ETL [13] allows the user to import data from relational databases to the popular graph database Neo4j. However, it does not provide automatic query translation, and users have to rewrite all previous SQL queries to Cypher queries manually [15] (the graph querying language used by Neo4j). More critically, Neo4j-ETL is not well maintained and has many issues at present[1] (e.g. error loading large dataset and error when mapping relations).

Motivated by the above reasons, we develop and demonstrate *SQL2Cypher*. *SQL2Cypher* focuses on graph databases adopting the property graph model, in which each vertex/edge in the graph can have an arbitrary number of key-value pairs to represent its properties. *SQL2Cypher* automatically allows users to migrate from relational databases to property graph databases. It automatically derives the relationships among migrated tables, maps the relational model to the property graph model, imports the data from RDBMS to GDBMS, and finally translates the corresponding SQL queries to Cypher queries. It supports Open Database Connectivity (ODBC) [59] compliant relational databases (e.g., MySQL, PostgreSQL and Microsoft SQL Server) and Cypher-based graph databases (e.g., Neo4j, SAP HANA Graph [60] and PatMat [51]). In addition, several optimization strategies are implemented based on predictive interaction framework [16] and duplication detection [17] to further improve the speed and the quality of the migration.

[1] https://github.com/neo4j-contrib/neo4j-etl/issues

3.2 System Overall

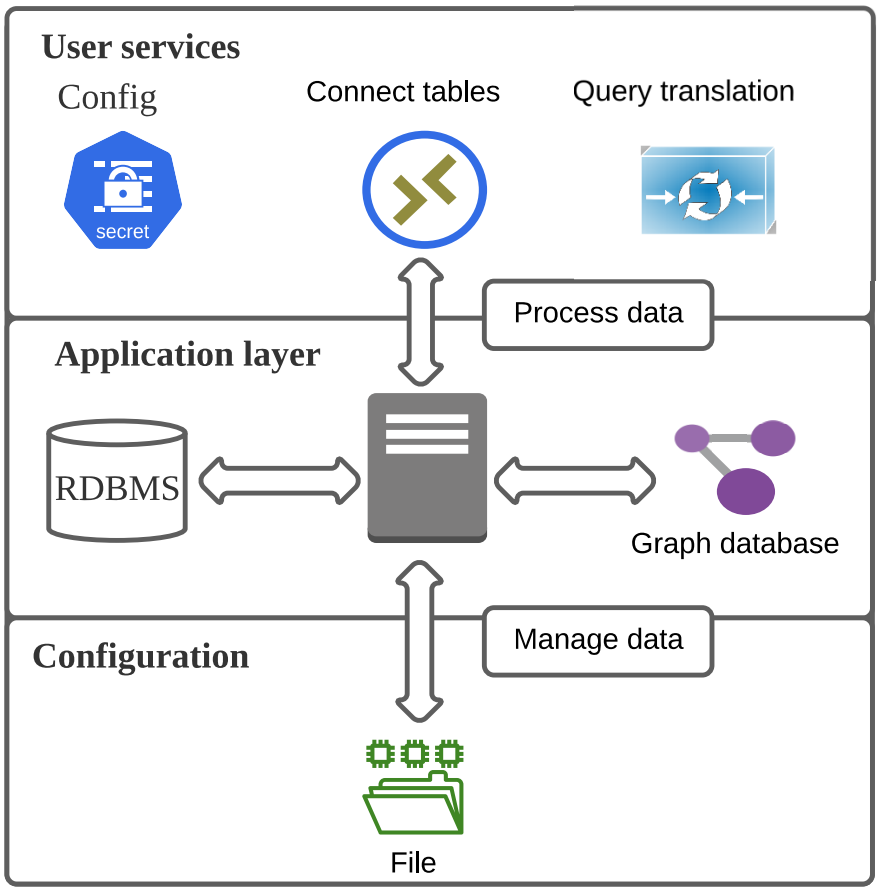


Figure 3.1: SQL2Cypher architecture

SQL2Cypher consists of three-layer as shown in Figure 3.1. The first is the user services layer, which is comprised of user interfaces (both graphical and command-line) where the user can operate the system. Second, the application layer will receive commands from the user services and complete background processing. Finally, in the configuration layer, we efficiently store and manage different system configurations on the disk. We present the details of these three layers as follows.

3.2.1 User services

Regarding the user services, as shown at the top of Figure 3.1, we build several user-friendly graphical interfaces. User services pass the information entered by the user to the application layer and receive data from the application layer that needs to be displayed. User services are comprised of three sections: configuration, table connections and query translations. In the configuration section, users can configure the essential information of the RDBMS and GDBMS (i.e., username, password and database URL). The table connection section presents the relationship between the tables to the user in a graph. Users can modify the relationship between tables if any are incorrect or missing. Lastly, users can use the system to translate SQL queries to Cypher queries after the application layer processes the relationships among tables. As shown in Figure 3.3, we use the *d3*[2] and *layui*[3] libraries to provide graph visualization and *CodeMirror*[4] to provide code highlighting. To improve usability, a command-line interface (CLI) is also provided with has the same workflow.

3.2.2 Application layer

The application layer connects the RDBMS and GDBMS. The server in this layer processes all background tasks. In order to make our system flexible and allow it to adapt to different database systems (e.g., MySQL, PostgreSQL and Microsoft SQL Server), the application layer connects RDBMS via Open Database Connectivity (ODBC) [59]. ODBC achieves database independence by using the ODBC driver as the translation layer between the application and RDBMS. Applications written using ODBC can be ported to other platforms on the client and server sides without changing the data access code. In the following section, we will explain in detail how the data migration is done in our system.

[2]https://d3js.org/

[3]https://www.layui.com/

[4]https://codemirror.net/

Algorithm 1: Generate edges

Input: $tuples = tablerelationshiptuples$

Output: E

$PE = \emptyset$
$GE = \emptyset$
$tuple = next(tuples)$
while $tuple \neq \emptyset$ **do**
    if $isJoinTable(src, dst)$ **then**
        $e \leftarrow$ Make a property edge with src, dst and $label$
        $PE.add(e)$
    else
        if $isConnect(src, dst)$ **then**
            $e \leftarrow$ Make a edge without property with src and dst
            $GE.add(e)$
    $tuple = next(tuples)$
$E = GE \cup PE$
return E

3.2.2.1 Parsing table relationship.

Table relationships can be seen as graph structures where tables are vertexes and edges are relationships. Our system can extract the relationships between tables based on the RDBMS schema; these relationships will be displayed as a graphical user interface or as a relationship path in the CLI. After data migration, the relationships between tables are stored in the configuration layer. For the naive approach, we store all the tables in the RDBMS as nodes in our GDBMS, then create edges for all the connected tables (connected by foreign keys). However, this approach can lead to high memory usage, and we will analyze memory usage and focus on optimization strategy in the following section (optimization analysis).

Algorithm 1 describes the transformation of the RDBMS to the GDBMS. The input value $tuples$ is a set of tuples consisting of source, destination and label. Edges and vertexes are accessed via several functions: $IsJoinTable$ and $IsConnect$. $IsJoinTable$ is to check whether there exists a *join table* (a table that contains two foreign keys) between connected tables. $IsConnect$ is to check whether the source table and destination table are connected. The processing of generating edges can encounter two cases.

Case 1. In this case, we implement an optimization strategy based on [16]. The strategy is to convert the *join table* to an edge and the attributes in the *join table* as properties of the edge. This strategy will save memory usage and make the data more adaptable to the nature of the graph database. In the algorithm 1 lines 4-6 can convert the *join table* to an edge with properties. We will explain this process with an example.

Optimization analysis. Our optimization strategy will reduce the amount of vertex storage and edge storage. We will show this result with a theoretical analysis. In the IMDB database, the *principal* is connected to the *name* and *title* tables respectively. Suppose *name* has n rows of data, *title* has m rows of data, and *principal* has e rows of data. For the naive approach, we need to connect *principal* with *name* and *principal* with *title*, so we need to store $n + m + e$ nodes and $2 * e$ edges. For our optimization strategies, we can connect *principal*, *name* and *title* together without saving the *principal* nodes. We only need to store $n + m$ nodes and e edges. Thus, we will

save $2 * e$ of storage compared to the naive approach. In the real world, the edges of the graph are usually very numerous , so our system can help users save more storage space after data migration.

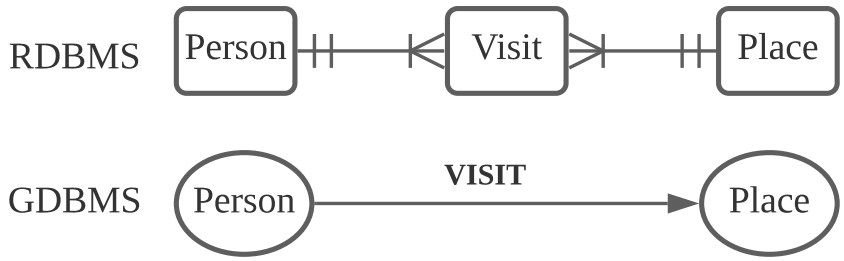


Figure 3.2: Example of extracting table as edge

Example 4. *In the top area of Figure 3.2, there are three tables named Person, Visit and Place. The Visit table connects the Person table and the Place table and the Visit table also has two attributes which are startTime and endTime (the start time and end time of visiting a place) besides two foreign keys. In our system, the Visit table will be seen as a join table, so it will be converted to an edge and startTime and endTime attributes will be converted as the properties of that edge. In Cypher query pseudocode, this is like: (Person)-[:VISIT{startTime: value, endTime: value}]→(Place)*

Case 2. In algorithm 1 lines 8-10, if there is no *join table*, the system will detect the connection of two tables (at line 8). If src and dst are connected, the system will build an edge without properties (line 9).

3.2.2.2 Translating query language.

We also provide conjunctive queries (e.g. selections, join, projection and insertion), which are expressed in SQL that translate to Cypher queries based on relationships path traversal operations. In our mechanism, we parse SQL into a list of tokens by building an abstract syntax tree (AST) and then traverse different operations to translate separately based on the relationships that are stored in the configuration layer. For example, as shown in Figure 3.4, the box on the left-hand

side shows the SQL query, when we click the convert button, the SQL will be translated to the Cypher query (the box on the right-hand side).

3.2.3 Configuration

In this layer, all the configuration, database information and the relationship between the tables is stored in a *pickle*[5] file. The relationships are then used to translate the query language. The system keeps the relationships and configuration updated based on the user input.

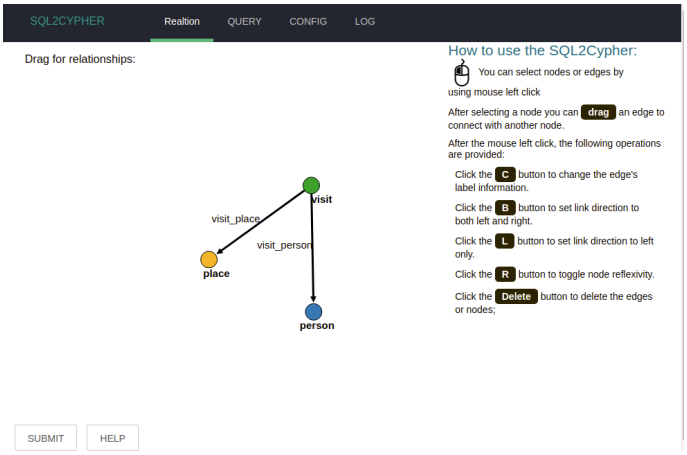


Figure 3.3: Modify relationships among tables

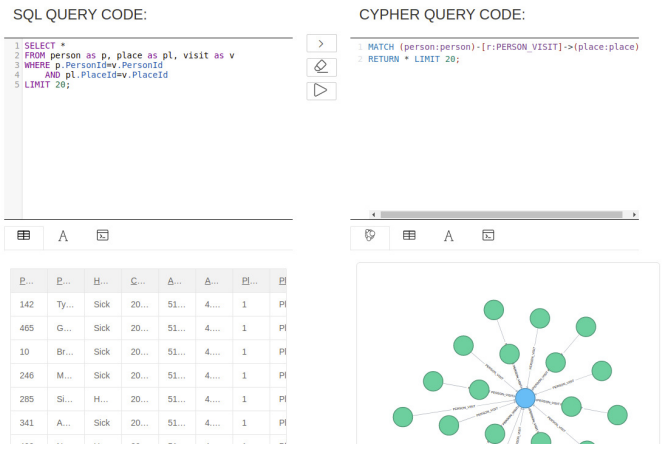


Figure 3.4: Query language translation

[5]https://docs.python.org/3/library/pickle.html

3.3 Demonstration

We will use two real-life example scenarios to demonstrate the overall experience of *SQL2Cypher*. The first scenario is COVID-19 spread which contains information about the COVID-19 test day of a person, the places visited by an infected person and the corresponding time. The purpose of the data migration is to effectively find people who have come into contact with infected people to avoid COVID-19 spreading. The second scenario is IMDB, which contains basic movie information, title, crew, rating, person name, etc. The second scenario aims to demonstrate the optimization strategy of the system.

SQL2Cypher processes the schema in the RDBMS to form a graph structure, and then the graph will be presented in the user interface as shown in Figure 3.3 and Figure 3.4. Figure 3.4 presents queries translation and the execution result. In our demo, we will conduct the following sections to explain how to use our system to migrate data and translate queries.

3.3.1 Data migration

Figure 3.3 displays the original relationships between person, visit and place tables from the first scenario. Users can modify (delete, add, and change) the connections between tables. After submission, our system will process the *join table* automatically.

The IMDB dataset contains movie names, aliases, basic information, episodes, crew, actors, directors, movie ratings and personnel information, and relationships between the data. The IMDB dataset contains the relationship between tables that can be converted to edge attributes. Therefore, based on the IMDB dataset, we demonstrate relationship modification during the data migration processing, as some tables need to be connected by reference key form. For example, the IMDB dataset contain a *principal* table, and the principal table connects with *title* and *name* tables by using *reference key*. Unfortunately, RDBMS can not detect *reference key*, therefore, users need to add edges among *principal*, *title* and *name* tables. To ensure that the migrated data is accurate, we do several queries on MySQL and Neo4j to compare the results. The queries contain the number of nodes/tables, values in nodes/tables and the value of relationships between tables.

Table 3.1: Data comparison

Table	Name	Title	Episode	Rating	Aka	Crew	Principal
RMDBS	11,684,622	8,974,009	6,740,446	1,251,067	32,265,777	8,974,009	50,646,116
GDBMS	11,684,622	8,974,009	6,740,446	1,251,067	32,265,777	8,974,009	50,646,116

3.3.2 Query translation

This section will use the first scenario to demonstrate our system's use to find potentially infected persons. A person is considered potentially infected if the person stays in the same place as an infected person at the same time. We can use the following SQL join operations to accomplish this task.

SELECT $*$ **FROM** $person\ AS\ p,\ place\ AS\ pl,\ visit\ AS\ v$
WHERE $p.PersonID\ =\ v.PersonID$
AND $pl.PlaceID\ =\ v.PlaceID$ **AND** $p.Healthstatus\ =\ "Sick"$;

After translating, *SQL2Cypher* will generate a graph pattern to find the potentially infected people using the following code:

MATCH $(p : person\{Healthstatus : "Sick"\}) - [r : VISIT] - > (pl : place)$
RETURN $*$;

In addition, users are able to execute both SQL and Cypher queries in the user interface, and we provide several different forms of displaying the result. As shown in Figure 3.4, we demonstrate SQL queries result with tabular data and demonstrate graph structure for Cypher queries result. Graph structure makes finding relationships between data easier than tabular data.

3.4 System evaluation

Migrating speed is not an essential evaluation aspect of our system. The migrated result accuracy is essential to evaluate. In this section, we evaluate our system based on the IMDB dataset, and the experiment is run on a Linux server with an Intel I5-8500T processor, 16GB main memory and 500GB hard disk. For RDBMS and GDBMS, we use MySQL 8.0 and Neo4j 3.5.14, respectively.

As shown in Table 3.1, IMDB contains seven tables in RDBMS. After the database migration, we can observe that the RDBMS and GDBMS data are the same. Note that the *Principal* table will be considered as the edge with properties between *Title* and *Name* tables. Therefore, we compare the number of edges between *Title* and *Name* tables in GDBMS with table rows in RDBMS.

3.5 Conclusion

In this work, we proposed a system that automatically converts relational databases into graph databases and translates SQL queries into Cypher queries. The system saves time and labour costs. The system is also optimized for repetitive data and graph database attributes.

Chapter 4

FSPS: Accelerating Subgraph and Path Queries Using FPGA

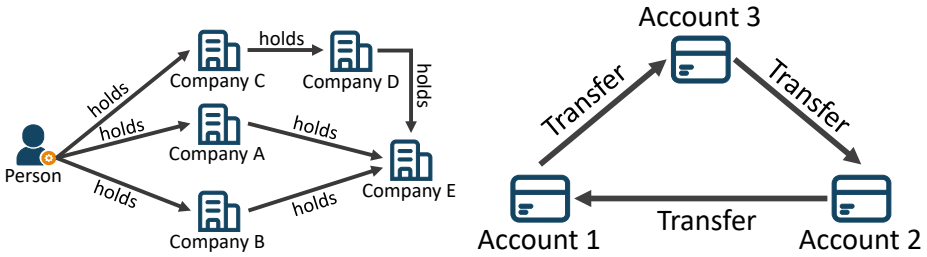
4.1 Introduction

Graph has been playing an increasingly important role in data management with the prevalence of graph data in different application domains in recent years. There are two fundamental types of graph queries in graph analysis [61], namely subgraph queries and path queries. Given a pattern graph q and a data graph G, subgraph queries aim to find all subgraph instances in G that are isomorphic to q. As for path queries, they navigate the graph to investigate the relations between a source vertex s and a target vertex t within k hops to find all paths. Subgraph and path queries are associated with a wide spectrum of applications in the areas of network & IT operations, finance, e-commerce, cyber security, bioinformatics, chemistry, social science, etc. Below we present two real-life scenarios.

Scenario I. In enterprise relationship analysis, it is common to investigate how one person could control a company. Investors can use a path query algorithm to determine if a competent person owns the company to decide whether to invest. In the example of Figure 4.1(a), given a person as the source vertex and a company 'E' as the target vertex, we want to enumerate all paths from

the source person to the target company within k hops ($k = 3$ in this example). As a result, one can notice that although the person does not hold company 'E' directly, he holds it via companies 'A', 'B', 'C', and 'D'. Finally, the investor can decide whether to invest after finding out that the person controls the company.

Scenario II. In practice, the entire fraud process may involve complex chains of transactions through many entities, which require complex cycle detection with various constraints [62]. A simple financial network can be represented as a graph where each account is a vertex, and the transactions in between are edges. Figure 4.1(b) demonstrates a subgraph pattern for possible financial fraud [20]. In this pattern, the three accounts' transactions form a triangle that may indicate money laundering.



(a) Enterprise relationship analysis (b) Fraud Detection

Figure 4.1: The real-life graph analysis scenarios.

Considerable efforts are made in both industry and academia to develop efficient systems for subgraph and path queries [18–22]. However, almost all solutions are developed on CPUs which have the following limitations when handling graph data: 1) CPUs do not offer flexible high-degree parallelism, and 2) CPU caches do not work effectively for irregular graph processing with limited data locality.

With the recent advance of field-programmable gate arrays (FPGAs), people are provided with a new alternative to accelerate graph computations at the hardware level. FPGAs have shown significant advantages over multi-core CPUs in parallelism due to their pipelining design and highly efficient hardware circuit. For instance, one FPGA card can efficiently parallelize a loop with 1,000 iterations, while we have to find a host equipped with 1,000 CPU cores to offer the same parallelism. Furthermore, compared with GPUs, FPGAs are more energy-efficient and can

handle irregular graph processing with more stable parallelism by fully exploiting their pipeline mechanism [23]. FPGAs are now widely deployed by enterprises and cloud service providers such as Microsoft, Alibaba, Tencent, Huawei, and Amazon Web Services (AWS).

[24] and [25] present the latest idea of subgraph and path query algorithms based on FPGA. However, these algorithms do not store graph data (i.e., temporally load in the memory), parse the query and property filtering, etc. Furthermore, these algorithms do not provide an interface for the user to query, which is not easy for the user to use. To improve the efficiency of the graph database, we explored the acceleration of graph database queries based on FPGA. Therefore, motivated by the above reasons and based on the latest research efforts [24,25], we develop and demonstrate the prototype of the first FPGA-based Subgraph and Path querying System, called FSPS. Specifically, FSPS has the following features:

- *A CPU-FPGA co-designed architecture.* FSPS employs a CPU-FPGA co-designed framework. The CPU is in charge of parsing, prepossessing, and scheduling, whereas the FPGA is responsible for the de facto computation for subgraph and path queries.
- *Fully pipelined execution on FPGA.* We design and implement the system on the FPGA side in a fully pipelined manner. This allows FSPS to achieve massive parallelism and maximised efficiency.
- *Reduced data transfer from FPGA's external memory.* FPGAs have small sizes of on-chip memory (BRAM) that is usually only tens of megabytes. As fetching data from FPGA's external memory (DRAM) is very expensive, FSPS applies partition and caching techniques to reduce the data transfer between DRAM and BRAM to improve the efficiency further.

4.2 System overall

Figure 4.2 shows the system architecture of FSPS. On the front end, FSPS provides the user interface for entering different types of queries, displaying query results and loading graph data from the local disk. Users are able to draw the subgraph and path queries when entering queries,

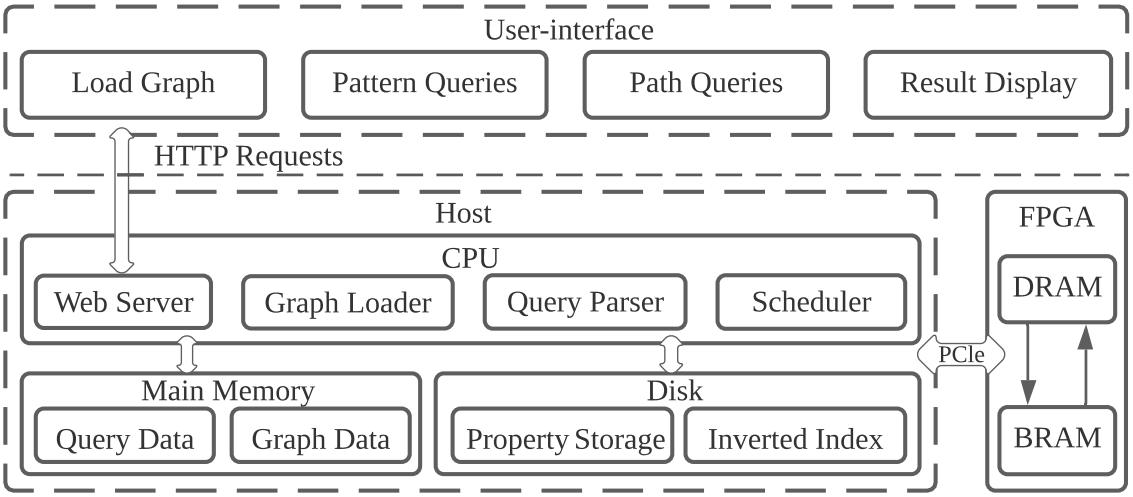


Figure 4.2: System Architecture

and FSPS will return and display the results either in a graph view or a table view. FSPS uses *d3*[1] and *layui*[2] libraries to build the front end. It communicates with the back-end web server on HTTP requests.

In the back end, FSPS employs the CPU-FPGA co-designed framework. On the host side, FSPS features a web server, a graph loader, a query parser, and a scheduler. The FPGA card is PCIe attached to the host, and the scheduler will schedule the task on the host and FPGA and coordinate data transfer between them.

FSPS supports property graphs [19] in which each vertex and edge can have arbitrary properties (i.e. any number of key-value pairs). We store the query and the topological structure of the data graph (together with the label information) in-memory (in compressed sparse row format) for fast access and the properties in the data graph on disk using a local key-value store in RocksDB[3]. FSPS can also build an additional inverted index of user-defined properties in the key-value store. As an interesting further work, we will investigate more complex index structures such as a B-tree.

Upon receiving a query, FSPS will first parse the query (by the query parser), send it to the

[1]https://d3js.org/

[2]https://www.layui.com/

[3]https://rocksdb.org/

scheduler, and finally execute it on FPGA. The CPU can access the graph data (either from the main memory or the disk) and send the data to FPGA via PCIe bus when needed. The backend of FSPS is implemented in C++, except the webserver, which is implemented in Python with the help of the *flask*[4] library.

In the following, we will discuss in detail how subgraph and path queries are executed on FPGA, respectively.

4.2.1 Subgraph Queries

We adopt the subgraph matching algorithm in [24] for fast and efficient subgraph queries on FPGA.

Host Implementation. When the user submits a subgraph query $Q = (q, G)$, the host first builds an auxiliary data structure upon q and G called *candidate search tree* (CST), which serves as the complete search space of the query. Limited by on-chip resources on FPGAs, CST is often too large to be fully loaded into FPGA's BRAM, and it dramatically decreases overall performance to access CST from DRAM rather than BRAM. Hence, the CST is recursively partitioned and offloaded to FPGA one by one. The partition strategy in [24] guarantees there is no overlap in the search space between the partitioned CST, and each partitioned CST can be completely loaded into BRAM. After finishing the CST partition, the host can share a small portion of matching tasks to improve the throughput as a whole further.

FPGA Implementation. Once a partitioned CST has been loaded into BRAM. We start the computation on FPGA. In the typical backtracking algorithms, one partial result is expanded at a time by matching the next vertex to a candidate vertex following the matching order. This sequential design cannot be pipelined because of data dependencies among iterations. To solve this, we decompose the matching process into three steps as follows: 1) *Generator* expands partial results by matching the next vertex in the matching order; 2) *Validator* verifies whether a new partial result is valid; 3) *Synchronizer* collects results. Different from the typical algorithms, our

[4]https://flask.palletsprojects.com/

method processes thousands of partial results at a time in these steps so that each step can fully utilize the pipeline mechanism of FPGA. Each round, we process those partial results that map most query vertices and pre-allocate enough space, which prevents the overflow of partial results buffer in BRAM. Two extra optimisations, namely *task parallelism* and *generator separation*, are employed to further improve the efficiency. The interested reader can refer to the original paper of [24] for more details.

4.2.2 Path Queries

We adopt the FPGA-based k-hop constrained s-t simple path enumeration algorithm in [25] for path queries. Given a source vertex s, a target vertex t, a hop constraint k, and an optional edge label set C, the algorithm can enumerate all the paths between s and t whose length is no more than k and the edge label is in C. Note that when $\mathcal{C} = \emptyset$, we do not consider the edge label constraint.

Host Implementation. When user submits the query Q, the host parses Q to extract and store s, t, k, $\mathcal{C}$, and G in main memory. To reduce the data graph G's size and search space, it first starts a preprocessing on the host using two $(k-1)$-hop breadth-fist search (BFS) starting form s and t, respectively. This can generate two tables, denoted as sd_s and sd_t, of the shortest distances of each visited vertex to s and t. For each visited vertex, we add it to a new graph G' with its neighbours if and only if the sum of its shortest path to s and t is no more than k. It has been proved in [25] that processing on G' is enough to get all results. When the host preprocessing is finished, the scheduler will transfer s, t, k, G' and sd_t to FPGA (to the DRAM) through the PCIe bus.

FPGA Implementation. Once the data is transferred to FPGA. We can start the computation on FPGA. In general, we adopt the BFS-based paradigm to utilize the pipeline mechanism of FPGA fully. The whole process on FPGA can be concluded as an *expansion-and-verification* framework, which can be dissected into three steps: 1) Expand the intermediate paths with one-hop successor vertices; 2) Verify if each expanded path is a valid path; 3) Write back the valid paths to the intermediate path set. The algorithm terminates when the intermediate path set is

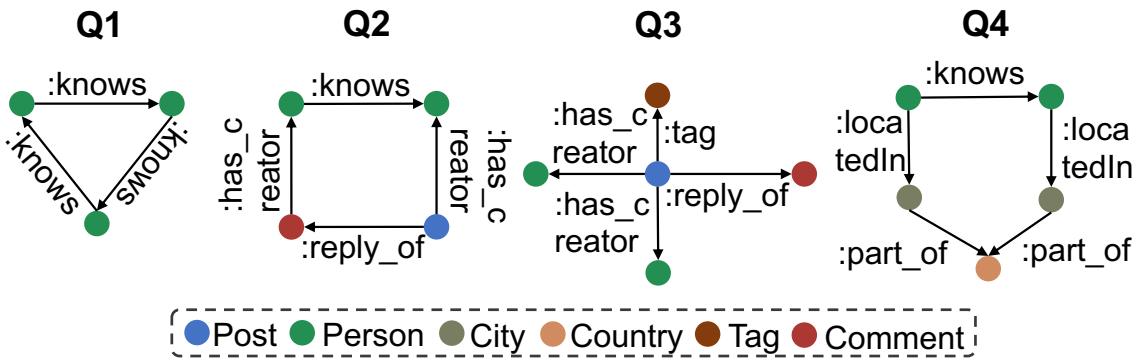


Figure 4.3: The LDBC queries

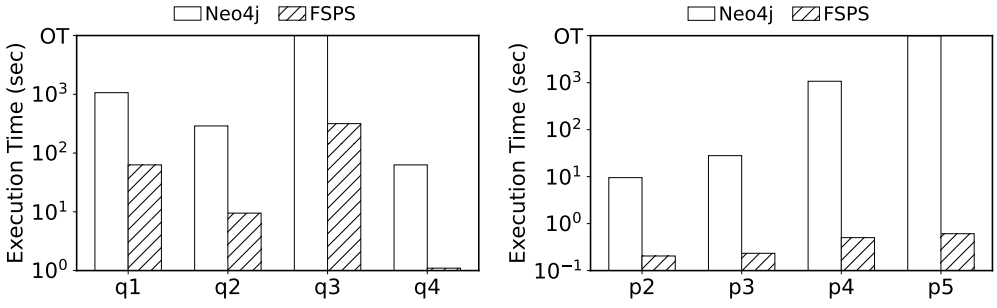
empty. To reduce the data transfer between DRAM and BRAM, we cache necessary data as much as possible on BRAM to improve the system latency. Furthermore, as discovered in [25], the longer path tends to have stronger pruning power in length check, which indicates it will produce fewer valid intermediate paths to save memory. Hence, by regarding the given intermediate path set S as a stack, we always fetch/write a batch of paths from/to its top, which ensures that the longest paths are always processed first. Finally *data separation* is applied to further improve the parallelism level. Please refer to [25] for the details.

4.3 Performance Evaluation

Following existing works [18, 24, 63, 64], we adopt the LDBC social network benchmark (SNB) [65] to evaluate the performance of FSPS. SNB provides a data generator that generates a synthetic social network together with a set of queries. We use two datasets DG10 and DG60 in our experiments. DG10 has 29.99 million vertices and 176.48 million edges. DG60 has 187.11 million vertices and 1.25 billion edges. These datasets are generated by simulating a real social network akin to Facebook with a duration of 3 years.

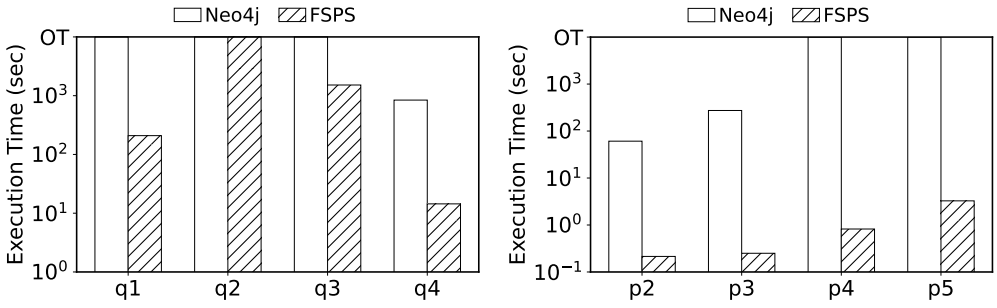
We compare FSPS with the most popular graph database system Neo4j [8]. We deploy two servers as follows: 1) a server equipped with an 8-core Intel Xeon E5-2620 v4 CPU (2.1GHz) with 250G memory, 2TB hard disk, and an Alveo U200 Data Center Accelerator Card with 64GB off-chip DRAM, 35MB on-chip BRAM, communicating with the host through PCIe gen3 $\times$ 16; 2) a server equipped with two 20-core Intel Xeon CPU E5-2698 v4 (2.20GHz) with 400GB memory, and 2TB

hard disk. Since part of the FSPS algorithm needs to be run on FPGAs card for acceleration, Neo4j is run on a better server (Server 2) for the fairness of the experiment. Therefore, we run FSPS on Server 1 (300MHz on the FPGA card), and Neo4j on Server 2. We allow 1 hour time limit. Overtime and out of memory queries are marked as OT and OOM, respectively. For each test, OT and OOM indicate a test case running out of the time limit and out of memory, respectively.



(a) Subgraph Query (b) Path Query

Figure 4.4: DG10

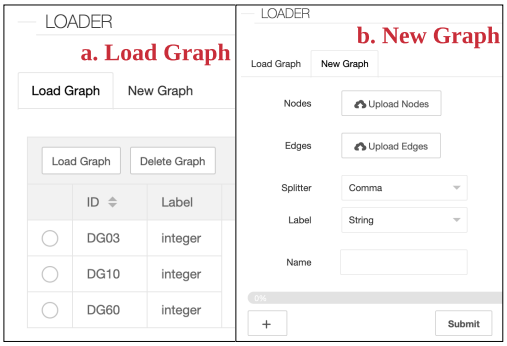


(a) Subgraph Query (b) Path Query

Figure 4.5: DG60

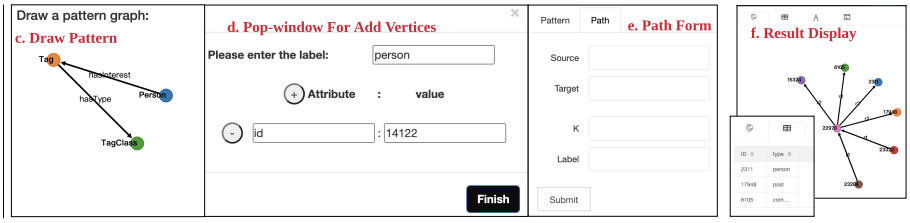
For subgraph queries, we use four representative subgraph patterns selected from the complex workload of LDBC-SNB as shown in Figure 4.3. For path queries, we vary the length constraint k from 2 to 5 (denoted as P_2 to P_5, respectively), and set the edge label constraint to $\emptyset$. For each k, we randomly generate 10 query pairs $\{s, t\}$, and report the average time.

The results for the data graphs DG10 and DG60 are demonstrated in Figures 4.4 and 4.5, respectively. FSPS significantly outperforms Neo4j on both subgraph and path queries in the DG10 dataset, achieving a 100% completion rate, whereas Neo4j only completes 56% of queries. In addition, in the DG60 dataset, FSPS achieves a 75% completion rate where Neo4j only completes



Step 1: Load Graph

Figure 4.6: Basic processing pipeline (part one)



Step 2: Add Query **Step3: Result Display**

Figure 4.7: Basic processing pipeline (part two)

25% of queries. For DG10, FSPS achieves an average speedup of 13 times on subgraph queries (up to 105 times on query Q_1) and an average speedup of 1180 times on path queries (up to 2133 times on query P_3) compared with Neo4j. For DG60, Neo4j is only able to complete Q_4 among the four subgraph queries. FSPS achieves a 58 times speedup on subgraph query Q_4, and an average speedup of 689 times on path queries (up to 1095 times on query P_2).

4.4 Demonstration Overview

The demonstration mainly presents 1) the basic processing pipeline of FSPS; 2) LDBC queries; and 3) real-life applications.

4.4.1 Processing Pipeline

In this section, we guide the user to experience the whole processing pipeline of FSPS. The basic pipeline is shown in Figure 4.6 and 4.7, which includes the following three steps.

1. *Load/Import graph.* The first step is to load or import the data graph. Users are able to import graphs to FSPS for CSV files and create optional indices on selected properties. Once graphs have been imported, they can be selected as the data graph to run queries as shown in Figure 4.6 steps 1.a and 1.b.

2. *Draw patterns/paths.* The next step is to enter subgraph or path queries. One can draw the pattern through our user interface. Query information such as the vertex/edge label in subgraph query and hop constraint in path query can be entered in pop-up windows as demonstrated in Figure 4.7 step 2.c and step 2.d. For path queries, the user can enter the source, target and constraint k in the form as shown in Figure 4.7 step 2.e.

3. *Display results.* FSPS will display the results at the final step. FSPS supports results display in graph view (which renders the results into a new graph) and table view as shown in Figure 4.7 step 3. Users can also check the meta-data returned or download the results for future use.

4.4.2 LDBC Queries

In this scenario, we will pre-load an LDBC dataset on the server and allow the attendee to specify one of the benchmark queries. The query will be executed using both FSPS and Neo4j. The performance metrics will be delivered back to the scene and demonstrated to the attendee to show the performance advantages of FSPS.

4.4.3 Real-life Applications

In this scenario, we use two real-life graphs, *weibo* and *DBLP*, to demonstrate how FSPS can be applied in real-life applications.

Friend recommendation in the social network. Friend recommendation in social networks. In the first application, we demonstrate how subgraph queries can be applied to provide friend recommendations in social networks. The data in real life is extensive, such as Facebook generating 4 petabytes daily. CPU-based subgraph matching algorithms can lead to long execution times when processing large amounts of graph data. Therefore, we can use FPGA to accelerate the subgraph matching, which can be a shorter latency to give users a better experience. We use the Weibo dataset obtained from [66]. The dataset is crawled from China's biggest social media platform, Weibo, similar to Twitter. It models a social network where each vertex represents a user, and the edge between two users represents the following relationship. To find possible friends that the user may know but has not followed yet, we use a pattern graph that is a 4-clique with one missing edge from the given user to the potential friend. Intuitively, if three persons 'A', 'B' and 'C' know each other, and both 'B' and 'C' knows 'D', it is very likely that 'A' also knows 'D'.

Finding connections in co-author network. In this application, we show how path queries can help researchers find paths to reach out other researchers. To do so, we extract authors who published papers from the last 5 years in top-tier database and data mining conferences, including SIGMOD, SIGKDD, VLDB, ICDE, CIKM and ICDM. The vertices represent the authors, and the edges represent co-authorship. The attendees can specify the length constraint k (e.g. $k = 3$), and the name of the author he/she wants to collaborate with. By given so, FSPS can enumerate all paths within k hops between the two authors. For example, if one wants to collaborate with Jiawei Han, he/she can network through the paths from him/her to Jiawei Han to reach out to him. This can also be used when assigning reviewers to a paper. If the author of a paper has many paths to a reviewer, this potentially means they have close cooperation. Thus, we may not want to assign this paper to the reviewer.

4.5 Conclusion

We developed the first FPGA-based prototype system for subgraph and path queries in this work. FSPS employs a CPU-FPGA co-designed framework, fully pipelined execution on FPGA, and reduced data transfer from FPGA's external memory.

Chapter 5

Discovering Hierarchy of Bipartite Graphs with Cohesive Subgraphs

My main contribution to this work [66] is the design of the bipartite hierarchy construction algorithms (e.g., HC-TD and HC-BU). The code implementation of HC-TD and HC-BU algorithms and the efficiency experiments are also conducted by me. In addition, I have obtained the permission from Dr. Kai Wang, Prof. Wenjie Zhang, Prof. Xuemin Lin and Prof. Ying Zhang to use this work in my thesis. My contribution in this work includes Section 5.4, 5.5 and 5.6.

5.1 Introduction

Bipartite graphs are naturally used to model relationships between two different types of entities such as customer-product [67], user-item [68], and author-paper [69]. Driven by numerous applications including fraud detection [70, 71], bioinformatics analysis [72, 73], and network visualization [32, 74], cohesive subgraph models (e.g., (α, β)-core [53, 54, 70, 75], k-bitruss [35, 37, 38], and biclique [72]) have been widely studied in bipartite graphs. Most graph databases do not have specialized operators for bipartite graph models such as PatMat [51], Neo4j [19], etc. In the previous work, we implemented the FSPS graph database system. In the future work, we can investigate

the application and feasibility of adding the bipartite graph model to the graph database. In this work, we focus on discovering the hierarchy of bipartite graphs based on cohesive subgraphs.

Applications. The following applications can be directly benefited by discovering the hierarchy of bipartite graphs.

• *Bipartite network visualization.* One of the most natural applications of our work is bipartite network visualization. By visualizing the hierarchy of bipartite networks/graphs, we can reveal the relationships of vertices (and the dense regions) in bipartite networks. We are also able to generate insights for network analysis from multiple perspectives. For instance, in user-movie networks, we can analyze the behaviours of users (or the attractiveness of movies) at different levels of granularity. In team-project networks, we can obtain the hierarchy of teams and the participation level of projects for management purposes.

• *Supporting efficient community search.* In bipartite graphs, given a query vertex q, the set of vertices in the connected component of the (α, β)-core containing q are considered in the community of q [53, 70, 76]. These vertices are closely related to q, and searching such communities can support many real-world applications (e.g., recommendation [53]). Note that the existing study [70] propose the bicore index to support retrieval of the vertex set of an arbitrary (α, β)-core in optimal time. However, a further BFS search is needed to obtain the community containing a query vertex. Since the bipartite hierarchy organizes the vertices in (α, β)-cores hierarchically by considering the connectivity, it can support solving the problem more efficiently with the same space cost as the bicore index (i.e., $O(m)$, where m is the number of edges in a given bipartite graph G). As evaluated in our experiments, the bipartite-hierarchy-based query algorithm significantly outperforms the bicore-index-based query algorithm by up to three orders of magnitude.

A new method for finding the hierarchy of bipartite graphs is worth exploring. In bipartite graphs, (α, β)-core is a well-studied cohesive subgraph model that ensures the degree of each upper (lower) vertex is at least α (β) in the subgraph [53, 54, 70, 75]. In this work, we are the first to discover the hierarchy of bipartite graphs based on (α, β)-cores and their connectivity. Specifically, given a bipartite graph G, we aim to build the *bipartite hierarchy* that consists of an upper hierarchy for the upper vertices and a lower hierarchy for the lower vertices according to the (α,

β)-cores containing the vertices. The intuition of using the (α, β)-core model is that it is not only vertex-centric (that considers two vertex layers separately) but also has two parameters α and β (that provides fine-grained analysis of bipartite graphs in two-dimensions).

Challenges and our contributions. We summarize the principal contributions and technical challenges as follows.

• *A systematical study of the hierarchy discovering problem in bipartite graphs.* By analyzing the drawbacks of existing studies and the special characteristic of bipartite graphs, we propose the *bipartite hierarchy* model, which is the first to reveal the hierarchy of bipartite graphs based on (α, β)-cores and graph connectivity. The bipartite hierarchy has a two-dimensional structure to analyze bipartite graphs with different granularity levels. Notably, it only has a linear space usage and can clearly depict the hierarchical tree structure of bipartite graphs. By utilizing the nested property of (α, β)-core and exploring possible cost-sharing, we also propose efficient algorithms to construct the bipartite hierarchy.

• *Efficient algorithms for maintaining the bipartite hierarchy on dynamic bipartite graphs.* In real-world scenarios, graphs are usually dynamically changing. For instance, new purchase records are growing on online shopping platforms, and we need dynamic bipartite graphs to model such user-item networks. Consequently, it is essential to maintain the bipartite hierarchy on dynamic bipartite graphs rather than recomputing it from scratch. Although existing works proposed algorithms on maintaining the index for extracting the (α, β)-core [76], the connectivity and the hierarchy of (α, β)-cores are not considered. In this work, we propose algorithms to maintain the bipartite hierarchy incrementally regarding the edge insertion/deletion cases. The proposed algorithms can effectively identify the affected regions to limit the computation scope and achieve high efficiency.

• *Comprehensive experimental evaluations on 10 real-world graphs.* We conduct comprehensive empirical studies on 10 real-world bipartite graphs. The effectiveness of our bipartite hierarchy model is demonstrated by case studies. We also evaluate the performance of the hierarchy construction algorithms via different experimental settings. Experimental results validate both the effectiveness and efficiency of our proposed techniques.

5.2 Related Work

This section reviews two closely related areas: hierarchical decomposition of unipartite graphs and cohesive subgraph models on bipartite graphs.

Hierarchical decomposition of unipartite graphs. On unipartite graphs, graph hierarchy decomposition is conducted based on cohesive subgraph models such as k-core [26, 77–81] and k-truss [82–84]. Based on k-core, [27, 58, 80, 85–88] study the core decomposition problem. In [27, 58], the authors propose an in-memory algorithm for core decomposition with linear time complexity. The problem is also studied in distributed environments [80, 87], graph streams [85], multi-thread frameworks [86], and semi-external settings [88]. In addition, Liu et al. [79] propose efficient algorithm for computing CoreCube decomposition in multi-layer graphs. [89] study the attributed community search problem using k-core in attributed graphs. Considering both cohesiveness and connectivity, the forest k-cores (i.e., the k-core hierarchy) is proposed in [26]. In addition, Lin et al. [81] study maintenance algorithms for the k-core hierarchy on dynamic graphs. [28, 29, 38, 83, 90, 91] study the truss decomposition problem.

Cohesive subgraph models on bipartite graphs. On bipartite graphs, several existing works [38, 53, 54, 75, 76] study the (α, β)-core model and propose efficient algorithms to find (α, β)-cores. Specifically, Ding et al. [53] propose an online algorithm to find the (α, β)-core by given α and β. In [76], the authors present index-based algorithms to find the vertex set of an (α, β)-core in optimal time. Wang et al. [38] study the problem of (α, β)-community search on weighted bipartite graphs. However, the connectivity and the hierarchy of (α, β)-cores are not considered in the works mentioned above.

By extending k-truss to bipartite graphs, [35, 38] study the k-bitruss model, which is the maximal subgraph where each edge is contained in at least k butteries. [35] present peeling algorithms to find the dense subgraphs based on k-bitruss/k-tip (i.e., the bitruss/tip decomposition). Here, k-tip is the maximal subgraph where each vertex is contained in at least k butteries. The parallel tip decomposition algorithm is also recently studied in [92]. Wang et al. [38] propose a novel online index and a new bitruss decomposition algorithm. [72, 93–96] study the biclique enumeration problem. [93] study the problem of enumeration of maximal bicliques from a large graph by using

MapReduce. [94, 95] study the problem of maintenance for maximal bicliques in bipartite graph streams. Zhang et al. [72] propose an algorithm that generates all maximal bicliques for diverse biological bipartite graphs. In [96], the authors explore an algorithm for implementing pivot-based enumeration pruning. Since the above works study different cohesive subgraph models, the algorithms in these works cannot be used to solve our problem here.

5.3 Problem Definition

In this section, we formally introduce the notations and concepts for defining the bipartite hierarchy. Mathematical notations used throughout this paper are summarized in Table 5.1.

Table 5.1: The summary of notations

Notation	Definition
G	a bipartite graph
$V(G)/E(G)$	the vertex/edge set of G
$U(G), L(G)$	the upper layer and lower layer of G
$\mathsf{size}(G)$	the size of $G = \|E\|$
u, v, w, x, q	a vertex in a bipartite graph
$(u, v), e$	an edge in a bipartite graph
$R_{\alpha,\beta}$	the (α, β)-core of G
$C_{\alpha,\beta}$	an (α, β)-component
$N(u, G)$	the set of neighbors of u in G
n, m	the number of vertices and edges in G ($m > n$)
$\mathcal{I}$	the bipartite hierarchy of G
$\mathcal{I}^U/\mathcal{I}^L$	the upper/lower hierarchy of $\mathcal{I}$
$\mathcal{I}^U_\alpha/\mathcal{I}^L_\beta$	an upper/lower tree
p	a tree node
$s_a(u, \alpha)$	α-offset of a vertex u
$s_b(v, \beta)$	β-offset of a vertex v

We consider an unweighted and undirected bipartite graph $G(V{=}(U, L), E)$. Here $U(G)$ denotes the set of upper layer vertices, $L(G)$ denotes the set of lower layer vertices, $U(G) \cap L(G) = \emptyset$. In addition, $V(G) = U(G) \cup L(G)$ denotes the total vertex set, and $E(G) \subseteq U(G) \times L(G)$ denotes the total edge set. An edge e between two vertices $u, v \in G$ is denoted as (u, v) or (v, u). The set of neighbors of a vertex $u \in G$ is denoted as $N(u, G) = \{v \in V(G) \mid (u, v) \in E(G)\}$, and the degree of u is denoted as $deg(u, G) = |N(u, G)|$. Note that G is omitted when the context is clear. We use n and m to denote the number of vertices and edges in G, respectively, and we

assume each vertex has at least one incident edge.

Before formally defining the bipartite hierarchy, we introduce the following critical concepts.

Definition 8 ((α, β)-core)**.** *Given a bipartite graph G and degree constraints α and β, a subgraph $R_{\alpha,\beta}$ is the (α, β)-core of G if (1) $deg(u, R_{\alpha,\beta}) \geq \alpha$ for each $u \in U(R_{\alpha,\beta})$ and $deg(v, R_{\alpha,\beta}) \geq \beta$ for each $v \in L(R_{\alpha,\beta})$; (2) $R_{\alpha,\beta}$ is maximal, i.e., any supergraph $G' \supset R_{\alpha,\beta}$ is not an (α, β)-core . We use α_{max} (or β_{max}) to denote the maximal α (or β) value where the $(\alpha, 1)$-core (or $(1, \beta)$-core) is not empty in G, respectively. Given an α (or β) value, we also use $\beta_{max}(\alpha)$ (or $\alpha_{max}(\beta)$) to denote the maximal integer where the $(\alpha, \beta_{max}(\alpha))$-core (or $(\alpha_{max}(\beta), \beta)$-core) is not empty, respectively.*

In the definition of (α, β)-core , it does not require graph connectivity, which is an important characteristic when adopting the core-based models in real-world scenarios. By further applying the connectivity constraint, we introduce the definition of (α, β)-component as follows.

Definition 9 ((α, β)-component)**.** *Given a bipartite graph G and the (α, β)-core $R_{\alpha,\beta}$, a subgraph $C_{\alpha,\beta}$ is a (α, β)-component if (1) $C_{\alpha,\beta} \subseteq R_{\alpha,\beta}$ and $C_{\alpha,\beta}$ is connected; (2) $C_{\alpha,\beta}$ is maximal, i.e., any supergraph $G' \supset C_{\alpha,\beta}$ is not a (α, β)-component. For a vertex u, we use $C_{\alpha,\beta}(u)$ to denote the (α, β)-component containing u.*

By the above definitions, when fixing α (β), if an upper vertex u (or a lower vertex v) belongs to an (α, β)-component with a larger β (α), it is usually contained in a group of vertices with more connections. To measure such properties, we introduce the concept of α-/β-offsets.

Definition 10 (α-/β-offset)**.** *Given a vertex $u \in V(G)$ and $\alpha \in [1, deg(u)]$, the α-offset denoted as $s_a(u, \alpha)$ is the maximal β value where u can be contained in an (α, β)-component. Similarly, given a vertex $v \in V(G)$ and $\beta \in [1, deg(v)]$, the β-offset $s_b(v, \beta)$ of v is the maximal α value where v can be contained in an (α, β)-component. The maximal α-offset (β-offset) value for α (β) is denoted as $\beta_{max}(\alpha)$ ($\alpha_{max}(\beta)$).*

The hierarchy of a bipartite graph can be defined as follows according to 1) the disjointedness of subgraphs based on the concept of (α, β)-component; 2) the containment relationships based on the concept of α-/β-offsets.

Definition 11 (bipartite hierarchy)**.** *Given a bipartite graph G, its bipartite hierarchy $\mathcal{I}$ consists of the upper hierarchy $\mathcal{I}^U$ and the lower hierarchy $\mathcal{I}^L$ (i.e., $\mathcal{I} = \mathcal{I}^U \cup \mathcal{I}^L$). $\mathcal{I}^U$ and $\mathcal{I}^L$ contain α_{max} and β_{max} trees, respectively.*

For each $\alpha \in [1, \alpha_{max}]$, the upper tree $\mathcal{I}^U_\alpha \in \mathcal{I}^U$ organizes all the upper vertices with non-zero α-offsets. For each tree node p in the i-th level of $\mathcal{I}^U_\alpha$, p is corresponding to an (α, i)-component (denoted by $C_{\alpha,i}(p)$) with a unique upper vertex set in $\mathcal{I}^U_\alpha$ (i.e., the subtree rooted by p contains the set of upper vertices in $C_{\alpha,i}(p)$). In the tree node p, the set of upper vertices in $U(C_{\alpha,i}(p))$ with their α-offsets equal to i are stored (denoted by $V(p)$). For a tree node p' on the i'-th level of $\mathcal{I}^U_\alpha$ ($i' > i$), it is a child of p if (1) $U(C_{\alpha,i'}(p')) \subset U(C_{\alpha,i}(p))$; (2) there is no other tree node p'' on the i''-th level which is corresponding to $C_{\alpha,i''}(p'')$ satisfying $U(C_{\alpha,i'}(p')) \subset U(C_{\alpha,i''}(p'')) \subset U(C_{\alpha,i}(p))$.

For each $\beta \in [1, \beta_{max}]$, the lower tree $\mathcal{I}^L_\beta \in \mathcal{I}^L$ organizes all the lower vertices with non-zero β-offsets symmetrically.

Note that for each vertex, we also record the addresses of the tree nodes containing it to quickly obtain its locations in the hierarchy.

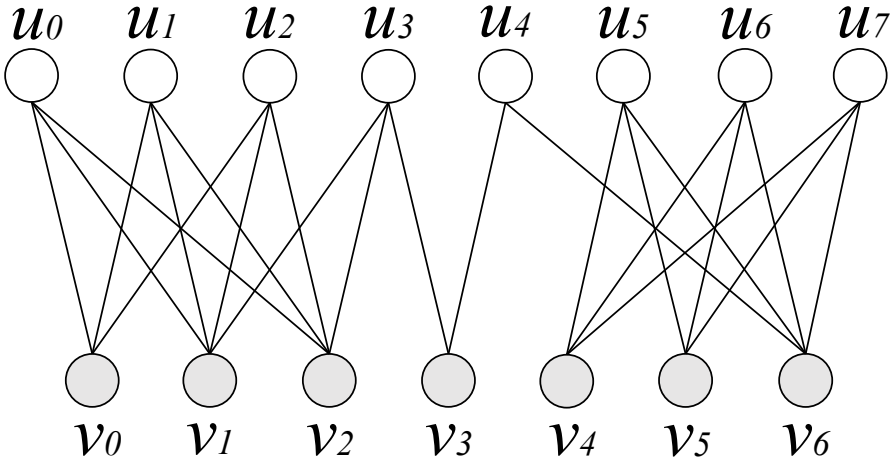


Figure 5.1: A bipartite graph G

Example 5. *Consider the bipartite graph G in Figure 5.1. The bipartite hierarchy $\mathcal{I}$ of G is shown in Figure 5.2. $\mathcal{I}$ contains three upper trees and four lower trees. For instance, $\mathcal{I}^U_1$ contains three tree nodes, one in the second level and two in the fourth level. The tree node in the second level corresponds to a (1, 2)-component, which has two children in the fourth level. The upper vertices*

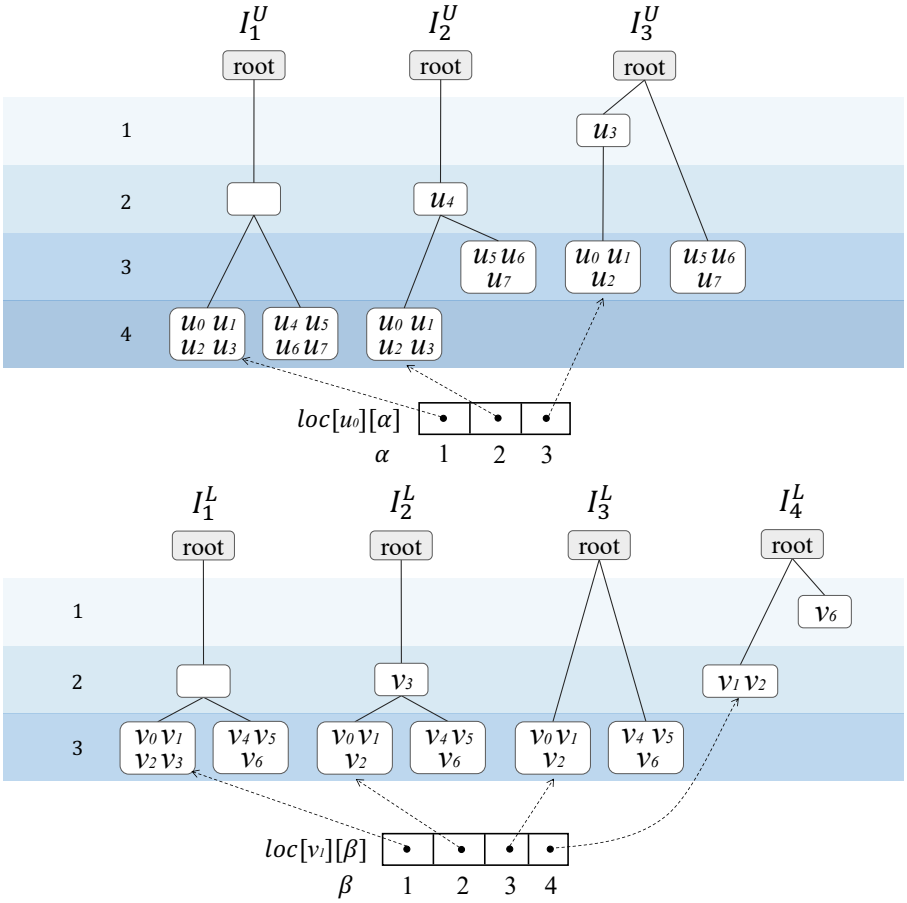


Figure 5.2: The bipartite hierarchy of G

in these components are contained in the tree nodes. The location array (i.e., loc) records the address of the tree node containing each vertex in different trees. For example, $loc[u_0][1]$ records the address of the tree node containing u_0 in $\mathcal{I}_1^U$.

Theorem 1. *Given a bipartite graph G, it needs $O(m)$ space to store the bipartite hierarchy of G.*

Proof. Firstly, since each upper/lower vertex v only appears in each upper/lower tree once and v can be contained in at most $deg(v)$ trees, the number of vertices in the hierarchy (and the location array) is bounded by $O(\sum_{v \in V(G)} deg(v)) = O(m)$. Accordingly, there are at most $O(m)$ non-empty tree nodes (i.e., the tree nodes that contain at least one vertex). Based on the structure of the bipartite hierarchy, each empty tree node links at least two non-empty tree nodes together to combine their corresponding connected components as one. Thus, there exist at most $O(m-1)$

empty tree nodes. The number of links between two tree nodes is also bounded by $O(m)$ due to the tree structure. In total, it needs $O(m)$ space to store the bipartite hierarchy. □

5.4 Hierarchy Construction

In this section, we present algorithms for building the bipartite hierarchy.

5.4.1 A top-down approach

Algorithm 2: The HC-TD Algorithm

Input: G: a bipartite graph

Output: $\mathcal{I} = \mathcal{I}^U \cup \mathcal{I}^L$: the bipartite hierarchy of G

1 initialize α_{max} trees in $\mathcal{I}^U$ and initialize β_{max} trees in $\mathcal{I}^L$;
2 initialize an array loc to record the locations of vertices in $\mathcal{I}$;
3 **for** $\alpha \leftarrow 1$ to α_{max} **do**
4 **for** $\beta \leftarrow 1$ to $\beta_{max}(\alpha)$ **do**
5 retrieve the (α, β)-core $R_{\alpha,\beta}$;
6 **if** $\beta > 1 \wedge$ *upper vertices in each component of* $R_{\alpha,\beta}$ *are unchanged compared with* $R_{\alpha,\beta-1}$ **then**
7 move tree nodes in β-1 level of $\mathcal{I}^U_\alpha$ to β level;
8 **continue**;
9 **foreach** *(α, β)-component* $C_{\alpha,\beta} \subseteq R_{\alpha,\beta}$ **do**
10 $S \leftarrow$ upper vertices with α-offset = β in $C_{\alpha,\beta}$;
11 make a tree node p and set $V(p) \leftarrow S$;
12 put p in the β-th level of $\mathcal{I}^U_\alpha$ and link p with its parent in $\mathcal{I}^U_\alpha$;
13 **foreach** *upper vertex* $u \in \mathcal{I}^U_\alpha$ **do**
14 $loc[u][\alpha] \leftarrow$ the address of the tree node containing u in $\mathcal{I}^U_\alpha$;
15 build the lower hierarchy $\mathcal{I}^L$ similarly as Lines 3 - 14;
16 **return** $\mathcal{I}$

Since the (α, β)-components in each (α, β)-core can be easily obtained using breath-first-search, it is straightforward to build each upper tree $\mathcal{I}^U_\alpha$ (or each lower tree $\mathcal{I}^L_\beta$) in a top-down manner by

following the peeling paradigm in prior works [53, 70]. Algorithm 2 shows the details of the index construction algorithm HC-TD. For each $\alpha \in [1, \alpha_{max}]$, HC-TD builds $\mathcal{I}_\alpha^U$ from the root node (Line 3). Specifically, for each $\beta \in [1, \beta_{max}(\alpha)]$, it first retrieves the (α, β)-core $R_{\alpha,\beta}$ (Line 5). Note that the (α, β)-core can be obtained by peeling the $(\alpha - 1, \beta)$-core (or $(\alpha, \beta - 1)$-core) in the former iteration rather than computing it from scratch. If $\beta > 1$ and the upper vertices in each connected component of $R_{\alpha,\beta}$ are unchanged compared with $R_{\alpha,\beta-1}$, we just move tree nodes in the β-1 level of $\mathcal{I}_\alpha^U$ to the β level. Otherwise, we obtain each (α, β)-component $C_{\alpha,\beta}$ from the (α, β)-core easily using breadth-first search. For each $C_{\alpha,\beta}$, we obtain the set of upper vertices S with α-offset $= \beta$ to build the tree node p in the β level of $\mathcal{I}_\alpha^U$. Then, we link p with its parent according to the containment relationships between the (α, β)-components they correspond to. After that, for each upper vertex $u \in \mathcal{I}_\alpha^U$, we record the address of the tree node it belongs to. The lower hierarchy $\mathcal{I}^L$ can be built in a similar way. Note that Algorithm 2 builds the bipartite hierarchy in $O((\alpha_{max} + \beta_{max}) \cdot n \cdot m)$ time. This is because there are $(\alpha_{max} + \beta_{max})$ trees in the hierarchy. For each tree, it has at most $O(n)$ tree nodes and building each of them needs $O(m)$ time using BFS.

5.4.2 A bottom-up approach

Although the algorithm HC-TD is easy to implement, it is inefficient when handling large-scale bipartite graphs since it needs to compute the vertex set of each tree node separately. As (α, β)-cores are nested naturally w.r.t. α and β, we propose the HC-BU algorithm that can utilize such nested structure information to accelerate the construction process.

Build one tree. We first show how to build an upper tree $\mathcal{I}_\alpha^U$ in the upper hierarchy using HC-BU. The following lemma follows directly from the definition of (α, β)-core .

Lemma 1. *Given $\alpha \in [1, \alpha_{max}]$, $\beta \in [2, \beta_{max}(\alpha)]$, and an (α, β)-component C_1, there must exist an $(\alpha, \beta - 1)$-component C_2 where $C_1 \subseteq C_2$.*

Since each tree node in an upper tree corresponds to an (α, β)-component, the above lemma depicts the nested relationships among the tree nodes in one tree. According to this lemma, we can follow

a bottom-up manner to first build the tree nodes in the deepest tree level and then incrementally build the tree nodes in higher tree levels based on them. To build the tree nodes in a given tree level, we still face the following two challenges: 1) for one vertex in the level, how to identify the tree node it should belong to; and 2) for one tree node, how to know which tree nodes in the deeper level are its children. To address the above issues, we have the following observations.

Lemma 2. *Given an upper tree I_α^U, for two vertices in the β level of I_α^U, they are contained by the same tree node if they belong to the same connected component in the (α, β)-core . For a tree node in the β level and one of its children in the β' level ($\beta' > \beta$) of I_α^U, the vertices in these two tree nodes must belong to the same connected component in the (α, β)-core .*

Proof. This lemma directly follows from the structure of the bipartite hierarchy. □

Based on the above lemma, the vertex sets of tree nodes (in one tree level) can be obtained via retrieving the connected components of (α, β)-cores. Since we follow a bottom-up manner to build the tree nodes, we can maintain the connected components in constant amortized time with the union-find data structure, which is widely used in the literature [97–99]. Specifically, when building the β level of the upper tree I_α^U, we can only process each vertex with α-offset = β and check which connected component it should belong to with the union-find structure.

Build the hierarchy. Utilizing the above approach to build one tree, we show the details of the algorithm to build the hierarchy in Algorithm 3. For each $\alpha \in [1, \alpha_{max}]$, HC-BU puts the vertices in $(\alpha, 1)$-core into different sets S_1, S_2, ..., $S_{\beta_{max}(\alpha)}$ according to their α-offsets (Line 4). Then, it builds $\mathcal{I}_\alpha^U$ in a bottom-up manner and starts from the $\beta_{max}(\alpha)$ level (Lines 5 - 9). In the $\beta_{max}(\alpha)$ level, HC-BU computes the set $\mathcal{E}$ of connected components from the vertices in $S_{\beta_{max}(\alpha)}$ and their incident edges (using the union-find structure). For each connected component $C \in \mathcal{E}$, HC-BU makes a tree node p with the upper vertices in C (Line 9). Then, to build the $\beta_{max}(\alpha) - 1$ level, we can maintain the connected components in $\mathcal{E}$ using the incoming vertices in $S_{\beta_{max}(\alpha)-1}$. For each connected component $C \in \mathcal{E}$, HC-BU makes a tree node p with the upper vertices in $S_{\beta_{max}(\alpha)-1} \cap C$. In addition, HC-BU links p with its children node according to the vertices in the connected component. Note that we skip creating the tree nodes in one level if the upper vertices

in each connected component of $\mathcal{E}$ are unchanged compared with the previous level. Following this manner, we can also build the other levels of $\mathcal{I}_\alpha^U$. Finally, we record the address of the tree node where each upper vertex in $\mathcal{I}_\alpha^U$ belongs to. The lower hierarchy $\mathcal{I}^L$ can be built in a similar fashion.

Algorithm 3: The HC-BU Algorithm

Input: G: a bipartite graph

Output: $\mathcal{I} = \mathcal{I}^U \cup \mathcal{I}^L$: the bipartite hierarchy of G

```
1  initialize α_max trees in I^U and initialize β_max trees in I^L;
2  initialize an array loc to record the locations of vertices in I;
3  for α ← 1 to α_max do
4      put vertices in (α,1)-core into different sets S_1, S_2, ..., S_{β_max(α)} according to their offsets;
5      for β ← β_max(α) to 1 do
6          if β = β_max(α) then
7              compute the connected components from S_β and put them into E;
8              foreach component C ∈ E do
9                  make a tree node p with the upper vertices in C on the β-th level of I_α^U;
10         else
11             compute the connected components from S_β ∪ E and put them into E;
12             if the upper vertices in each connected component of E are unchanged then
13                 continue;
14             foreach connected component C ∈ E do
15                 make a tree node p with the upper vertices in S_β ∩ C on the β-th level of I_α^U;
16                 link p with its children according to C;
17     foreach upper vertex u ∈ I_α^U do
18         loc[u][α] ← the address of the tree node containing u in I_α^U;
19 build the lower hierarchy I^L similarly as Lines 3 - 18;
20 return I
```

Lemma 3. *Given a bipartite graph G, Algorithm 2 uses $O(m + \sum_{\alpha=1}^{\alpha_{max}} \mathsf{size}((\alpha,1)\text{-}core) + \sum_{\beta=1}^{\beta_{max}} \mathsf{size}((1,\beta)\text{-}core))$ time to build the bipartite hierarchy.*

Proof. Firstly, it needs $O(m)$ time in total to retrieve each $(\alpha, 1)$-core (from α = 1 to $\alpha = \alpha_{max}$) and each $(1, \beta)$-core (from β = 1 to $\beta = \beta_{max}$) incrementally. Then, when building each upper tree

$\mathcal{I}_\alpha^U$ (or lower tree $\mathcal{I}_\beta^L$), it needs $O(\mathsf{size}((\alpha,1)\text{-}core))$ (or $O(\mathsf{size}((1,\beta)\text{-}core))$ time if we consider that the union-find operations run in constant time. □

Example 6. *Consider the bipartite graph in Figure 5.1. We show how to build $\mathcal{I}_1^U$ in Figure 5.2 using* HC-BU. *We first put vertices of (1, 1)-core into different sets based on their α-offsets. In this case, S_2 = {v_3}, S_3 = {v_0, v_4, v_5}, S_4= $\{u_0$-$u_7, v_1, v_2, v_6\}$. Then, when $\beta = 4$, we obtain $\mathcal{E}$ containing two (1, 4)-components and create two tree nodes that contain the upper vertices in S_4. After that, when $\beta = 3$, we add the vertices in S_3 and their incident edges into $\mathcal{E}$. We observe that the upper vertices in the (1, 3)-components are unchanged compared with the (1, 4)-components. Thus, we move to the next level. When $\beta = 2$, S_2 = {v_3} and $\mathcal{E}$ now has only one $(1,2)$-component. Then, we create a tree node and link the tree nodes in the fourth level as its children. When $\beta = 1$, S_1 is empty, and we do not need to create any tree nodes in the first level. Finally, for each vertex in the upper tree $\mathcal{I}_1^U$, we record the address of the tree node it belongs to.*

Algorithm 4: Community Search

Input: $\mathcal{I}, q, \alpha, \beta$

Output: $\mathcal{R}$

1 $p = loc[q][\alpha]$
2 **while** $p \neq null$ *and* $p.level \leq \beta$ **do**
3 $\quad p = p.parent$
4 $roots = p.childs$
5 **foreach** $child \in roost$ **do**
6 $\quad \mathcal{R} = \mathcal{R} \cup child.vertices$
7 $\quad roots = roots \cup child.childs$
8 $p = null, roots = \emptyset$
9 **foreach** $v \in N(q)$ **do**
10 $\quad$ **if** $loc[v][\beta].level \geq \alpha$ **then**
11 $\quad\quad p = loc[v][beta]$
12 repeat lines 2 -3 and set $p.level \leq \alpha$
13 repeat lines 4 - 7
14 **return** $\mathcal{R}$

5.5 Support Efficient Community Search

In bipartite graphs, given a query vertex q and parameters α, β, the group of vertices in the connected component of the (α, β)-core containing q is considered in the community of q [70]. In [70], the authors only propose an index to support the retrieval of the (α, β)-core without considering the connectivity. Thus, a further BFS is needed to search the community of q. Note that this community of q can be efficiently obtained via the bipartite hierarchy using the following algorithm. Here, we introduce the community search algorithm as shown In Algorithm 4. Suppose q is an upper vertex. We first run the following steps to find the upper vertices in the community (lines 1 - 7).

- *Step 1.* In line 1, we find the tree node p containing q in the upper tree $\mathcal{I}_\alpha^U$ using the location array (i.e., $loc[q][\alpha]$).
- *Step 2.* In lines 2 - 3, we retrieve the ancestor node p' of p that stays in the β level of $\mathcal{I}_\alpha^U$. If it does not exist, we assign p' as the first ancestor node of p below the β level.
- *Step 3.* We traverse the subtree rooted by p' and retrieve all the vertices in the subtree, which are the upper vertices in the community (Lines 4 - 7).

To find the lower vertices in the community, in lines 9 - 11, we first identify a neighbor of q that stays in the α' level of $\mathcal{I}_\beta^L$ ($\alpha' > \alpha$). Then, we run a similar approach as Step 2 and Step 3 (Lines 12 - 13). Note that the time complexity of the above algorithm is $O(|\mathcal{I}_\alpha^U[\beta]| + |\mathcal{I}_\beta^L[\alpha]|)$. Here $|\mathcal{I}_\alpha^U[\beta]|$ denotes the total size of the subtrees (including the tree nodes and vertices in the tree nodes) in the β level of $\mathcal{I}_\alpha^U$. The space complexity of the algorithm is $O(m)$.

Example 7. *Consider the bipartite graph in Figure 5.1. We show how to find the community of u_0 using the bipartite hierarchy in Figure 5.2 by given parameters $\alpha = 2, \beta = 2$. We first find the tree node p containing u_0 in the fourth level of $\mathcal{I}_2^U$. Then, we retrieve the ancestor node p' of p in the second level since $\beta = 2$. After that, we traverse the subtree rooted by p' and retrieve all the upper vertices in the subtree (i.e., $u_0, u_1, u_2, u_3, u_4, u_5, u_6$, and u_7). To find the lower vertices in the community, we first get v_0, which is the neighbor of u_0 and is located in the third level of $\mathcal{I}_2^L$.*

Then, we find the tree node p'' in the second level of $\mathcal{I}_2^L$ through v_0, and traverse the subtree rooted by p''. After that, we can get the lower vertices in the community, which are $v_0, v_1, v_2, v_3, v_4, v_5$, and v_6.

Table 5.2: Summary of Datasets

Dataset	$\|E(G)\|$	Type of E	$\|U(G)\|$	Type of U	$\|L(G)\|$	Type of L
RL	233,286	Membership	168,337	Artist	18,421	Record
YG	293,360	Membership	94,238	User	30,087	Group
GH	440,237	Membership	56,519	User	120,867	Project
TM	1,366,466	Membership	901,130	Athlete	34,461	Team
IM	2,715,604	Association	685,568	Person	186,414	Work
WC	3,795,796	Inclusion	1,853,493	Article	182,947	Category
FG	8,545,307	Membership	395,979	User	103,631	Group
PA	12,282,059	Authorship	1,953,085	Author	5,624,219	Publication
ML	25,000,095	Rating	162,541	User	59,047	Movie
DUI	101,798,955	Tag	833,081	User	33,778,221	URL

5.6 Experiments

In this section, we first evaluate the bipartite hierarchy for some real applications. Then, we evaluate the algorithms for building and maintaining the bipartite hierarchy.

5.6.1 Experimental setting

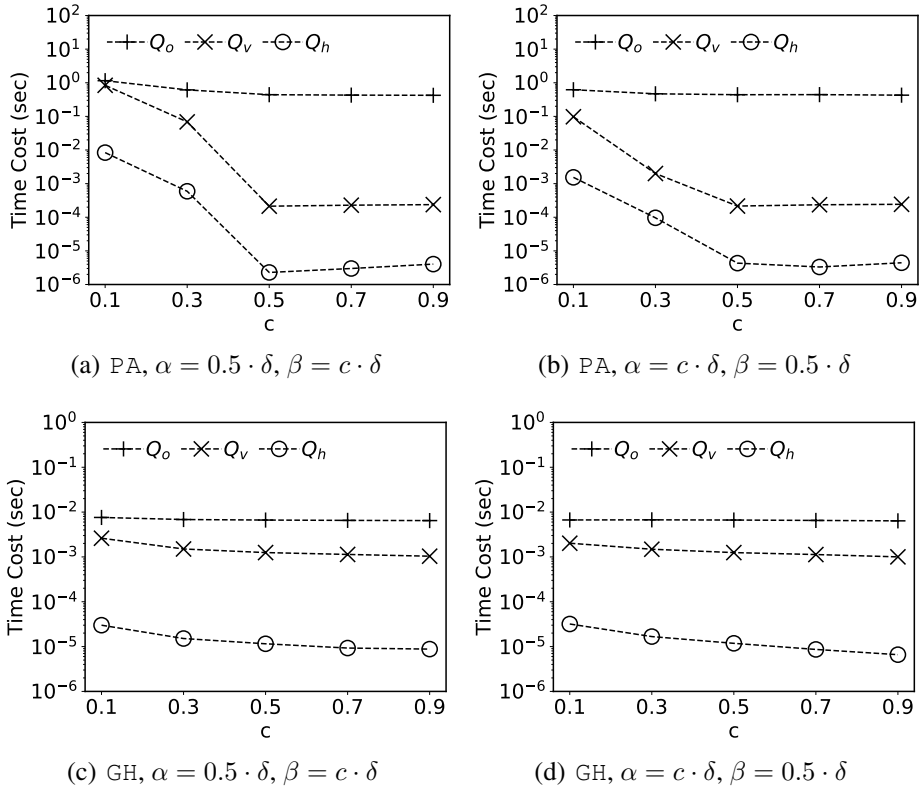
Algorithms. Our empirical studies are conducted against the following designs:

The hierarchy construction algorithms. 1) the top-down hierarchy construction algorithm HC-TD in Section 5.4.1, and 2) the bottom-up hierarchy construction algorithm HC-BU in Section 5.4.2. We also report the size of the bipartite hierarchy.

The algorithms for searching (α, β)-core -based communities. 1) the online peeling algorithm Q_o in [53] that finds an (α, β)-core in linear time, 2) the query algorithm Q_v based on the bicore index I_v proposed in [70] that can optimally retrieve the vertex set of an (α, β)-core , and 3) the query algorithm Q_h based on our bipartite hierarchy in Section 5.5. Note that for the algorithms Q_o and

Q_v, we need to further conduct a BFS search to retrieve the community of a query vertex after finding the (α, β)-core .

We implement the algorithms with C++, and all the experiments are run on a Linux server with Intel Xeon E5-2698 processor and 512GB main memory. *We terminate an algorithm if the running time is more than* 10^5 *seconds*.



(a) PA, $\alpha = 0.5 \cdot \delta$, $\beta = c \cdot \delta$ (b) PA, $\alpha = c \cdot \delta$, $\beta = 0.5 \cdot \delta$

(c) GH, $\alpha = 0.5 \cdot \delta$, $\beta = c \cdot \delta$ (d) GH, $\alpha = c \cdot \delta$, $\beta = 0.5 \cdot \delta$

Figure 5.3: Retrieving the (α, β)-communities, varying α and β

Datasets. We use 10 real-world datasets in our experiments, which are `Record labels` (RL), `YouTube` (YG), `Github` (GH), `Dbpedia Team` (TM), `IMDB` (IM), `Wikipedia categories` (WC), `Flickr` (FG), `DBLP` (PA), `MovieLens` (ML) and `Delicious-ui` (DUI). All the used datasets can be found in KONECT [1].

The summary of datasets is shown in Table 5.2. U and L are vertex layers, E denotes the edge set. We also show the types of edges and vertices in the table.

[1] http://konect.cc/

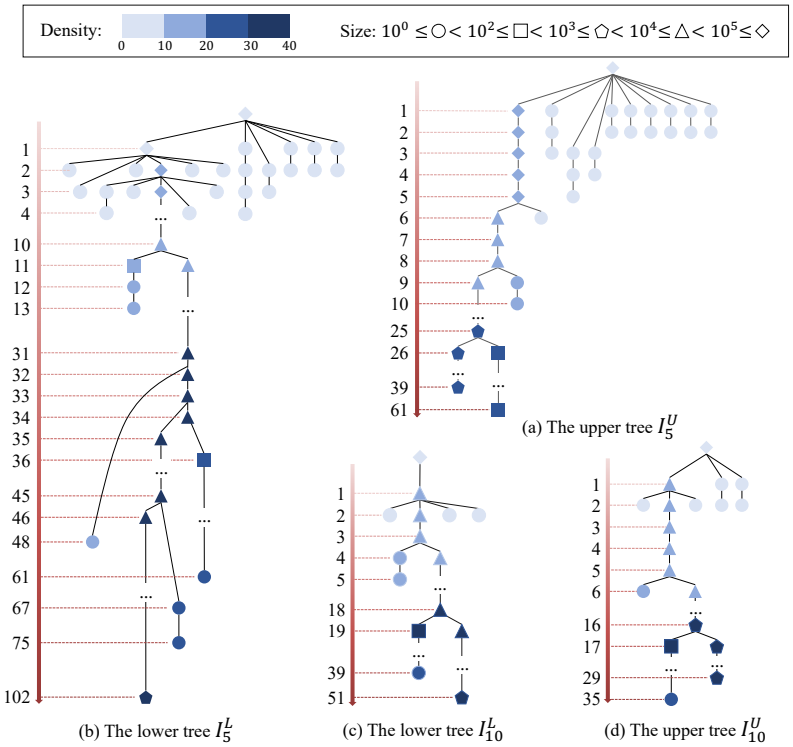


Figure 5.4: Part of the bipartite hierarchy for IMDB

5.6.2 Application on network visualization

Here, we visualize a part of the bipartite hierarchy of IMDB in Figure 5.4, which shows two upper trees and two lower trees with a clearly branching and hierarchy structure. $\mathcal{I}_5^U$, $\mathcal{I}_{10}^U$, $\mathcal{I}_5^L$, and $\mathcal{I}_{10}^L$ contains 95, 57, 694, and 245 tree nodes, respectively. Due to the short of space, we omit the tree nodes that (1) do not have any children and directly link to the root; (2) or locate in the middle of a chain. We show the tree nodes with different colors and shapes, where the color denotes the density of its corresponding (α, β)-component computed with the equation $d(G) = |E(G)|/\sqrt{|U(G)||L(G)|}$ [100], and the shape denotes the size of its (α, β)-component. The tree level of each tree node is also shown on the left of the tree. We can observe that in the same tree, with the increasing of tree level, the density of the component is generally increased, and the size of the component is decreased. In addition, $\mathcal{I}_5^U$ has more tree nodes than $\mathcal{I}_{10}^U$ and the deepest tree level of $\mathcal{I}_5^U$ is greater than $\mathcal{I}_{10}^U$. However, at the same level, the component of $\mathcal{I}_{10}^U$ is denser than $\mathcal{I}_5^U$. Similar observations can also be made in lower trees. In this case study, we validate that our bipartite hierarchy can clearly show the hierarchy of a bipartite network at different levels of

granularity and can be used to find the dense regions.

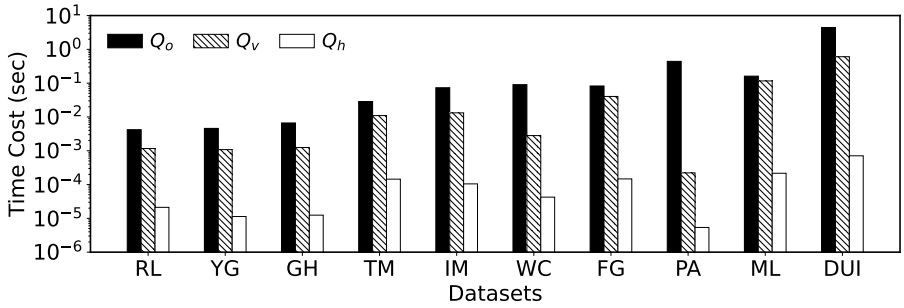


Figure 5.5: Performance on searching communities

5.6.3 Application on efficient community search

Given a query vertex q and parameters α, β, we aim to search the set of vertices in the (α, β)-component of q. In Figure 5.5, we evaluate the performance of community search algorithms on all the datasets by setting $\alpha = \beta = 0.5\delta$. Here, δ is the maximal value where a (δ, δ)-core exists in a bipartite graph. We can see that, based on the bipartite hierarchy, the query algorithm Q_h significantly outperforms the online algorithm Q_o and the bicore-index-based algorithm Q_v by several orders of magnitude. This is because Q_h does not need a further BFS to obtain the connected components of the query vertex. We also vary the parameters α and β to evaluate the performance of these algorithms in Figure 5.3. We can see that when α and β become larger, Q_h can be much faster than Q_o and Q_v. This is because with α and β increases, the size of the community is decreased, and the vertices that need to be retrieved in the bipartite hierarchy is decreased.

5.6.4 Evaluations of the bipartite hierarchy

Evaluating the hierarchy construction time and size. Here, we evaluate the construction time and size of the hierarchy.

1) Hierarchy construction time. In Figure 5.6, we evaluate the index construction algorithms HC-TD and HC-BU on all the datasets. We can see that the bottom-up approach HC-BU significantly

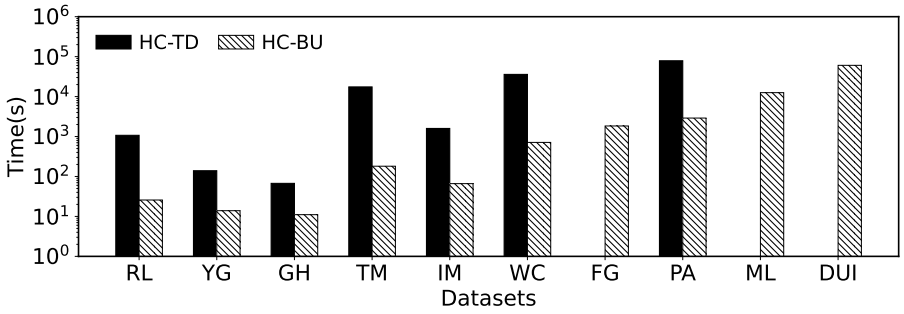


Figure 5.6: The construction time of hierarchy

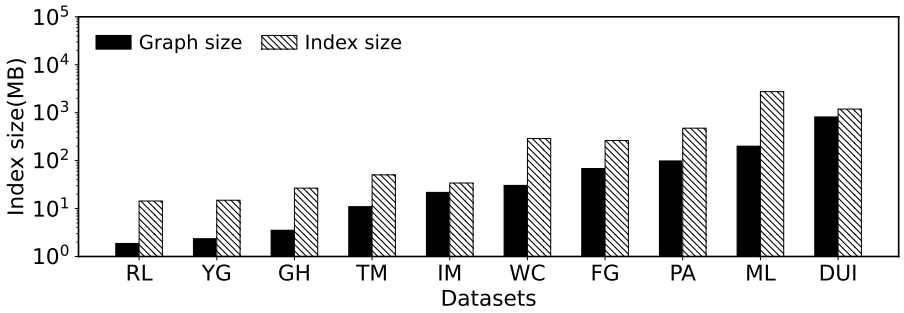


Figure 5.7: The size of hierarchy

outperforms the top-down approach HC-TD on all the datasets. This is because HC-BU can utilize the nested structure of the (α, β)-core model to share some construction costs. Note that HC-TD cannot build the hierarchy on large datasets `FG`, `ML` and `DUI` within the given time limit.

2) The size of hierarchy. As shown in Figure 5.7, we evaluate the size of the bipartite hierarchy by comparing it with the graph size (i.e., $|G|$). We can see that the size of the bipartite hierarchy is only $1.4\times$-$13.7\times$ to the graph size, which is very space-efficient in practice.

5.7 Conclusion

In this work, we discover the hierarchy of bipartite graphs based on the (α, β)-core model. We propose the bipartite hierarchy model and devise efficient algorithms for constructing the hierarchy algorithms. We conduct extensive experiments to validate the effectiveness and efficiency of the proposed techniques.

Chapter 6

Conclusion and Future Directions

This chapter will summarise our work and several possible directions for future work. Specifically, in Section 6.1 we will summarize the significant contribution of this thesis. And in Section 6.2, we will propose several possible directions for future work.

6.1 Conclusions

In this thesis, we are mainly working on improving the usability and efficiency of the graph database. We designed SQL2Cypher system to automatically convert RDBMS to graph database to enhance the availability of graph database. FSPS improves the efficiency of graph database queries by FPGA acceleration. Since several graphs databases currently do not support cohesive subgraph mining, such as PatMat [51], Neo4j [19], and so on. Therefore, we explore this aspect of the graph database and try to implement the bipartite graph model in the graph database. Different models and algorithms can increase the usability of the graph database. The challenges we address in the thesis: (1) Automatic migration of relational databases to graph databases and support for query language translation, (2) Store graph data and support graph database query language and (3) Explore new bipartite graph models and design the construction and maintenance algorithms.

In this thesis, we mainly optimize the availability and operation of the graph database system. To improve the usability, in the first work , we propose a system called SQL2Cypher that can convert the relational database system to a graph database system. The system will detect whether the table can be converted into an edge with the properties to save disk space and adapt relational databases to graph databases. In addition, Our system performs data similarity checks when migrating data to save storage space. To translate the SQL queries to Cypher queries with relationships, we store the relationships of migrated tables as a graph structure. We also provide a user-friendly and interactive interface for the ease of users. To improve the efficiency of graph database operation, in the second work, we propose a system called SQL2Cypher that can convert the relational database system to a graph database system. The system will detect whether the table can be converted into a property edge to save disk space and adapt relational databases to graph databases. In addition, Our system performs data similarity checks when migrating data to save storage space. To translate the SQL queries to Cypher queries with relationships, we store the relationships of migrated tables as a graph structure. We also provide a user-friendly and interactive interface for the ease of users. Lastly, in order to adapt the graph database to a wider range of problems, we explored the model of cohesive subgraphs. In this work, we discover the hierarchy of bipartite graphs based on the (α, β)-core model. We propose the bipartite hierarchy model and devise efficient algorithms for building the hierarchy. In addition, we also propose hierarchy maintenance algorithms to handle the cases when edges are inserted/deleted dynamically. We conduct extensive experiments to validate the model's effectiveness and the algorithms' efficiency in real-world graphs..

6.2 Directions for future work

In this section, we will propose several possible directions for future work. **SQL2Cypher: Automated Data and Query Migration from RDBMS to GDBMS.** There are many relational databases and graph databases that we do not support yet to support more conversions between relational and graph databases in the future. For relational databases, currently SQL2Cypher supports MySQL, PostgreSQL and Microsoft SQL. For graph databases, we currently support Neo4j and PatMat. We may also need to convert the graph database to the relational database in real life.

So in the future, we can add this function to our system. In addition, we can use machine learning to train and identify schema in relational databases, which will allow our system to adapt to more databases.

FSPS: Accelerating Subgraph and Path Queries Using FPGA. Currently, FSPS supports the two most commonly used algorithms, both subgraph matching and path query. For the subgraph pattern matching, the user only can draw the patterns with properties now. This query approach is not a formal way of searching, and there is no way to do the batch search. In future work, we can support Cypher query language for our system. In addition, it would be interesting to support more algorithms for our system, such as k-core community search, bipartite graph algorithms (e.g., (α, β)-core) and k-truss. In addition, instead of adding algorithms, we can explore to support the distributed framework for our system. Furthermore, We can design FSPS as a distributed database, which can effectively improve the query performance of the system.

Discovering Hierarchy of Bipartite Graphs with Cohesive Subgraphs. With the popularity of distributed frameworks in recent years, many graph algorithms are using distributed frameworks for graph processing, such as subgraph matching [47, 101]. For *bipartite hierarchy*, every bipartite hierarchy can be processed in parallel. It would be interesting to see how to construct the *bipartite hierarchy* in a distributed framework. In addition, using FPGA to accelerate *bipartite hierarchy* construction might be an interesting issue to be investigated.

References

[1] M. A. Saleem, R. Kumar, T. Calders, X. Xie, and T. B. Pedersen, "Location influence in location-based social networks," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, 2017, pp. 621–630.

[2] T. Cai, J. Li, N. A. H. Haldar, A. Mian, J. Yearwood, and T. Sellis, "Anchored vertex exploration for community engagement in social networks," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 409–420.

[3] Y. Arfat, R. Mehmood, and A. Albeshri, "Parallel shortest path graph computations of united states road network data on apache spark," in *International Conference on Smart Cities, Infrastructure, Technologies and Applications*. Springer, 2017, pp. 323–336.

[4] Z. Yu, X. Yu, N. Koudas, Y. Liu, Y. Li, Y. Chen, and D. Yang, "Distributed processing of k shortest path queries over dynamic road networks," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 665–679.

[5] H. Yuan, G. Li, Z. Bao, and L. Feng, "Effective travel time estimation: When historical trajectories over road networks matter," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2135–2149.

[6] A. Thomas, R. Cannings, N. A. Monk, and C. Cannings, "On the structure of protein–protein interaction networks," 2003.

[7] S. Zhang, S. Li, and J. Yang, "Gaddi: distance index based subgraph matching in biological networks," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009, pp. 192–203.

[8] "Database trend," 2020. [Online]. Available: $https://db-engines.com/en/ranking_categories$

[9] E. Codd, "A relational model of data for large shared data banks. 1970," *MD Comput*, vol. 15, pp. 162–166, 1998.

[10] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proc. VLDB Endow.*, vol. 11, no. 4, p. 420–431, Dec. 2017.

[11] R. Wang, Z. Yang, W. Zhang, and X. Lin, "An empirical study on recent graph database systems," in *Knowledge Science, Engineering and Management*. Cham: Springer International Publishing, 2020, pp. 328–340.

[12] R. De Virgilio, A. Maccioni, and R. Torlone, "Converting relational to graph databases," in *First International Workshop on Graph Data Management Experiences and Systems*, 2013, pp. 1–6.

[13] "neo4j-etl," https://neo4j.com/developer/neo4j-etl/.

[14] C. Bizer, "D2r map-a database to rdf mapping language," 2003.

[15] "Cypher," 2021. [Online]. Available: https://neo4j.com/developer/cypher

[16] J. Heer, J. M. Hellerstein, and S. Kandel, "Predictive interaction for data transformation." in *CIDR*, 2015.

[17] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Transactions on knowledge and data engineering*, vol. 19, no. 1, pp. 1–16, 2006.

[18] K. Hao, Z. Yang, L. Lai, Z. Lai, X. Jin, and X. Lin, "Patmat: a distributed pattern matching engine with cypher," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 2921–2924.

[19] "Neo4j," 2020. [Online]. Available: https://neo4j.com

[20] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1695–1698.

[21] Y. Tian, W. Sun, S. J. Tong, E. L. Xu, M. H. Pirahesh, and W. Zhao, "Synergistic graph and sql analytics inside ibm db2," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 1782–1785, 2019.

[22] Y. Peng, Y. Zhang, X. Lin, W. Zhang, L. Qin, and J. Zhou, "Towards bridging theory and practice: Hop-constrained s-t simple path enumeration," *Proc. VLDB Endow.*, vol. 13, no. 4, p. 463–476, Dec. 2019. [Online]. Available: https://doi.org/10.14778/3372716.3372720

[23] M. Besta, D. Stanojevic, J. D. F. Licht, T. Ben-Nun, and T. Hoefler, "Graph processing on fpgas: Taxonomy, survey, challenges," *arXiv preprint arXiv:1903.06697*, 2019.

[24] X. Jin, Z. Yang, X. Lin, S. Yang, L. Qin, and Y. Peng, "Fast: Fpga-based subgraph matching on massive graphs," *arXiv preprint arXiv:2102.10768*, 2021.

[25] Z. Lai, Y. Peng, S. Yang, X. Lin, and W. Zhang, "Pefp: Efficient k-hop constrained s-t simple path enumeration on fpga," *arXiv preprint arXiv:2012.11128*, 2020.

[26] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *J. ACM*, vol. 30, no. 3, pp. 417–427, 1983. [Online]. Available: https://doi.org/10.1145/2402.322385

[27] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003. [Online]. Available: http://arxiv.org/abs/cs/0310049

[28] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *SIGMOD*, 2014, pp. 1311–1322.

[29] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," *Proc. VLDB Endow.*, vol. 9, no. 4, pp. 276–287, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol9/p276-huang.pdf

[30] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang, "Efficiently computing k-edge connected components via graph decomposition," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. ACM, 2013, pp. 205–216. [Online]. Available: https://doi.org/10.1145/2463676.2465323

[31] R. Zhou, C. Liu, J. X. Yu, W. Liang, B. Chen, and J. Li, "Finding maximal k-edge-connected subgraphs from a large graph," in *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, E. A. Rundensteiner, V. Markl, I. Manolescu, S. Amer-Yahia, F. Naumann, and I. Ari, Eds. ACM, 2012, pp. 480–491. [Online]. Available: https://doi.org/10.1145/2247596.2247652

[32] A. E. Sariyüce, C. Seshadhri, A. Pinar, and Ü. V. Çatalyürek, "Finding the hierarchy of dense subgraphs using nucleus decompositions," in *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, A. Gangemi, S. Leonardi, and A. Panconesi, Eds. ACM, 2015, pp. 927–937. [Online]. Available: https://doi.org/10.1145/2736277.2741640

[33] A. E. Sariyüce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *Proc. VLDB Endow.*, vol. 12, no. 1, pp. 43–56, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol12/p43-sariyuce.pdf

[34] M. E. Newman, "Scientific collaboration networks. i. network construction and fundamental results," *Physical review E*, vol. 64, no. 1, p. 016131, 2001.

[35] A. E. Sariyüce and A. Pinar, "Peeling bipartite networks for dense subgraph discovery," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, WSDM 2018, Marina Del Rey, CA, USA, February 5-9, 2018*, Y. Chang, C. Zhai, Y. Liu, and Y. Maarek, Eds. ACM, 2018, pp. 504–512. [Online]. Available: https://doi.org/10.1145/3159652.3159678

[36] G. A. Pavlopoulos, P. I. Kontou, A. Pavlopoulou, C. Bouyioukos, E. Markou, and P. G. Bagos, "Bipartite graphs in systems biology and medicine: a survey of methods and applications," *GigaScience*, vol. 7, no. 4, p. giy014, 2018.

[37] Z. Zou, "Bitruss decomposition of bipartite graphs," in *Database Systems for Advanced Applications - 21st International Conference, DASFAA 2016, Dallas, TX, USA, April 16-19, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. B. Navathe, W. Wu, S. Shekhar, X. Du, X. S. Wang, and H. Xiong, Eds., vol. 9643. Springer, 2016, pp. 218–233. [Online]. Available: https://doi.org/10.1007/978-3-319-32049-6_14

[38] K. Wang, W. Zhang, X. Lin, Y. Zhang, L. Qin, and Y. Zhang, "Efficient and effective community search on large-scale bipartite graphs," *arXiv preprint arXiv:2011.08399*, 2020.

[39] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang *et al.*, "Tidb: a raft-based htap database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3072–3084, 2020.

[40] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[41] S. Sivasubramanian, "Amazon dynamodb: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 729–730.

[42] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[43] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[44] A. Spark, "Apache spark," *Retrieved January*, vol. 17, no. 1, p. 2018, 2018.

[45] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[46] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang *et al.*, "A survey and experimental analysis of distributed subgraph matching," *arXiv preprint arXiv:1906.11518*, 2019.

[47] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *arXiv preprint arXiv:1205.6691*, 2012.

[48] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce," *Proceedings of the VLDB Endowment*, vol. 8, no. 10, pp. 974–985, 2015.

[49] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 217–228, 2016.

[50] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," *Journal of the ACM (JACM)*, vol. 65, no. 3, pp. 1–40, 2018.

[51] K. Hao, Z. Yang, L. Lai, Z. Lai, X. Jin, and X. Lin, "Patmat: A distributed pattern matching engine with cypher," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 2921–2924.

[52] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation: An index-based approach," in *The World Wide Web Conference*, 2019, pp. 1130–1141.

[53] D. Ding, H. Li, Z. Huang, and N. Mamoulis, "Efficient fault-tolerant group recommendation using alpha-beta-core," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 2047–2050.

[54] A. Ahmed, V. Batagelj, X. Fu, S.-H. Hong, D. Merrick, and A. Mrvar, "Visualisation and analysis of the internet movie database," in *2007 6th International Asia-Pacific Symposium on Visualization*. IEEE, 2007, pp. 17–24.

[55] D. S. Hochbaum, "Approximating clique and biclique problems," *Journal of Algorithms*, vol. 29, no. 1, pp. 174–200, 1998.

[56] S. Gunnemann, E. Muller, S. Raubach, and T. Seidl, "Flexible fault tolerant subspace clustering for data with missing values," in *2011 IEEE 11th International Conference on Data Mining*. IEEE, 2011, pp. 231–240.

[57] A. K. Poernomo and V. Gopalkrishnan, "Towards efficient mining of proportional fault-tolerant frequent itemsets," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009, pp. 697–706.

[58] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.

[59] "Odbc," 2021. [Online]. Available: https://docs.microsoft.com/en-us/sql/odbc

[60] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: data management for modern business applications," *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012.

[61] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Comput. Surv.*, vol. 50, no. 5, Sep. 2017. [Online]. Available: https://doi.org/10.1145/3104031

[62] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1876–1888, 2018.

[63] R. Wang, Z. Yang, W. Zhang, and X. Lin, "An empirical study on recent graph database systems," in *Knowledge Science, Engineering and Management*, 2020, pp. 328–340.

[64] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang, Y. Zhang, Z. Qian, and J. Zhou, "Distributed subgraph matching on timely dataflow," *Proc. VLDB Endow.*, vol. 12, no. 10, p. 1099–1112, Jun. 2019. [Online]. Available: https://doi.org/10.14778/3339490.3339494

[65] LDBC, "Ldbc benchmark," 2021. [Online]. Available: http://ldbcouncil.org/benchmarks

[66] K. Wang, W. Zhang, X. Lin, Y. Zhang, and S. Li, "Discovering hierarchy of bipartite graphs with cohesive subgraphs," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022.

[67] J. Wang, A. P. de Vries, and M. J. T. Reinders, "Unifying user-based and item-based collaborative filtering approaches by similarity fusion," in *SIGIR 2006: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Seattle, Washington, USA, August 6-11, 2006*, E. N. Efthimiadis, S. T. Dumais, D. Hawking, and K. Järvelin, Eds. ACM, 2006, pp. 501–508. [Online]. Available: https://doi.org/10.1145/1148170.1148257

[68] A. Beutel, W. Xu, V. Guruswami, C. Palow, and C. Faloutsos, "Copycatch: stopping group attacks by spotting lockstep behavior in social networks," in *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, D. Schwabe, V. A. F. Almeida, H. Glaser, R. Baeza-Yates, and S. B. Moon, Eds. International World Wide Web Conferences Steering Committee / ACM, 2013, pp. 119–130. [Online]. Available: https://doi.org/10.1145/2488388.2488400

[69] M. Ley, "The DBLP computer science bibliography: Evolution, research issues, perspectives," in *String Processing and Information Retrieval, 9th International Symposium, SPIRE 2002, Lisbon, Portugal, September 11-13, 2002, Proceedings*, ser. Lecture Notes in Computer Science, A. H. F. Laender and A. L. Oliveira, Eds., vol. 2476. Springer, 2002, pp. 1–10. [Online]. Available: https://doi.org/10.1007/3-540-45735-6_1

[70] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (a,β)-core computation: an index-based approach," in *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, L. Liu, R. W. White, A. Mantrach, F. Silvestri, J. J. McAuley, R. Baeza-Yates, and L. Zia, Eds. ACM, 2019, pp. 1130–1141. [Online]. Available: https://doi.org/10.1145/3308558.3313522

[71] B. Lyu, L. Qin, X. Lin, Y. Zhang, Z. Qian, and J. Zhou, "Maximum biclique search at billion scale," *Proc. VLDB Endow.*, vol. 13, no. 9, pp. 1359–1372, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p1359-lyu.pdf

[72] Y. Zhang, C. A. Phillips, G. L. Rogers, E. J. Baker, E. J. Chesler, and M. A. Langston, "On finding bicliques in bipartite graphs: a novel algorithm and its application to the

integration of diverse biological data types," *BMC Bioinform.*, vol. 15, p. 110, 2014. [Online]. Available: https://doi.org/10.1186/1471-2105-15-110

[73] A. Tanay, R. Sharan, M. Kupiec, and R. Shamir, "Revealing modularity and organization in the yeast molecular network by integrated analysis of highly heterogeneous genomewide data," *Proceedings of the National Academy of Sciences*, vol. 101, no. 9, pp. 2981–2986, 2004.

[74] S. P. Borgatti, "2-mode concepts in social network analysis," *Encyclopedia of complexity and system science*, vol. 6, pp. 8279–8291, 2009.

[75] M. Cerinsek and V. Batagelj, "Generalized two-mode cores," *Soc. Networks*, vol. 42, pp. 80–87, 2015. [Online]. Available: https://doi.org/10.1016/j.socnet.2015.04.001

[76] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation in bipartite graphs," *The VLDB Journal*, vol. 29, no. 5, pp. 1075–1099, 2020.

[77] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu, "Effective and efficient community search over large directed graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 11, pp. 2093–2107, 2019. [Online]. Available: https://doi.org/10.1109/TKDE.2018.2872982

[78] Q. Linghu, F. Zhang, X. Lin, W. Zhang, and Y. Zhang, "Global reinforcement of social networks: The anchored coreness problem," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 2211–2226. [Online]. Available: https://doi.org/10.1145/3318464.3389744

[79] B. Liu, F. Zhang, C. Zhang, W. Zhang, and X. Lin, "Corecube: Core decomposition in multilayer graphs," in *Web Information Systems Engineering - WISE 2019 - 20th International Conference, Hong Kong, China, November 26-30, 2019, Proceedings*, ser. Lecture Notes in Computer Science, R. Cheng, N. Mamoulis, Y. Sun, and X. Huang, Eds., vol. 11881. Springer, 2019, pp. 694–710. [Online]. Available: https://doi.org/10.1007/978-3-030-34223-4_44

[80] A. Montresor, F. D. Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Trans. Parallel Distributed Syst.*, vol. 24, no. 2, pp. 288–300, 2013. [Online]. Available: https://doi.org/10.1109/TPDS.2012.124

[81] Z. Lin, F. Zhang, X. Lin, W. Zhang, and Z. Tian, "Hierarchical core maintenance on large dynamic graphs," *Proc. VLDB Endow.*, vol. 14, no. 5, pp. 757–770, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p757-zhang.pdf

[82] Y. Shao, L. Chen, and B. Cui, "Efficient cohesive subgraphs detection in parallel," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C. E. Dyreson, F. Li, and M. T. Özsu, Eds. ACM, 2014, pp. 613–624. [Online]. Available: https://doi.org/10.1145/2588555.2593665

[83] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 812–823, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p812_jiawang_vldb2012.pdf

[84] F. Zhao and A. K. H. Tung, "Large scale cohesive subgraphs discovery for social network visual analysis," *Proc. VLDB Endow.*, vol. 6, no. 2, pp. 85–96, 2012. [Online]. Available: http://www.vldb.org/pvldb/vol6/p85-feng.pdf

[85] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proc. VLDB Endow.*, vol. 6, no. 6, pp. 433–444, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p433-sariyuce.pdf

[86] M. Ghaffari, S. Lattanzi, and S. Mitrovic, "Improved parallel algorithms for density-based network clustering," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 2201–2210. [Online]. Available: http://proceedings.mlr.press/v97/ghaffari19a.html

[87] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, "Scalable big graph processing in mapreduce," in *International Conference on Management of Data, SIGMOD*

2014, Snowbird, UT, USA, June 22-27, 2014, C. E. Dyreson, F. Li, and M. T. Özsu, Eds. ACM, 2014, pp. 827–838. [Online]. Available: https://doi.org/10.1145/2588555.2593661

[88] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu, "I/O efficient core graph decomposition at web scale," in *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 2016, pp. 133–144. [Online]. Available: https://doi.org/10.1109/ICDE.2016.7498235

[89] Y. Fang, R. Cheng, S. Luo, and J. Hu, "Effective community search for large attributed graphs," *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 1233–1244, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol9/p1233-fang.pdf

[90] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National security agency technical report*, vol. 16, pp. 3–1, 2008.

[91] B. Liu, F. Zhang, W. Zhang, X. Lin, and Y. Zhang, "Efficient community search with size constraint," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 97–108. [Online]. Available: https://doi.org/10.1109/ICDE51399.2021.00016

[92] K. Lakhotia, R. Kannan, V. K. Prasanna, and C. A. F. D. Rose, "RECEIPT: refine coarse-grained independent tasks for parallel tip decomposition of bipartite graphs," *Proc. VLDB Endow.*, vol. 14, no. 3, pp. 404–417, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol14/p404-lakhotia.pdf

[93] A. P. Mukherjee and S. Tirthapura, "Enumerating maximal bicliques from a large graph using mapreduce," *IEEE Trans. Serv. Comput.*, vol. 10, no. 5, pp. 771–784, 2017. [Online]. Available: https://doi.org/10.1109/TSC.2016.2523997

[94] A. Das and S. Tirthapura, "Incremental maintenance of maximal bicliques in a dynamic bipartite graph," *IEEE Trans. Multi Scale Comput. Syst.*, vol. 4, no. 3, pp. 231–242, 2018. [Online]. Available: https://doi.org/10.1109/TMSCS.2018.2802920

[95] Z. Ma, Y. Liu, Y. Hu, J. Yang, C. Liu, and H. Dai, "Efficient maintenance for maximal bicliques in bipartite graph streams," *World Wide Web*, pp. 1–21, 2021.

[96] A. Abidi, R. Zhou, L. Chen, and C. Liu, "Pivot-based maximal biclique enumeration," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, C. Bessiere, Ed. ijcai.org, 2020, pp. 3558–3564. [Online]. Available: https://doi.org/10.24963/ijcai.2020/492

[97] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. [Online]. Available: http://mitpress.mit.edu/books/introduction-algorithms

[98] F. Bi, L. Chang, X. Lin, and W. Zhang, "An optimal and progressive approach to online search of top-k influential communities," *Proc. VLDB Endow.*, vol. 11, no. 9, pp. 1056–1068, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1056-bi.pdf

[99] L. Chang, X. Lin, L. Qin, J. X. Yu, and W. Zhang, "Index-based optimal algorithms for computing steiner components with maximum connectivity," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 459–474. [Online]. Available: https://doi.org/10.1145/2723372.2746486

[100] R. Kannan and V. Vinay, *Analyzing the structure of large graphs*. Rheinische Friedrich-Wilhelms-Universität Bonn Bonn, 1999.

[101] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang *et al.*, "Distributed subgraph matching on timely dataflow," *Proceedings of the VLDB Endowment*, vol. 12, no. 10, pp. 1099–1112, 2019.