Using a Conceptual Model in Plug-and-Play SQL

Shubham Swami[1], Santosh Aryal[1], Sourav S. Bhowmick[2], and Curtis Dyreson[1](✉)

[1] Department of Computer Science, Utah State University, Logan, UT 84322, USA
{a02345936,curtis.dyreson}@usu.edu
[2] School of Computer Engineering, Nanyang Technical University, Singapore, Singapore
assourav@ntu.edu.sg
https://www.usu.edu/cs/people/CurtisDyreson ,
https://personal.ntu.edu.sg/assourav/

Abstract. We propose using a conceptual model for a database query's input type. The input type is the shape of the data needed by a query. Pairing a conceptual model with a query creates a plug-and-play query that can be type matched to a database's schema to determine whether the query can be safely evaluated. Plug-and-play queries are portable, easier to write, and are type safe. We describe a simple conceptual model based on virtual hierarchies, show how a virtual hierarchy is type matched to a relational schema, and how to transform an SQL query into one that can be evaluated on the matched schema.

Keywords: SQL · query evaluation · hierarchical data · query guards

1 Introduction

This paper proposes using a conceptual model to improve database queries. More specifically we propose using a conceptual model as the query's *input type*. 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.

Suppose that instead of a generic type or a subset of the schema we used a conceptual model as the input type. The conceptual model describes the *mini-world* in which the query needs to be evaluated. The idea is depicted in Fig. 1.

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

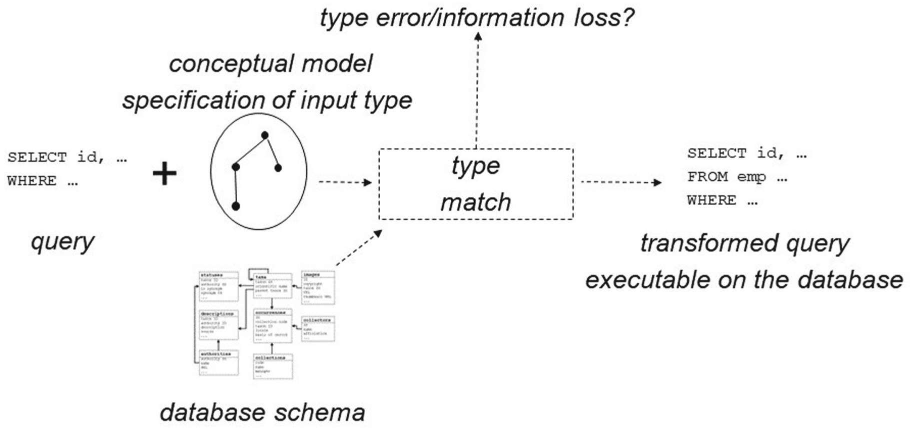


Fig. 1. Using a conceptual model as the input type

A query together with the conceptual model of the data needed by the query 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.

- *Type Safety* - A query that ignores the input type is said to *lack type safety.* The evaluation of the query cannot determine whether the query is malformed, *e.g.,* a name in a path expression is misspelled, or whether there is no data that matches the query since both cases produce the empty set. A type safe query, on the other hand, evaluates the structure of the input to determine if it conforms to that expressed by the conceptual model, *i.e.,* needed by the query.
- *Portability* - A query is *portable* if it can be type safely evaluated on different data collections. The conceptual model is not only critical to describing the input type to safeguard the query, but the model 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 conceptual model what the query needs to evaluate and match the conceptual model to the current schema.

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

This paper describes a system that has these potential benefits and specifically does not address the issue of determining *what is the best conceptual model to use as the input type*. We address instead the research question of *what are the potential benefits of using a conceptual model as the input type*. We chose to use the hierarchical model as the input type, surprisingly, for queries in SQL, a relational query language. The advantages of the hierarchical model are simplicity and prior research by others in *virtual hierarchies*, which are hierarchies that are not stored, rather they are constructed during query evaluation.

In his 1970 seminal paper on the relational model, E. F. Codd argued in favor of the relational model by describing important drawbacks of the popular (at that time) hierarchical model [7]. One of the drawbacks of the hierarchical model that Codd identified was *access path dependence*. Codd pointed out that queries in a hierarchical (or network) model necessarily have to use access paths ("dot" operators) to navigate to desired data. The access paths tightly couple the query to a specific hierarchy, which is problematic since the same data could be organized in different hierarchies, so a query written for one hierarchy would fail if the same data were organized differently. Access path dependence decreases query portability and increases the brittleness of queries to changes in the structure of the data.

But hierarchical data also makes some aspects of querying easier. First, access paths in hierarchical queries are simpler and more straightforward to express than joins in a relational database, an advantage also present in graph queries in languages such as Cypher and GQL, and in SQL for SurrealDB, which uses a RELATE clause to build relationships between tables that can be navigated by path expressions. Joins are implicitly embedded in a hierarchical data structure, performed when creating the data model, and these embedded joins in the data are easily navigated with a path expression. Second, grouping and aggregation can be more naturally expressed in hierarchical data. Their expression in SQL has been shown to be cognitively challenging for many users, especially programmers learning SQL [1,17,18]. Third, Codd's critique of access path dependence applies only to *stored* hierarchies. *Virtual hierarchies* are dynamically constructed as needed for query evaluation, hence have no such dependence.

This paper leverages virtual hierarchies as a conceptual model to support *plug-and-play* SQL. We propose coupling a query to a hierarchical specification of its input type, we call the specification a *query guard*, to create a plug-and-play query. A plug-and-play query is similar to 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, if the data source provides data in a sufficient structure specified by its input type or guard, it will "play" producing a 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 Fig. 2 to retrieve the names of botanists who collected *Asteraceae* (plants in the Daisy family)

```
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. 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. 3. Retrieve the names of those who collected *Asteraceae* specimens in 2023

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 Fig. 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).

One big advantage of plug-and-play SQL queries is that they are *portable*. The query in Fig. 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 Fig. 4. Querying against a hierarchy simplifies grouping and aggregation (as in XQuery and Cypher).

This paper focuses on matching and transforming the *shape* of the data. We 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. The focus of our research is on the shape of the data and because the problem is orthogonal 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 Fig. 3 are present to help in the semantic matching. The guard

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

Fig. 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. 5. Simplified guard for the query in Fig. 3

could be simplified to that shown in Fig. 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 schema to those in the guard.

This paper makes the following contributions.

- We describe using a conceptual model as the input type for an SQL query. We call the model a query guard.
- We show how to match the query guard to the schema of a relational database.
- We give a denotational semantics for converting a plug-and-play query to SQL.
- We report on the implementation of plug-and-play SQL.

This paper is organized as follows. The next section describes how an input type specification is matched to a schema hierarchy and how the match is used to rewrite the query to one that can be evaluated. We then explain how we implemented query guards and give a brief evaluation. Section 5 covers related work. The paper concludes with a short summary and gives some avenues for continuing the research in future.

2 Model

In this section we describe, at an abstract level, how the virtual hierarchy is constructed for a guard when evaluated on a relational database. The key ideas are to model *data-relatedness* using a multigraph of associations among relations. A spanning tree in the multigraph determines how to best relate names in a hierarchical context. The tree is used to construct an SQL query to extract data for formatting in the shape specified by a guard.

2.1 Guard

A *guard* is a specification or declaration of the structure of the data needed by the query.

Definition 1 (Guard). *Let database, D, consist of names $N = n_1, \ldots, n_k$. Then guard $G = (M, E)$ where $M \subseteq N$ and*

$$E = \{(n_p, n_c) \mid n_p, n_c \in M\}$$

forms a connected, acyclic graph. ∎

Essentially a guard is a tree of database column or table names.

We assume that a guard is specified using JSON-like syntax, as is common in other tools, *e.g.,* GraphQL.

Definition 2 (Guard JSON Specification). *A guard conforms to the EBNF grammar given below.*

```
guard  ← GUARD pair
pair   ← name obj?
name   ← TABLE_NAME | COLUMN_NAME
obj    ← { pair (, pair)* }
```
∎

The tree of names in a guard is built from the nested values of name/obj pairs in the guard specification. For example, the guard in Fig. 3 is the tree consisting of nodes {`collectors`, `occurrences`, `name`, `family`, and `year`} and edges { (`collectors`, `name`), (`collectors`, `occurrences`), (`occurrences`, `family`), (`occurrences`, `year`) }.

2.2 Association Multigraph

For our purposes a relational database, D, is a set of relations, $\{R_1, \ldots, R_n\}$, and a set of *associations* among attributes in the relations $K = f_1, \ldots, f_m$, *e.g.,* K could be a set of foreign key constraints, inclusion dependencies, or user specified "edges" (such as specified by the RELATE clause in SurrealDB SQL). Each relation in D has some number of attributes, that is, the schema for relation R_i is $(A_1, \ldots, A_k)$ and each relationship in K is of the form $R_j \rightarrow R_m$, that is, relation R_j is related to R_m, *e.g.,* there is a foreign key from R_j to R_m.

Definition 3 (Association Multigraph). *The association multigraph, $G = (V, E)$, for D is an undirected multigraph where $V = \{R_1, \ldots, R_n\}$ is the set of vertices and $E = \{(R_j, R_m, i) \mid f_i \in K \wedge f_i = R_j \rightarrow R_m\}$ is the set of edges (i is the label of the edge).* ∎

Note that there is one edge per association. The edge is labelled with the identifier for the association. As there could be more than one association, *e.g.,* more than one foreign key, between a pair of relations, there can be more than one edge between nodes, but each edge will have a different label.

As an example consider the relational schema depicted in Fig. 6. The schema is for Symbiota, a commonly used biodiversity data management system [11]. The schema depicted is a small part of Symbiota's schema, which has 74 tables and 97 foreign keys. Symbiota stores specimen biodiversity data such as `occurrences` of `taxa` that are part of `collections` housed in herbaria, natural history museums, and private collections. A collection may involve various `collectors` and `images` of the specimens. The `taxa` table is the taxonomic hierarchy of scientific names that may be synonyms (recorded in the `statuses` table) as stipulated by taxonomic `authorities`. The `taxa` records also have `descriptions` derived from taxonomic treatments. The `taxa` table has a foreign key to itself that associates child to parent taxa. The association multigraph for the schema in Fig. 6 is given below and depicted in Fig. 7.

- $V = \{$`statuses`, `descriptions`, `authorities`, `taxa`, `occurrences`, `collections`, `images`, `collectors`$\}$
- $E = \{$(`authorities`, `statuses`, 1), (`authorities`, `descriptions`, 2), (`statuses`, `taxa`, 3), (`descriptions`, `taxa`, 4), (`occurrences`, `taxa`, 5), (`collections`, `occurrences`, 6), (`taxa`, `images`, 7), (`collectors`, `occurrences`, 8), (`taxa`, `taxa`, 9)$\}$

In general in this paper we will utilize foreign keys for the associations. We focus on foreign keys not only because foreign keys describe important semantic connections between tables, but also because the keys are stored in the schema and so can be automatically read and used. But the multigraph could be constructed by computing other associations among relations, *e.g.,* inclusion dependencies, and the techniques described in this paper would be the same.

2.3 Relating Data Through Names

The association multigraph can be used to relate names in a guard based on the notion of *closeness* [20]. Closeness can be described as the property that two data items are *related* if they are connected (by a path) and that no shorter paths that connect items of the same *type* exists. In the context of relational databases the *type* of a datum is the domain (an attribute in a relation) to which it belongs.

Suppose that a guard specifies that `affiliation` should be related to `scientific name`. The `affiliation` type exists in the `collectors` relation, while `scientific name` is part of `taxa`. There is a path of length two that connects `collectors` to `taxa` as well as paths of length greater than two (by traversing the link from `taxa` to itself). Closeness stipulates that the shortest path is preferred.

Definition 4 (Parent/Child Closeness). *Let plug P have parent p with child c where p is an attribute of relation R_p and c is an attribute of relation R_c.*

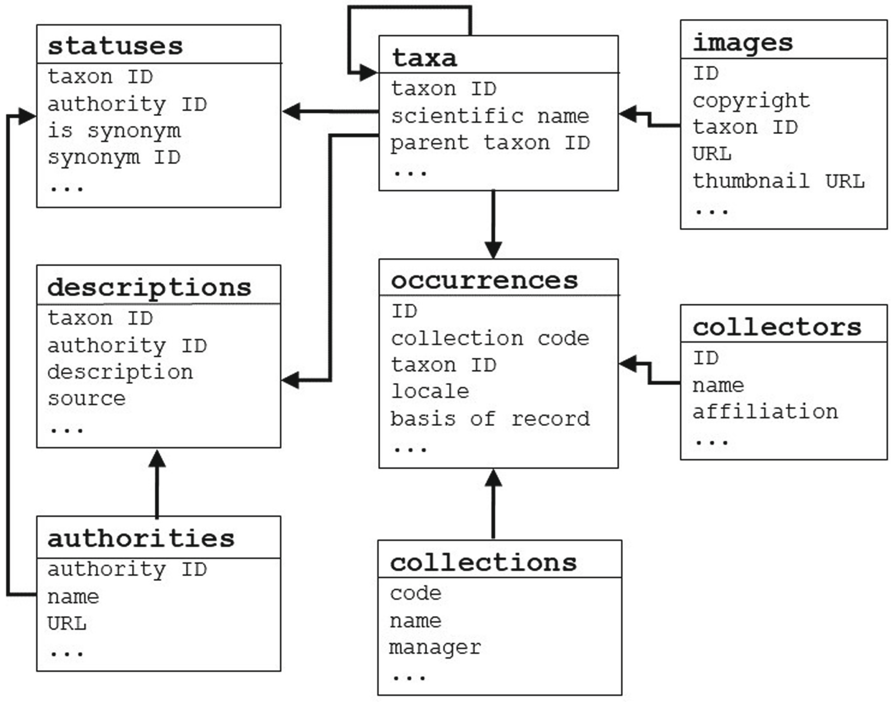


Fig. 6. Reduced schema of the Symbiota2 database

Closeness stipulates that a path from R_p to R_c makes p closest to c if and only if there is no shorter path between R_p and R_c in a association multigraph, F, that is,

$$\otimes(F, P, p, c) = \{(R_p, R_1, i_1), \ldots, (R_n, R_c, i_n)\}$$

$\otimes$ is the closest operator and $(R_p, R_1, i_1), \ldots, (R_n, R_c, i_n)$ is a shortest path. ∎

As an example assume the pattern contains `affiliation` (in relation `collectors`) and `description` (in relation `descriptions`), then the shortest path is below.

```
{(collectors, occurrences, 8), (occurrences, taxa, 5),
 (taxa, descriptions, 4)}
```

Parent/child closeness relates a pair of names in a guard, but a guard could contain many names. Closeness for the guard is built from parent/child closeness.

Definition 5 (Guard Closeness). *Let P be the set of parent child relationships, (p, c), in a guard. Then for association multigraph F the data relationship operator, $\bigotimes$, is defined as follows.*

$$\bigotimes(F, P) = \bigcup_{\forall (p,c) \in P} \otimes(F, P, p, c)$$ ∎

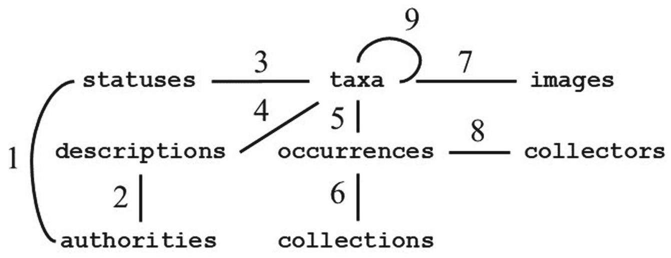


Fig. 7. Association multigraph for the Symbiota2 database

Guard closeness defines a spanning tree within the graph over the nodes corresponding to relations that have attributes in the guard. To relate data in a guard, P, the paths on the plug are joined using an in-order walk of the tree for $\bigotimes(F, P)$.

Definition 6 (Relating Data). *Given a guard, P, and an association multigraph, F, with spanning tree, C, for $\bigotimes(F, P)$ that relates names $x_1, \dots, x_k$ in P, let an inorder walk of the spanning tree yield the list of relations $[R_1, \dots, R_n]$. Then the data relationship operator, $⟕_P$, is defined as follows.*

$$⟕_P(C, [x_1, \dots, x_k]) = \pi_{x_1,\dots,x_k}(⟕[R_1, \dots, R_n])$$

where ⟕ is the left outer join (on the attributes in the foreign keys). ∎

For example, to relate `affiliation` to `scientific name`, the inorder walk for the spanning tree is [`collectors`, `occurrences`, `taxa`]. The data relationship operator applied to this list yields the query given below.

$$\pi_{\texttt{affiliation,scientific name}}(\texttt{ collectors} ⟕ \texttt{occurrences} ⟕ \texttt{taxa })$$

2.4 Potential Information Loss

There may be more than one closeness spanning tree that connects pairs of names. For instance there are two paths of length two from `authorities` to `taxa`, one through `statuses` and one through `descriptions`. To determine which spanning tree to use, we rank the trees by their *potential information loss.*

Definition 7 (Loss Ranking). *Let spanning trees $T_1, \dots, T_n$ connect names $x_1, \dots, x_k$. Then T_i is the most complete spanning tree if the data relatedness of the tree produces the most tuples,* i.e.,

$$|\bigotimes(T_i, [x_1, \dots, x_k])| = \mathbf{max_{1 \leq j \leq n}} |\bigotimes(\mathbf{T_j}, [\mathbf{x_1}, \dots, \mathbf{x_k}])|.$$ ∎

```
taxa {
  name {
     manager
  }
}
```

Fig. 8. Taxa and the managers who manage collections of them

The idea of loss ranking is to choose the spanning tree that produces the most tuples since such a join represents the most complete connection among the set of relations. Note that the loss ranking is an instantaneous measure, that is, it produces a ranking with respect to the state of a database as of when the query is evaluated. Since relations change over time an alternative path may represent the most complete connection at some future time. To compute a measure of the completeness the association multigraph can be annotated with *join selectivity*. Suppose foreign key f is from relation R to relation S, that is, R borrows a key from S. Then the join selectivity for f be $L_f = |S \ltimes R|/|S|$. Note that $S \ltimes R$ using f will produce between 0 and $|S|$ tuples. We can annotate the association multigraph with join selectivities and multiply the selectivities along branches in a spanning tree to get total completeness; alternative spanning trees can be ranked by their total completeness.

Completeness factors can also be used to categorize plugs by the amount of information loss. A completeness factor of 1 for a plug represents that the construction of a hierarchy loses no information, *i.e.,* it is *complete* in the sense that every value at a leaf can be reached from the root. A completeness factor of less than 1 indicates that some leaf values might not be represented. For example, consider the guard specified in Fig. 8. which relates `taxa` to `collections`. The guard specifies joins along the following path: `taxa`, `occurrences`, and `collections`. If the completeness factor is 1 then every taxon is part of some collection that has a manager. On the other hand, a completeness factor less than 1 indicates that some `taxa` may be unrelated to a `manager` (are not in a collection). Note that because we are using outer joins to compute the hierarchy those `taxa` will still be present in the hierarchy, but the `manager` will be a null value.

Guard closeness as defined above is based on the closeness of parent/child relationships in a guard rather than the minimal number of relationships overall in a guard. An alternative is to use the Steiner tree, which is a minimal spanning tree among a subset of nodes in the multigraph. Computing the Steiner tree is NP-complete [14], even for an unweighted multigraph. Though approximation techniques exist [5], it is unclear if the Steiner tree gives a better intuitive solution to the data-relatedness problem since a guard designer may construct a guard by reasoning about parent/child relationships in a hierarchy rather than overall minimality of the edges in a guard.

```
$\llbracket$ GUARD $G$
  SELECT $s_1, \ldots, s_n$
  WHERE $Q$
$\rrbracket(D) \equiv$
  SELECT A.$s_1$, ..., A.$s_n$
  FROM (
    SELECT $s_1, \ldots, s_n$
    FROM $⟕_P(C, s_1, \ldots, s_n)$ where $C$ is the spanning tree derived from $G$ for $D$
    WHERE $Q$
  ) A
```

Fig. 9. Denotational rule for translating a plug-and-play query into SQL

```
SELECT A.name
FROM (SELECT name
      FROM collectors
         LEFT OUTER JOIN occurrences ON collectors.id = occurrences.collid
         LEFT OUTER JOIN taxa ON occurrences.'taxon ID' = taxa.'taxon ID'
      WHERE family = 'Asteraceae' AND year = 2023) A
```

Fig. 10. SQL for retrieving who collected *Asteraceae* specimens in 2023

2.5 Combining the Guard with the Query

In this section we give the denotational semantics of a plug-and-play query. There are two cases: with and without an aggregate function. We consider the without case first.

If a query does not have an aggregate function then the transformation is relatively straightforward using the data-relatedness operator, $⟕_P$. In the rule given in Fig. 9, D is the database on which the query is evaluated. As an example, the transformation of the query in Fig. 3 is given in Fig. 10. Note that the outer join operator generates a path from `collectors` to `taxa` through `occurrences` to relate `name` to `family`, hence the final hop in the path back to `occurrences` is not needed in the join expression.

A query with an aggregate function has to add grouping (more than one aggregate is a repetition of this case). In the denotation rule given in Fig. 11 we assume a is an aggregate applied to a name at level k in the tree (with ancestor names g_1 to g_k). We further assume a is both in the `SELECT` and the `WHERE` clause. As an example, the transformation of the query in Fig. 4 is given in Fig. 12.

3 Implementation

In this section, we describe the code structure for our application. Most of the code is written in Java. We used ANTLR for parsing and translation and modified the grammar for SQLite. The code structure for the application is shown

```
⟦ GUARD $G$
   SELECT $s_1, \ldots, s_n, a$ where $a$ is an aggregate
   WHERE $Q$ $AND$ $Q_A$ where $Q_A$ is the predicate part of the aggregate
⟧$(D) \equiv$
   SELECT A.$s_1$, ..., A.$s_n$, A.$a$
   FROM (
      SELECT $s_1, \ldots, s_k, a$
      FROM $\bowtie_P(C, s_1, \ldots, s_n)$ where $C$ is the spanning tree derived from $G$ for $D$
      WHERE $Q$
      GROUP BY $s_1, \ldots, s_k$
   ) A
   WHERE $Q_A$
```

Fig. 11. Denotational rule for translating a plug-and-play query with an aggregate into SQL

```
SELECT A.name
FROM (SELECT name, count(*) as C
      FROM collectors
             LEFT OUTER JOIN occurrences ON collectors.id = occurrences.collid
             LEFT OUTER JOIN taxa ON occurrences.'taxon ID' = taxa.'taxon ID'
      WHERE family = 'Asteraceae' AND year = 2023
      GROUP BY name) A
WHERE A.C > 40
```

Fig. 12. SQL for retrieving who collected *Asteraceae* specimens in 2023

in Fig. 13. It consists of five modules. The `database` module is handles database communication We used JDBC for communicating. The `grammar` module contains the lexer and parser rules for the SQL and query guard, and a custom listener to implement the denotational semantics for the translation of a plug-and-play SQL query into SQL. The `data pull` module contains the logic to evaluate a query and display results. The `join graph` module builds and maintains the association multigraph. Lastly, the `tree module` communicates with the listener and the data pull module to generate the queries.

Figure 14 shows a screenshot of our JavaFX application that displays the generated query (the guard and query are in the context of a baseball database). As shown in Fig. 15, the user selects the query they want to execute and hits the `Execute Query` button to generate the result.

4 Plug-and-Play Evaluation

We provide a comparative analysis of the run-time cost of ordinary SQL queries with plug-and-play queries. Of course the plug-and-play queries were easier to write, but in this evaluation we focus on the run-time cost. We wrote six plug-and-play queries on a baseball database with 2GB of data (the Lahman baseball

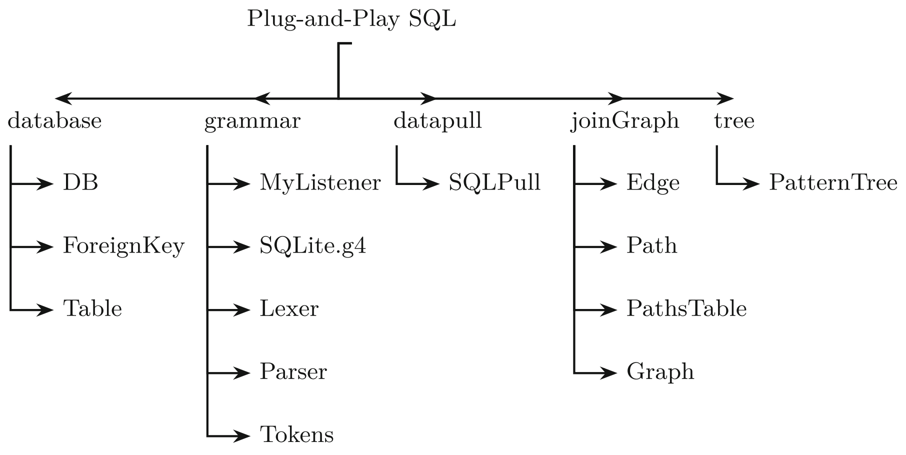


Fig. 13. Plug and Play SQL- Code Structure

Table 1. Cost Analysis

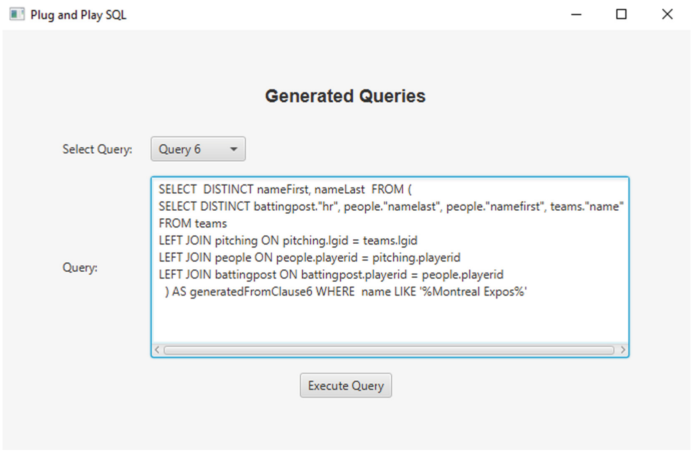
Query No.	Manual Query Cost	Generated Query Cost
Query 1	318.61	2009.71
Query 2	3684.72	3684.72
Query 3	0.29	709
Query 4	3906.91	3924.3
Query 5	2500.25	2677.22
Query 6	3040.94	3040.94
Query 7	2004.75	8958.4

database is publicly available). We ran the queries using Postgres version 14.7 on a Linux system running Ubuntu with 16GB of RAM. Table 1 shows the cost comparison of the manually created SQL queries compared to the queries generated by the plug-and-play application. We observe that the plug-and-play queries are often the same cost as the hand-crafted queries, but sometimes incur higher cost due to the cost of left outer joins versus inner joins. We plan to focus on optimizing queries to consider edge cases in future work.

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 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



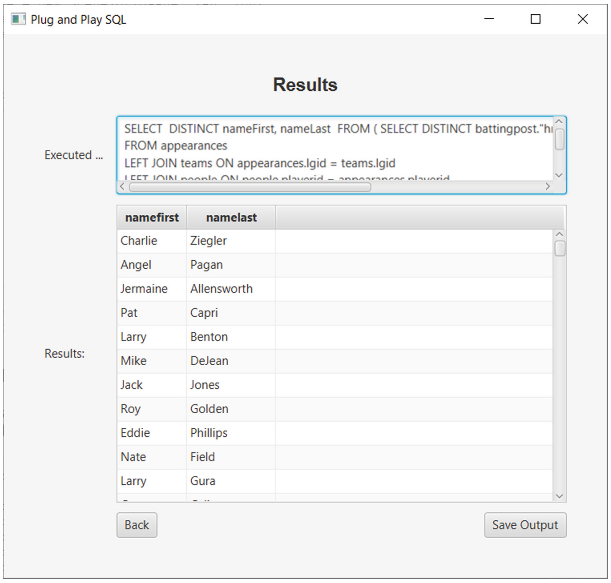
```
SELECT  DISTINCT nameFirst, nameLast  FROM (
SELECT DISTINCT battingpost."hr", people."namelast", people."namefirst", teams."name"
FROM teams
LEFT JOIN pitching ON pitching.lgid = teams.lgid
LEFT JOIN people ON people.playerid = pitching.playerid
LEFT JOIN battingpost ON battingpost.playerid = people.playerid
 ) AS generatedFromClause6 WHERE  name LIKE '%Montreal Expos%'
```

Fig. 14. Generated Queries

approximately match them to the data within a given edit distance [2,3,13]. 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 [8] and refinement [4] approaches are also best at exploring only small changes to the data.

Declarative Transformations. There are declarative languages for specifying transformations of (hierarchical) data [15,16]. 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 [6]. 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 [9,12]. 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 auto-



Results

Executed ...

```
SELECT  DISTINCT nameFirst, nameLast  FROM ( SELECT DISTINCT battingpost."h
FROM appearances
LEFT JOIN teams ON appearances.lgid = teams.lgid
LEFT JOIN people ON people.playerid = appearances.playerid
```

Results:

namefirst	namelast
Charlie	Ziegler
Angel	Pagan
Jermaine	Allensworth
Pat	Capri
Larry	Benton
Mike	DeJean
Jack	Jones
Roy	Golden
Eddie	Phillips
Nate	Field
Larry	Gura

Fig. 15. Results

matic. 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 [10]. 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.

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.

6 Conclusions and Future Work

This paper describes how to pair a query with a conceptual model, which we call a query guard. The query guard is a specification of the query's input type, that is, the structure or shape of the data that the query needs in order to correctly evaluate. The combination of query guard and query creates a plug-and-play query. Plug-and-play queries are more portable, more reliable because they are input type safe, and are potentially easier to write.

In this paper we chose a very simple conceptual model for expressing a query guard, namely, a hierarchical specification. We used this specification for a relational query language, thereby demonstrating that the model for the input type can be independent of the data model for the query. Though we focused on how to run a plug-and-play query on a relational database, a plug-and-play query could be equally run on JSON data or graph data. But the input type must be matched to a given data model. We described how to match the query guard to a relational schema. Once the schema is matched the query can be transformed to a query that can be safely evaluated on the relational database.

In future we plan to investigate whether there is a better way to express a query guard, *i.e.,* what is the best conceptual model to use? Concurrent with this effort we will conduct a user survey to help evaluate the effectiveness of plug-and-play SQL in lowering the time and effort to write queries. The user survey will investigate the use of different conceptual models using a randomized approach [19]. The user survey requires a separate treatment than this paper, which focuses on conceptual modeling. We also plan to expand the range of queries we handle to include subqueries, relational operations (union, intersection, and difference), and data modification. Another direction of future research is guard inference. Relying on programmers to specify query guards for plug-and-play queries has two problems: First, a programmer may change a query but forget to change the guard. Second, a programmer may give an incorrect guard, for instance, specify a guard that is in the wrong shape for a query. The best way to solve both problems is to automatically infer a guard, Q_p, from a query Q. Ideally, Q_p, will be minimal, that is we will infer Q_p such that there does not exist another guard, Q'_p, for Q, which is tighter than Q_p.

Acknowledgements. This work was supported in part by the National Science Foundation under Award No. DBI-1759965, *Collaborative Research: ABI Development: Symbiota2: Enabling greater collaboration and flexibility for mobilizing biodiversity data.* Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of NSF.

References

1. Ahadi, A., Prior, J., Behbood, V., Lister, R.: A quantitative study of the relative difficulty for novices of writing seven different types of SQL queries. In: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2015, pp. 201–206 (2015). https://doi.org/10.1145/2729094.2742620

2. Amer-Yahia, S., Cho, S., Srivastava, D.: Tree pattern relaxation. In: EDBT, pp. 496–513 (2002)
3. Augsten, N., Böhlen, M.H., Gamper, J.: The q-gram distance between ordered labeled trees. ACM Trans. Database Syst. **35**(1), 1–36 (2010)
4. Balmin, A., Colby, L.S., Curtmola, E., Li, Q., Ozcan, F.: Search driven analysis of heterogenous XML data. In: CIDR (2009)
5. Beyer, S., Chimani, M.: Strong Steiner tree approximations in practice. J. Exp. Algorithmics **24**(1), 1.7:1–1.7:33 (2019)
6. Bhide, M., Agarwal, M., Bar-Or, A., Padmanabhan, S., Mittapalli, S., Venkatachaliah, G.: XPEDIA: XML processing for data integration. PVLDB **2**(2), 1330–1341 (2009)
7. Codd, E.F.: A relational model of data for large shared data banks. CACM **13**(6), 377–387 (1970)
8. Cohen, S., Brodianskiy, T.: Correcting queries for XML. Inf. Syst. **34**(8), 690–710 (2009)
9. Fagin, R., Haas, L.M., Hernández, M., Miller, R.J., Popa, L., Velegrakis, Y.: Clio: schema mapping creation and data exchange. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 198–236. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02463-4_12
10. Fan, W., Bohannon, P.: Information preserving XML schema embedding. ACM Trans. Database Syst. **33**(1), 1–44 (2008)
11. Gries, C., Gilbert, E., Franz, N.: Symbiota - a virtual platform for creating voucher-based biodiversity information communities. Biodivers. Data J. **2**, e1114 (2014)
12. Jiang, H., Ho, H., Popa, L., Han, W.S.: Mapping-driven XML transformation. In: WWW, pp. 1063–1072 (2007)
13. Kanza, Y., Sagiv, Y.: Flexible queries over semistructured data. In: PODS (2001)
14. Karp, R.M.: Reducibility among combinatorial problems. In: Proceedings of a Symposium on the Complexity of Computer Computations, pp. 85–103 (1972)
15. Krishnamurthi, S., Gray, K.E., Graunke, P.T.: Transformation-by-example for XML. In: PADL, pp. 249–262 (2000)
16. Pankowski, T.: A high-level language for specifying XML data transformations. In: ADBIS, pp. 159–172 (2004)
17. Poulsen, S., Butler, L., Alawini, A., Herman, G.L.: Insights from student solutions to SQL homework problems. In: Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, pp. 404–410 (2020). https://doi.org/10.1145/3341525.3387391
18. Taipalus, T., Siponen, M., Vartiainen, T.: Errors and complications in SQL query formulation. ACM Trans. Comput. Educ. **18**(3), 1–29 (2018)
19. Uesbeck, P.M., Peterson, C.S., Sharif, B., Stefik, A.: A randomized controlled trial on the effects of embedded computer language switching. In: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, Virtual Event, USA, 8–13 November 2020, pp. 410–420. ACM (2020). https://doi.org/10.1145/3368089.3409701
20. Zhang, S., Dyreson, C.E.: Symmetrically exploiting XML. In: WWW, pp. 103–111 (2006)