

Utah State University

**DigitalCommons@USU**

---

All Graduate Theses and Dissertations

Graduate Studies

---

8-2021

## Plug-and-Play SQL

Shubham Swami

*Utah State University*

Follow this and additional works at: <https://digitalcommons.usu.edu/etd>



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Swami, Shubham, "Plug-and-Play SQL" (2021). *All Graduate Theses and Dissertations*. 8192.

<https://digitalcommons.usu.edu/etd/8192>

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact [digitalcommons@usu.edu](mailto:digitalcommons@usu.edu).



Plug-and-Play SQL

by

Shubham Swami

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

---

Dan Watson, Ph.D.  
Committee Member

---

Haitao Wang, Ph.D.  
Committee Member

---

D. Richard Cutler, Ph.D.  
Interim Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY  
Logan, Utah

2021

## ABSTRACT

## Plug-and-Play SQL

by

Shubham Swami, Master of Science

Utah State University, 2021

Major Professor: Curtis Dyreson, Ph.D.

Department: Computer Science

Programmers invest time and effort into developing a set of queries that meet the needs of the database's end-users. As new users and new user needs arise over time, new queries must be developed. The objective of this research project is to provide a system for plug-and-play queries for SQL, which is a standard language for accessing and manipulating databases along with the implementation of the proposed objective.

A plug-and-play query is a freestanding query that can determine if it can be evaluated without user intervention. We use hierarchies to improve SQL querying in a way that eliminates the need to write a view to construct virtual tables or a set of tables to run a query. The hierarchy would be a declarative specification of the desired shape of data rather than a description of how the data is organized. The advantage is that the common use of logical or semantic pointers in the SQL queries is eliminated and a natural way to group data for aggregation is provided. The plug-and-play queries have several advantages, they are portable and can be used to evaluate any data source.

(48 pages)

## PUBLIC ABSTRACT

## Plug-and-Play SQL

Shubham Swami

We present an efficient model to retrieve data from a database by implementing plug-and-play queries using the query guards. The model is efficient in the sense that it saves time when writing a query and promotes query portability and reuse. A plug-and-play query is a freestanding query that can couple to any data socket and self determine whether it can be evaluated reliably on the data. We use hierarchies to improve SQL querying in a way that eliminates the need to write a view to construct virtual tables or a set of tables to run a query. The hierarchy is a declarative specification of the desired shape of data rather than a description of how the data is organized. The advantage is that the common use of logical or semantic pointers in the SQL queries is eliminated and a natural way to group data for aggregation is provided. The plug-and-play queries have several advantages, they are portable and can be used to evaluate any data source.

## ACKNOWLEDGMENTS

I would like to offer my sincere gratitude to my thesis advisor, Dr. Curtis Dyreson. I am grateful for his unfailing support and patience with me in answering my questions and guiding me in the right direction. It was all his guidance and assistance that helped me through the outcome of this thesis.

Besides my advisor, I would like to thank my committee members, Dr. Haitao Wang and Dr. Dan Watson, for being such wonderful mentors.

In addition, I would like to thank my family and friends, for supporting me throughout my academic pursuits and through my life in general.

Shubham Swami

## CONTENTS

	Page
ABSTRACT . . . . .	ii
PUBLIC ABSTRACT . . . . .	iii
ACKNOWLEDGMENTS . . . . .	iv
LIST OF TABLES . . . . .	vii
LIST OF FIGURES . . . . .	viii
1 INTRODUCTION . . . . .	1
1.1 Problem Motivation . . . . .	1
1.2 Single motivating example . . . . .	4
1.3 Contribution . . . . .	6
2 EXAMPLES AND USE CASES . . . . .	7
2.1 The Baseball Database . . . . .	7
2.2 Data Cleaning . . . . .	8
2.3 The Baseball Database Schema: . . . . .	8
2.4 Queries . . . . .	8
3 PLUG-AND-PLAY SQL IMPLEMENTATION . . . . .	15
3.1 ANTLR . . . . .	15
3.1.1 Grammar . . . . .	15
3.1.2 Lexer . . . . .	17
3.1.3 Parser . . . . .	18
3.1.4 Listener . . . . .	18
3.1.5 ANTLR Summary . . . . .	18
3.2 Parsing the Query Guard . . . . .	19
3.3 Graph of Foreign Keys . . . . .	21
3.3.1 Edge . . . . .	21
3.3.2 Path . . . . .	22
3.3.3 Paths Table . . . . .	23
3.3.4 Graph . . . . .	23
3.4 Implementation Description . . . . .	23
3.4.1 Connect to Database . . . . .	23
3.4.2 Implement a query guard . . . . .	25
3.4.3 Parsing the Query Guard . . . . .	26
3.4.4 Obtaining the AST . . . . .	26
3.4.5 The Custom Listener . . . . .	26
3.4.6 Populating the Pattern Tree . . . . .	27
3.4.7 Compute the shortest paths . . . . .	27

3.4.8	Generate SQL Queries . . . . .	28
3.4.9	Evaluate the EXPLAIN plan . . . . .	28
3.4.10	Ordering the queries on the number of rows they produce: . . . . .	28
3.4.11	Emit the From Clause . . . . .	29
3.4.12	Execute Final Queries . . . . .	29
4	PLUG-AND-PLAY EVALUATION . . . . .	31
4.0.1	The Column Name Ambiguity . . . . .	31
4.0.2	Query Identification . . . . .	32
4.0.3	Solution . . . . .	32
5	RELATED WORK . . . . .	33
6	CONCLUSIONS AND FUTURE WORK . . . . .	36
6.1	Conclusions . . . . .	36
6.2	Future Work . . . . .	36
6.2.1	The column name ambiguity . . . . .	36
6.2.2	DBMS Support . . . . .	36
6.2.3	Multiple guard clauses . . . . .	37
6.2.4	The WITH clause . . . . .	37

## LIST OF TABLES

Table	Page
3.1 Lexical Analysis . . . . .	17
4.1 Cost Analysis . . . . .	31



## LIST OF FIGURES

Figure	Page
1.1 Querying hierarchical data . . . . .	2
1.2 Using a query guard . . . . .	2
2.1 Lahman's baseball database complete schema . . . . .	9
3.1 Plug and Play SQL- Code Structure . . . . .	16
3.2 ANTLR Compiler . . . . .	16
3.3 AST generated from tokens . . . . .	18
3.4 Lexical Analysis . . . . .	21
3.5 Abstract Syntax Tree- Query Guard . . . . .	22
3.6 Join Graph: Code Structure . . . . .	22
3.7 Implementation Flowchart . . . . .	24
3.8 Database Flowchart . . . . .	24
3.9 Establishing connection to the database . . . . .	25
3.10 Enter Query Guard to generate queries . . . . .	26
3.11 Generated Queries . . . . .	29
3.12 Results . . . . .	30

## CHAPTER 1

### INTRODUCTION

#### 1.1 Problem Motivation

The goal of this research project is to reduce the effort and time required to write accurate and efficient queries. The technique is to couple a query with a *query guard* creating a *plug-and-play query*. The idea of a plug-and-play query is similar to that of a plug-and-play device. Such a device can be plugged into any socket and if the socket provides the necessary (electrical) input or other required input, then the device will play. Similarly, a plug-and-play query can be plugged into any data source and the query will “play” producing a desired result.

Query guards for plug-and-play queries have been previously explored in the context of hierarchical data [1–7]. This research proposed the idea of a hierarchical query guard. The guard allows the query to couple to any hierarchy that can be converted, cast, or coerced to the type (hierarchy) needed by the query. Figure 1.1 shows how a non-plug-and-play query is evaluated on hierarchical data. The programmer must ensure that the type (hierarchy) of the data matches what the query expects by specifying the appropriate path expressions in the query. If the right path expressions are not given the query will fail in the sense that it produces an empty result; it does not produce an error since it is common in hierarchical data for the hierarchy to terminate with missing data (the data is referred to as semi-structured, meaning it lacks a completely defined structure, such data is common in the real world). The query guard approach, shown in Figure 1.2, is to pair a query with a guard. The guard protects the query by checking whether the data can be transformed, without losing information, to the type (hierarchy) needed by the query. Some transformations lose information when the hierarchy is manipulated, but the guard can alert the programmer to lossy transformations.

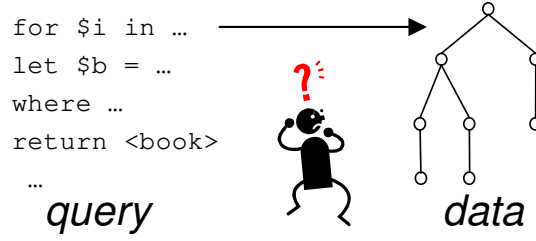


Fig. 1.1: Querying hierarchical data

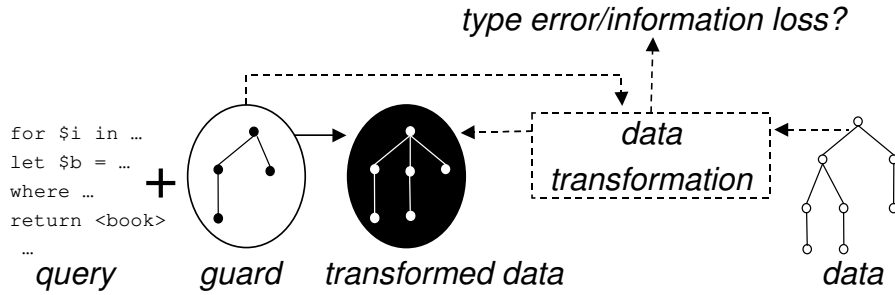


Fig. 1.2: Using a query guard

In this thesis, we develop query guards for SQL queries. Unlike previous research the SQL data model is not a hierarchy, rather it is a set of relations or a so-called “flat” structure in contrast to a hierarchy. We will call an SQL query with a query guard a *plug-and-play SQL query* or just a plug-and-play query. A query guard declares the structure of the data needed by the query. Surprisingly, this structure is declared as a hierarchy even for SQL queries for the reasons given below. A query guard turns an ordinary SQL query into a plug-and-play query and can be reused.

Data can be represented in several shapes. For instance, book data could be represented in a single (universal) table, or several tables related by foreign keys. A programmer should not have to first write a view to construct a virtual table or set of tables to run an SQL query. Instead, SQL queries should be plug-and-play queries that automatically (virtually) transform the data to that needed by the query.

Surprisingly we can use hierarchies to improve SQL querying. Unlike in a hierarchical model, our hierarchy will be a declarative specification of the desired shape of the data

rather than a description of how the data is organized. A hierarchy can eliminate the common use of “logical or semantic pointers” in SQL queries (*e.g.*, key to foreign key joins) and provide a natural way to “group” data for aggregation. For example, the following Symbiota query finds who collected *Asteraceae* specimens in 2018.

```
SELECT botanist.name
FROM taxa, collection, botanist
WHERE taxa.tid = collection.tid AND botanist.id = collection.pid
      AND taxa.taxon = 'Asteraceae' AND collection.year = 2018
```

The query does a join between the `taxa`, `collection`, and `botanist` tables and applies the appropriate selection conditions, and projects the name. The query explicitly uses logical pointers (foreign key to key associations) from the `taxa` table to the `collection` and `botanist` table. This query can be simplified with an SQL query guard as follows.

```
GUARD botanist.name {
    taxon {
        collection.year
    }
}
SELECT name
WHERE taxon = 'Asteraceae' AND year = 2018
```

The query uses the graph of foreign key relationships between tables to build a hierarchy, similar to using a `NATURAL JOIN` to join the tables. If there are several `name` columns in the database, the column is disambiguated in the guard. However, the above query is portable to data collections that have different shapes (*i.e.*, we do not care how many steps are involved in “joining” the tables) and naturally “groups” the data. Suppose for instance we only wanted those collectors who collected more than 40 specimens then we could modify the query as follows.

```
GUARD botanist.name {
```

```

        taxon {
            collection.year
        }
    }

SELECT botanist.name
WHERE taxon = 'Asteraceae' AND collection.year = 2018
      AND COUNT(taxon) > 40

```

Querying against a hierarchy simplifies grouping and aggregation (as in XQuery) which in SQL is specified in separate clauses (a COUNT is forbidden in a WHERE clause).

## 1.2 Single motivating example

In this example, we explore the use of a query guard from its specification through its implementation. This illustrates the use and advantages of query guards. Suppose that we want to retrieve the name of every person who has been the manager for the team “New York Yankees”. Note that we construct this query without much concern over how the data is physically structured, rather our goal is to specify a query that will *work on any data source*.

```

GUARD manager {
    firstName
    lastName
    teams {
        name
    }
}

SELECT DISTINCT firstName, lastName
WHERE teamName = 'New York Yankees'
ORDER BY firstName

```

Several names indicate how the query is intended to be used. There is the first and last name of a manager and a team name. Moreover, the manager must have managed the indicated team in the given year.

The first step in evaluating the guard is to semantically match the names in the guard to the names in the database. In this thesis we assume a perfect semantic match, our focus is on structural matching. If this guard is evaluated on the Lahman Baseball database (a history of baseball data from its inception through 2020) then labels in the guard match to columns or tables in the database, *e.g.*, `firstName` matches `nameFirst` in the `players` table. For the baseball database, the semantic matching also yields that the data needed involves three tables: the `players` table, the `managers` table, and the `teams` table.

The next step is to determine how these three tables are related. We use the graph of foreign keys to determine the shortest path among the tables. In the baseball database, the `managers` table has a foreign key into both the `teams` and `players` table, so all three tables are related through the `managers` table.

The final step is to generate a query on the baseball database. The generated query is given below.

```
SELECT  DISTINCT nameFirst, nameLast
FROM ( SELECT DISTINCT people."namelast", people."namefirst"
      FROM teams LEFT JOIN managers ON (managers.lgid = teams.lgid
      AND manager.yearid = teams.yearid
      AND maager.teamid = teams.teamid)
      LEFT JOIN people ON managers.playerid = people.playerid
    ) AS generatedFromClause
WHERE name='New York Yankees'
ORDER BY nameLast
```

Finally, if the user were to write the query manually, they would issue the following query.

```
SELECT DISTINCT nameFirst, nameLast
```

```
FROM people JOIN managers ON (people.playerID = managers.playerID)
      JOIN teams ON (teams.yearID = managers.yearID
                    AND teams.lgID = managers.lgID
                    AND teams.teamID = managers.teamID)
WHERE teams.name = 'New York Yankees'
ORDER BY nameLast
```

But this query would work only on the Lahman baseball database. If the database were structured differently, then the query would fail because the tables and joins between the tables would differ. In contrast, the query with the guard would succeed since it could adapt to the differently structured schema.

### 1.3 Contribution

This research project creates a system to evaluate plug-and-play queries for SQL, the most widely used database query language. In this thesis, we show that a hierarchical table structure can be used to generate SQL queries that are capable of retrieving data efficiently. We also provide a robust querying system to the programmers which would save their time and effort to write SQL queries that can retrieve data accurately. Through this research project, we will analyze, evaluate and describe the concept of Plug-and-Play SQL.

## CHAPTER 2

### EXAMPLES AND USE CASES

#### 2.1 The Baseball Database

We have used Lahman's Baseball Database for our project, This is an open-source collection of baseball data and is frequently updated and maintained. We configured the baseball data on the Postgres Relational Database Management System. This database has 29 tables as shown in Figure 2.1, however, we have used only a few tables from this database for our project. The open-source data is available in the form of comma-separated values (csv files) and since there is a regular update to these data values, adjustments were needed to achieve normalization to use it on this project.

Some of the adjustments are:

1. The data for the *divisions* and the *leagues* table was not explicitly provided, therefore these tables were manually added after extracting the csv data.
2. Initially, no constraints were included in the database, so the primary key and the foreign keys referencing the other tables in the database were manually added.
3. Upon adding the foreign keys and primary keys there were a lot of data anomalies and this resulted in a data loss.
4. The Postgres DBMS does not allow column names to start with a numeric character because of the reason that it becomes easy for the parser to decipher the SQL statements and the execution of these statements becomes faster too. For this reason, the column names that started with numbers were refactored *e.g.*, columns *2B* and *3B* were refactored to *H2B* and *H3B* respectively.



## 2.2 Data Cleaning

While setting up the database, when we added the key constraints there were a lot of cases where the referential integrity was violated. We had to remove all the conflicting data in the database. The *playerID*'s like "thompar01", "thompan01" and a few others were present in the tables like fieldingof, pitching and others but not in the people table (people table is the master table for this database) and so these tuples were truncated.

## 2.3 The Baseball Database Schema:

The schema for the baseball database is shown in Figure 2.1. However, we have confined our examples ( section 2.4) to a few tables (people, batting, pitching, appearances, managers, and teams table) for this project.

## 2.4 Queries

We have implemented a few queries and their corresponding plug-and-play queries to execute and evaluate the project. The running examples are as follows:

**Query 1:** Query to retrieve the name of every person who has been the manager for the team "New York Yankees".

### SQL Query:

```
SELECT DISTINCT nameFirst, nameLast
FROM (people JOIN managers ON
      (people.playerID = managers.playerID))
      JOIN teams ON (teams.yearID = managers.yearID AND
                    teams.lgID = managers.lgID AND teams.teamID = managers.teamID)
WHERE (teams.name = 'New York Yankees' AND yearID >= '1913') order by nameLast;
```

### Plug-and-Play Query:

```
GUARD
```

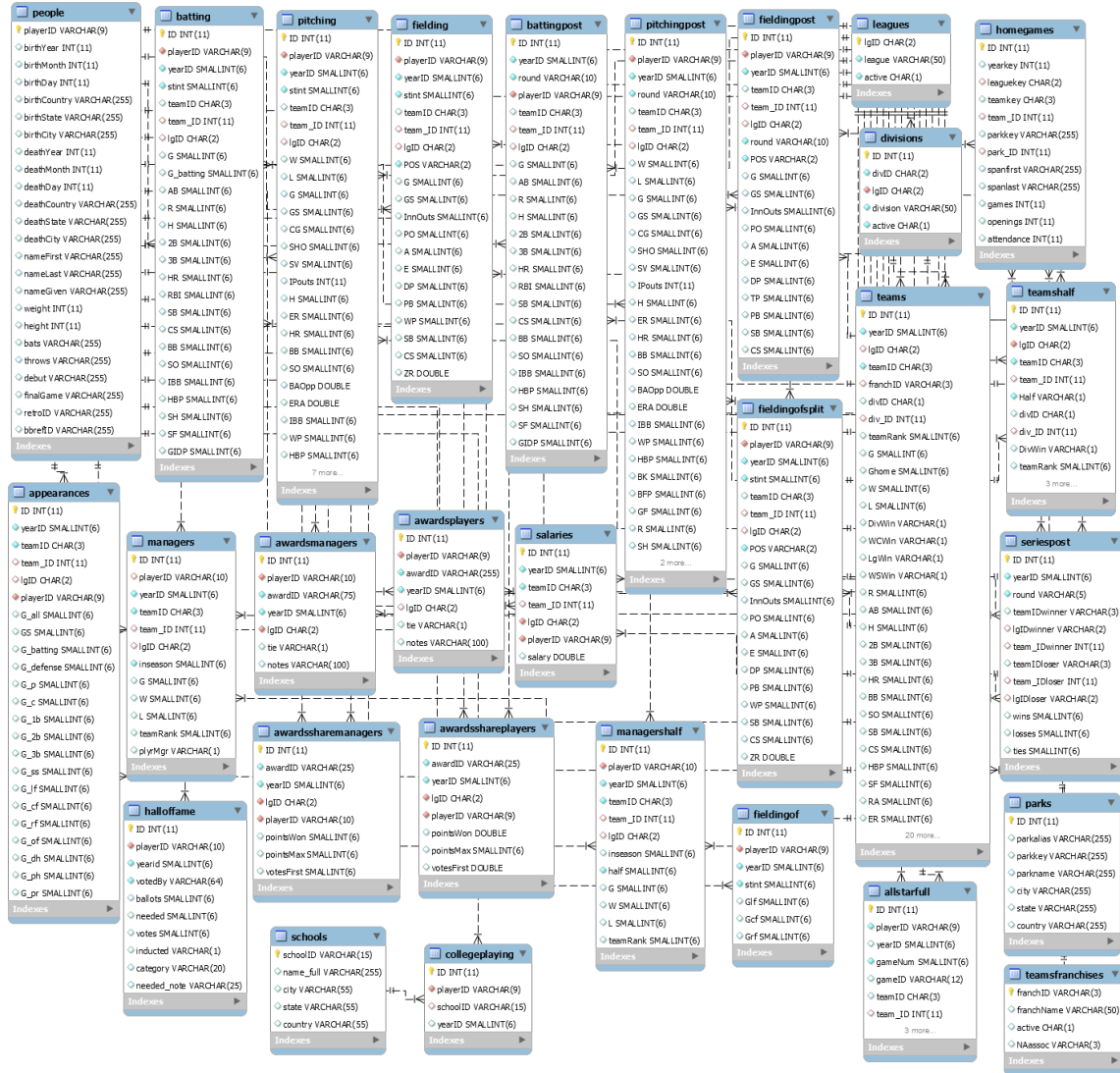


Fig. 2.1: Lahman's baseball database complete schema

```

managers{
    plyrmgr
    namefirst
    namelast{
        teams{
            yearID
            name
        }
    }
}

```

```

        }
    }

SELECT DISTINCT nameFirst, nameLast
WHERE (teams.name ='New York Yankees' AND yearID>= '1913')
ORDER BY nameLast;

```

**Query 2:** List the name of each player with more than 200 hits in a season in their career (hits made is the "H" column in the "batting" table).

**SQL Query:**

```

SELECT DISTINCT nameFirst, nameLast FROM people
JOIN batting ON people.playerID = batting.playerID
WHERE (batting.H >200 ) order by nameLast;

```

**Plug-and-Play Query:**

```

GUARD
batting {
    namefirst
    namelast
    H
}

SELECT DISTINCT nameFirst, nameLast
WHERE (H >200) order by nameLast;

```

**Query 3:** List all the pitchers who have a first name as "Roger".

**SQL Query:**

```

SELECT  nameFirst, nameLast from people
where nameFirst="Roger" AND playerID IN
(SELECT playerID from pitching);

```

**Plug-and-Play Query:**

```

GUARD
pitching {
  name{
    namefirst
    namelast
  }
}

SELECT DISTINCT nameFirst, nameLast
WHERE nameFirst ="Roger";

```

**Query 4:** List the first name and last name of every player that has played center field (use the “appearances” table) at any time in their career for the Atlanta Braves. List each player only once.

**SQL Query:**

```

SELECT DISTINCT  nameFirst, nameLast FROM people
JOIN  appearances ON people.playerID = appearances.playerID
WHERE (teamID="ATL" AND G_cf != 0 )
ORDER BY nameLast;

```

**Plug-and-Play Query:**

```

GUARD
appearances{
  g_cf
  team{
    teamID
  }
  namefirst
  namelast
}

```

```
SELECT DISTINCT nameFirst, nameLast
WHERE (teamID="ATL" AND G_cf != 0 ) ORDER BY nameLast;
```

**Query 5:** List the first name and last name of every player that has pitched for the team named the “Brooklyn Dodgers”. List each player only once.

**SQL Query:**

```
SELECT DISTINCT nameFirst, nameLast from people
WHERE playerID IN
    (SELECT masterID from pitching
     WHERE (teamID="BRO" AND
            (yearID = '1912' OR yearID= '1911' OR yearID>='1932' )));
```

**Plug-and-Play Query:**

GUARD

```
pitching {
    teamID
    yearID
    name {
        namefirst
        namelast
    }
}
```

```
SELECT DISTINCT nameFirst, nameLast
WHERE (teamID='BRO' AND (yearID = '1912' OR yearID= '1911' OR yearID>='1932'));
```

**Query 6:** Yankee Batters - For each year, list how many different players batted for the New York Yankees.

**SQL Query:**

```

SELECT yearID, COUNT(playerID) as 'Number of Batters' FROM batting
WHERE (teamID="NYA" AND yearID>="1913")
GROUP BY yearID ORDER BY yearID;

```

**Plug-and-Play Query:**

GUARD

batting {

    playerID

    yearID

    teamID

}

```

SELECT yearID, COUNT(playerID) as 'Number of Batters'
WHERE (teamID="NYA" AND yearID>="1913") GROUP BY yearID ORDER BY yearID;

```

**Query 7:** List the first and last names of people who pitched for “Montreal Expos”.

**SQL Query:**

```

SELECT DISTINCT m.nameFirst, m.nameLast FROM
people m,pitching a,teams t
WHERE (m.playerId = a.playerId
      AND t.teamId = a.teamId
      AND t.lgID = a.lgID
      AND t.name like '%Montreal Expos%'
      AND t.yearID = a.yearId);

```

**Plug-and-Play Query:**

GUARD

batting {

    hr

    name {

        nameFirst

```
        nameLast
      }
    team {
      name
    }
  }

SELECT DISTINCT nameFirst, nameLast
WHERE name like '%Montreal Expos%';
```

## CHAPTER 3

### PLUG-AND-PLAY SQL IMPLEMENTATION

In this section, we define the code structure for our application. Most of the code is written in Java and the ANTLR grammar has all the lexer and parser rules for SQL along with the proposed guard clause. The code structure for the application is shown in Figure 3.1. There are five main modules in the application, the **database** module is responsible for configuring the data source, establishing a connection, and extracting the data. The **grammar** module contains all the lexer and parser rules for the SQL and query guard, the parser and lexer codes that are generated by the ANTLR jar, and the custom listener, to parse the tree. The **data pull** module contains the logic to generate queries and evaluate each query based on its execution plan. The data pull model also re-formats the output and makes it suitable to display in the form of applications. The **join graph** module manipulates the paths and provides the infrastructure to obtain the shortest path between tables. Lastly, the **tree module** communicates with the listener and the data pull module to generate the queries, and glue them to the emitted from clause. These modules have been elaborated in individual sections below.

### 3.1 ANTLR

ANTLR is abbreviated as Another Tool for Language Recognition and is widely used to build languages, frameworks, and tools. In this project, we use ANTLR as a tool to generate the lexer and parser code for the implemented SQLite grammar. We use this parser to read, interpret and process the plug-and-play query. Figure 3.2 shows the working of the ANTLR jar.

#### 3.1.1 Grammar

Grammar is defined as a list of rules, namely the lexer and parser rules. The convention



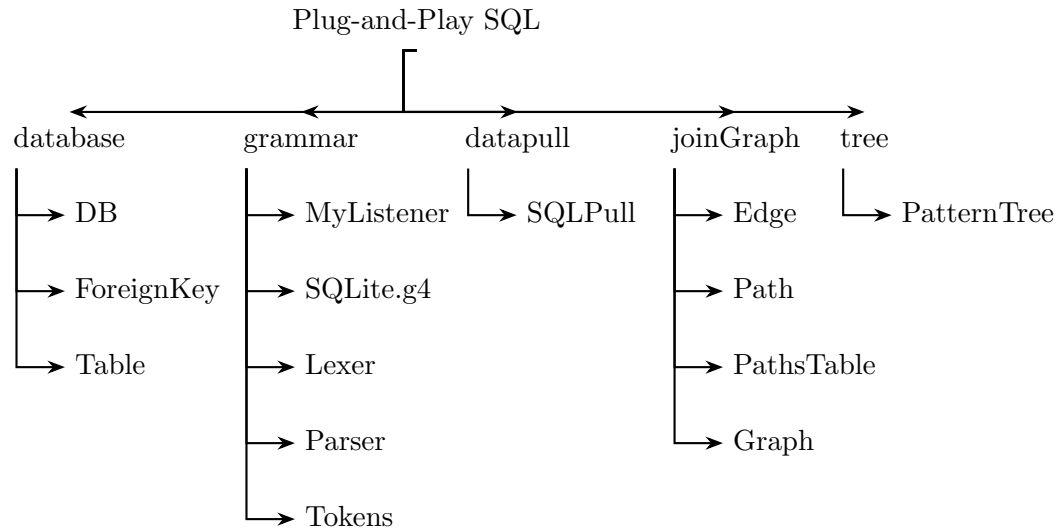


Fig. 3.1: Plug and Play SQL- Code Structure

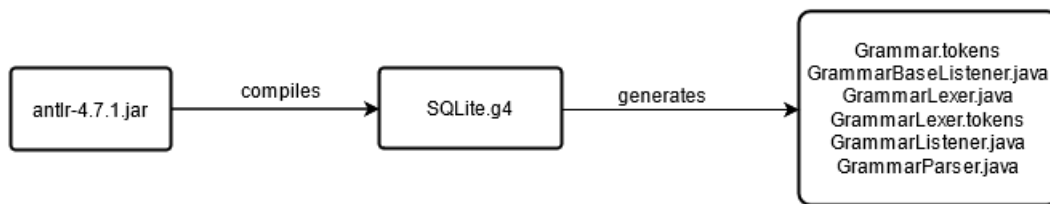


Fig. 3.2: ANTLR Compiler

followed is, writing the parser rules in lowercase characters and the lexer rules in uppercase characters. The syntax to write an ANTLR grammar is as follows:

```

grammar grammar_name;

@headers {...}

@members {...}

imports {...}

parser_rules{}

lexer_rules{}
  
```

The syntax for writing a rule is as follows:

```

Rule_Name : Rule_Definition ;
  
```

Type	Lexeme
K_GUARD	GUARD
IDENTIFIER	name
null	{
IDENTIFIER	hr
null	}

Table 3.1: Lexical Analysis

The ANTLR tool uses *SQLite.g4* grammar in this project and compiles it to generate the parser and lexer code.

To run the generated parser and lexer code we need the run time library of ANTLR along with the parser and lexer code. For this purpose, we have used antlr-4.7.1-complete.jar in our project.

### 3.1.2 Lexer

Lexer in ANTLR is like a prerequisite for the parser. A lexer can be defined as something that converts each character from the input string (plug-and-play query in this project) into a token.

A token is a meaningful translation of the input query string by the lexer and an alphanumeric string inside a token is a lexeme. *e.g.*, We have the plug-and-play query:

```

GUARD
  name {
    hr
  }

```

The lexer would obtain the Lexemes given in Table 3.1 from a plug-and-play query.

### 3.1.3 Parser

We can define a parser as a program that takes in the tokens as its input, evaluates these tokens for the rules, and creates a logical structure, called the Abstract Syntax Tree.

From Table 3.1 the parser takes in the tokens and generates the abstract syntax tree as shown in Figure 3.3.

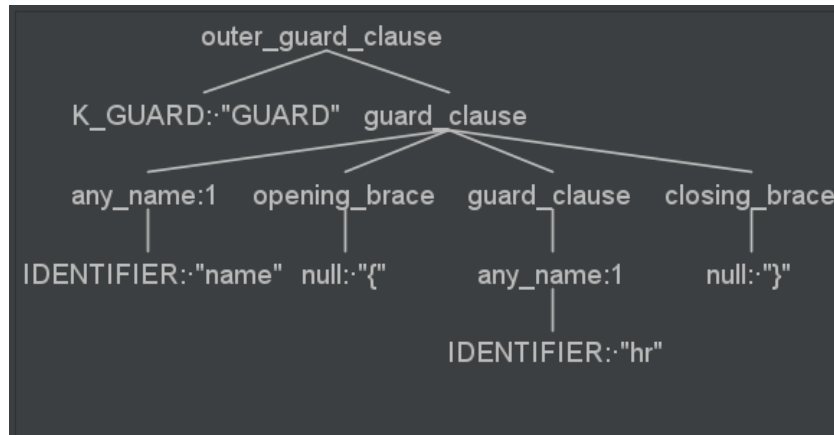


Fig. 3.3: AST generated from tokens

### 3.1.4 Listener

Since we have the AST generated by the parser, we will now move towards walking this tree structure. By walking this tree, we want to code the functionality for each node in the AST. To implement this we can write our custom listener class in java. The aforementioned listener class is described in detail in the implementation section.

### 3.1.5 ANTLR Summary

- The ANTLR grammar is a set of lexer and parser rules.
- The ANTLR run-time library compiles the grammar into useful programs, which include lexer and parser code.
- Lexer translates input string to tokens.
- The parser uses tokens from the lexer to generate an AST.

- A listener can be implemented to walk the AST and handle events corresponding to each node.

### 3.2 Parsing the Query Guard

A plug-and-play query is different from an ordinary query in the sense that it includes a guard clause that determines the hierarchy of the data. The guard clause is interpreted and compiled to emit a FROM clause. We glue the emitted FROM clause with the remaining portion of the query to generate an ordinary SQL query and retrieve the data.

The plug-and-play SQL query is different as it replaces the conditions required to join the tables, with a hierarchical structure enclosed in “{” and “}”.

Syntax:

```
GUARD {
    name1,name2 ... {
        name1, name2 ...
    }
}
```

Let’s compare an ordinary SQL query with a Plug and Play SQL Query. An example ordinary SQL query is given below.

```
SELECT
    DISTINCT p.nameFirst, p.nameLast
FROM
    people p,
    pitching a,
    teams t
WHERE
    p.masterId = a.masterId
    AND t.teamId = a.teamId
    AND t.lgID = a.lgID
```

```

AND t.name like '%Montreal Expos%'
AND t.yearID = a.yearId

```

The corresponding Plug-and-Play SQL query is given below.

GUARD

```

    batting {
        hr
        name {
            nameFirst
            nameLast
        }
    }
    team {
        name
    }
}

SELECT DISTINCT nameFirst, nameLast
WHERE name like '%Montreal Expos%';

```

The guard clause is interpreted by the parser because of the following rules defined in the ANTLR grammar.

```

outer_guard_clause[DB db]
:K_GUARD guard_clause
;

guard_clause
: (any_name (opening_brace guard_clause closing_brace)?)+
;

opening_brace

```

```

: '{'
;

```

```

closing_brace
: '}'
;

```

The grammar acts as input to the *antlr4.7.1.jar*. This jar file interprets and compiles the grammar into the corresponding lexer and parser.

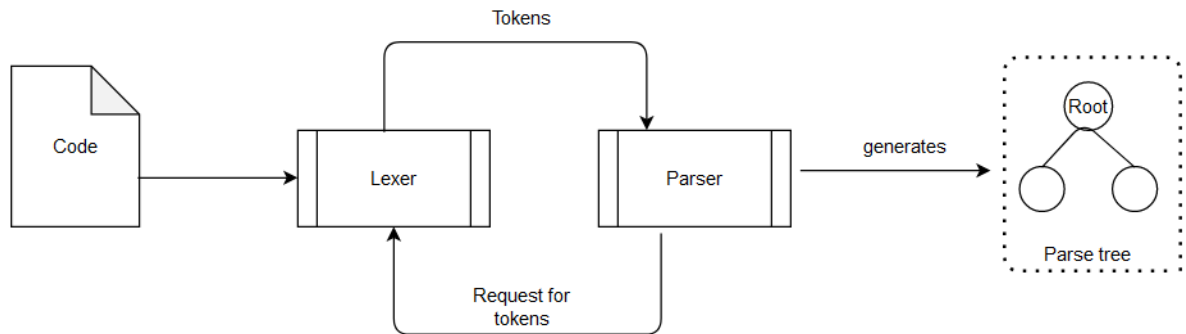


Fig. 3.4: Lexical Analysis

The generated parser is then used to parse the query guard and obtain the Abstract Syntax Tree as shown in Figure 3.5.

### 3.3 Graph of Foreign Keys

In this project, we have implemented a Graph structure to store, process, and retrieve paths between two tables in the database.

#### 3.3.1 Edge

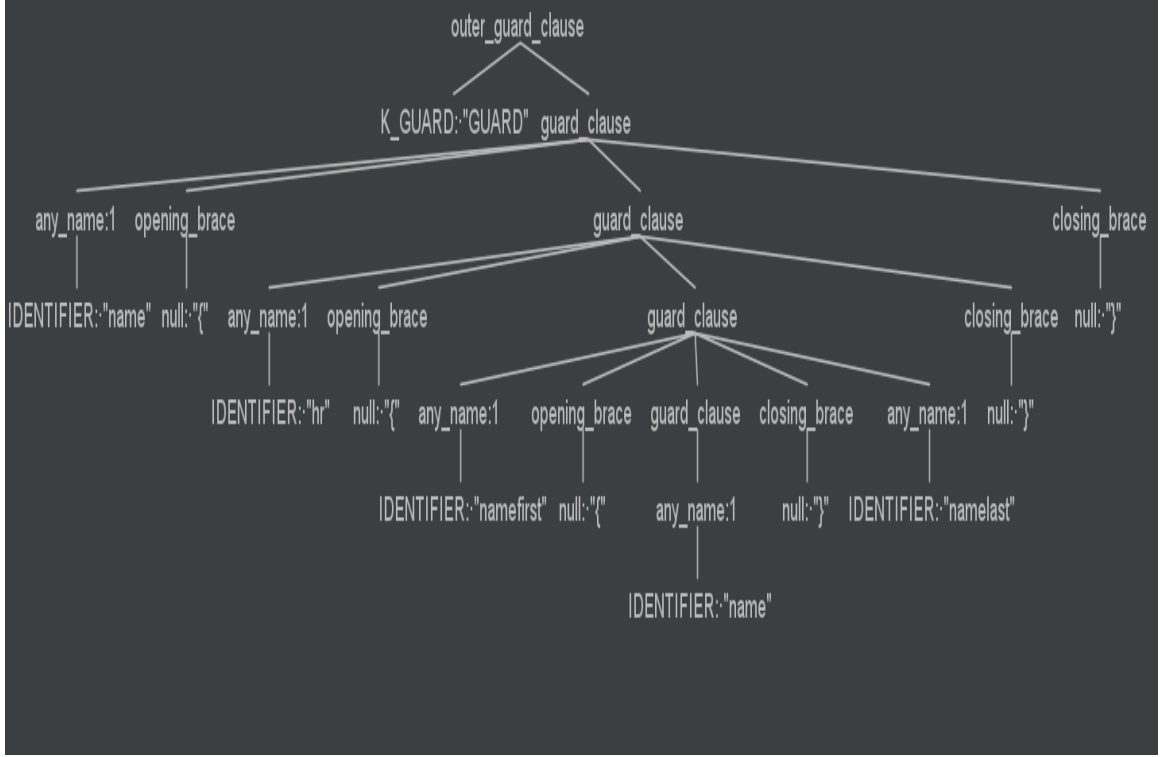


Fig. 3.5: Abstract Syntax Tree- Query Guard

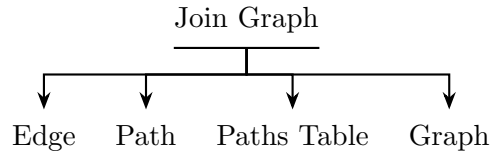


Fig. 3.6: Join Graph: Code Structure

An edge is a foreign key relationship between two vertices. In our code, we declare an Edge class that has a Foreign Key attribute that gets set in a constructor and we retrieve this edge or foreign key using a getter method.

### 3.3.2 Path

A path is a list of foreign keys. In subsection 3.3.1, we defined a foreign key relationship as an edge therefore a path in our code is a list of foreign keys. As its attributes, a path contains a *from table*, a *to table*, and the foreign key relationship between them.

The main function of the Path class is to generate a join condition in between the

two tables and join conditions are generated recursively after some processing on the list of foreign keys.

### 3.3.3 Paths Table

So, a paths table usually represents the shortest path that connects a pair of tables. The paths table initially stores all the paths between two tables but after some logical processing, the paths table is optimized to store the shortest paths.

### 3.3.4 Graph

This graph is a set of vertices and edges, where vertices are tables in the database. Our objective is to generate a join condition between two tables using the best path between them.

The graph of Foreign Keys represents the graph of join relationships established by the foreign keys. The Graph also plays an important role in computing the shortest paths.

## 3.4 Implementation Description

In this chapter, we will provide a detailed description of the project implementation. The flowchart to explain how the algorithm works has been described with the help of Figure 3.7.

A step by step description of how the plug-and-play query implementation is as follows:

### 3.4.1 Connect to Database

As stated earlier, we have used the baseball database by Sean Lehman for our project on the Postgres RDBMS.

The code has a database module that performs the following tasks:

1. Establishes a connection to the baseball database using JDBC and Postgres server.
2. The DB.java has been implemented as *Singleton Class* to ensure that only one instance exists throughout the project.



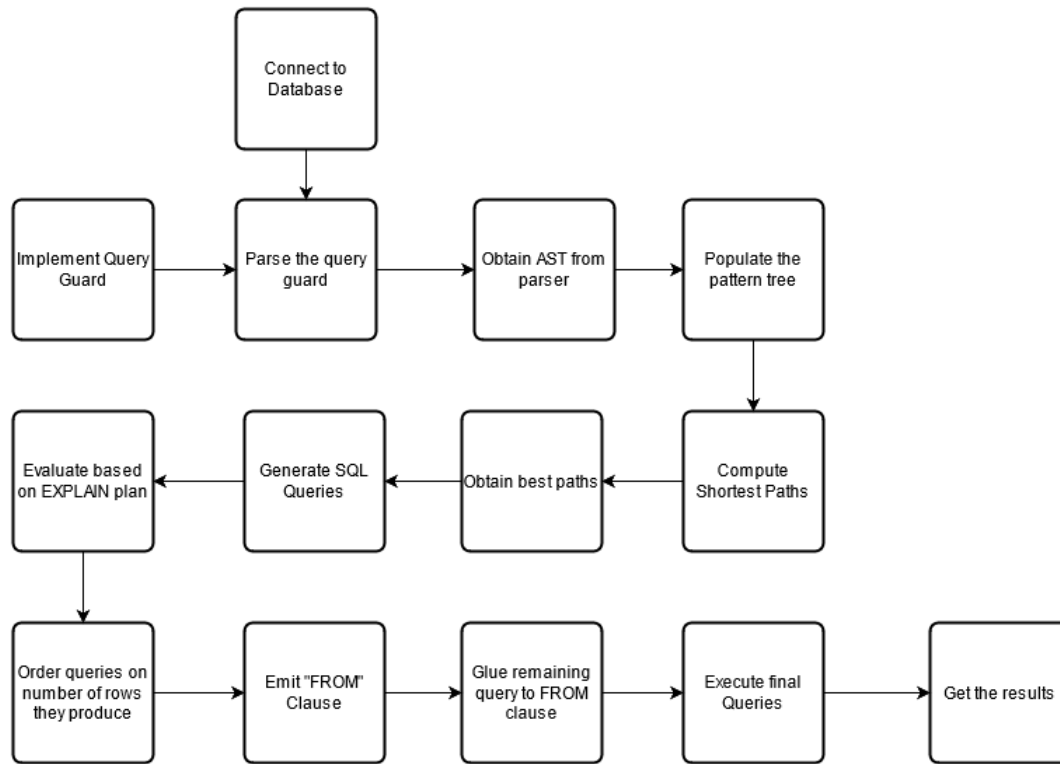


Fig. 3.7: Implementation Flowchart

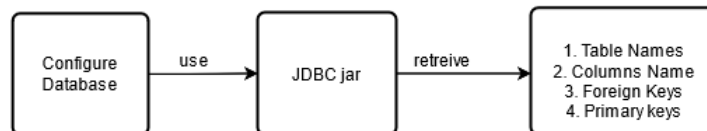
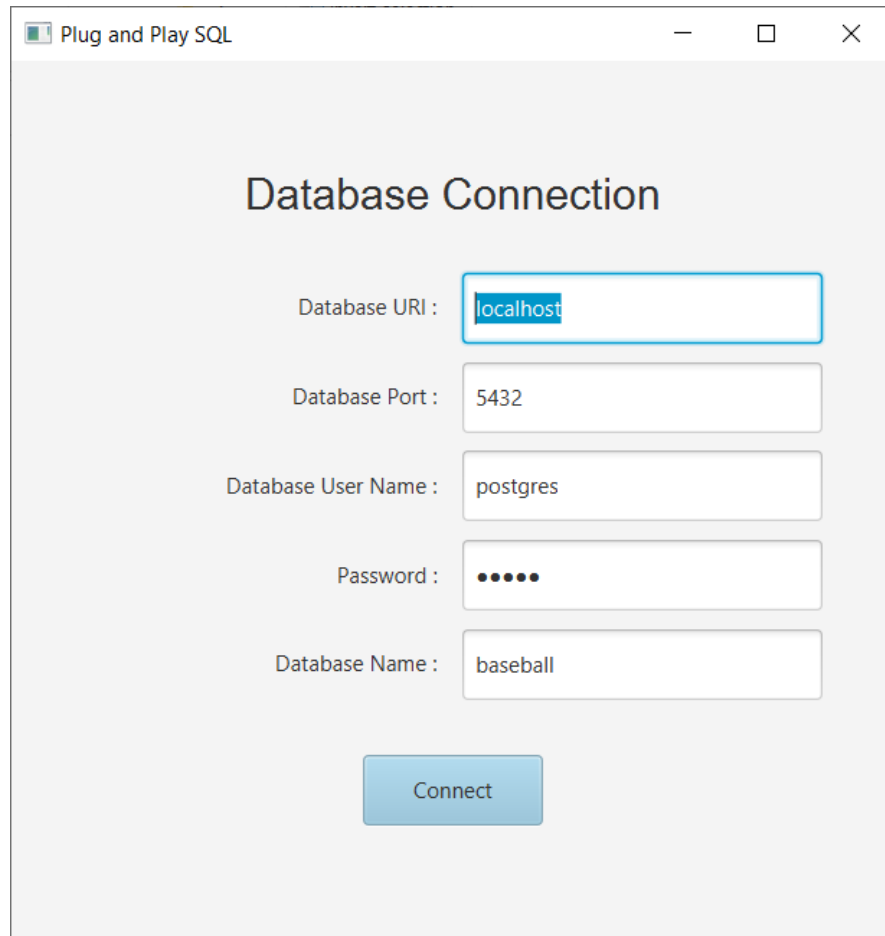


Fig. 3.8: Database Flowchart

3. Upon establishing a connection to the database successfully, the DB class retrieves the database metadata, including the table names, column names, primary keys, and foreign keys.
4. After the data retrieval step, the foreign keys are coupled with the path information, *i.e.*, the foreign keys along with the path (source and destination table and the corresponding column information) is stored into a list.

5. The stored paths are then evaluated and used to obtain the best paths.

Figure 3.9 shows how the connection to the Database is established in the implemented application.



The screenshot shows a window titled "Plug and Play SQL" with standard window controls (minimize, maximize, close). The main content area is titled "Database Connection" and contains five input fields and a "Connect" button. The fields are labeled "Database URI :", "Database Port :", "Database User Name :", "Password :", and "Database Name :". The values entered are "localhost", "5432", "postgres", a masked password (five dots), and "baseball" respectively. The "Connect" button is a blue rectangle located below the input fields.

Fig. 3.9: Establishing connection to the database

### 3.4.2 Implement a query guard

The difference between an ordinary SQL query and a plug-and-play query has been determined in Subsection 2.4 from Chapter 2. As a running example, we will use the same query guard to showcase the implementation.

The plug-and-play query used in Figure 3.10 uses the graph of foreign key relationships between the tables to build a hierarchy, similar to using a natural join to join the tables.

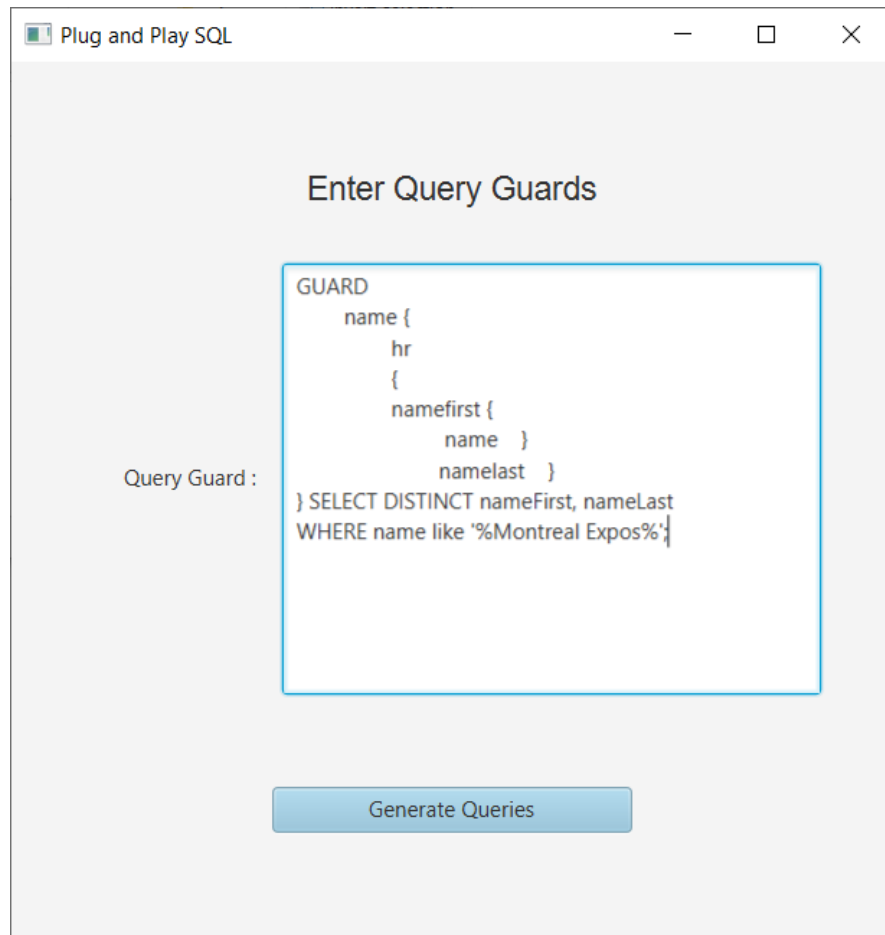


Fig. 3.10: Enter Query Guard to generate queries

### 3.4.3 Parsing the Query Guard

This project has the SQLite grammar implemented, which acts as an input to the ANTLR jar. The ANTLR jar compiles the grammar into the corresponding lexer and parser which are used to parse the query guard.

### 3.4.4 Obtaining the AST

Upon parsing the query guard, an Abstract Syntax Tree is generated. The AST for our query guard is shown in Figure 3.5

### 3.4.5 The Custom Listener

We have implemented a *MyListener* class, that is a custom listener. This custom

listener class extends the listener class generated by the ANTLR tool (SQLiteBaseListener). The listener as the name suggests listens to the events, or in simpler words, we parse the obtained tree using the custom listener we have implemented. The SQLiteBaseListener provides the enter and exit methods for every node that we visit while parsing the tree. These enter and exit methods are executed upon walking the tree.

While walking the tree, visiting and exiting each tree node calls the corresponding enter and exit methods for that tree node in MyListener. The generated base listener class has the same enter and exit methods too, however they are abstract (declarations without implementation) and so a custom listener is required to implement the enter and exit tree node functionality during our walk and so we override these methods in the MyListener class. The MyListener methods take the context of a treenode (ctx) as an argument, which contains data from our input SQL statement, the ctx contains the number of children nodes, index information, and other details about the input string. All this data makes it easier to populate the pattern tree while entering and exiting a tree node.

### **3.4.6 Populating the Pattern Tree**

As discussed earlier, we have implemented a Pattern Tree class in java that is populated when the query guard is parsed, The pattern tree holds the hierarchy of the tables and column values that are inside the query guard. Mostly the pattern tree works in synchronization with the MyListener and SQLPull classes so as to organize the guard into the hierarchical structure and support query generation.

### **3.4.7 Compute the shortest paths**

When the connection to the database is established, the paths that include the foreign keys, source, and destination tables, and source and destination column values are stored. We want to obtain the shortest or the best paths amongst all the paths between any two tables. The optimal paths would help us generate queries that can retrieve data from the database faster.

### 3.4.8 Generate SQL Queries

After computing the best paths between the tables in a hierarchy, we want to build a SQL query corresponding to the guard clause (The queries generated in this step correspond to the hierarchy from the guard clause only, they do not include the entire plug-and-play query). In the project, we have implemented an *SQLPull.java* class that performs the task of generating the SQL queries from the paths. The foreign keys, columns, and tables are used to generate queries. The unique foreign keys along with the from table and to table are used to generate the join conditions.

While generating the join condition, a left outer join is performed. A left outer join returns all the tuples from the left table and the matching tuples from the right table. After the join conditions have been generated, the generated condition acts as a *FROM* clause and the columns to be retrieved are glued to this *FROM* clause along with the *SELECT* statement and the *ORDER BY* clause if any to generate the queries which represent the query guard hierarchy.

### 3.4.9 Evaluate the EXPLAIN plan

After the queries representing the hierarchy of the query guard are generated, they are estimated for the rows they generate. So basically we evaluate the *EXPLAIN* plan for each of the generated queries by appending the *EXPLAIN* keyword to each query. The *EXPLAIN* plan of a query describes how a particular query executes and instead of giving the requested tuples as an output, the metadata on the query execution is rendered as an output which includes the row count that the query generates.

### 3.4.10 Ordering the queries on the number of rows they produce:

We store the queries generated and the number of rows it produces in a hash map and then sort this hash map by values in descending order to rank the queries according to the number of rows they produce.

### 3.4.11 Emit the From Clause

After ordering, these queries serve as a *FROM* clause for the SELECT statement from the initial plug-and-play query. The plug-and-play query excluding the query guard hierarchy is split and combined with the emitted *FROM* clause to obtain the final executable queries. These final queries are rendered on the application in the form of a combo box that gives the user an option to select the query they want to execute along with the corresponding generated query in a text area. Figure 3.11 shows the screenshot of our JavaFX application that displays all the generated queries.

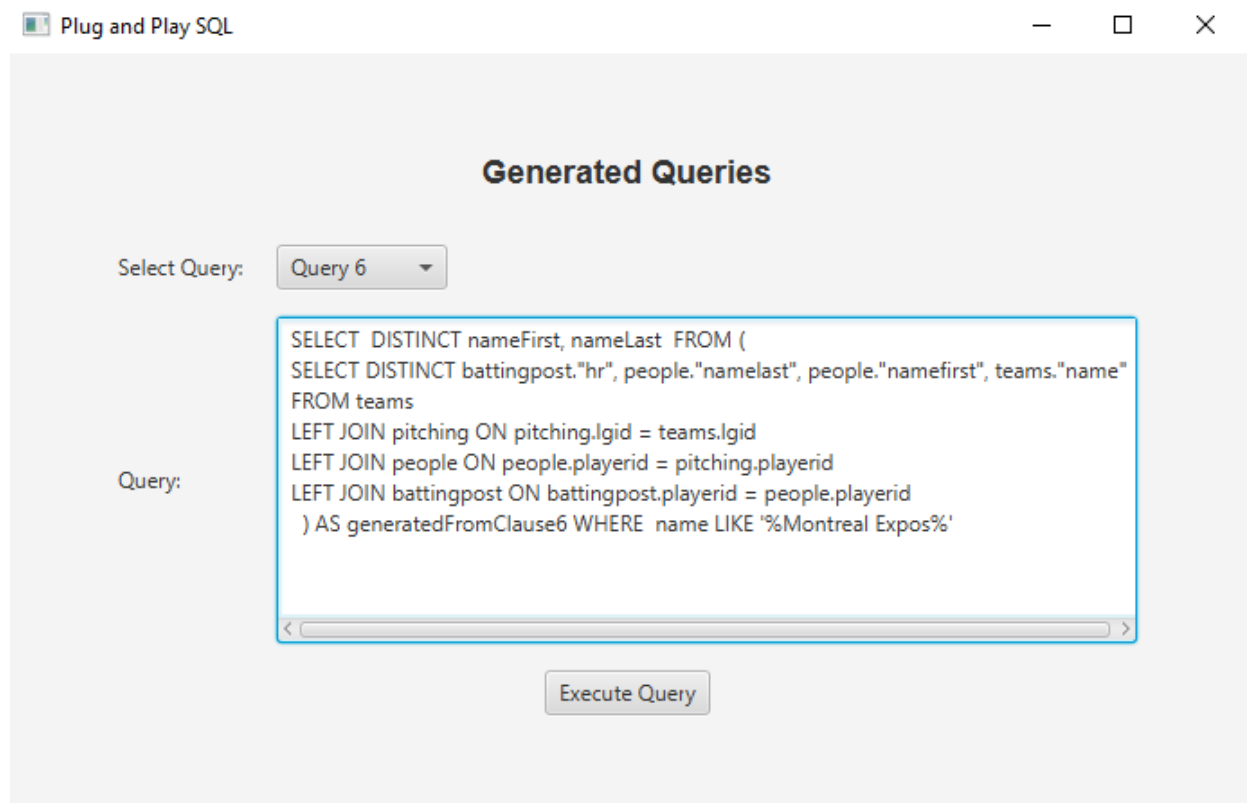


Fig. 3.11: Generated Queries

### 3.4.12 Execute Final Queries

As shown in Figure 3.12, the user selects the Query they want to execute and hits the Execute Query button on the application. The *DB.java* has the JDBC methods to execute the query, the final query selected by the user is executed to retrieve the respective

results. The results are then formatted and organized in the form of tabular data and then displayed on the result page of the application. The application also provides buttons to save the retrieved data in the form of a txt file.

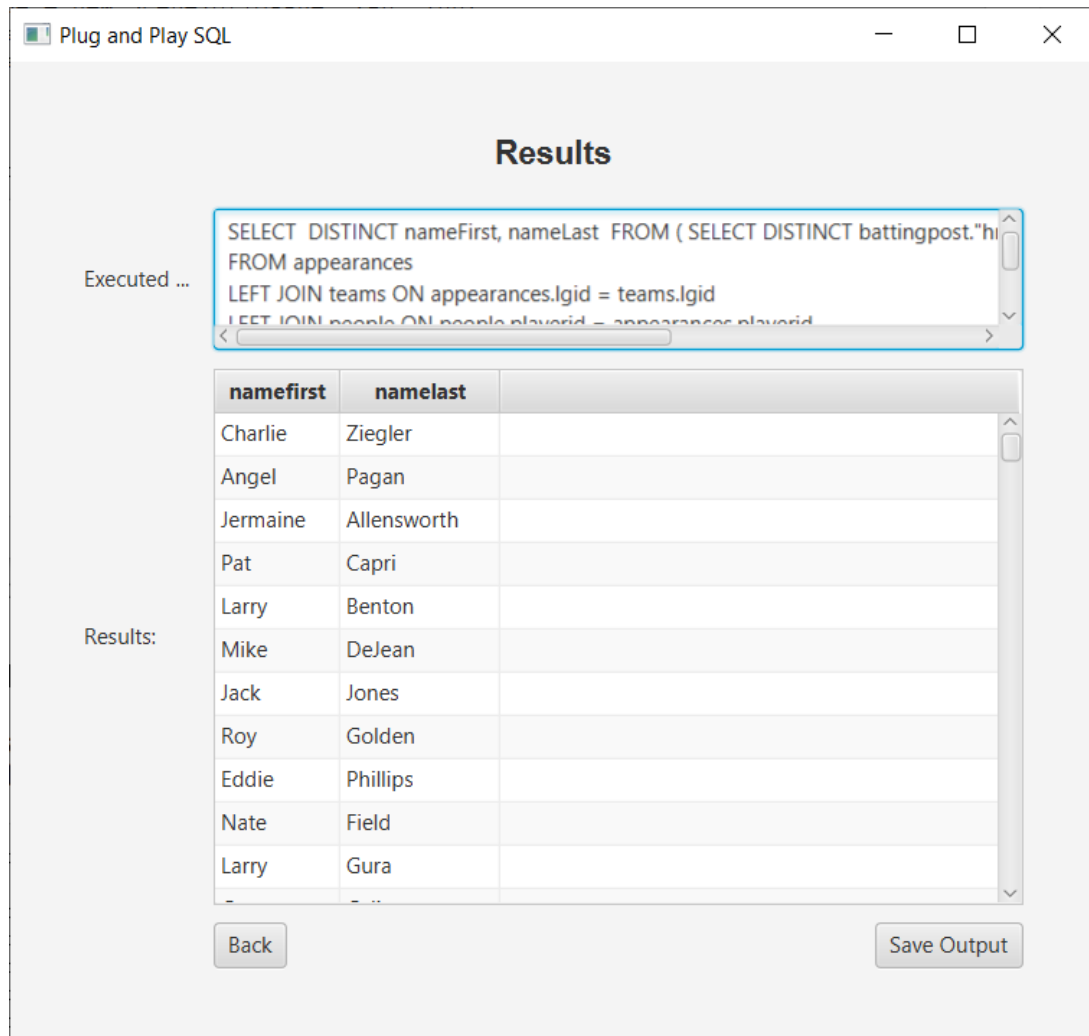


Fig. 3.12: Results

## CHAPTER 4

### PLUG-AND-PLAY EVALUATION

In this chapter, we provide a comparative analysis of the ordinary SQL queries with plug-and-play queries. We described a few example queries and their respective plug-and-play queries in Subsection 2.4 from Chapter 2.

Table 4.1 shows the cost comparison of manual SQL queries with the queries generated by the plug-and-play application.

Query No.	Manual Query Cost	Generated Query Cost
Query 1	318.61	2009.71
Query 2	3684.72	3684.72
Query 3	0.29	709
Query 4	3906.91	3924.3
Query 5	2500.25	2677.22
Query 6	3040.94	3040.94
Query 7	2004.75	8958.4

Table 4.1: Cost Analysis

Although, the Postgres documentation states that “the estimated costs and row counts may vary slightly because EXPLAIN’s statistics are random samples rather than exact and the costs are inherently somewhat platform-dependent.”, upon executing and evaluating these queries, we identified some work items that need to be addressed to achieve faster and more efficient query generation and execution.

#### 4.0.1 The Column Name Ambiguity

As a part of our current implementation, we configure the data source and store all the metadata from the database. The guard clause includes column names that we want



to retrieve. To retrieve this database we perform a search on all the tables in a database that have the specified column in the guard clause, evaluate the keys, and generate the join conditions in these tables.

There is a possibility that the column names are ambiguous *e.g.*, columns like *playerID*, *name*, and *teamID* are present in multiple tables, and to generate the queries all tables are looked up and evaluated.

After generating the join conditions, the EXPLAIN plan for all the queries is evaluated and the queries are ordered in the non-decreasing order of the number of rows they generate. This can lead to inaccurate results because of the ambiguity.

#### 4.0.2 Query Identification

As discussed earlier, the column name ambiguity generates queries on all the tables that have a particular column, and because of this the user needs to identify the correct query from the list of queries being generated.

#### 4.0.3 Solution

To overcome the column name ambiguity and the query identification problem, we propose the following solution: We will modify the ANTLR grammar to accept the *tableName.columnName* syntax. We will also make changes to the lookup logic to support the modified guard clause and this would help us in restricting the column search domain and instead of looking a particular column into all the tables, we will look up a column into the specified table which would result in less number of queries being generated, faster query generation and execution and accurate results.

## CHAPTER 5

### RELATED WORK

To the best of our knowledge, there is no previous work in querying SQL using hierarchies, in fact, the relational model replaced the hierarchical model [8] and is widely considered an improved successor. But there has been previous research in querying hierarchical data with the wrong shape that can be broadly classified into several categories.

**Query relaxation/approximation** - One way to loosen the tight coupling of path expressions to the hierarchy of data is to relax the path expressions or approximately match them to the data by exploring a space of hierarchies that are within a given edit distance [9–11]. Though such techniques work well for small variations in hierarchy or values, there can be a *very large* edit distance among a pair of instances which we would like to consider as the same data. Relaxing a query to explore all hierarchies within a large edit distance is overly permissive, and includes many hierarchies which do not have the same data. Query correction [12] and refinement [13] approaches are also best at exploring only small changes to the hierarchy.

**Hierarchical search engines** - Hierarchical search engines (*c.f.*, [14,15]) de-couple queries from specific hierarchies, similar to our aims, and can find data in differently-shaped hierarchies. But like query relaxation, search engines are overly permissive, and once data is found, a search engine does not transform that data to a hierarchy needed by a query.

**Structure-independent querying** - The idea of using a least common ancestor (LCA) has been explored for querying data independent of its hierarchy. Schema-free XQuery uses the meaningful LCA [16], XSeek exploits node interconnections [15], and others use the smallest LCA [17]. In contrast to all of the above research, none of these approaches are *data transformations*, that is they do not transform *values* (subtrees in the hierarchy), rather they can only utilize values from the source hierarchy; data transformations need to produce values in the shape of the target hierarchy.

**Declarative transformations** - There are declarative languages for specifying transformations of hierarchical data [18, 19]. These languages hide from users many specification details that would be needed in a language such as XSLT. 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.

**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 a hierarchy 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 hierarchy from which to extract the data. Specifying the source hierarchy will not work for a query guard, a query guard must be agnostic about the source. 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.

We also 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.

## CHAPTER 6

### CONCLUSIONS AND FUTURE WORK

#### 6.1 Conclusions

In this thesis, we have demonstrated the use of hierarchies to query a database. We also introduce and implement the plug-and-play query, to provide an efficient system to the programmers, that makes it easier to develop accurate queries. We have discussed how to write a query guard to translate a manual SQL query to a plug-and-play query. We have described how the query guards are manipulated and the shortest paths between the tables are computed using the graph of foreign keys. The application provides a fast way to generate queries and retrieve data from the database. The plug-and-play application is also easy to use and understand, by the use of ER diagrams or schemas it becomes easier for anyone to understand the structure of a database, which means that anyone can use the application without having to learn a lot. We have evaluated and analyzed this application, along with the aspects that can help in extending the application. Some of the features identified as future work are described in 6.2

#### 6.2 Future Work

##### 6.2.1 The column name ambiguity

As identified and described in chapter 4, the column name ambiguity is one of the aspects that needs to be resolved to achieve better efficiency and accuracy while generating and executing queries.

##### 6.2.2 DBMS Support

The Plug-and-Play SQL application currently works with the Postgres DBMS, thus generalizing this project to support all the relational database management systems to

obtain a generic application is another important feature.

### 6.2.3 Multiple guard clauses

Currently, our implementation supports one guard clause per SQL query, however, the grammar can be modified to extend this project to allow multiple guard clauses in a plug-and-play SQL query, and this would simplify implementing a lot of complex queries.

### 6.2.4 The **WITH** clause

Our implementation does not store the dynamic table structures that can be created using the *WITH* clause. Extending this application to store dynamically created tables and then use these dynamically created tables to retrieve requested data would be an important add-on.

This application was implemented in an effort to save time to implement queries, however, in a short period of time our application testing was done to a limited scope. This application, after implementing the identified future work items would need extensive testing to identify some important fixes or features.

## REFERENCES

- [1] C. Dyreson and S. Zhang, “The Benefits of Utilizing Closeness in XML,” in *DEXA Work.*, 2008, pp. 269–273.
- [2] C. Dyreson, S. Bhowmick, A. Jannu, K. Mallampalli, and S. Zhang, “XMorph: A Shape-polymorphic, Domain-specific XML Data Transformation Language,” in *ICDE*, 2010, pp. 844–847.
- [3] C. E. Dyreson, S. S. Bhowmick, and K. Mallampalli, “Using XMorph to Transform XML Data,” *PVLDB*, vol. 3, no. 2, pp. 1541–1544, 2010.
- [4] C. E. Dyreson and S. S. Bhowmick, “Querying XML Data: As You Shape It,” in *ICDE*, 2012, pp. 642–653.
- [5] B. Q. Truong, S. S. Bhowmick, and C. E. Dyreson, “SINBAD: Towards Structure-Independent Querying of Common Neighbors in XML Databases,” in *DASFAA (1)*, 2012, pp. 156–171.
- [6] 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>
- [7] —, “Virtual exist-db: Liberating hierarchical queries from the shackles of access path dependence,” *PVLDB*, vol. 8, no. 12, pp. 1932–1943, 2015. [Online]. Available: <http://www.vldb.org/pvldb/vol8/p1932-dyreson.pdf>
- [8] E. F. Codd, “A Relational Model of Data for Large Shared Data Banks,” *CACM*, vol. 13, no. 6, pp. 377–387, 1970.

- [9] S. Amer-Yahia, S. Cho, and D. Srivastava, “Tree Pattern Relaxation,” in *EDBT*, 2002, pp. 496–513.
- [10] 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.
- [11] Y. Kanza and Y. Sagiv, “Flexible Queries over Semistructured Data,” in *PODS*, 2001.
- [12] S. Cohen and T. Brodianskiy, “Correcting Queries for XML,” *Inf. Syst.*, vol. 34, no. 8, pp. 690–710, 2009.
- [13] A. Balmin, L. S. Colby, E. Curtmola, Q. Li, and F. Özcan, “Search Driven Analysis of Heterogenous XML Data,” in *CIDR*, 2009.
- [14] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, “XSEarch: A Semantic Search Engine for XML,” in *VLDB*, 2003, pp. 45–56.
- [15] Z. Liu, J. Walker, and Y. Chen, “XSeek: A Semantic XML Search Engine Using Keywords,” in *VLDB*, 2007, pp. 1330–1333.
- [16] Y. Li, C. Yu, and H. V. Jagadish, “Schema-Free XQuery,” in *VLDB*, 2004, pp. 72–83.
- [17] Y. Xu and Y. Papakonstantinou, “Efficient Keyword Search for Smallest LCAs in XML Databases,” in *SIGMOD Conference*, 2005, pp. 537–538.
- [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 *ADBS*, 2004, pp. 159–172.
- [20] M. Bhide, M. Agarwal, A. Bar-Or, S. Padmanabhan, S. Mittapalli, and G. Venkatchalialah, “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.