5-2024

Inferring a Hierarchical Input Type for an SQL Query

Santosh Aryal
Utah State University, a02345936@usu.edu

INFERRING A HIERARCHICAL INPUT TYPE FOR AN SQL QUERY

by

Santosh Aryal

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	John Edwards, 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

Inferring a Hierarchical Input Type for an SQL Query

by

Santosh Aryal, Master of Science

Utah State University, 2024

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

SQL queries, as the fundamental mechanism to interact with relational databases, have always faced challenges related to specifying and verifying input types. Ensuring the accuracy of input types is vital for achieving reliable results and efficient pe rformance. In Plug-and-play SQL the input type is specified as a h ierarchy. But relying on a programmer to specify the hierarchy could lead to a mismatch between the query and the hierarchy because the programmer makes a mistake or because the structure of the data changes.

Rather than relying on mistake-prone programmers to specify the hierarchy a better strategy is to automatically infer the input type from a query. This thesis introduces an automated system to infer hierarchical input type from an SQL query using ANTLR, a powerful parser generator, for a specialized SQLite grammar. We developed an input type inference algorithm for a common subset of SQL queries.

To demonstrate the applicability and robustness of our solution, we conducted an evaluation using a diverse set of real-world SQL queries. Preliminary results indicate that our system can consistently and accurately identify query input types, thus reducing the chances of query execution errors and increasing efficiency.

(51 pages)

PUBLIC ABSTRACT

Inferring a Hierarchical Input Type for an SQL Query

Santosh Aryal

SQL queries are a common method to retrieve information from databases, much like asking a detailed question and getting a precise answer. Plug-and-play queries simplify the process of querying. In a Plug-and-play SQL query a programmer sketches the shape of the input to the query as a hierarchy. But the programmer could make a mistake in specifying the hierarchy and it takes programmer time and effort to specify the hierarchy. A better solution is to automatically *infer* the hierarchy from a query. This thesis presents a system to infer a hierarchical input type for an SQL query. We consider two cases, with and without knowledge of the organization of the database. Knowledge of the database's organization can help to create a more precise hierarchy but it not necessary. This thesis describes an inference algorithm for both cases as well as an implementation of the algorithm. Finally, we evaluate the system on a set of database queries.

ACKNOWLEDGMENTS

First and foremost, I would like to express my profound gratitude to my major professor, Dr. Curtis Dyreson, for his unwavering guidance, patience, and support throughout the course of this research. His expertise and insights have been invaluable, and his encouragement has been a constant source of motivation.

I am equally grateful to my committee members, Dr. John Edwards and Dr. Steve Petruzza. Their feedback, constructive criticisms, and encouragement have significantly shaped this research, and their expertise has been instrumental in refining my work.

To my family back in Nepal: Your unwavering faith in my abilities, your love, and the sacrifices you have made to support my dreams have been my anchor. Even miles away, your presence has been felt every step of the way.

I would also like to extend my heartfelt thanks to my friends here at Utah State University. Your camaraderie, encouragement, and the countless moments of relief amidst the rigors of academic life have played a huge role in making this journey memorable and rewarding.

Lastly, to everyone who has been a part of this journey, directly or indirectly, thank you. This accomplishment is a reflection of the collective effort and belief of all those mentioned and many unmentioned.

Santosh Aryal

To my family

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Background

Relational databases have long supported our ever-increasing data needs [1]. Central to these systems is the Structured Query Language (SQL) [2], a standardized language used to access, modify, and manipulate the data stored in a relational database. SQL queries have well-defined *input* and *output* types. The input type is a subset of the *schema*, which is how the data is organized in the database. The input type is a set of table and column names. The output type is a table with a specified number of columns. It describes the structure of the data set that will be returned after the query execution [3].

For decades, SQL queries were crafted based on a fixed understanding of a database's structure. This traditional approach necessitated that the query's input type align with the database schema's specific tables and columns, allowing for predictable, though sometimes rigid, data extraction. The output type, meanwhile, was confined to the strict structures dictated by the query specifications

However, the landscape of database management has evolved, growing in both scale and complexity, prompting innovations in the mechanisms for interacting with them. One significant advancement is the introduction of *plug-and-play queries* [4–11]. This paradigm facilitates more flexible database interactions by employing hierarchical specifications for query input types. Instead of a flat, rigid structure, the input type is defined using a tree-like hierarchy of table and column names that guides SQL query execution, as illustrated in Figure 1.1. A query together with the conceptual model of the data needed by the query, in this case a hierarchy, is type matched to the schema of the database. The match produces a transformed query that is executable against the schema as well as a report on type errors or potential information loss in the transformation. There are several benefits that potentially

accrue.

- *Portability* - A query is *portable* if it can be type safely evaluated on different data collections. The hierarchy is not only critical to describing the input type to safeguard the query, but can be used to transform the query so that it can adapt to the data's type.

- *Simplicity* - A key challenge for query writers, especially novice query writers, is understanding the (conceptual model of a) database. It is simpler and easier for writers to express their conceptual understanding of the data needed by the query and let the compiler match the input type to the data's type, transforming the query to adapt to the data's type as needed.

- *Resilience* - Queries written with respect to a specific schema are brittle in the sense that if the schema changes, even small changes, the query may fail. To make a query resilient to schema evolution it is best to capture in a hierarchy what the query needs to evaluate and match the hierarchy to the current schema.

In summary using a hierarchy as the input type potentially makes a query portable, easier to code, and more resilient to schema changes.

By focusing on the input type and adopting a hierarchical perspective, we can create queries that are more adaptable and capable of navigating the complex relationships within data. This hierarchy-centric approach, promising a more intuitive and streamlined interaction with databases, allows queries to be more dynamic. However, it also introduces a unique set of challenges. These include the need to accurately infer the hierarchical input type from a query, ensuring it reflects the true structure and relationships in the underlying data, which is crucial for maintaining the integrity and reliability of the query results.

1.2 Problem Statement

As the plug-and-play approach gains traction, two main challenges have emerged. First is the issue of manual hierarchy specification. While offering more flexibility, the responsibility of defining these tree-like structures rests on the users. Manual specification raises

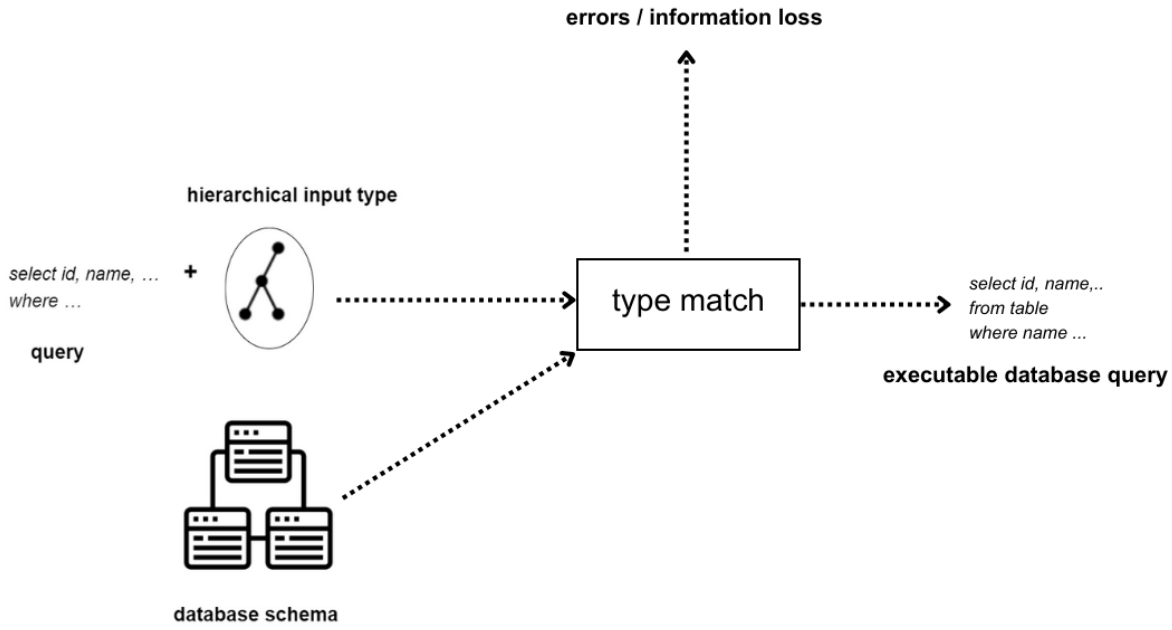

Fig. 1.1: Using hierarchical model as input type

the potential of inconsistencies or outright errors in defining the hierarchies, leading to inaccurate data retrieval. The second challenge arises from the dynamic nature of databases and queries. Even if an initial hierarchy is defined correctly, any modification to either the database structure or the SQL query can lead to mismatches. This misalignment results in either errors during query execution or unintended results, both of which can have significant implications, especially in critical applications like finance, healthcare, or research.

Addressing these challenges isn't just about improving efficiency but is pivotal for ensuring the accuracy and reliability of data extraction in the modern age [12]. Given the critical role of databases in today's world, an automated system that can navigate the complexity of hierarchical input types in SQL queries is a necessity.

1.3 Research Objectives

The goal of the research reported in this thesis is to design and implement an automated system that can accurately infer the hierarchical input type of an SQL query based on its textual representation. This system aims to address the challenges posed by manual

specification of hierarchies in the context of plug-and-play SQL and ensure the reliability and efficiency of SQL query execution. To achieve this, our research is grounded in the following four objectives.

1. System Design and Implementation

   Objective : Develop a system capable of recognizing the input type of SQL queries by analyzing their textual representations.

   Rationale : By automating the process of hierarchy identification, we can minimize manual errors, improve query reliability, and optimize the querying process.

2. Grammar Integration using ANTLR

   Objective : Incorporate a specialized grammar tailored for SQLite and utilize ANTLR (ANother Tool for Language Recognition) for parsing and analyzing SQL queries.

   Rationale : ANTLR is a potent tool for understanding textual input. By combining it with a grammar designed for SQLite, we aim to achieve a high level of precision in analyzing and understanding SQL queries.

3. Rule and Constraint Establishment

   Objective : Define a comprehensive set of rules and constraints that guide the system in accurately inferring query input types.

   Rationale : SQL query syntax possesses nuances and variations. By establishing specific rules and constraints, we can ensure consistent recognition of hierarchical structures across various query forms.

4. System Evaluation

   Objective : Rigorously test the system's efficiency and accuracy using a diverse set of real-world SQL queries.

   Rationale : The practical applicability of our solution hinges on its performance in real-world scenarios. A thorough evaluation will provide insights into the

system's strengths, potential areas of improvement, and its overall feasibility in addressing the challenges of hierarchical querying.

By fulfilling these objectives, this research aims to make a significant contribution to the field of relational database systems, enhancing the robustness and reliability of SQL query execution in the evolving landscape of hierarchical data interactions.

1.4 Plug-And-Play SQL

Plug-and-play SQL is a technique that aims to simplify the process of writing accurate and efficient queries. It involves coupling a query with a *query guard*, *e.g.,* a hierarchical specification of the input type, to create a plug-and-play query that can be used with any data source [11]. The idea is similar to that of a plug-and-play device, where a device can be plugged into any socket and will work if the necessary input is provided. Similarly, a plug-and-play query can be plugged into any data source and will produce the desired result.

We motivate the utility of query guards with an example. Suppose that we have a relational database with data about biological specimens collected in the field. A user could query the database using the query in Figure 1.2 to retrieve the names of botanists who collected *Asteraceae* (plants in the Daisy family) specimens in 2023. The query does a join between the `taxa`, `occurrences`, and `collectors` tables, applies the appropriate selection conditions, and projects the name of the botanist. The query explicitly uses logical pointers (foreign key to key associations) from the `taxa` table to the `occurrences` and `collectors` tables.

We can rewrite the query as a plug-and-play SQL query using a query guard as shown in Figure 1.3. The guard specifies the *shape* or type of the *input* to the query. The guard stipulates that the query can be evaluated on any data collection that has this hierarchy, or that can be converted or transformed to the desired shape (within information loss guarantees).

```
SELECT collectors.name
FROM taxa, occurrences, collectors
WHERE taxa.tid = occurrences.tid AND collectors.id = occurrences.collid
  AND taxa.family = 'Asteraceae' AND occurrences.year = 2023
```

Fig. 1.2: Retrieve the names of botanists who collected *Asteraceae* specimens in 2023

```
GUARD collectors {
        name,
        occurrences {
          family,
          year
        }
      }
SELECT name
WHERE family = 'Asteraceae' AND year = 2023
```

Fig. 1.3: Retrieve the names of those who collected *Asteraceae* specimens in 2023

One big advantage of plug-and-play SQL queries is that they are *portable*. The query in Figure 1.3 is portable to data collections that have different shapes (*i.e.,* we do not care how many steps are involved in "joining" the tables to construct the hierarchy). A second advantage is that the hierarchy naturally *groups* the data, and the grouping can be exploited in a query for aggregation. Suppose for instance we only wanted those collectors who collected more than 40 specimens then we could modify the query as shown in Figure 1.4. Querying against a hierarchy simplifies grouping and aggregation (as in XQuery and Cypher).

Play-and-play SQL queries focus on matching and transforming the *shape* of the data. They are agnostic about the *semantic* matching of labels between the guard and the source, *e.g.,* does `person` in the guard mean the same as `person` in the data, because the *semantic matching problem* is already being researched by other communities, *e.g.,* work on ontologies in the Semantic Web community. We can add Semantic Web solutions to plug-and-play queries to address the problem of semantic mismatch. Note that the table names in the query guard in Figure 1.3 are present to help in the semantic matching. The guard could be simplified to that shown in Figure 1.5. To better combine the output of any semantic matching technique with the guard, a `MATCH` clause could be added that maps names in the

```
GUARD collectors {
        name,
        occurrences {
          family,
          year
        }
      }
SELECT name
WHERE family = 'Asteraceae' AND year = 2023 AND COUNT(*) > 40
```

Fig. 1.4: Retrieve the names of botanists who collected more than 40 *Asteraceae* specimens in 2023

```
GUARD name {
        family,
        year
      }
SELECT name
WHERE family = 'Asteraceae' AND year = 2023
```

Fig. 1.5: Simplified guard for the query in Figure 1.3

schema to those in the guard.

One of the key features of Plug-and-Play SQL is the use of hierarchies to eliminate the need to write a view to construct virtual tables or a set of tables to run a query. Instead, the hierarchy is a declarative specification of the desired shape of data, which provides a natural way to group data for aggregation and eliminates the common use of logical or semantic pointers in SQL queries.

Plug-and-Play SQL queries have several advantages, including portability and the ability to be used to evaluate any data source. They also promote query reuse and reduce the time and effort required to write new queries as new users and user needs arise over time.

However, manual specification of these hierarchies poses a significant challenge, opening doors to potential errors and inconsistencies, particularly when changes are made to the database or the query. By automating the inference of hierarchical input types, we can dramatically reduce the margin for human error. An automated system can consistently and accurately determine the correct hierarchy, ensuring that the query seamlessly integrates

with the intended data source every time. Moreover, as data structures evolve or when there's a need to plug the query into a new or modified data source, an automated system can swiftly adapt, realigning the hierarchical specifications as needed. In essence, by automating the hierarchical inference process, we not only fortify the reliability of Plug-and-Play SQL but also amplify its core strength.

CHAPTER 2

ALGORITHM AND EXAMPLES

In this chapter, we explore how to figure out the hierarchical input types for SQL queries. The chapter is divided into four main sections. First, we describe some overall design goals for the algorithm. Next, we introduce the Algorithm, called the Guard Generation Algorithm, which is a detailed guide for building the hierarchy of an SQL query based on two overall cases, one where the schema is available and one where the schema is not available. Following this, we show this algorithm in action with real SQL queries. Finally, after showcasing these examples, we conduct a thorough Big-O analysis of the algorithms.

2.1 Constraints for Hierarchical Input Type Inference

The process of inferring a hierarchical input type has several constraints that should be observed to produce a "good" hierarchy.

- **Completeness** : The most important constraint is completeness. When data is in a hierarchy a child only exists if a parent exists. Therefore, in inferring a hierarchy, we should try to ensure that for every child there exists a corresponding parent.
- **Accuracy** : The inferred hierarchy must accurately mirror the query's structural demands, reflecting the precise relationships between the tables involved in the query.
- **Declarative and reusable**: The hierarchy should maintain a standard form, free from specific data dependencies, enabling its reuse across multiple queries with similar structural demands.
- **Concise**: The hierarchy should eliminate all non-essential elements, promoting ease of interpretation and modification.

- **Data model consistency**: To maintain semantic integrity, the hierarchy should utilize nomenclature and concepts that resonate with the existing database schema, promoting uniformity and comprehension.

2.2 Guard Generation Algorithm and Examples

This section presents the guard generation algorithm, first for the case where the schema is available, and then for the case where a schema is not known. Knowledge of the schema helps to construct a more complete hierarchy.

2.2.1 Algorithm when schema is available

The guard generation algorithm when the schema is available is described in Algorithm 1. Initially, the algorithm starts by initializing an empty structure called guard G and creating lists for outer and inner tables, along with a dictionary to map tables to their columns. The next step involves decomposing the SQL query into its constituent parts: the SELECT, FROM, and WHERE clauses. This decomposition helps identify the output columns, the tables involved, and any specific conditions linked to these tables. The outer table is then determined based on certain criteria, such as the presence of a subquery or JOIN clauses. Following this, the algorithm populates the lists of outer and inner tables, and associates columns with these tables. The core of the algorithm is the hierarchical construction of the Guard: starting with the outer table as the root, it systematically adds tables and their associated columns as child nodes, creating a nested structure that accurately represents the query's hierarchy. Additionally, it refines this structure by removing columns that are only used for joining tables and not present in the output or WHERE clause conditions. The final output of the algorithm is the guard G, which represents the hierarchical input type of the given SQL query. The pseudo code for this algorithm is given in Figure 2.1.

2.2.2 Examples when schema is available

This section provides a detailed walk through of inferring a hierarchical input type

Algorithm 1: Guard Inference with Schema Knowledge

Input: SQL query Q
Output: Guard G representing the hierarchical input type

`/* Initialization */`
Create an empty Guard structure G
Create lists for outer and inner tables: `OuterTables` and `InnerTables`
Create an empty dictionary, `TableColumns`, `TableRelationships`, and `ColumnWeights`
`/* Decompose SQL Query */`
Extract columns from the clauses
Prioritize columns in the WHERE, GROUP BY, and HAVING clauses
for *each column in these clauses* **do**
    Assign weights: Higher Weight for columns in GROUP BY or HAVING, Lower Weight for columns in SELECT or WHERE
end
`/* Identify Outer Table */`
if *there's a subquery* **then**
    Outer table is the one from the subquery
else
    if *there's a JOIN between tables* **then**
        Relationships based on foreign keys, where child table is the one that borrows the key
    else
        Outer table is the table from which the output columns (in SELECT) are directly derived
    end
end
`/* Populate Lists */`
Add the identified outer table to `OuterTables`
Add other tables from the FROM clause and subquery (if any) to `InnerTables`
`/* Build the Guard Hierarchically */`
Start with the Outer table as the root of G
for *each table in* `OuterTables` *and* `InnerTables` **do**
    Add associated columns (from `TableColumns` dictionary) as child nodes ensuring columns with higher weights are placed higher in the hierarchy
    If there are JOINs, structure the hierarchy to reflect these relationships
end
`/* Refinement */`
Remove any columns from G that are only used for joining tables and are not present in the output or specific conditions in the WHERE clause
return G

from a specific SQL query when the schema is available. We will go through each step of the algorithm previously discussed.

Example 1. Simple Select Our first example is for a query with a single predicate in the WHERE clause and one table. Consider the SQL query given below.

```
SELECT first_name, last_name
FROM employees
WHERE department = "Sales";
```

The resulting, inferred guard is given below.

```
{
  "employees": {
    "department": {
      "first_name",
      "last_name"
    }
  }
}
```

To infer the guard, the algorithm goes through the following steps.

1. **ExtractSelectColumns**: The SELECT clause specifies `first_name` and `last_name` as the output columns.
2. **ExtractFromTables**: The FROM clause indicates that the data is retrieved from the `employees` table.
3. **ExtractWhereConditions**: The WHERE clause applies a condition on the `department` column. As the condition does not involve an `IS NULL` comparison the `department` column is given the largest weight.
4. **Identify Outer Table**: Since there's no subquery or JOIN, the Outer table is straightforwardly `employees`.
5. **Associate columns with tables**: The columns `first_name`, `last_name`, and `department` are associated with the `employees` table.
6. **Build Guard:**

- Start with `employees` as the root of the guard, reflecting it as the Outer table.
- Place the `department` column as the child of `employees` as it is used in the WHERE clause.
- Add `first_name` and `last_name` as children of `department`, representing the selected output columns.

7. **Refinement**: No refinement is needed as all included columns are relevant to the output or the query's filtering condition.

Example 2. Subquery The second example is for a query with a subquery in the WHERE clause. This query given below performs a semi-join between two tables.

```
SELECT customer_name
FROM customers
WHERE customer_id IN (
    SELECT customer_id
    FROM orders
    WHERE order_total > 100
);
```

To infer the guard, the algorithm goes through the following steps.

1. **ExtractSelectColumns:** Extracts '{`customer_name`}'.
2. **ExtractFromTables:** For the main query, it's '{`customers`}', and for the subquery, it's '{`orders`}'.
3. **ExtractWhereConditions:** Main query conditions are related to '{`customer_id`}', and subquery conditions are related to '{`customer_name, order_total`}'.
4. **Identify Outer Table:** Due to the presence of a subquery, the Outer table is taken from the subquery, which is '`orders`'.
5. **Associate Columns with Tables:**
   - For the '`customers`' table, we have '{`customer_name, customer_id`}'

- For the 'orders' table, we have '{customer_id, order_total}'

6. **Build Guard:**
   - Starting with 'orders' as the root, given the Outer identification from the subquery.
   - Extend hierarchy to the 'customers table based on the IN condition.
   - Add associated columns from each table as child nodes.
7. **Refinement:** During refinement, customer_id is excluded from the guard. It's used in the query to link orders and customers, but since it's neither a part of the output nor a specific filtering condition in the context of the Outer table (orders), it's not included in the guard.

The resulting, inferred guard is given below

```
{
  "orders": {
    "order_total",
    "customers": {
      "customer_name"
    }
  }
}
```

Example 3. Subquery The third example is for a query that joins two tables. This query given below performs an inner join between two tables.

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

To infer the guard, the algorithm goes through the following steps.

1. **ExtractSelectColumns:** The SELECT clause includes: 'e.first_name', 'e.last_name', and 'd.department_name'.

2. **ExtractFromTables:** The query uses two tables: 'employees' (aliased as e) and 'departments' (aliased as d).

3. **ExtractWhereConditions:** The JOIN condition is 'e.department_id = d.department_id. There are no specific WHERE conditions.

4. **Identify Outer Table:** Based on the JOIN condition, 'employees can be considered the child table (as it likely contains the foreign key department_id), and departments the parent table.

5. **Associate Columns with Tables:**

   - 'employees contributes 'first_name', and 'last_name'.
   - 'departments' contributes 'department_name'.

6. **Build Guard:**

   - Start with 'departments' as the Outer table.
   - Include 'department_name under departments.
   - Nest 'employees within 'departments, indicating the JOIN relationship.
   - Under 'employees', include 'first_name' and 'last_name'.

7. **Refinement:** Exclude columns that are only used for joining tables and not part of the output or specific filtering conditions. In this case, 'department_id' is used for joining but is not part of the output, so it is omitted from the guard.

The resulting, inferred guard is given below

```
{
  "departments": {
    "department_name",
    "employees": {
      "first_name",
      "last_name"
    }
  }
}
```

Example 4. Subquery The fourth example is also for a query that joins two tables but with group by, having clause and count aggregation. This query given below performs an inner join between two tables and also use group by and having clause.

```
SELECT d.department_name, d.location, COUNT(e.employee_id) AS employee_count
FROM departments d
JOIN employees e ON d.department_id = e.department_id
GROUP BY d.department_name
HAVING COUNT(e.employee_id) > 3;
```

We will apply the Guard Generation Algorithm to deduce the Guard for this query.

1. **ExtractSelectColumns:** The SELECT clause includes: `'d.department_name'`, `'d.location'` and `'COUNT(e.employee_id)` as the output columns.

2. **ExtractFromTables:** The query uses two tables: '`departments`' (aliased as `d`). and '`employees`' (aliased as `e`).

3. **ExtractWhereConditions:** The JOIN condition is '`d.department_id = e.department_id` and there's a '`HAVING` clause with '`COUNT(e.employee_id) > 3`'.

4. **Assign Weights to Columns:**

   - '`d.department_name`' in the `GROUP BY` clause gets a higher weight (e.g. weight 3).
   - '`e.employee_id`' used in the COUNT aggregation and HAVING clause also gets a higher weight (e.g., weight 3).
   - '`d.location`' is in the SELECT clause but not in GROUP BY or HAVING, so it gets a lower weight (e.g. weight 1)

5. **Identify Outer Table:** Considering the JOIN relationship, departments is likely the parent table (holding the primary key '`department_id`), and employees is the child table (with the foreign key '`department_id`).

6. **Associate Columns with Tables:**

- 'departments' contributes 'department_name' (higher weight) and 'location' (lower weight).
- 'employees' contributes 'employee_id '(for the COUNT aggregation).

7. **Build Guard:**
   - Start with 'departments' as the Outer table.
   - Include 'department_name and then 'location' under departments due to the assigned column weights.
   - Nest 'employees within 'departments, indicating the JOIN relationship.
   - Under 'employees', include 'employee_id' for the COUNT aggregation.
8. **Refinement:** Exclude columns used only for joining that aren't part of the output or specific filtering conditions. 'department_id' is used for the JOIN but not as a selected output column or in the HAVING clause, so it's omitted.

The resulting, inferred guard is given below.

```
{
  "departments": {
    "department_name",
    "location",
    "employees": {
      "employee_id"
    }
  }
}
```

2.2.3 Algorithm when schema is not available

The guard generation algorithm when schema is not available is described in Algorithm 2. The algorithm initializes an empty structure called 'Guard G' and creates lists and dictionaries to hold tables and their potential columns, along with any explicit associations between them. It then decomposed the SQL query into its SELECT, FROM, and

WHERE clauses to identify involved tables, output columns, and conditions. The algorithm pays special attention to explicit column associations, denoted by the 'table.column' format, and records these in a dedicated dictionary. In determining the primary or 'Outer Table', it considers the presence of subqueries or chooses the table with the most referenced columns. Subsequently, the algorithm constructs the Guard in a hierarchical manner. For each table, it includes only those columns that are explicitly associated with the table or, in the absence of explicit associations, all possible columns. This approach ensures that the Guard accurately represents the query's structure, with a focus on maintaining the correct associations between tables and columns. The final step involves refining the Guard by removing any nodes used solely for joining purposes and not part of the query's output or conditions. The resulting Guard G thus provides a hierarchical representation of the SQL query when the schema is not present. The pseudo code for this algorithm is given in Figure 2.2

2.2.4 Examples when schema is not available

This section provides a detailed walk through of inferring a hierarchical input type from a specific SQL query when the schema is not available.

Example 1. Simple Select We take the previous example, a query with a single predicate in the WHERE clause and one table given below.

```
SELECT first_name, last_name
FROM employees
WHERE department = "Sales";
```

Here every step will be similar as that with the available schema, since there is only one table and all the columns will be the child of that table. And from this we get the same inferred guard.

The resulting, inferred guard is given below.

```
{
  "employees": {
    "department": {
```

Algorithm 2: Guard Inference without Schema Knowledge

Input: SQL query Q
Output: Guard G representing the hierarchical input type
`/* Initialization */`
Create an empty Guard structure G
Create lists for outer and inner tables: `OuterTables` and `InnerTables`
Create an empty dictionary, `TableColumns`, `TableRelationships`, `ExplicitAssociations` and `ColumnWeights`
`/* Decompose SQL Query */`
Extract columns from the clauses
Prioritize columns in the WHERE, GROUP BY, and HAVING clauses
for *each column in these clauses* **do**
    Assign weights: Higher Weight for columns in GROUP BY or HAVING, Lower Weight for columns in SELECT or WHERE
    If a column is referenced with a table (like 'employees.first name'), note this explicit association
end
`/* Identify Outer Table */`
if *there's a subquery* **then**
    Outer table is the one from the subquery
else
    if *there's a JOIN between tables* **then**
        Relationships based on foreign keys, where child table is the one that borrows the key
    else
        Outer table is the table from which the output columns (in SELECT) are directly derived
    end
end
`/* Populate Lists */`
Add the identified outer table to `OuterTables`
Add other tables from the FROM clause and subquery (if any) to `InnerTables`
Populate the `AllColumns` dictionary with all columns that could potentially belong to each table
`/* Build the Guard Hierarchically */`
Start with the Outer table as the root of G
for *each table in* `OuterTables` *and* `InnerTables` **do**
    If explicit column associations are available, add only those columns as child nodes in G
    Else, add all columns ensuring columns with higher weights are placed higher in the hierarchy
    If there are JOINs, structure the hierarchy to reflect these relationships
end
`/* Refinement */`
Remove any columns from G that are only used for joining tables and are not present in the output or specific conditions in the WHERE clause
return G

```
      "first_name",
      "last_name"
    }
  }
}
```

Example 2. Select from multiple tables This example is for a query which selects from multiple tables and without the schema knowledge we don't know which column belongs to which table.

```
SELECT name, location, status
FROM employees, departments
WHERE employees.department_id = departments.department_id
AND departments.state = 'UT';
```

To infer the guard, the algorithm goes through the following steps.

1. **ExtractSelectColumns:** The SELECT clause includes: `'name', 'location', and 'status'`.

2. **ExtractFromTables:** The query uses two tables: '`employees`' and '`departments`'.

3. **ExtractWhereConditions:** Conditions include '`employees.department_id = departments.depa` and '`departments.state = 'UT'`'.

4. **Identify Outer Table:** Since we don't know which table is primary or secondary without the schema, we consider both '`employees`' and '`departments`' at the root level.

5. **Associate Columns with Tables:**

   - All possible columns are considered for each table due to the absence of explicit schema information except for '`state`' which is under '`department`' table.

6. **Build Guard:**

   - Since we don't know which table is primary or secondary, we consider both '`employees`' and '`departments`' at the root level.

- Since explicit column associations is only for '`state`', we add all other known columns to each table.

The resulting, inferred guard is given below

```
{
    "employees": {
        "name",
        "location",
        "status",
    },
    "departments": {
        "name",
        "location",
        "status",
        "state"
    }
}
```

2.3 Big-O Analysis

The analysis considers each case of the algorithm separately.

2.3.1 When Schema is Available

- **ExtractSelectColumns**, **ExtractFromTables**, and **ExtractWhereConditions**: These functions parse different parts of the SQL query. Each function typically iterates over the length of the query once, leading to $\mathcal{O}(n)$ complexity for each, where n is the length of the SQL query.
- **Assigning Weights to Columns**: This process involves iterating over the extracted columns and assigning weights based on their presence in different clauses. The complexity is $\mathcal{O}(m)$, where m is the number of columns involved in the query.
- **Determining the table relationships:** Analyzing the relationships between tables, especially for JOIN operations and subqueries, involves parsing and understanding the structure of the query. This complexity can vary but is generally $\mathcal{O}(f)$, where f is the number of foreign key relationships or JOIN conditions in the query.

- **Populating Lists and Dictionaries:** Populating `OuterTables`, `InnerTables`, and `TableColumns` involves iterating over the list of involved tables and the columns. This step can be estimated as $\mathcal{O}(t + t * c)$, where t is the number of tables, and c is the average number of columns per table.

- **Building the Guard:** Constructing the hierarchical guard structure involves nested iterations over tables and their associated columns, considering the weights. This is typically $\mathcal{O}(t * c)$, taking into account the hierarchy construction.

- **Refinement loop:** This step involves checking each node in the guard structure for its necessity based on the query's conditions, which can be approximated as $\mathcal{O}(n)$.

Combining these, the overall time complexity for the algorithm when schema information is available is $\mathcal{O}(n + m + f + t * c)$.

Where:

- n is the length of the SQL query.

- m is the number of columns involved in the SQL query.

- f is the number of foreign key relationships between tables.

- t is the number of tables.

- c is the average number of columns associated with each table.

2.3.2 When Schema is Not Available:

In the absence of schema information, the algorithm assumes all columns can belong to all tables unless explicitly associated, which impacts the complexity

- **Extracting Query Components and Assigning Weights:** Similar to the schema-available case, this is $\mathcal{O}(n)$ for extraction steps and $\mathcal{O}(m)$ for assigning weights.

- **Building the Guard Without Schema Information:** Without schema details, the guard's construction might involve more assumptions, potentially considering all

columns for each table. However, if the algorithm still assumes a single root table and a hierarchy beneath it, the complexity remains $\mathcal{O}(t * c)$, where t is the number of tables and c is the total number of columns mentioned in the query.

- **Refinement Loop:** Similar to the schema-available case, this step remains $\mathcal{O}(n)$.

Therefore, the overall time complexity for the algorithm when schema information is not available is also $\mathcal{O}(n + m + f + t * c)$.

In both scenarios, the complexity is dominated by the steps involving parsing the SQL query and constructing the guard structure.

```
FUNCTION GenerateGuard(Q: SQL Query) -> Guard
    Initialize an empty Guard G
    Initialize lists OuterTables, InnerTables
    Initialize dictionary TableColumns, TableRelationships, ColumnWeights

    // Decompose the SQL Query
    involved_tables = ExtractFromTables(Q)
    conditions = ExtractWhereConditions(Q)
    output_columns, group_by_columns, having_columns, where_columns = ExtractQueryComponents(Q)
    AssignWeightsToColumns(output_columns, group_by_columns,
                           having_columns, where_columns, ColumnWeights)
    TableRelationships = DetermineTableRelationships(involved_tables, conditions)

    // Identify Outer Table
    IF there's a subquery in Q:
        OuterTables.append(table from the subquery)
    ELSE:
        OuterTables.append(table directly linked to output_columns)

    FOR table in involved_tables:
        IF table is not in OuterTables:
            InnerTables.append(table)

    // Associate Columns with Tables
    FOR table in involved_tables:
        TableColumns[table] = columns associated with table in Q

    // Build Guard
    G = OuterTables[0] // Root
    FOR table in OuterTables + InnerTables:
        parent = TableRelationships.get(table, None)
        IF parent:
            IF parent not in G:
                G[parent] = {}
            ConsiderWeightsAndBuildHierarchy(G, parent, table, TableColumns, ColumnWeights)
        ELSE:
            G[table] = {}
            AddColumnsWithWeights(G[table], TableColumns[table], ColumnWeights)

    // Refinement
    FOR node in G:
        IF node only used for joining and not in output or specific conditions:
            Remove node from G

    RETURN G
```

Fig. 2.1: Pseudo code for the guard generation algorithm when schema is available

```
FUNCTION GenerateGuard(Q: SQL Query) -> Guard
    Initialize an empty Guard G, lists OuterTables, InnerTables, AllColumnsName
    Initialize dictionary AllColumns, ExplicitAssociations, TableRelationships, ColumnWeights

    // Decompose the SQL Query
    output_columns = ExtractSelectColumns(Q)
    involved_tables = ExtractFromTables(Q)
    conditions = ExtractWhereConditions(Q)
    AllColumnsName = GetAllColumns(output_columns, conditions)
    AssignWeights(output_columns, conditions, ColumnWeights)
    TableRelationships = DetermineTableRelationships(involved_tables, conditions)

    // Identify Outer Table
    IF there's a subquery in Q:
        OuterTables.append(table from the subquery)
    ELSE: OuterTables.append(table directly linked to output_columns)

    FOR table in involved_tables:
        IF table is not in OuterTables:
            InnerTables.append(table)
        AllColumns[table] = allColumnsName (List of all possible column names)

    // Associate Columns with Tables
    FOR column in AllColumnsName:
        IF column is in format 'table.column':
            table, col = Split(column, '.')
            IF table in OuterTables or InnerTables:
                IF table not in ExplicitAssociations:
                    ExplicitAssociations[table] = Set()
                ExplicitAssociations[table].add(col)

    // Build Guard
    G = OuterTables[0] // Root
    FOR table in OuterTables + InnerTables:
        parent = TableRelationships.get(table, None)
        IF parent:
            IF parent not in G:
                G[parent] = {}
            ConsiderWeightsAndBuildHierarchy(G, parent, table, TableColumns, ColumnWeights)
        ELSE: G[table] = {}
        FOR column in AllColumns[table]:
            IF table in ExplicitAssociations:
                IF explicitColumn in ExplicitAssociations[table]:
                    AddColumnsWithWeights(G[table][explicitColumn],
                    TableColumns[table], ColumnWeights)
            ELSE:
                AddColumnsWithWeights(G[table], TableColumns[table], ColumnWeights)
    // Refinement
    FOR node in G:
        IF node only used for joining and not in output or specific conditions:
            Remove node from G
    RETURN G
```

Fig. 2.2: Pseudo code for the guard generation algorithm when schema is not available

CHAPTER 3

IMPLEMENTAION

This chapter describes the implementation of the algorithm using Python and ANTLR. The overall workflow of the program is shown in Figure 3.1. Our program takes an SQL query as input and parses it using ANTLR tools, yielding a parse tree for the query. A custom listener then walks the parse tree to capture specific elements from the query, particularly focusing on the SELECT and WHERE clauses. These elements are used to infer the query's hierarchy, which is output in readable JSON format.

3.1 ANTLR

ANother Tool for Language Recognition (ANTLR) is a powerful parser generator tool used for reading, processing, executing, or translating structured text or binary files. Unlike mere regular-expression based matching, ANTLR facilitates the development of full-fledged parsers that can understand the intricate syntax of programming languages, configuration files, or data formats. Driven by its adaptability, it's been chosen by many for the creation of domain-specific languages (DSLs), interpreters, compilers, and translators. The basic ANTLR workflow is illustrated in the Figure 3.2

Grammar in ANTLR: Every language, be it spoken or computer-based, is governed by a set of rules called a grammar. Within ANTLR, the grammar forms the linchpin—it's the blueprint that delineates the structure of the expected input. Grammars in ANTLR are crafted using a set of defined rules that indicate valid sequences of tokens. Two primary rule types shape this:

- **Lexer Rules:** These dictate how sequences of input characters are clustered into coherent tokens. Each lexer rule usually corresponds to potential substrings in your input, like keywords, operators, or identifiers.

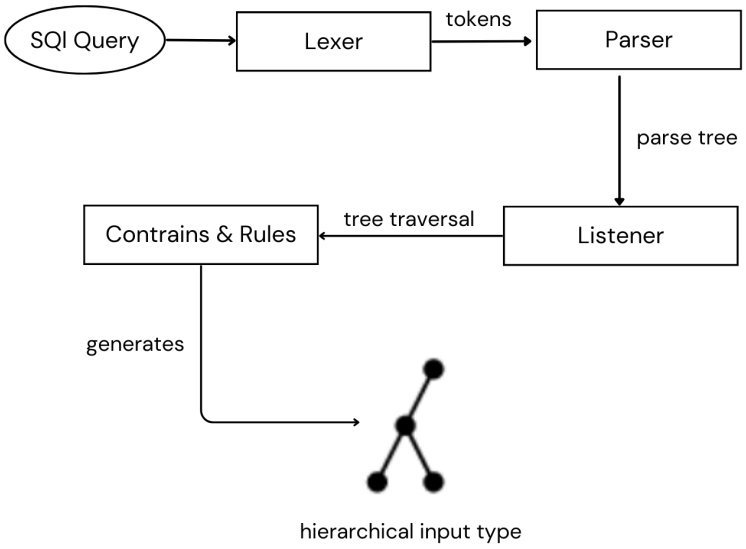

Fig. 3.1: Program workflow

- **Parser Rules:** Post tokenization, the parser rules determine the patterns, sequences, and hierarchies these tokens should adhere to.

SQLite Grammar: SQLite stands out as a unique database system, balancing lightweight operation with rich features. Adapting ANTLR for SQLite required a specialized grammar, capturing SQLite's particular flavor of SQL. This grammar serves as a translator between SQLite's syntactical intricacies and ANTLR's parsing machinery, ensuring any SQL query specific to SQLite is parsed with utmost precision.

Lexer and Parser: The lexer and parser play symbiotic roles in ANTLR's operational workflow. Initially, the lexer scans the raw input stream, meticulously transforming it into a series of tokens, each being a representative symbol of recognizable patterns defined by the lexer rules. After tokenization, the parser takes center stage. With parser rules as its guide, it arranges these tokens into a logically structured form. This structure, typically visualized as an abstract syntax tree (AST) or more commonly in ANTLR as a parse tree, captures the hierarchical relationships between tokens, reflecting the grammar's intended structure.

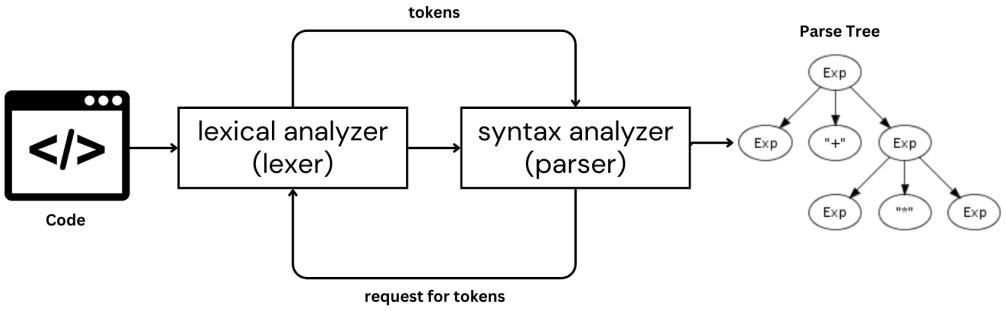

Fig. 3.2: Basic ANTLR workflow

Listener and Walker: With a parse tree in hand, the actual processing or information extraction can commence. ANTLR furnishes two paradigms for this: listeners and visitors (often referred to as the walker pattern). The more automated of the two, the listener, is a reactive interface. As the system navigates through the parse tree, the listener responds to "enter" and "exit" events at each node. For instance, when encountering a specific SQL clause in the tree, the listener can be programmed to execute certain actions. This reactive nature offers a hands-off approach to tree traversal.

In the context of our research, ANTLR, with its robust capabilities, acts as the backbone for automatically inferring the hierarchical input type of queries. Leveraging the SQLite grammar, we ensure that our system understands and processes SQL queries accurately. The lexer and parser collaborate to translate the textual representation of queries into structured forms, while the listener and walker mechanisms enable the extraction of necessary hierarchical details.

3.1.1 Using ANTLR

ANTLR is instrumental in constructing compilers and interpreters, but it's also a handy

tool for building any application that requires a sophisticated understanding of textual input. Here's how we can set up ANTLR on Windows for Python to be used on this research project.

1. Java Installation: ANTLR4 requires Java; we need to ensure it's installed on our machine. We can download it from the official Oracle website or use OpenJDK. Set the JAVA_HOME environment variable and update the system PATH to include Java binaries.

2. ANTLR 4 Setup: Download the ANTLR 4 JAR file from the GitHub releases page and place it in a known directory. Create an environment variable (e.g., ANTLR4) pointing to the jar file. Update the system PATH to include the path to the ANTLR jar file.

3. Python Runtime for ANTLR: The Python runtime is necessary for running the parser and lexer code generated by ANTLR. We can install the Python Runtime for ANTLR using pip:

```
\textit{pip install antlr4-python3-runtime.}
```

4. Generate the lexer and parser: For this research we are using SQLite grammer (SQLiteLexer.g4, SQLiteParser.g4) available on ANTLR GitHub releases page. Then we use ANTLR to generate the lexer and parser in Python. We can use the following command in the command line:

```
antlr4 -Dlanguage=Python3 SQLiteLexer.g4
antlr4 -Dlanguage=Python3 SQLiteParser.g4
```

5. Finally we write a Python script and custom listener to use the lexer and parser to generate hierarchical input type for SQL query. The visual representation of the ANTLR setup on Windows for Python used for this research project is shown in Figure 3.3

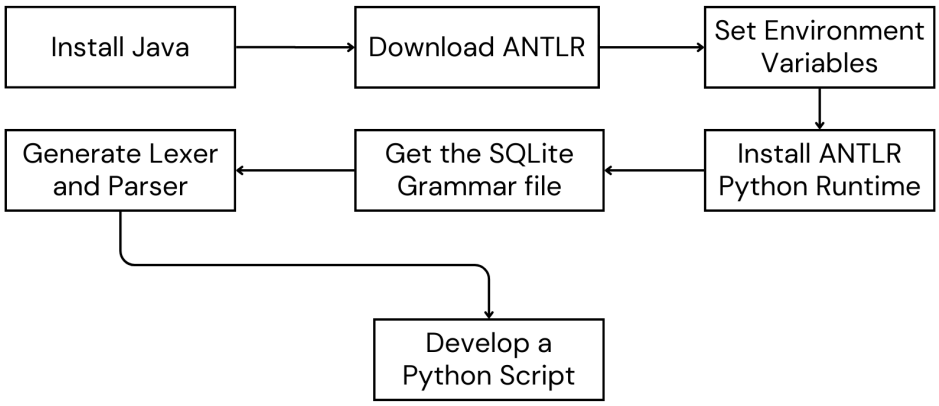

Fig. 3.3: ANTLR setup on Windows for Python

3.2 Implementation Details

This section focus on the technical aspects of our system designed to infer hierarchical input types from SQL queries. Our approach utilizes ANTLR, a powerful parser generator, for parsing SQL queries. This is achieved by defining a custom listener class, CustomSQLiteListener, which extends the functionality of the base listener provided by the ANTLR-generated SQLite parser.

- **Column and table extraction**: Two key methods, **enterExpr** and **enterResult_column**, are utilized for capturing column names from both the **SELECT** and **WHERE** clauses. These methods extract column names along with their associated table names, ensuring a comprehensive capture of all relevant fields in the query. The **enterTable_or_subquery** method is responsible for extracting table names referenced in the query. It captures tables both from the main body of the query and from any subqueries.

- **Hierarchical Guard Construction**: The construction of the hierarchical guard structure occurs in the **exitSelect_stmt** method. This method is called upon the

completion of parsing a **SELECT** statement, indicating that all relevant tables and columns have been identified. The method identifies the primary and secondary tables. If a subquery is present, the primary table is considered to be the one referenced in the subquery. Otherwise, the primary table is the first one mentioned in the query.

- **Guard Structure**: The guard is a dictionary where keys are table names, and values are lists of associated column names. If a query involves multiple tables, the structure nests secondary tables within the primary table's entry, reflecting the hierarchical nature of the query. For each table, columns are listed in a simple array format, providing a clear and concise representation of the table's schema as inferred from the query.

- **Column Weight Assignment**: A new method, **assignColumnWeights**, assigns weights to columns based on their occurrence in the query. Columns in GROUP BY or HAVING clauses receive higher weights due to their guaranteed non-null values. Columns in SELECT or WHERE clauses receive lower weights as they could potentially contain null values. These weights are used to prioritize columns in the guard structure, ensuring columns with higher weights are placed higher in the hierarchy.

Helper Methods

The implementation has several helper methods.

- **Context Finding Methods**: **find_table_name_context** and **find_column_name_context** are recursive helper methods used to navigate the ANTLR-generated parse tree. They locate the specific contexts for table and column names, respectively, ensuring accurate extraction of this information.

- **Primary Table Identification**: The **identify_primary_table** method determines the primary table in the query. It checks for the presence of a subquery using the **has_subquery** method. If a subquery is found, the primary table is considered to be from the subquery; otherwise, it is the first table in the query.

- **Subquery Detection**: The **has_subquery** method recursively examines the parse tree to determine if the current context contains a subquery. This is crucial for correctly identifying the primary table and for constructing an accurate hierarchical guard.

Output Format

The final output is a JSON-style hierarchical structure that accurately represents the relationships between tables and their associated columns as inferred from the SQL query. This format is chosen for its clarity, readability, and ease of use in further data processing or analysis tasks.

CHAPTER 4

EVALUATION

In this section, we will assess the effectiveness of our algorithm using the Northwind sample database. This well-known database is often used for testing and learning about databases, and it includes a variety of tables and data that are similar to what you might find in a real business. The primary criterion for evaluation is the successful extraction of the hierarchy, which is pivotal in ensuring the integrity and applicability of the algorithm in real-world scenarios. By using the Northwind database, we can put our algorithm to the test with practical, real-world examples.

4.1 Northwind Database

The Northwind database is a sample database originally created by Microsoft for its Access database product. It has since become a popular resource for demonstrating the features of various database management systems. The database represents a fictional company named Northwind Traders, which imports and exports specialty foods from around the world. The schema this database is shown in Figure 4.1.

4.2 Benchmark Query Set and Testing

To thoroughly evaluate our algorithm's performance, we have curated a set of SQL queries that span a wide range of database operations and complexities commonly encountered in real-world scenarios. Each query in this benchmark set is processed through our algorithm. The inferred guard structure is then compared against the expected hierarchy based on the known relationships in the Northwind database schema. We then describe the results based on the accuracy of the inferred hierarchy, the algorithm's handling of different SQL constructs, and any discrepancies observed.

- **Simple SELECT Query**:

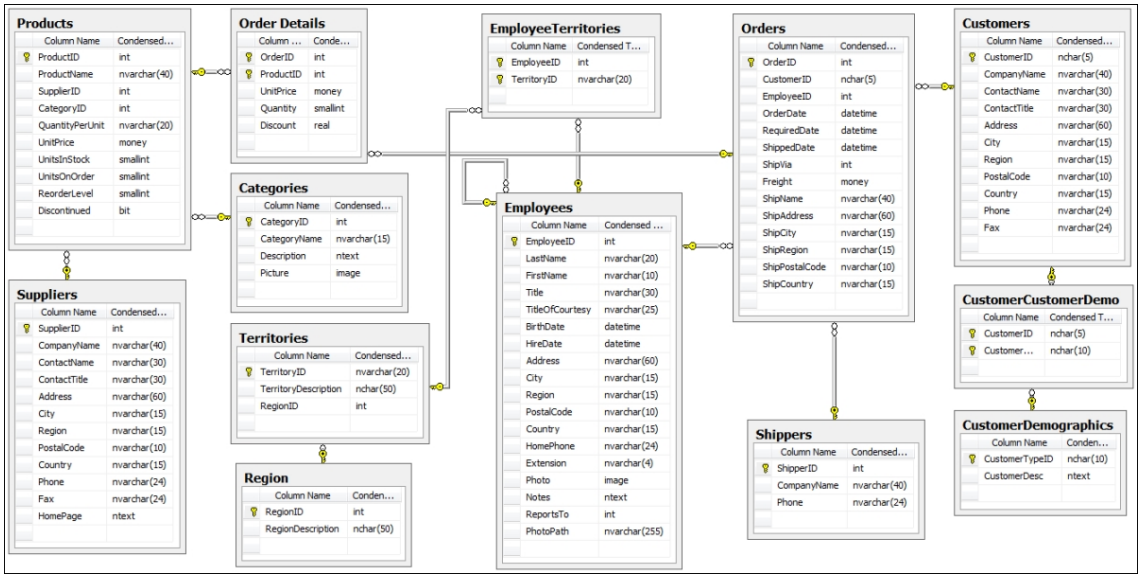

Fig. 4.1: Northwind Database Schema

```
SELECT ContactName, CompanyName
FROM Customers WHERE City = 'London';
```

```
Purpose: Evaluates basic column and table extraction.
```

- Inference: The algorithm correctly identifies the `'Customers'` table and the columns `'ContactName'`, `'CompanyName'` and `'City'`.
- Observation: The algorithm performs well in simple query structures without joins.

- **Aggregation with GROUP BY**:

```
SELECT EmployeeID, COUNT(OrderID) AS TotalOrders
FROM Orders GROUP BY EmployeeID;
```

```
Purpose: Tests the algorithm's handling of aggregate functions
and GROUP BY clauses.
```

- Inference: Successfully captures `'EmployeeID'` and assigns the require column weights and the aggregate function on 'OrderID'.
- Observation: The algorithm performs well in simple query structures with GROUP BY.

- **JOIN Operation**:

```
SELECT C.CustomerID, O.OrderID FROM
Customers C INNER JOIN Orders O ON C.CustomerID = O.CustomerID;
```

Purpose: Assesses the algorithm's ability to handle JOINs and infer relationships between tables.

  - Inference: Identifies the JOIN between 'Customers' and 'Orders' but it will struggle with determining the primary and secondary tables without schema knowledge.
  - Improvement: Enhancements are needed in JOIN interpretation and primary table detection in multi-table queries.

- **Complex Query with Multiple Joins and Subquery**:

```
SELECT E.LastName, C.CategoryName, COUNT(P.ProductID) AS TotalProducts
FROM Employees E
JOIN Orders O ON E.EmployeeID = O.EmployeeID
JOIN [Order Details] OD ON O.OrderID = OD.OrderID
JOIN Products P ON OD.ProductID = P.ProductID
JOIN Categories C ON P.CategoryID = C.CategoryID
WHERE E.Country = 'USA'
GROUP BY E.LastName, C.CategoryName;
```

Purpose: Tests complex JOIN operations, subqueries, and GROUP BY handling.

  - Inference: The complex relationships are partially captured, but the algorithm misinterprets the hierarchical structure due to multiple JOINs.
  - Improvement: Handling nested joins, identifying the parent and child tables based on multiple join relations and accurately placing columns in the hierarchy in complex queries needs improvement

- **Query with Subquery in WHERE Clause**:

```
SELECT ProductName FROM Products
WHERE ProductID IN
(SELECT ProductID FROM [Order Details] WHERE Quantity > 10);
```

Purpose: Evaluates handling of subqueries within WHERE clauses.

  - Inference: The algorithm correctly identifies the subquery and places it in the right hierarchy.
  - Observation: The algorithm performs well in simple subqueries within WHERE clauses.

- **Nested Subqueries**:

```
SELECT SupplierID, CompanyName FROM Suppliers
WHERE EXISTS (SELECT ProductID FROM Products
WHERE SupplierID = Suppliers.SupplierID AND UnitsInStock > 0);
```

```
Purpose: Assesses algorithm's ability to handle nested subqueries
and EXISTS clause..
```

  - Inference: The algorithm struggles with nested subqueries and the EXISTS clause.
  - Observation: The algorithm's current design has limited capability in handling deeply nested subqueries.

4.2.1 General Observations

The algorithm performs well with simple queries, subqueries and basic JOIN operations, correctly identifying tables and columns. But for complex queries involving multiple joins, nested subqueries pose significant challenges. While basic queries are accurately processed, complex queries often result in partial or incorrect hierarchical structures.

CHAPTER 5

RELATED WORK

To the best of our knowledge, there is only our previous work in querying SQL using hierarchies [11], in fact, the relational model replaced the hierarchical model and is widely considered an improved successor. But there has been previous research in querying with input types that can be broadly classified into several categories.

Query relaxation/approximation - One way to loosen the tight coupling of the input type to the data is to relax the path expressions in a query or approximately match them to the data within a given edit distance [13–15]. Though such techniques work well for small variations in data structure or values, there can be a *very large* edit distance among the same data organized in different structures, which we would like to consider as the same data. Relaxing a query to explore all data shapes within a large edit distance is overly permissive, and includes many shapes which do not have the same data. Query correction [16] and refinement [17] approaches are also best at exploring only small changes to the data.

Declarative transformations - There are declarative languages for specifying transformations of (hierarchical) data [18,19]. However, each transformation depends on the hierarchy of the input and would have to be re-programmed for a different hierarchy. It would be more desirable if a programmer could simply declare the desired hierarchy in a single guard.

Existing query languages - In a broad sense a database query has an input type and an output type; the query transforms data from the input to the output type. The input type is either a generic type, *e.g., Any*, or (a subset of) a database's schema. In languages for schemaless databases, like XQuery and Cypher, the input type is generic. There is no compiler type check for the input type, instead a query will evaluate on any data collection, producing an empty result if a path expression in the query fails to navigate to desired

data.[1] In languages for databases that have a schema, such as SQL, the input type is the names of tables and columns that appear in the query, which is a subset of a schema. The compiler checks the input type and generates an error if there is a mismatch.

Schema integration - Data can be integrated from one or more source schemas to a target schema by specifying a mapping to carry out a specific, fixed transformation of the data [20]. Once the data is in the target schema, there is still the problem of queries that need data in some schema other than the target schema. In some sense schema mediators integrate data to a fixed schema, which is the starting point for what query guards do. The different problem leads to a difference in techniques used to map or transform the data. For instance, tuple-generating dependencies (TGDs) are a popular technique for integrating schemas [21, 22]. Part of a TGD is a specification of the source structure from which to extract the data. Specifying the source schema will not work for a query guard, a query guard must be agnostic about the schema and work for any given schema (work in the sense that the input type can be matched or the matching produces information about potential data loss or errors). A second concern for query guards is that the transformation must be fully automatic. A third difference is the need to determine potential information loss, which is an important part of a query guard, but absent from such mappings for data integration. For schema mediation, if a programmer programs a data transformation that loses information, that information is gone and subsequent queries on the transformed data will never know about the information loss. Fan and Bohannon explored preserving information in data integration, namely by describing schema embeddings that ensure invertible mappings that are query preserving [23]. Query guards focus on an important special case of the mappings they investigated. Query preservation concerns all possible queries, while query guards are designed to check a single query. Our approach for quickly determining whether a mapping is invertible (or in our terminology reversible) is based on the concept of *closeness*, and in those cases where mapping is not reversible we can identify weaker, but still useful classes of mappings that permit some information loss.

[1] XML and Graph schema specifications are used and checked for data modification, rather than (read) query evaluation.

Finally we note that our research focuses only on the *structure, not the semantics,* of the data because Semantic Web technologies, *i.e.,* ontologies, already address the orthogonal semantic matching problem. Hence, solutions developed by the Semantic Web community can be used to semantically match in plug-and-play queries.

This thesis is a continuation and expansion of the ideas presented researched by Swami *et al.* [11]. That work laid the groundwork for the use of conceptual models in SQL queries, emphasizing the advantages of a hierarchical model in query formulation. In our approach, a conceptual model, termed a "query guard," is paired with an SQL query, transforming it into a plug-and-play query. This methodology enhances portability, type safety, simplicity, and resilience of SQL queries.

Building on this concept, our research delves deeper into the automatic inference of hierarchical input types for SQL queries. We extend the idea of query guards by implementing an algorithm that infers these guards without explicit schema knowledge, thereby addressing a significant limitation in the original work.

CHAPTER 6

CONCLUSIONS

In this thesis, we implemented an algorithm to infer hierarchical input types for SQL queries, both when schema information is available and when it is not. Our method leverages ANTLR for parsing SQL queries and a custom listener class for constructing a hierarchical guard structure. We tested our method with the Northwind database and found it works well for simple queries including simple selections, aggregations, joins, and subqueries. It can identify tables and columns correctly and manage basic connections between tables. However, challenges were observed in complex scenarios involving multiple joins, nested subqueries, and advanced SQL features. In such cases, the algorithm struggled to accurately infer the hierarchical structure, highlighting areas for improvement

To enhance the robustness and accuracy of the Guard Generation Algorithm, future work could focus on the following areas.

- **Improved Handling of Complex Queries**: Enhancing the algorithm to better manage complex queries involving multiple joins and nested subqueries. This could involve developing more sophisticated methods for identifying primary and secondary tables and accurately placing columns in the hierarchy in multi-table queries.

- **Advanced SQL Features**: Expanding the algorithm's capability to handle more advanced SQL features such as window functions, common table expressions (CTEs), and recursive queries. This would increase the algorithm's applicability to a wider range of real-world scenarios.

- **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.

- **Testing with Diverse Database**: Conducting further evaluations using a wider range of databases and schemas, beyond the Northwind sample database, to test the algorithm's versatility and adaptability to different data models.

- **User Interface Development**: Developing a user-friendly interface for the algorithm that could be integrated into database management tools, enhancing its accessibility and ease of use for database administrators and developers.

By addressing these areas, the Guard Generation Algorithm can be further refined to effectively handle a broader spectrum of SQL queries and become a more versatile tool.

REFERENCES

[1] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, p. 377–387, jun 1970. [Online]. Available: https://doi.org/10.1145/362384.362685

[2] Y. Silva, I. Almeida, and M. Queiroz, "Sql: From traditional databases to big data," 02 2016, pp. 413–418.

[3] D. D. Chamberlin and R. F. Boyce, "Sequel: A structured english query language," in *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET '74. New York, NY, USA: Association for Computing Machinery, 1974, p. 249–264. [Online]. Available: https://doi.org/10.1145/800296.811515

[4] C. E. Dyreson and S. S. Bhowmick, "Plug-and-play queries for temporal data sockets," in *Flexible Query Answering Systems*, H. Christiansen, H. Jaudoin, P. Chountas, T. Andreasen, and H. Legind Larsen, Eds. Cham: Springer International Publishing, 2017, pp. 124–136.

[5] A. Jain, C. Dyreson, and S. S. Bhowmick, "Generating plugs and data sockets for plug-and-play database web services," in *Cooperative Information Systems*, M. Sellami, P. Ceravolo, H. A. Reijers, W. Gaaloul, and H. Panetto, Eds. Cham: Springer International Publishing, 2022, pp. 279–288.

[6] C. Dyreson and S. Zhang, "The benefits of utilizing closeness in xml," in *2008 19th International Workshop on Database and Expert Systems Applications*, 2008, pp. 269–273.

[7] C. E. Dyreson, S. S. Bhowmick, and K. Mallampalli, "Using xmorph to transform xml data," *Proc. VLDB Endow.*, vol. 3, no. 1–2, p. 1541–1544, sep 2010. [Online]. Available: https://doi.org/10.14778/1920841.1921033

[8] C. Dyreson, S. Bhowmick, A. R. Jannu, K. Mallampalli, and S. Zhang, "Xmorph: A shape-polymorphic, domain-specific xml data transformation language," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, 2010, pp. 844–847.

[9] C. E. Dyreson, S. S. Bhowmick, and R. Grapp, "Querying virtual hierarchies using virtual prefix-based numbers," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 791–802. [Online]. Available: https://doi.org/10.1145/2588555.2610506

[10] ——, "Virtual exist-db: Liberating hierarchical queries from the shackles of access path dependence," *Proc. VLDB Endow.*, vol. 8, no. 12, p. 1932–1935, aug 2015. [Online]. Available: https://doi.org/10.14778/2824032.2824104

[11] S. Swami, S. Aryal, S. S. Bhowmick, and C. Dyreson, "Using a conceptual model in plug-and-play sql," in *Conceptual Modeling*, J. P. A. Almeida, J. Borbinha, G. Guizzardi, S. Link, and J. Zdravkovic, Eds. Cham: Springer Nature Switzerland, 2023, pp. 145–161.

[12] V. Yesin, M. Karpinski, M. Yesina, V. Vilihura, and K. Warwas, "Ensuring data integrity in databases with the universal basis of relations," *Applied Sciences*, vol. 11, no. 18, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/18/8781

[13] S. Amer-Yahia, S. Cho, and D. Srivastava, "Tree Pattern Relaxation," in *EDBT*, 2002, pp. 496–513.

[14] N. Augsten, M. H. Böhlen, and J. Gamper, "The q-gram distance between ordered labeled trees," *ACM Trans. Database Syst.*, vol. 35, no. 1, 2010.

[15] Y. Kanza and Y. Sagiv, "Flexible Queries over Semistructured Data," in *PODS*, 2001.

[16] S. Cohen and T. Brodianskiy, "Correcting Queries for XML," *Inf. Syst.*, vol. 34, no. 8, pp. 690–710, 2009.

[17] A. Balmin, L. S. Colby, E. Curtmola, Q. Li, and F. Ozcan, "Search Driven Analysis of Heterogenous XML Data," in *CIDR*, 2009.

[18] S. Krishnamurthi, K. E. Gray, and P. T. Graunke, "Transformation-by-Example for XML," in *PADL*, 2000, pp. 249–262.

[19] T. Pankowski, "A High-Level Language for Specifying XML Data Transformations," in *ADBIS*, 2004, pp. 159–172.

[20] M. Bhide, M. Agarwal, A. Bar-Or, S. Padmanabhan, S. Mittapalli, and G. Venkatachaliah, "XPEDIA: XML ProcEssing for Data Integration," *PVLDB*, vol. 2, no. 2, pp. 1330–1341, 2009.

[21] H. Jiang, H. Ho, L. Popa, and W.-S. Han, "Mapping-driven XML Transformation," in *WWW*, 2007, pp. 1063–1072.

[22] R. Fagin, L. Haas, M. Hernandez, R. Miller, L. Popa, and Y. Velegrakis, "Clio: Schema Mapping Creation and Data Exchange," in *LNCS 5600*, 2009, pp. 198–236.

[23] W. Fan and P. Bohannon, "Information preserving xml schema embedding," *ACM Trans. Database Syst.*, vol. 33, no. 1, 2008.