8-2024

Creating a Virtual Hierarchy From a Relational Database

Yucong Mo
Utah State University

CREATING A VIRTUAL HIERARCHY FROM A RELATIONAL DATABASE

by

Yucong Mo

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Curtis Dyreson, Ph.D. Major Professor	Shuhan Yuan, Ph.D. Committee Member
Steve Petruzza, Ph.D. Committee Member	D. Richard Cutler, Ph.D. Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2024

ABSTRACT

Creating a Virtual Hierarchy from a Relational Database

by

Yucong Mo, Master of Science

Utah State University, 2024

Major Professor: Curtis Dyreson, Ph.D.
Department: Computer Science

JOIN operations in relational databases are generally considered expensive. Although there are many strategies to improve JOIN's performance, most of these strategies are applied at runtime which can be costly. This thesis explores another approach to improve the overall efficiency of databases by revisiting the hierarchical model. We introduce a set of algorithms to impose virtual hierarchies on relational schemas. Our method shows full preservation of data integrity and relationships and constraints among tables. To accurately represent these relationships in hierarchies, we will need to manufacture additional nodes which leads to extra space costs. We also show that the algorithm minimize the cost to a modest level.

(44 pages)

PUBLIC ABSTRACT

Creating a Virtual Hierarchy from a Relational Database

Yucong Mo

In data management and modeling, the value of the hierarchical model is that it does not require expensive JOIN operations at runtime; once the hierarchy is built, the relationships among data are embedded in the tree-like hierarchical structure, and thus querying data could be much faster than using a relational database. Today most data is stored in relational databases, but if the data were stored in hierarchies, what would these hierarchies look like? And more importantly, would this transition lead to a more efficient database? This thesis explores these questions by introducing a set of algorithms to convert a relational schema to a hierarchy, that is, a tree-like structure. We show that our algorithms minimize space cost for creating tree nodes while preserving all the relationships and constraints in the schema. Finally, we evaluate the hierarchies on a native XML DBMS with a set of queries.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my major professor Dr. Curtis Dyreson, for his invaluable guidance throughout my Master's journey, and for his expertise in this research project. He was always available when I had questions about the thesis. His support and encouragement have helped me through many challenges. I could not imagine a better mentor and advisor.

I would also like to thank the rest of my committee members: Dr. Shuhan Yuan and Dr. Steve Petruzza. Their insightful feedback and constructive criticism have been indispensable to the development of this thesis.

Lastly, I express my heartfelt thanks to my family back in China. Their endless love and support have been my source of motivation and strength.

Yucong Mo

To my family

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This thesis explores the concept of virtual hierarchies in database management, focusing on the transformation of relational data into hierarchical models to enhance performance and data management efficiency. The introduction outlines the theoretical background, the research objectives, related work in the field, and the contributions made by this research.

The introduction section begins with a discussion on the relational model of databases, highlighting its widespread use and the inherent challenges associated with JOIN operations. It contrasts this with the hierarchical model, which, by embedding relationships directly within its structure, offers significant performance advantages for certain applications. This sets the stage for a deeper exploration into merging these two models to leverage the benefits of both.

1.1 Background

The relational model has been the center of database research ever since E. F. Codd introduced it about 50 years ago. One critical advantage of modeling the data in the relational model is that the data is logically separated into relations of tuples. Tuples in relations are related dynamically in a query by a `JOIN` or Cartesian product operation that combines relations. These operations are in general expensive to perform, even though there are many strategies to improve them [1–3].

The hierarchical model, on the other hand, statically embeds relationships in a tree-like structure. Each child datum in the tree is related to a single parent datum. Once the hierarchy is built, there is no need to dynamically combine or `JOIN` data at runtime to construct parent/child relationships because these relationships are embedded in the hierarchy. Transitively data items are connected through the hierarchy to ancestors and descendants. The hierarchical model is still very popular in applications that require very

high performance and availability such as banking, health care, and telecommunications, especially encoded as JSON or XML.

1.2 Research Objectives

Despite the advantages of the hierarchical model in specific applications, most data are still stored in relational databases. We are interested to see if we can improve the overall efficiency of databases, whether they are relational or non-relational, by creating virtual hierarchies directly from well established relation schemas. This conversion seeks to bridge the gap between the relational and hierarchical data models.

This thesis aims to develop and evaluate a comprehensive algorithm capable of converting relational database schemata. There are two primary objectives.

1. Algorithm Development:

   To design an algorithm to construct a virtual hierarchy from a relational schema. The virtual hierarchy accurately represents the relationships and constraints of a relational schema within a hierarchical model ensuring data integrity and query compatibility.

2. Performance Evaluation:

   To evaluate the performance of the virtual hierarchical structures generated by the algorithm within eXist-db, a native XML database system, in order to demonstrate the viability of the conversion in terms of query efficiency and data accessibility.

The research uses three sample relational database: Sakila, Northwind, and airportDB. These databases are used due to their complexity and widespread recognition within the database community. The evaluation will focus on specific aspects of the hierarchical model, including accuracy, completeness, and query performance.

1.3 Related Work

Previous work on data model transformation has mainly revolved around converting hierarchical databases to relational databases or storing hierarchical data directly in relational databases [4–7].

The first paper that attempted to create a hierarchical view for relational databases uses a *universal* relation for the data, essentially joining all the data into a single, large relation [8]. The universal relation approach suffers from massive space blow-up. The paper does not investigate the mapping in terms of data completeness and minimizing space cost. In contrast, our methods pay particular attention to full schema preservation and relationship conversion.

With the advent of XML in the 2000s, several methods for converting relational data to XML documents were proposed [9–11]. These approaches bridge the gap between the two models but still largely rely on complex relational engines for the transformation. Our algorithms introduce a simpler way to represent hierarchies without using additional systems.

1.4 Contribution

This thesis makes the following contributions.

- We propose an algorithm that correctly converts relational data into hierarchical data without losing relationships and constraint between tables.
- We show the algorithm minimizes the duplication of nodes when handling mapping from one-to-many and many-to-many relationships in hierarchy.
- We explore the nuances of data model transformations and offered insights into optimizing hierarchical structures for performance and storage efficiency.

CHAPTER 2

ALGORITHM AND EXAMPLES

In this chapter, we introduce the algorithm for creating a virtual hierarchy from a relational database. The chapter is organized into three main sections. First, we outline the design principles and goals of the algorithm. Next, we present the algorithm as a detailed guide for creating a virtual hierarchy from a relational schema, which is abstracted as a multi-graph. Finally we run the algorithm with example schema graph from a relational database.

2.1 Constraints for Input Relational Schema and Output Hierarchy

Creating a hierarchy from a relational schema requires several correct and meaningful constraints on the input relational schema and the output hierarchy.

- **Relational Abstraction and Terminology** : The first constraint we will enforce is the abstraction of the input relational schema. We model a schema as a multi-graph. Each node represents a relation from the schema. Inside the graph, the edges and their directions are used to denote foreign-key relationships between nodes. Namely, in the multi-gragh, if there is an edge coming from node A to node B, we read it as relation B borrows a foreign key from relation A, at the schema level. Since there may be more than one foreign key between tables, the graph can have multiple edges between nodes.

  Nodes in the graph can be categorized into four types in looking at foreign keys:

  1. No foreign keys going into or out of: These are independent entities without any foreign key relationships. They are often standalone tables, and these nodes mean the graph does not have to be connected.
  2. Source Nodes: Entities that have only outgoing foreign key references. These typically represent associative entities in relationships, capturing many-to-many

relationships in relational databases. We call this source node and it occurs with any table representing a relationship type (in the ER model). Relationship type tables never have their keys borrowed as foreign keys.

3. Sink Nodes: Entities with only incoming foreign keys, representing strong entity types that are often primary tables in the database schema.
4. Nodes with both incoming and outgoing edges. These are also always (weak or strong) entity types, multivalued attribute types, or subclasses in the ER model.

Cycles in the graph are possible when the relation borrows a key from itself, but would be rare in well-designed schemas in other situations. We remove reflexive foreign keys from the graph to create the multi-graph.

- **Completeness and Accuracy** : The second and most important constraint is completeness and accuracy. The inferred hierarchy should preserve all nodes, and correctly record edge information that indicates different types of relationships among relations, including one-to-one, one-to-many, and many-to-many.

- **Dummy Nodes in Hierarchy** : In a hierarchy, a child node exists only if its parent does. In scenarios where the database design allows null values for foreign keys, a supposed child node may not have an associated parent node. To maintain hierarchical integrity, we need to create *dummy nodes* for those supposed child nodes. For example, in a `Customer` and `Order` scenario, and `CustomerID` is the parent of `Order`, if an `Order` record has a null `CustomerID`, but the hierarchical model requires every `Order` to have a `Customer` parent, then a dummy `Customer` node must be generated to maintain structural integrity in the hierarchy. Suppose nodes of type A are parent nodes of nodes of type B. We can calculate the exact number of dummy nodes needed by subtracting the number of distinct non-null foreign key values in B (which point to A) from the total number of A nodes. This calculation helps identify how many A-type records are actually referenced by B-type records. If all B-type nodes reference an A-type node, no dummy nodes are needed.

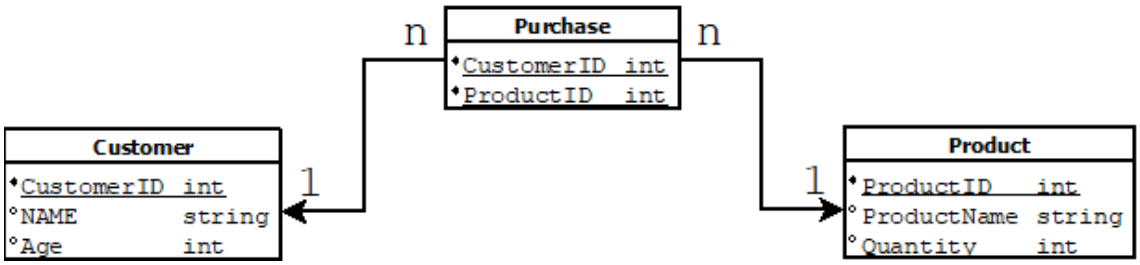

Fig. 2.1: Node Duplicates Scenario

- **Node Duplication in Hierarchy** : Node duplicates in a hierarchy represent a duplication of a relation. Sometimes they are unavoidable in order to ensure the completeness of many to many relationships. For example, in Figure 2.1, if `Customer (type A nodes)` and `Product (type B nodes)` are connected through a `Purchase (type C nodes)` table where each `Customer` can buy multiple `Products` and each `Product` can be bought by multiple `Customers`, each product purchased by multiple customers needs to appear under each customer in the hierarchy. If `Product 1` is bought by both `Customer A` and `Customer B`, then `Product 1` will need to be duplicated in the hierarchy under both customers. We can also calculate the number of duplicates with: `total number of B to A pointers - number of distinct B to A pointers`. We will also show our algorithm introduces the construction of edge nodes and relationship type of nodes (source nodes) which help minimizing the manufacturing of these duplicates.

- **Transferable Data Format**: The resulting hierarchy should be in JSON or XML format that is widely used in practice.

2.2 Conversion Algorithm

In this section we present the main steps of conversion algorithm that creates the virtual hierarchy from a relational schema, followed by detailed examples on its use. The idea of minimizing the manufacturing of dummy nodes and duplicate nodes are applied in the algorithm.

Before introducing conversion algorithm, we first need to map input relational schema to a multi-graph. which leads to Schema Graph Mapping Algorithm, as given in algorithm 1.

2.2.1 Schema Graph Mapping Algorithm

The Schema Graph Mapping Algorithm is designed to transform a relational database schema into a graph representation, where nodes represent relations and directed edges symbolize foreign key relationships among these relations. The six main steps of the algorithm are given below.

1. **Obtaining the Schema**: Depending on the database management system (DBMS) in use, relational schema can be acquired through various methods.

    - **Data Definition Language (DDL) Scripts:** DDL scripts provide a textual representation of the schema through SQL statements that define relations, relationships, and constraints. These scripts can be generated by most DBMSs as part of their schema export functionality.
    - **Database Metadata Queries**: The schema can also be generated by direct querying on database metadata. We can achieve this by executing specific SQL queries or API calls that retrieve schema details from the DBMS's systems.
    - **Manual Extraction**: In the case where automated tools are not available, the schema may also be derived manually. This method, while time consuming and requires careful maintenance to ensure accuracy, allows for custom documentation and flexible schema representation.

2. **Schema Parsing**: After the schema has been obtained the algorithm parses the schema, creating nodes for each relation and identifying primary and foreign key constraints, as well as total and partial participation relationships among relations if an entity-relationship (ER) diagram is available or the cardinality of foreign key relationships is computed through queries.

3. **Identifying Nodes with Null Foreign Key Value**: Dummy nodes typically arise when a child node in a hierarchy doesn't have a corresponding parent node due to null foreign keys. We find all tuple records with null foreign key value and deal and pay special attention to them when constructing the hierarchy later.

4. **Node Creation**: A node is created in the graph for each relation. This node includes detailed information about the relation's structure and constraints.

5. **Edge Creation**: A directed edge is added between nodes for each foreign key relationship. Each edge is directed from the child relation (foreign key holder) to the parent relation (key source). As mentioned earlier, multi edges are allowed between nodes to denote more than one foreign key between nodes.

   We introduce weights in graph to represent two cases, one where tuple records of the database are available, and second where we only have the schema and no access to the actual population of tuples. In the former case, edges with same directions are merged and a weight, i.e, the number of edges used for merging, is assigned to the new edge. The latter case is common in practice, and weights are considered 0.

6. **Handling Associative Relations**: Relations identified as associative (junction relations) for many-to-many relationships are represented by nodes with only outgoing edges, i.e. source nodes.

7. **Output Generation**: The final multi-graph is then available for visualization, analysis, or further processing to transform to a hierarchy. This graphical representation can be used to identify root relations (without incoming edges), leaf relations (without outgoing edges), and the overall structure and dependencies within the schema.

Figure 2.2 shows the output multi-graph after running the schema mapping algorithm on a simple relational schema. Figure 2.3 shows the output multi-graph on a more complex schema.

Algorithm 1: Generate Weighted Multi-Graph from Relational Schema

Input: Relational database schema S
Output: Weighted Directed Multi-Graph G with edges weighted according to relationship type and participation

`/* Initialization */`
Create an empty Directed Multi-Graph G
Initialize an empty dictionary $EdgeWeights$ to store the weights of edges based on relationship type and participation
`/* Schema Parsing */`
Parse the schema, identifying primary and foreign key constraints
Determine the classification of each relation (sink, source, or intermediate nodes based on foreign key relationships)
`/* Node Creation and Classification */`
for *each relation R in schema S* **do**
    Create a node in G for R
    if *R has no outgoing foreign keys and no incoming foreign keys* **then**
        Mark R as a standalone node
    else
        if *R has only outgoing foreign keys* **then**
            Mark R as a source node
        else
            if *R has only incoming foreign keys* **then**
                Mark R as a sink node
            else
                Mark R as a regular node
            end
        end
    end
end
`/* Edge Creation and Weight Assignment */`
for *each foreign key relationship FK between relations R_i and R_j in S* **do**
    Add a directed edge from the child relation R_i (foreign key holder) to the parent relation R_j (key source)
    if *tuple records are available* **then**
        Merge edges with the same direction between R_i and R_j, assigning a weight equal to the number of merged edges
    else
        Assign a default weight of 0 to each edge as a placeholder, indicating the absence of tuple records
    end
end
`/* Post-processing to Finalize the Multi-Graph */`
for *each edge (R_i, R_j) in G* **do**
    Label edge (R_i, R_j) with its weight from $EdgeWeights[(R_i, R_j)]$
end
return G

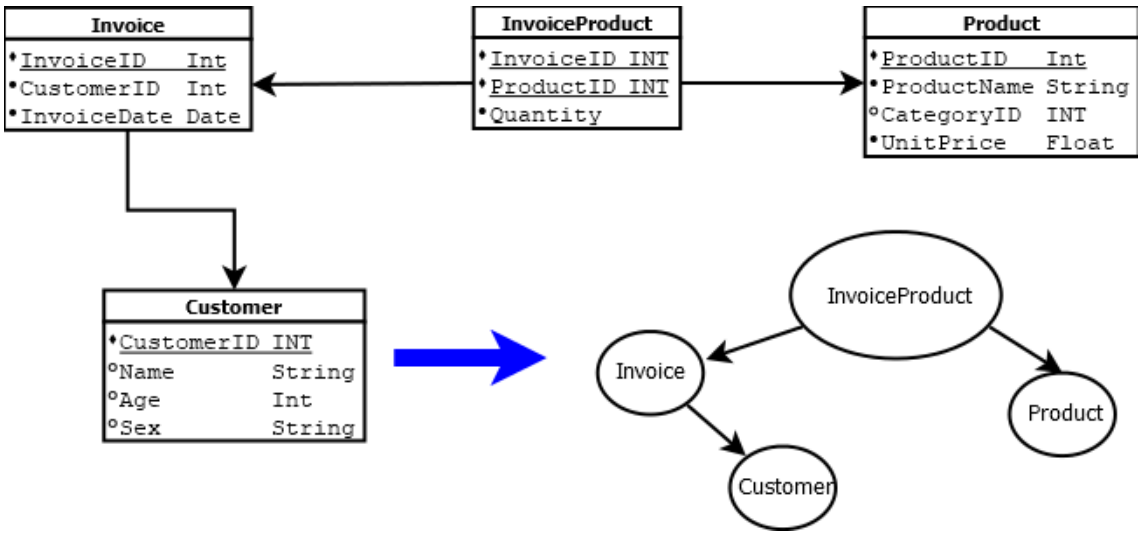

Fig. 2.2: Simple Schema Mapping to a multi-graph

2.2.2 Conversion Algorithm

The conversion algorithm given in Algorithm 2 includes a helper Algorithm 3. This helper algorithm is for handling the mapping of many-to-many relationship.

The design of Algorithm 2 is based on three core principles aimed at ensuring a faithful transformation at low space cost.

1. **Hierarchical Clarity**: By converting the multi-graph into a hierarchy, the algorithm aims to present the relational schema in a more intuitive and comprehensible format.

2. **Preservation of Relationships**: Critical to the algorithm's design is the retention of the schema's intrinsic relationship details, including complex n-nary relationships, within the hierarchy.

3. **Adaptability**: The algorithm is designed to handle both connected and disjoint graphs, ensuring robustness and versatility in varying schema complexities.

4. **Minimizing Dummy Nodes and Duplicate Nodes**: The algorithm is designed to reduce space cost of dummy nodes duplicate nodes in the output hierarchy.

If the input multi-graph given by algorithm 1is weighted, we have access to tuple records in original database. In this case the conversion has the following eight steps:

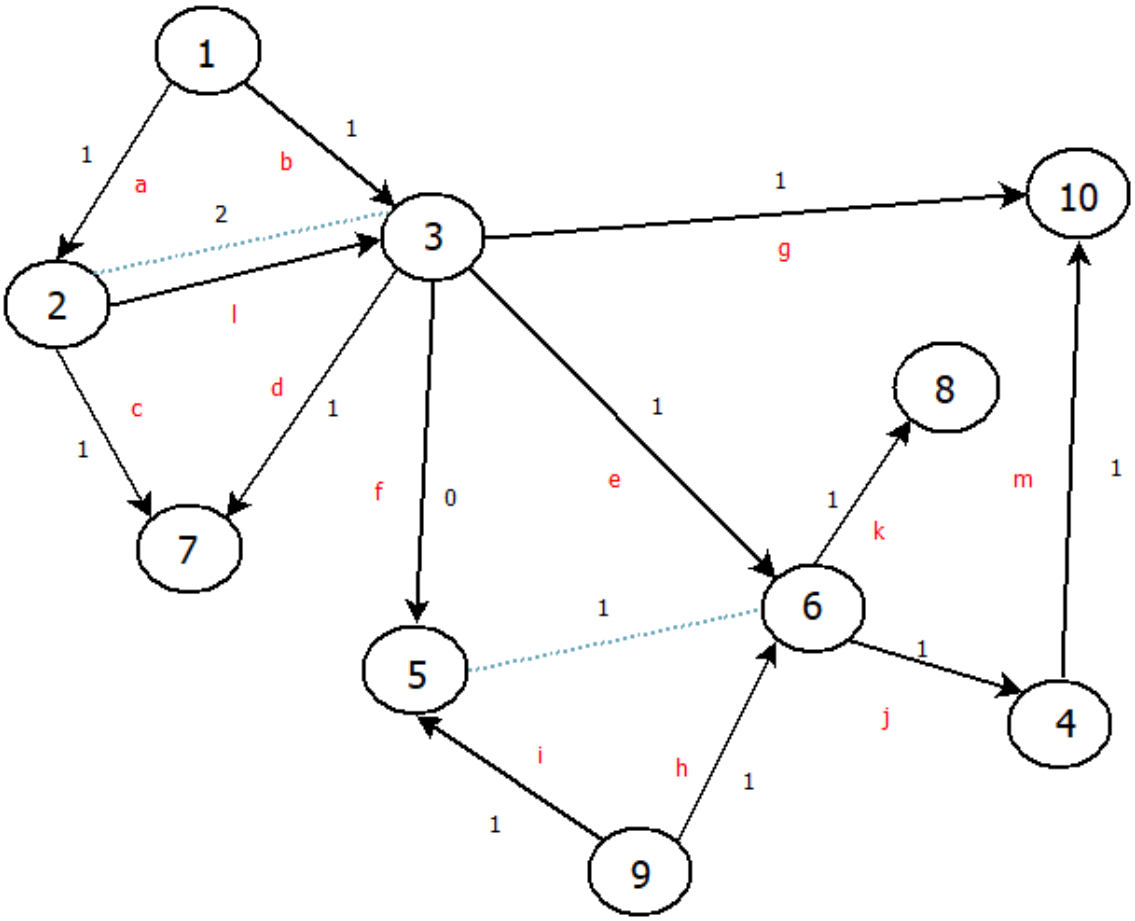

Fig. 2.3: A more complex Schema Mapped to a multi-graph

1. **Definition and Removal of Source Nodes:** Initially, nodes in the multi-graph that represent n-nary relationship types (source nodes), denoted as P, are identified. These nodes, also called source nodes, are characterized by having only outgoing edges, signifying an in-degree of zero. Subsequent to their identification, these nodes are temporarily removed from the multi-graph. This is to simplify the initial construction phase of the hierarchy by focusing on the mapping of one-to-many and one-to-one relationships first. Source nodes will be reintegrated to hierarchy later.

   (Note: The removal of P nodes may result in a disjoint or connected graph, necessitating separate handling strategies for each scenario.)

2. **Longest Path Identification**:

Following the simplification of the multi-graph, the algorithm then identifies the longest weighted path(s) within the graph. In a rare case where there are multiple longest paths, any one can be selected. The chosen path is used to form the backbone of the resulting hierarchy, T. The reason for choosing the longest path is we want to capture most relations for initial hierarchy construction and to *branch out* from nodes in the path. The longest path is the best candidate for this regard and it has the same runtime as finding the shortest path.

3. **Backward Path Traversal and Hierarchy Construction**:

   The traversal begins from the last node of the chosen path, designated as the root of T. The last node from the path is guaranteed to only have incoming edges, which makes it an ideal root node for our hierarchy. Each other node encountered during this backward traversal, termed I, is then integrated into T through rigid rules detailed as follows:

   - Nodes E directly connected to I are made children of I in T, with a an edge node, Q, instantiated beneath E to document this connection.
   - For each edge in multi-graph that has been visited, we remove that edge, and if the removal causes the node degree drops to 0, we remove the node as well.

   By doing so, every `node` that is 1 degree away from I is added to the hierarchy. If the graph is not exhausted after backtracking, we run the algorithm again on the reduced graph. We chose to only add one node at a time instead of a chain of nodes(for example, nodes that are not in the longest path and are n degrees away from I) because we want to minimize the destruction of relationships between nodes while removing them from multi-graph. As adding nodes to hierarchy requires removing nodes from multi-graph, the removal of a chain of nodes may commit greater semantics error to the original relational schema, which makes it harder to trace when completing our hierarchy.

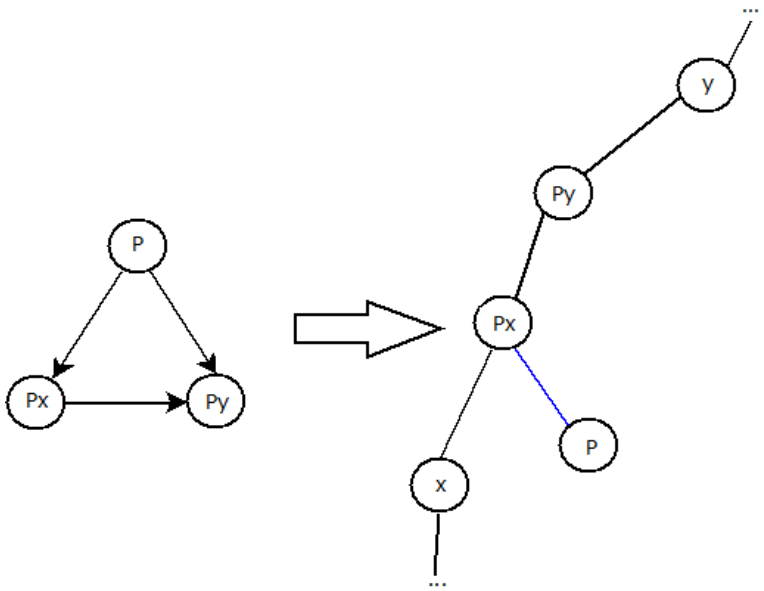

Fig. 2.4: Edge Node as Leaf Node for Recording M-N Relationship

4. **Reintegration of Relationship Type Nodes**:

   This step adds back the relationship type of nodes P removed in step 1. First we identify the nodes that P connected to in the original multi-graph, termed P_i. Notice after previous steps all P_i are now added to T. Next we find the P_i of deepest level in T, termed P_x, and make P a child node of P_x. We make P as leaf node because it shows the relationships between P's parent, termed U, and U's parent, termed V, and in this case, U and V are in a many-to-many relationship. We observe the pattern for creating leaf nodes to record relationships between concrete nodes(nodes that represent relations) rather simple and easy to use: by going only two levels up from itself to show relations, and we continue to apply this pattern when recording other types of relationship, as detailed in further steps. Figure 2.4 shows an example transformation for a simple relationship.

   For all the other P_is, if there exist a P_y, such that P_y does not share a parent-child(P-C) relationship with P_x, we promote the tree rooted as P_x to be a sub-tree of the lowest common ancestor(LCA) of P_y and P_x, termed P_z(Notice P_z can be P_x or P_y) . For the original parent node that P_x connected to, and nodes along the path before

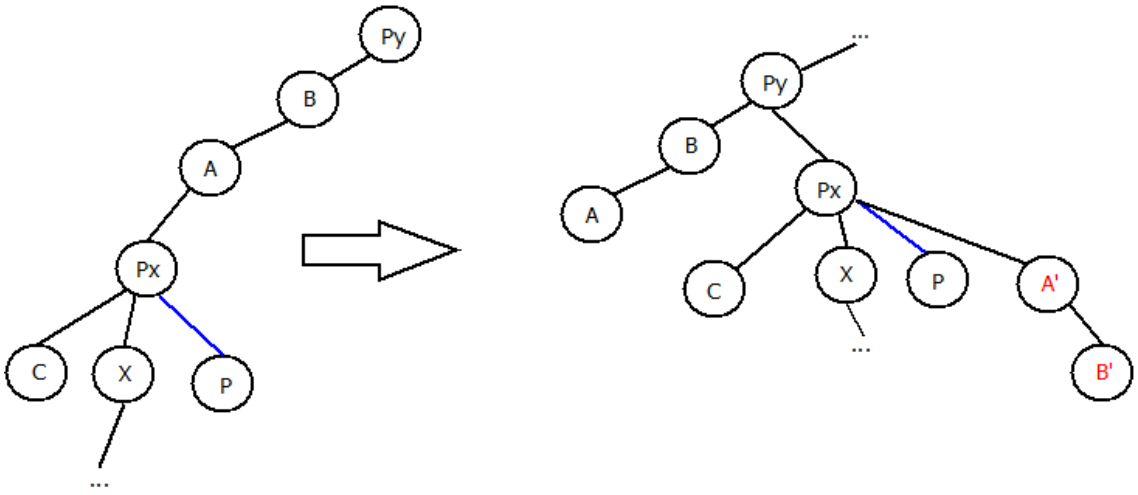

Fig. 2.5: Promoting P_x to the LCA of P_x and P_y

reaching P_y, create duplicates under P_x after the promotion. The primary reason for this promotion is to maintain a direct many-to-many connection of nodes in the hierarchy, because otherwise when P_z needs to visit nodes under P_x it has to travel through the nodes between P_x and P_z, which is not reflected in the original multi-graph schema. Another advantage of this promotion is we shorten the tree height. Creating duplicates are for respecting original schema and former hierarchy, as shown in Figure 2.5: After the promotion, if node B needs to visit node C or X, it will have to go through node A and P_x as required before the promotion, and this is accurately recorded in our hierarchy because now node B can travel from its duplicate, B', going upwards to visit X, which yields the same path in original multi-graph and previous hierarchy. Finally in the cases where P_x and P_y are in different sub-tree, we promote both trees rooted as P_x and P_y to be a sub-tree of P_z. This is a result from removal of P in step 1, yielding disjoint sub-graphs.

If P_y and P_x are in a P-C relationship, and they are both total participation or partial participation we put the one with more join selectivity as the parent. If one of them is different from the other in participation type, we put the one that is total participation at the top.

When considering which node should be put on top, total vs. partial participation

and join cardinality are most important factors. This is because the former indicates which entity(relation) is more *central* or *essential* in the relationship and the latter informs which the ones with more data has a higher degree of connections from one side. Therefore entities in total participation and greater join cardinality are should be placed higher in the hierarchy. There other factors to be considered such as semantic meanings and query patterns and use cases, but they are more difficult to infer from input schema.

5. **Edge Nodes for Minimizing Node Duplicates**:

   As discussed previously, the placement of relational type nodes inspires the introduction of edge nodes. Edge nodes are created as leaf nodes for the node with deeper level in T, and they exist only for tracking relationship between exact two nodes. Each edge from multi-graph is mapped to exact 1 edge node in T. This means at worst scenario we miss documenting a relationship rather than recording existing ones again. There are two case where duplicates are necessary: one is when a node in multi-graph has more than 2 out-degrees, whether it is a concrete node or relational type node, this is because a hierarchy forbids a node to have more than one parent; the second case is in promotion placement of P_x copying chain of nodes is unavoidable as showed in previous step. For the first case, edge node helps setting the upper bound of node duplicates to exactly the node's out-going degree. It ensures self-copying are only needed when it is pointing to more than one node, and the number of copies we need is less than or equal $out - degree - 1$ by the construction of these edge nodes, this is because otherwise there will be edge nodes recording non-existing relationship in the hierarchy.

6. **Iterative Process and Finalization**:

   The outlined steps above are reiterated until the original graph is fully exhausted, returning the hierarchy T.

The algorithm ends here if the input multi-graph given by algorithm 1 is unweighted. We have created a hierarchy where each node represents a type of data rather than actual tuple records in relational database.

If the input multi-graph is weighted, we expand nodes in hierarchy with tuple records and execute the following steps:

7. **Creating and minimizing Dummy Nodes**: For nodes of type B and their parent nodes of type A in T, if a B node has a null foreign key value to its supposed parent node of an A node, then a dummy A has to be created to record this relationship in T.

   To minimize these dummy nodes, instead of creating one dummy node for every node with null foreign key value, termed L, we create a generic dummy node for L, and link L nodes of same type to their respective dummy nodes, as shown in Figure 2.6, effectively reducing creations of dummy nodes.

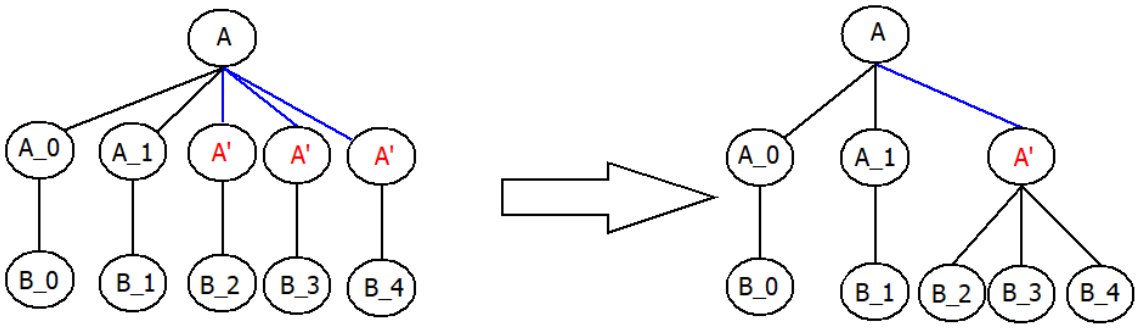

Fig. 2.6: Minimizing Dummy Nodes

8. **Creating and minimizing Duplicate Nodes**: Suppose again nodes of type B and their parent nodes of type A in T. If their relationship are introduced by source node C, i.e, M-N relationship types. In order to represent C every association of an A with a B has to be present in the hierarchy (with a C child). So if nodes 1 and 2 of type A are related to nodes 3 and 4 of type B, then the B nodes have to be duplicated

since 3 and 4 must be a child of 1, and 3 and 4 must also be a child of 2. Moreover, the entire tree rooted at 3 and 4 must be duplicated to preserve data and relational integrity.

To minimize the impacts of creating duplicate nodes to T, we push these nodes as far down the tree as possible. By doing so, the duplication is restricted to the deeper parts of the hierarchy where it is less disruptive and more contextually appropriate. Moreover, it keeps the upper levels of the hierarchy clean and minimize redundancy at these levels can improve the performance of operations that involve traversing or manipulating the hierarchy. Figure 2.7 shows this process. Notice the duplicated C nodes record relationships of duplicated B_3 and B_4 with A_2, instead of the nodes 2 level above C.

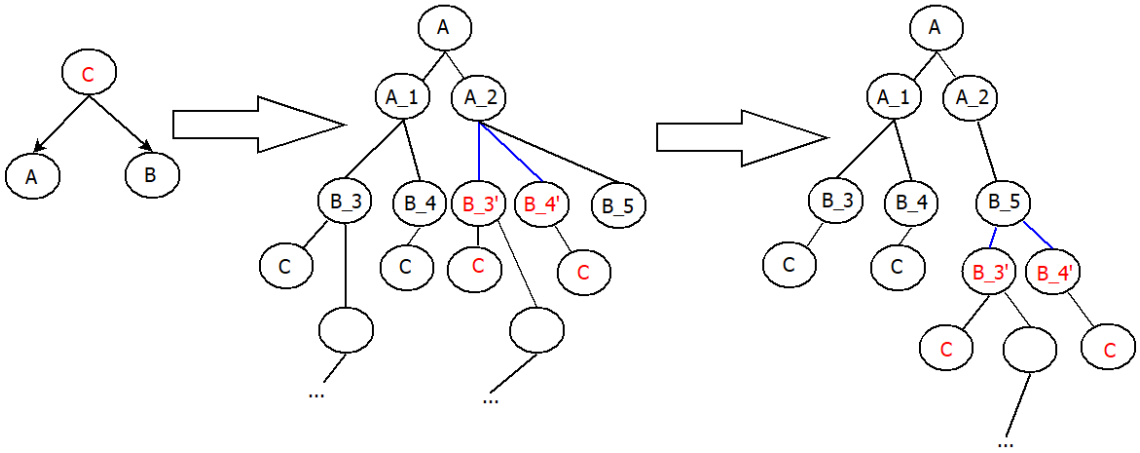

Fig. 2.7: Minimizing Duplicate Nodes

Figure 2.8 shows a virtual hierarchy after running Algorithm 2 on Figure 2.3. It records all the information in original schema accurately. The nodes marked red are duplicated and edges marked blue are edge nodes.

Algorithm 2: Creating a Virtual Hierarchy from Relation Multi-Graph

Input: Weighted Directed Acyclic Graph multi-graph representing a relational schema

Output: A Virtual Hierarchy T representing the schema

`/* Initialization */`

Initialize an empty hierarchy T

Identify and set aside n-nary relationship nodes, termed P, within the multi-graph

`/* Simplify multi-graph by removing source nodes first */`

Remove P nodes from multi-graph, noting their connections for later reintegration

(Note: This may result in disjoint or connected graphs)

if *Step 4 yields connected Graphs* **then**

  `/* Identify the foundational structure of` T `*/`

  Find the weighted longest path in multi-graph Select the path's end node as T's root

  `/* Integrate nodes into` T `*/`

  for *each node I in the longest path, starting from the root, going backwards* **do**

    add I to T.

    for *every node E that connects to I* **do**

      if *E is I's parent and E is in T* **then**

        skip E.

      end

      add E as a child node of I, create an edge node Q under E to record the relationship of E and I.

    end

  end

  Remove all the edges that were added to T.

  Remove nodes whose degree become 0.

  `/* Reintegrate n-nary relationship nodes */`

  for *each n-nary relationship node P* **do**

    execute algorithm 3.

  end

  if *multi-graph is not empty* **then**

    run step 6 to step 27

  end

  for *every graph G in disjoint Graphs* **do**

    run steps 6 to step 27.

  end

else

  Merge disjoint graphs by number of non-relationship type of nodes by descending order.

end

return *virtual hierarchy T*

Algorithm 3: Relationship Type Nodes Integration and Optimization Algorithm

Input: Source Node P, Virtual Hierarchy T, multi-graph from Algorithm 2
Output: Optimized Virtual Hierarchy T

(This is a helper algorithm for Algorithm 2.)
Identify the list of nodes that P connects to in the multi-graph, termed P_i.
Find the P_i of **deepest** level in T, termed P_x, and add P as a child node of P_x.
if *No other P_i is in a parent-child relationship with P_x* **then**
    Find P_i with the **highest** level in T, termed P_y
    Promote P_x to be a child of P_y and P_x's lowest common ancestor, termed, P_z.
    Move the sub-tree of P_z that includes P under P_x.
end
`/* Creating and minimizing Dummy Nodes */`
for *each sink node with a missing parent due to null foreign keys* **do**
    Create a generic dummy node for each type where the foreign key is null.
    Link all nodes of the same type with null foreign keys to their respective dummy node.
end
`/* Creating and minimizing Duplicate Nodes */`
for *each source node P in T* **do**
    Identify two types of nodes that P references to(by going two levels up from itself), termed A and B, with A being the parent of B.
    for *each B node that associates with an A node through a P* **do**
        Duplicate tree rooted under node B .
        Modified node P in duplicate tree to record relationship of A and A.
        Set duplicate trees the children of an A node of deepest level in T.
    end
end
return *optimized virtual hierarchy T*

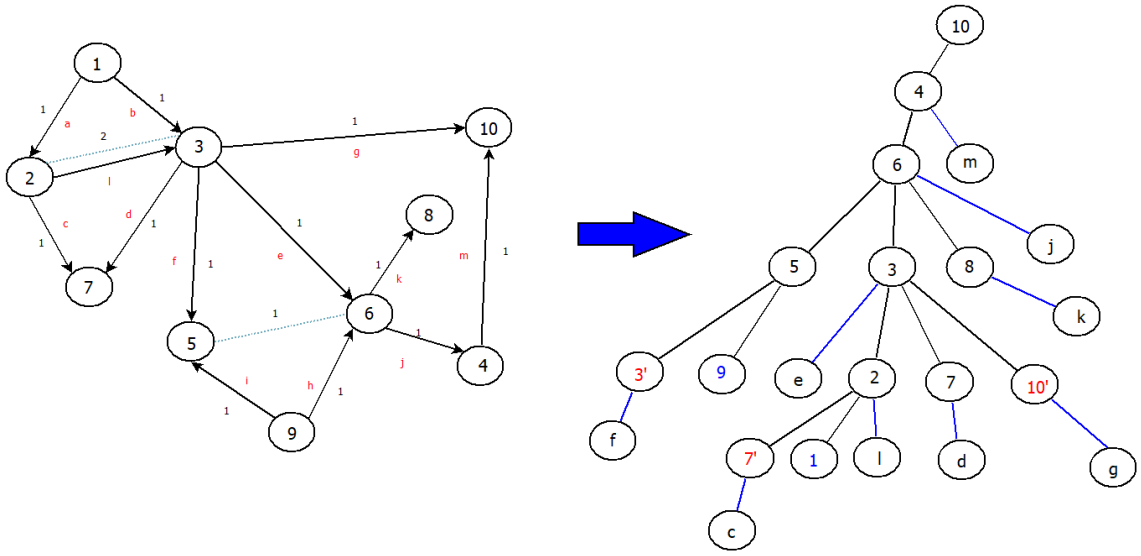

Fig. 2.8: Algorithm 2 output for the Complex Schema

2.3 Example Walk Through

In this section we will present a detailed walk through on executing Algorithm 2 on Figure 2.3.

Step 1. Remove Relationship Type of Nodes To initialize, we remove nodes that only has out-degrees. These nodes are called relationship type of nodes and they serve as connecting nodes for many-to-many relationship among other nodes. In our example, node 1 and node 9 are these type of nodes.They are temporarily removed to simplify our graph structure. If this removal results in disjoint graphs, at the end we will merge these graphs after processing them separately.

Step 2. Identify Weighted Longest Path We need to find the weighted longest path in this multi-graph. If there are multiple candidates, which is a rare case, we can choose any one of them to go on with our algorithm. If multiple longest paths exist after first iteration, we select the one that includes the most nodes in our hierarchy T. In Figure 2.3, $2-3-6-4-10$ is longest path. We set node 10 as the root node.

Step 3. Backtracking from Root Node Next we go backwards from root node and process each node in the path. For each node, we add them to hierarchy T if it

is not already in T, and set all other nodes that it connects to as its child node. To record the relationship of existing nodes and newly added node in T, an edge node is created as a leaf node under the newly added node. Since a newly added node is created under an existing node, the edge node effectively documents and shows their relationships by going at most two levels up from itself. For all the nodes that are added to T, remove the edges that are used to traverse to them. If the nodes' degree become 0, we remove those nodes.

In our multi-graph, node 4 is connected to 6 and 10. Since 10 is already in T, and that its relation with 4 would be recorded by edge node right after 4 is added to T, we skip 10 and set the other nodes as a child for 4, and we also create corresponding edge nodes. We then go backwards to node 6 again and apply the logic: Add 6 to T, create 3 and 8 as children nodes for 6, and spawn edge nodes recording relationships. Notice nodes 1 and 9 were removed in step 1. When it's time to examine node 3, it is showed to be connected to node 10, which is already added to T, but that addition was contributed by edge m from node 4. In order to record the relationship of node 3 and 10 given by edge g, we duplicate 10 as a child for 3, and spawn an edge node for them.

Figure 2.9 shows the result of running step 3.

Step 4. Iterating Graphs and Edge Nodes After processing every node in the longest path in step 3, if there still nodes existing in multi-graph, we run step 2 and step 3 again until every node and edge are visited. We create edge nodes for every new relations added to T.

The result of step 4 is showed in Figure 2.10

Step 5. Reintegrating Relationship Type Nodes Now we add back the relationship type nodes that were removed in step 1. Relationship type nodes introduce many-to-many relationships to the nodes they point to only.

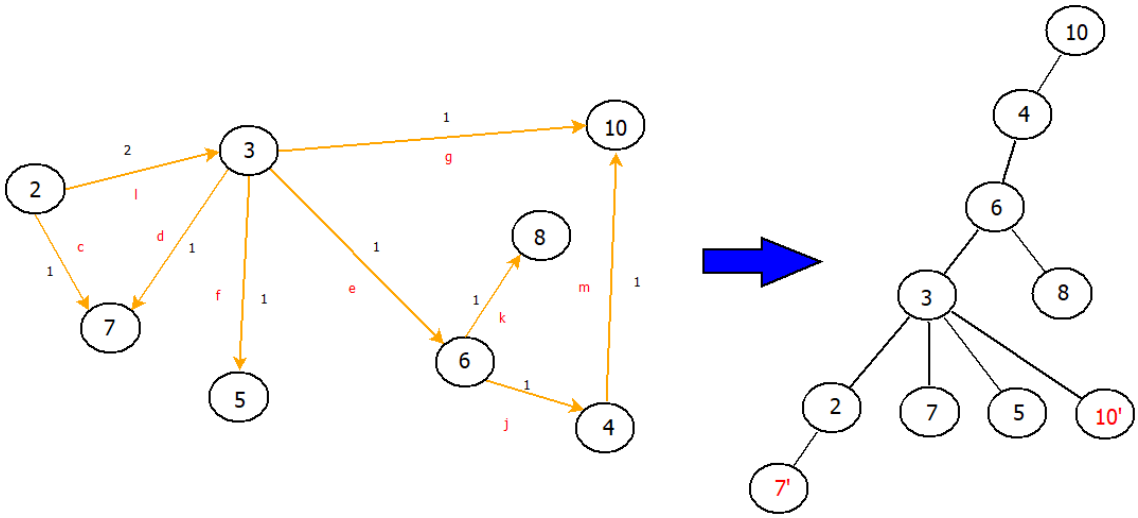

Fig. 2.9: Executing Step 3

In our example node 9 points to 5 and 6. We first add node 9 to T under the deeper level between 5 and 6, in this case, node 5. Next we notice 5 and 6 do not have direct parent-child relationship, in our case this means node 6 is at least 2 levels above node 5. To effectively reduce the height of T (tree) without losing correct interpretation of node relationships, we promote the tree rooted as node 5 as a new sub-tree under node 6. Now node 9 is recording the relationship of 5 and 6 in the same fashion as an edge node: it shows what is happening between the nodes two levels above itself. For node 5's original parent, node 3, we create a duplicate of it under the new position of node 5.

We continue to process node 1 the same way. Since Node 2 and 3 are already in a parent-child relationship in T, we do not need to adjust the structure.

The algorithm exists here if there is no disjoint graphs after step 1. Figure 2.8 shows the final result of T.

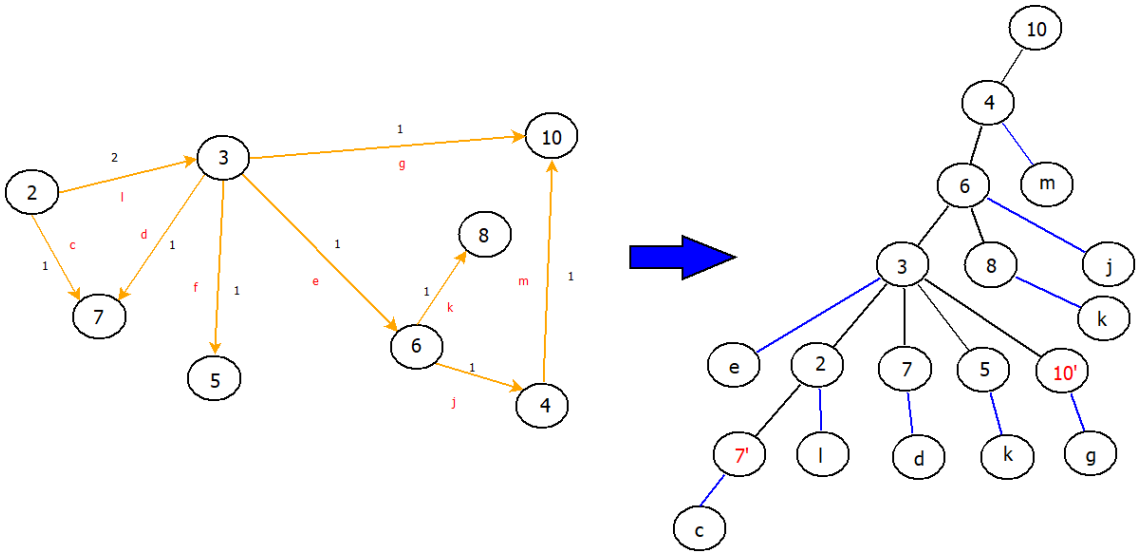

Fig. 2.10: Executing Step 4

CHAPTER 3

IMPLEMENTAION

This chapter describes the implementation details for the virtual hierarchy conversion system. The overall workflow of the program is shown in Figure 3.1. Our program first uses a schema parser to extract the relational schema from a RDBMS with SQL queries and its metadata. Then Algorithm 1 comes in and maps the schema to a directed acyclic graph. Next Algorithm 2 takes in the multi-graph and outputs our virtual hierarchy. Finally we put the virtual hierarchy into eXistDB, an open source XML databases (hierarchical) for experiments.

3.1 Schema Parser

Many relational database management system provides tools for extracting schema from a database. The output of these tools is usually an SQL script to recreate the database. The script accurately records all the tables and foreign key constraints between the tables. The downside is it does not directly provide the relationships between these tables, which is essential for our Schema Mapping Problem. For this, we need to develop our own parser to document the tables as well as relationships between tables. MySQL provides a reverse database engineering option and it allows us to view the Enhanced Entity Relationship diagram generated from an example database called Sakila in MySQL, as shown in Figure 3.2. This is helpful for checking the correctness of parsing and mapping of multi-graph.

- **Table Identification** : The parser scans the SQL script to find every `CREATE TABLE` statement and records these tables as nodes in a list.
- **Relationship Mapping**: Next the parser identifies the `FOREIGN KEY` constraints embedded within the table definitions. These constraints are the edges of our multi-graph, representing the pathways and connections between the tables. Each foreign key relationship adds a directed edge in our multi-graph.

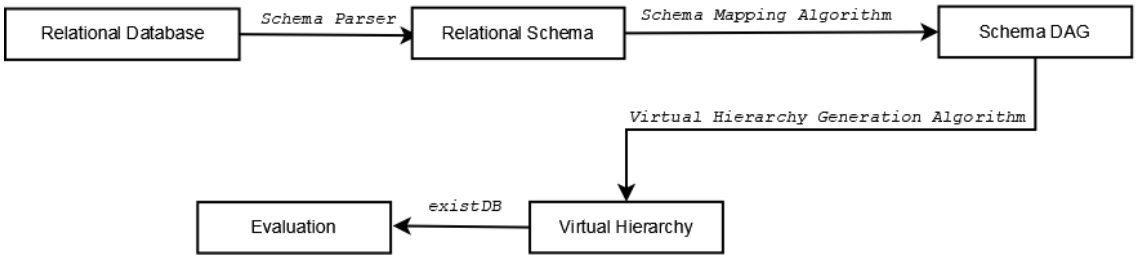

Fig. 3.1: Program workflow

The output of the parser is a structured representation of the database schema in a Python dictionary, where each table name serves as a key, and its values are the names of tables to which it has foreign key relationships. The following are sample outputs from our parser:

```
Tables and Their Relationships:

actor
    Primary Key: actor_id
    No foreign key relationships.

country
    Primary Key: country_id
    No foreign key relationships.

city
    Primary Key: city_id
    Foreign Keys:
        country_id references country(country_id)

address
    Primary Key: address_id
    Foreign Keys:
        city_id references city(city_id)

store
    Primary Key: store_id
    Foreign Keys:
        manager_staff_id references staff(staff_id)
        address_id references address(address_id)
```

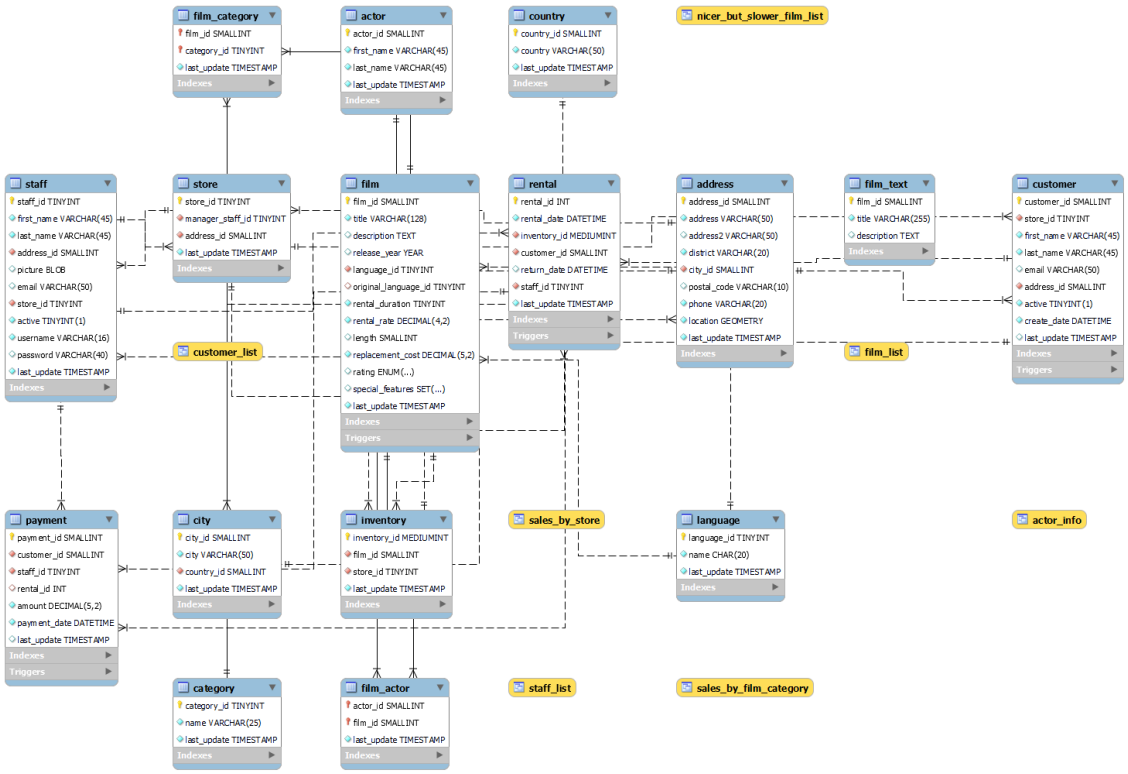

Fig. 3.2: E-R Diagram for Sakila Database

3.2 Schema Mapping Algorithm

Once the schema is parsed and structured, Algorithm 1 transforms this representation into a multi-graph. This process iterates through the dictionary output from the parser, creating nodes for each table and directed edges for each relationship. The creation of the multi-graph is crucial as it lays the groundwork for the next step in our system workflow.

The key methods in this algorithm are:

- **calculate_edge_weight Function** : This function assigns weights to edges based on the relationship type and participation level. The weights reflect the complexity and significance of the relationships, adhering to the algorithm's specification.
- **generate_graph Function** : This is the core function that iterates through the defined schema, initializing nodes for each table and creating directed edges with weights for each relationship. The resulting multi-graph is a Python dictionary where

each table (node) maps to a list of dictionaries representing its outgoing relationships (edges) and their weights.

3.3 Conversion Algorithm

Next, Algorithm 2 takes in the output multi-graph then produces a virtual hierarchy. The key implementation details include the findings of longest path, back-tracking depth first search approach from the root node, handling disjoint graphs after removal of relationship type of nodes, and reintegration of relationship type of nodes, which is given in Algorithm 3.

- **Longest Path Identification:** A foundational step in Algorithm 2 involves identifying the longest path within the multi-graph. This process determines the hierarchy's backbone, providing a structured approach to laying out the primary lineage of nodes. By pinpointing the longest path, we establish a starting point for the hierarchy, ensuring that the most substantial relationships are prioritized in the hierarchical construction.

- **Depth-First Search with Back-Tracking:** Following the identification of the longest path, Algorithm 2 employs a depth-first search (DFS) strategy, along with back-tracking, to traverse from the identified root node. This approach enforces a thorough exploration of the graph, and it allows a systematic construction of the hierarchy by progressively building out branches based on the relationships between tables.

- **Handling Disjoint Graphs:** The removal of relationship type nodes, such as those representing many-to-many relationships, can result in disjoint graphs. Algorithm 2 addresses this challenge by separately processing each disjoint graph to construct sub-hierarchies, which are then integrated into the main hierarchy. This ensures that all relationships, even those temporarily abstracted away, are accurately represented in the final hierarchical structure.

- **Reintegration of Relationship Type Nodes:** To reintegrate of the relationship type nodes previously set aside, we introduced Algorithm 3, which involves strategically placing these nodes back into the hierarchy, ensuring that the many-to-many relationships they represent are accurately depicted. The reintegration is guided by the hierarchical positions of related entities, maintaining the logical integrity and relational context of the original schema.

The following is the output of the truncated XML file for figure 3.2.

```
...
<database name="Sakila">
    <table name="payment">
        <parentTable>null</parentTable>
        <childTable name="staff">
            <duplicate>yes</duplicate>
            <relationship>one-to-many</relationship>
                <invertedEdge>no</invertedEdge>
        </childTable>
        <childTable name="customer">
            <duplicate>no</duplicate>
            <relationship>one-to-many</relationship>
            <invertedEdge>no</invertedEdge>
        </childTable>
    </table>
    <table name="address">
        <parentTable>customer</parentTable>
        <childTable name="city">
            <duplicate>no</duplicate>
            <relationship>one-to-one</relationship>
            <invertedEdge>no</invertedEdge>
        </childTable>
    </table>
    <table name="country">
        <parentTable>address</parentTable>
        <childTable name="city">
            <duplicate>no</duplicate>
            <relationship>one-to-many</relationship>
            <invertedEdge>no</invertedEdge>
        </childTable>
    </table>
    <table name="city">
        <parentTable>country</parentTable>
        <childTable name="rental">
            <duplicate>no</duplicate>></duplicate>
            <relationship>one-to-many</relationship>
            <invertedEdge>no</invertedEdge>
        </childTable>
    </table>
    <table name="rental">
        <parentTable>city</parentTable>
        <childTable name="staff">
            <duplicate>no</duplicate>></duplicate>
            <relationship>one-to-many</relationship>
            <invertedEdge>no</invertedEdge>
        </childTable>
        <childTable name="inventory">
            <duplicate>no</duplicate>
        </childTable>
    </table>
...
```

Fig. 3.3: Truncated Output XML for Sakila Database

CHAPTER 4

EVALUATION

We experimented our algorithms on several sample public databases including Sakila, Northwind and Airportdb. These databases are available on MySQL's website. Our experiments include measuring the accuracy and completeness of transformation from relation schema to virtual hierarchies, and performance benchmarking on a set of queries. These queries are designed to target the structural relationships and theoretical data distribution across the hierarchy. The output virtual hierarchies are imported to eXist-db, a fully functional and open source XML DBMS.

4.1 Testing Database

The main testing databases are the sample ones created by MySQL, such as Sakila, World and Airportdb. We chose them because of several reasons. First, they are designed for education and practice. They contain a variety of tables, relationships and database design, which make them ideal resources to practice and test on different types of quires. Second, they also are modeled on realistic scenarios. For example, the Sakila database is modeled after a DVD rental store and Northwind database models a food importing company. This makes them good candidates for handling data we might encounter in real-world applications.

4.2 Completeness and Accuracy

Our first experiment is to verify the integrity of nodes (representing database tables) and their relationships (reflecting foreign key constraints and associations between tables) in the original database schema. The evaluation was conducted through a set of automated tests and manual inspection. Automated tests were designed to traverse the virtual hierarchy using XQuery, checking for the presence of nodes and the accuracy of their rela-

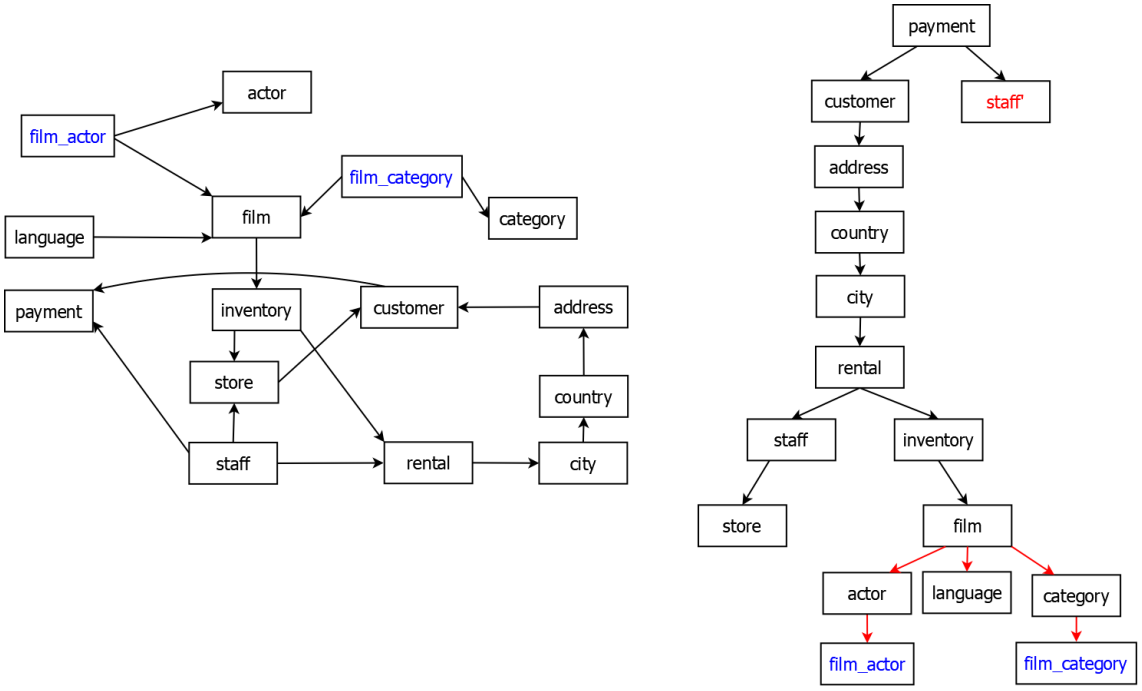

Fig. 4.1: Virtual Hierarchy for Sakila Database

tionships. Manual inspection complemented these tests by providing a thorough review of the hierarchy's logical structure and the correctness of the relationship mappings.

The experiment finds that our algorithm correctly records all the tables to our virtual hierarchies if the testing database can be mapped to a connected graph. The algorithm fails to capture all tables if there are multiple, disconnected groups of tables, for example, when parts of the database schema do not have direct or indirect relationships with each other. We also find that the output hierarchy accurately records relationship between nodes because of the additional creation of edge nodes. However, the more introduction of many-to-many relationship there is in a complex database like EmployeeDB, the need for edge direction inversion increases. Edge inversion is not ideal because it might break directionality given by the original E-R diagram. Figure 4.1 shows the example virtual hierarchy after running our algorithm. For simplicity reasons we omitted the edge nodes in the hierarchy.

4.3 Query Performance

The second experiment focuses on benchmarking the performance of the virtual hierarchies in eXist-db. We crafted a set of Xquery designed to test several aspects of the hierarchical structure, including simple lookups, complex joins, and aggregation queries. We will continue to use Sakila database to demonstrate our experiment process.

- **Query 1: Simple Retrieval**

```
for $payment in doc("sakila.xml")//table[@name="payment"]/childTable
return $payment/@name
```

  - Purpose: This query tests the function and performance of retrieving direct child elements of a given node.
  - output: `name="staff" name="customer"`
  - observation: The algorithm performs well in direct hierarchical retrieval query.

- **Query 2: Deep Hierarchy Navigation**

```
for $actor in doc("sakila.xml")//table[@name="actor"]
let $films := $actor//childTable[@name="film"]
return <result>{$actor/@name}{$films/@name}</result>
```

  - Purpose: Assess the performance of navigating through multiple levels of the hierarchy.
  - output: `<result name="actor"/>`
  - observation: Since there's no actual data beyond schema level, the query returns incorrect result.

- **Query 3: Aggregate Query Simulation**

```
let $categories := doc("sakila.xml")//table[@name="category"]
return <categories count="{count($categories)}">{$categories/@name}</categories>
```

  - Purpose: Evaluate how well the database handles aggregation-like operations within the hierarchy.
  - output: `<categories count="1" name="category"/>`
  - observation: Hierarchy can correctly check parent child relationship.

- **Query 4: Path Existence Check**

```
let $exist := exists(//table[@name="inventory"]/childTable[@name="film"])
return <path-exists>{$exist}</path-exists>
```

  - Purpose: Test the performance of checking for the existence of in a parent-child relationship.
  - output: `<path-exists>true</path-exists>`
  - observation: Hierarchy can correctly handle aggregation-like operations.

- **Query 5: Long Path Existence Check**

```
let $exist := exists(//table[@name="customer"]/childTable[@name="inventory"])
return <path-exists>{$exist}</path-exists>
```

  - Purpose: Test the performance of checking for the existence of two nodes in a ancestor relationship.
  - output: `<path-exists>false</path-exists>`
  - observation: Hierarchy does not recognize long paths between nodes..

In summary, the experiment has demonstrated that the virtual hierarchies generally provides a functional and efficient framework for data querying at a schema level. The path existence check query however fails when there is not direct parent-child relationship between the nodes, even if one node is another node's ancestor.

CHAPTER 5

CONCLUSIONS

We have proposed a set of algorithms to impose virtual hierarchies on existing relational databases. A virtual hierarchy is a tree-like structural representation of a collection of relation schemes in which nodes, edges and paths are navigational routes. Our methods minimize node duplicates when converting one-to-many and many-to-many relationship to hierarchies. We also introduced edge nodes and relational type of nodes in the construction of virtual hierarchies. The benefits of these nodes are they record relationships of concrete nodes at low space cost, and they are always set as leaf nodes to the nodes they associate with which keeps output hierarchies simple. Finally we evaluated the output virtual hierarchies in eXistDB, and found the algorithms correctly capture all the tables, relationships and constraints from relational schema, and can provide a functional hierarchy framework for simple data querying at schema level. We also observed the system performs poorly against queries for long path-existence check among nodes.

There are several improvements can be made to this work in the future:

- **Reducing Tree Height**: The height of output hierarchy T is largely determined by the longest path searching in Algorithm 1, this may lead T to become a unsatisfactory tall tree which affects overall efficiency for the databases.

- **Data Record Level Mapping**: Our algorithms target relational schema rather than the actual data records insides it. A more robust and comprehensive transition system should be capable of mapping entire databases from one to another, including schemata and data.

- **Applying Virtual Prefixed Numbering System**: As observed in evaluation process, the hierarchies are unable to detect long paths relationship when two nodes are far away. This can solved by applying virtual prefixed number system or vPBN [12]

- **Optimization for Performance**: Optimizing the algorithm for performance, especially when processing large and complex queries. This could involve refining the algorithm's parsing and guard construction processes to reduce computational complexity.

- **Integration with Big Data Technologies:** Integrating virtual hierarchies with big data processing frameworks such as Hadoop and Spark could bring new opportunities for efficient data processing. Further Research could focus on how virtual hierarchies can improve data querying and aggregation operations in big data systems.

- **Testing with Diverse and Complex Database**: The testing sets of databases we used are mostly for educational purposes and therefore the results we gathered might not provide sufficient insights for use in production environment. Further evaluations should include using a wider range of databases and testing with practical applications.

REFERENCES

[1] L. D. Shapiro, "Proceedings of the 2001 acm symposium on document engineering," in *ACM Transactions on Database Systems*, vol. 11, 1986, pp. 239–264.

[2] P. M. abd PictureMargaret H. Eich, "Proceedings of the 2001 acm symposium on document engineering," in *ACM Computing Surveys*, vol. 24, 1992, p. 63–113.

[3] Y. C. Tay, "On the optimality of strategies for multiple join," in *Journal of the ACM*, vol. 40, 1993, p. 1067–1086.

[4] J.Shanmugasundaram, E.Shekita, J.Kiernan, R. Krishnamurthy, E. Viglas, and I. T. J. Naughton, "A general technique for querying xml documents using a relational database system," in *SIGMOD Record*, vol. 30, 2001, pp. 20–26.

[5] S. Amer-Yahia, F. Du, and J. Freire., "A comprehensive solution to the xml-to-relational mapping problem," in *Proceedings of the 6th annual ACM international workshop on Web information and data management*, vol. 30, 2004, pp. 31–38.

[6] N. Yaghmazadeh, X. Wang, and I. Dillig., "Automated migration of hierarchical data to relational tables using programming-by-example," in *Proceedings of the VLDB Endowment*, vol. 11, 2004, pp. 580–593.

[7] R. Krishnamurthy, R. Kaushik, and J. F. Naughton, "Xml-to-sql query translation literature: The state of the art and open problems," Ph.D. dissertation, University of Wisconsin-Madison, US, 2007.

[8] Y. E. Lien., "Hierarchical schemata for relational databases," in *ACM Transactions on Database Systems*, vol. 6, 2004, pp. 48–69.

[9] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald, "Efficiently publishing relational data as xml documents," in *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, 2000, pp. 65–76.

[10] M. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, W.-C. Tan, R. Barr, M. J. Carey, B. G. Lindsay, and B. R. Hamid Pirahesh, "Efficiently publishing relational data as xml documents," in *ACM Transactions on Database SystemsVolume*, vol. 4, 2001, p. 438–493.

[11] I. Varlamis and M. Vazirgiannis, "Proceedings of the 2001 acm symposium on document engineering," in *ACM Transactions on Database Systems*, vol. 4, 2001, pp. 105–114.

[12] C. E. Dyreson, S. S. Bhowmick, and R. Grapp, "Querying virtual hierarchies using virtual prefix-based numbers," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014, pp. 791–802. [Online]. Available: http://doi.acm.org/10.1145/2588555.2610506