ABSTRACT
We study the problem of synthesizing a core fragment of relational queries called select-project-join (SPJ) queries from input-output examples. Search-based synthesis techniques are suited to synthesizing projections and joins by navigating the network of relational tables but require additional supervision for synthesizing comparison predicates. On the other hand, decision tree learning techniques are suited to synthesizing comparison predicates when the input database can be summarized as a single labelled relational table. In this paper, we adapt and interleave methods from the domains of relational query synthesis and decision tree learning, and present an end-to-end framework for synthesizing relational queries with categorical and numerical comparison predicates. Our technique guarantees the completeness of the synthesis procedure and strongly encourages minimality of the synthesized program. We present Libra, an implementation of this technique and evaluate it on a benchmark suite of 1,475 instances of queries over 159 databases with multiple tables. Libra solves 1,361 of these instances in an average of 59 seconds per instance. It outperforms state-of-the-art methods in terms of both the running time and the quality of the synthesized programs.

PVLDB Reference Format:
doi:10.14778/3626292.3626306

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/aadityanaik/libra.

1 INTRODUCTION
The problem of learning relational logic programs from input-output data has been widely studied in inductive logic programming (ILP) and program synthesis [8, 13]. Such programs offer a variety of benefits by virtue of being explainable, interpretable, generalizable, and summarizing information from complex relational databases as well as data exploration and discovery without requiring in-depth knowledge of the underlying schema [8, 33, 39, 43] (see [42] for how interactions with human experts on Stack Overflow to search for the desired SQL query can be replaced by synthesis from input-output examples).

These programs involve two core components: joins across multiple tables in a database, and comparison predicates that filter out rows. As such, learning logic queries involves addressing two fundamental and inter-dependent challenges, which are synthesizing joins and synthesizing comparison predicates. Recent works in learning logic programs focus on addressing either of the two challenges in an automated manner, but not both. To address both challenges simultaneously, they rely on additional user supervision.

Consider the example of university records of students taking courses, subjects students are majoring in, and departments in a university, as shown in Figure 1. Suppose the user intends to discover a concept that explains students ’Alice’ and ’Bob’, but excludes ’Charlie’ and ’David’. The simplest explanation is that ’Alice’ and ’Bob’ take an undergraduate course (a course with ID less than 500) in the Engineering school. This explanation can be expressed as the SQL query shown in Figure 1c. The output examples, both positive and negative, are represented by entries from a single column (or a subset of columns) as in Figure 1b. We illustrate the two aforementioned challenges using this example.

Challenge 1: Learning the Join Policy. Learning the join policy corresponds to learning the projections and joins that correspond to the SELECT and FROM clauses in Figure 1c. Search-based query synthesis techniques have made significant strides in learning relational queries over multiple tables [27, 28, 39, 40, 43] by effectively enumerating the possible ways in which tables may be joined, thereby specializing in navigating the network of tables in a relational database. ILP techniques use language bias mechanisms such as mode declarations and meta-rules while program synthesis methods enumerate candidate programs using syntactic constraints or an explicit list of candidate rules to define a hypothesis space and explore the different key-foreign key pairs to join tables. Example-guided techniques rely on the underlying patterns in the data to discover them. All of these techniques struggle to address the challenge of synthesizing comparison predicates like courseID < 500 or school = Engineering required by the target query.
is interpreted as input relations $I$. Such a single table and then applying a feature selection method typically done by performing key-foreign key joins to construct a supervisory process to produce the single table they take as input. This is so given only the output examples. To obtain the correct decision tree, though there is no clear way to do this process. Additionally, the user must provide accurate labels to for databases with several tables, necessitating the automation of the learning algorithm. Manually doing so also becomes intractable.

This process can introduce data redundancy and impact the efficiency of the queries. On the other hand, decision tree learning techniques [30, 44] require additional supervision from the user in the form of an exhaustive list of constants that may be used in comparisons. On the other hand, decision tree learning techniques [30, 44] solve this problem in a limited setting in which the data is provided as a single table, where each row represents an instance of the input, and each column represents a feature of the instance [25, 29]. In such a setting, the labeling is given by a partition of the instances into positive and negative examples. These techniques then construct classifiers using greedy information gain heuristics to search for and combine locally optimal comparison predicates.

However, using these techniques requires additional user supervision to produce the single table they take as input. This is typically done by performing key-foreign key joins to construct such a single table and then applying a feature selection method [17]. For example, the user would have to provide the table from Figure 2a to the learning algorithm, along with the correct labels for each row, to obtain the decision tree in Figure 2b that corresponds to the WHERE clause in the target query. To obtain this table, the user must manually join the registration and department tables over the deptCode column. This process is tedious and protracted as it requires a careful analysis of the relational database, and errors in the process can introduce data redundancy and impact the efficiency of the learning algorithm. Manually doing so also becomes intractable for databases with several tables, necessitating the automation of this process. Additionally, the user must provide accurate labels to obtain the correct decision tree, though there is no clear way to do so given only the output examples.

Our Approach. We thus observe a dichotomy of existing techniques — they either support multi-table databases (as with search-based relational query synthesis) or excel at learning comparison predicates (as in the case of decision trees), but not both. In this paper, we leverage the strengths of the two paradigms to design an end-to-end algorithm for the synthesis of relational queries that feature both comparison predicates and joins across multiple tables. Specifically, we focus on the class of select-project-join (SPJ) queries like the one in Figure 1c which constitute an important fragment of relational algebra [20]. These queries feature equi-joins, that is joins across tables parameterized by a set of columns (with matching types), as well as categorical and numerical comparisons for selections.

Our key insight is to interpret the query synthesis problem as a search across a two-dimensional space defined by comparison predicates on one side and candidate joins on the other. To efficiently search through this space, we introduce an interleaved approach that allows us to leverage the strengths of both decision tree learning and search-based synthesis.

This interleaved approach seeks to address the challenge of finding the optimal join policy by enumerating different projection and join policies as partial queries, each producing a single intermediate table over which a decision tree could be learned to generate the comparison predicates for the target query. We use the example-guided search strategy [40] to enumerate these queries as it prioritize joins with fewer tables, thus synthesizing queries that contain only sufficient and necessary features from the database.

Once we generate a candidate single intermediate table, the next step is to synthesize comparison predicates. Classical decision tree
learning techniques require the rows of the intermediate table to be labeled, while in our setting, only certain constants (or tuples of constants) are labeled. There is no straightforward way to handle this discrepancy without additional user supervision. We fundamentally modify the classical decision tree learning algorithm ID3 [30] by changing the definition of entropy and information gain used by it and present it in Section 3.2.

Together, the search for candidate joins and the modified decision tree learning procedure can work in tandem to synthesize the relational queries with categorical and numerical comparisons. In practice, our algorithm synthesizes queries that are general (that is, it does not overfit the data) and of minimal size. We also prove the completeness of the algorithm—it synthesizes a query consistent with the training data iff there exists such a query. We implement this interleaved approach as Libra, and evaluate it on a benchmark suite of 1,475 instances of SPJ queries from the SPIDER [46] and GEOGRAPHY [12, 21, 47] datasets over 160 different databases, each with multiple tables. Libra solves 1,361 of these instances with a timeout of 10 minutes per task, and takes 58.9 seconds on average per instance. We also compare with state-of-the-art tools Scythe [12, 21, 47] and ParSQL that can synthesize select-project-join queries. They can solve 195 and 673 instances, and take 139.50 and 23.13 seconds on average per task respectively. All the benchmarks solved by the baselines are also solved by at least one instance of our framework, and our framework additionally solves a significantly larger set of the total benchmarks which the baselines fail to solve.

In summary, our work makes the following contributions:

1. We propose a novel approach to split the relational query synthesis problem into two separate sub-problems: synthesizing joins and learning comparison predicates.
2. We implement Libra, an end-to-end synthesis tool that realizes this algorithm, and demonstrate that it outperforms state-of-the-art approaches on a variety of tasks in terms of synthesis time as well as the quality of the synthesized programs.
3. Further, we show that the algorithm is complete and in practice generates general and minimal queries.

2 PROBLEM FORMULATION

In this section, we briefly review the syntax and semantics of relational queries and formulate the synthesis problem.

2.1 Syntax and Semantics

In this paper, we study select-project-join queries where selection supports categorical and numerical comparisons, and all joins are equi-joins. We will use SQL syntax to denote all queries.

To define the syntax of these queries, we first fix a set of input tables and a set of output tables. For simplicity, the columns of each table are of either of the kinds: categorical, numerical, and incomparable.

The syntax of the select-project-join queries is defined by the grammar in Figure 3. The JOIN operator featured in this query is an equi-join, that is a join parameterized by a set of columns \( \Theta \). Comparisons of the form \((T.c = k)\) or \((T.c \neq k)\) are supported only for columns of the categorical kind, all other comparisons are only supported for the numerical kind, and no comparisons are supported for the incomparable kind.

For this paper, the size of a query \( Q \) is defined as the sum of the number of input tables that are joined in it (that is the expansion of non-terminal \( J \)) and the size of the comparisons for the selection operator (\( \sigma \)) in disjunctive normal form. For instance, the size of the query \( Q_{\text{ex}} \) is 4 as it has a join of two tables (registration and department) and two comparison operators (registration.courseID \( \geq \) 500) and (department.school = "Engineering").

The semantics for these queries are as defined in classical works on relational algebra [1, 10, 20]. We denote the set of tuples produced by query \( Q \) on input tables \( I \) as \( \llbracket Q \rrbracket (I) \). In this paper we consider the set-semantics and not bag-semantics, that is, a relation is a set of literals with the same predicate (such as registration, instructor, and department).

2.2 Query Synthesis Problem

Our ultimate goal is to synthesize a select-project-join query (in the syntax of Figure 3) that is consistent with the given input-output
Algorithm 1 summarises the procedure and Figure 4 presents its input tables.

**Problem 2.1 (Query Synthesis).** Given input tables $I$ and a set of output tuples partitioned as $O^+$ and $O^-$, return a query $Q$ such that $O^+ \subseteq \{Q\}(I)$ and $O^- \cap \{Q\}(I) = \emptyset$, if such a query exists, and unsat otherwise.

We further refine the problem specification by identifying the columns of the input tables $I$ as either numerical, categorical, or incomparable. Numerical columns support ordered comparisons while categorical supports only equality checks. Consider the running example from the Introduction as in Figure 1. Given the input tables $I$ as registration, department, and major, we aim to synthesize a query $Q$ such that it generates both the positively labelled tuples in $O^+$ (Alice and Bob) and neither of the negatively labelled tuples in $O^-$ (Charlie and David). The query $Q_{EX}$ in Figure 1c is an example of such a query. Here, courseID is numerical while school is categorical.

### 3 ALGORITHM

In this section we describe the end-to-end Libra algorithm, which takes input-output examples $E = (I, O^+, O^-)$ as input and returns a relational query $Q$ consistent with $E$, that is, it solves Problem 2.1. Algorithm 1 summarises the procedure and Figure 4 presents its architecture.

We start with an example-guided search to construct a partial query with projection and joins (and without the comparisons for the selection operator). This partial query is constructed by analyzing patterns of co-occurrence of constants in the input-output examples Section 3.1 formalizes these patterns as enumeration contexts that can be translated into partial queries with projection and join operators. These correspond to the step 2 of the algorithm.

To synthesize categorical and numerical comparisons for the selection operator we turn to supervised learning. We maintain the enumeration contexts in a priority queue $L$ ordered by increasing size, where the size of each context is given by the number of tuples forming that context. For each context $C$ in $L$, we convert $C$ into a single table $T_C$ through a join of the input relation tables that occur in $C$. This is followed by the modified decision tree learning procedure (DTL) which completes the query. This corresponds to step 4a-4c.

---

**Algorithm 1** Libra($I, O^+, O^-$), where $I$ is the set of input tuples, and $O^+$ and $O^-$ are the sets of positively and negatively labeled output tuples respectively.

1. Set $\text{ans} = \text{unsat}$ and $N = \infty$.
2. For an arbitrary $t \in O^+$, let $C_t$ be the initial contexts that explain $t$ as defined in Equation 1.
3. Initialize the priority queue as $L = C_t$.
4. While $L$ is non-empty:
   a. Pick the smallest size element $C \in L$, and remove it from the queue: $L := L \setminus \{C\}$.
   b. If $|C| > N$, exit the loop and go to Step 5.
   c. For each table $T_C$ constructed using Equation 4:
      i. Let $\top$ be a node of an empty decision tree. Run $\text{DTL}(T_C, \top, O^+, O^-)$.
      ii. If $\text{DTL}(T_C, \top, O^+, O^-)$ returns a tree $\Delta$ such that $|\Delta| + |C| \leq N$ and entropy of $|\Delta|$ is 0, (A) Set $\text{ans} = Q(T_C, \Delta)$ as defined in Equation 6. (B) Set $N = |C| + |\Delta|$.
   d. For each tuple $t'$ that shares a constant with a tuple in $C$, update:
      $$L = L \cup \{C \cup \{t'\}\}.$$ 
5. Return $\text{ans}$.

In step 4d, we expand the context by one tuple. This corresponds to considering a join with an additional table. Therefore, the steps 4a-4c explore comparison predicates and step 4d explores joins. We thus search through the two-dimensional search space.

Throughout the algorithm, we maintain the size of the query $Q$ as variable $N$ and guarantee that the query $Q$ has minimal size among all queries consistent with the input-output example (subject to the optimality of the decision tree). Additionally, if no such query exists, the algorithm terminates and returns unsat. We prove this completeness result in Theorem 3.1.

Problem 2.1 is known to be co-NP complete in the case of queries without selection with categorical or numerical operators [40], and therefore we propose Algorithm 1 that runs in EXPTIME. Observe that the DTL sub-process is analogous to the ID3 algorithm and hence its running time is $O(nm \log m)$, where $n$ is the cardinality of input tuples $I$ and $m$ is the total size of output tuples $O^*$ and $O^-$ [31]. Therefore, Algorithm 2 runs in time polynomial in the size of the input. In step 4, for each context $C$ in queue $L$, we construct the possible tables $T_C$ which are polynomial in the size of the input and for each of them the process DTL is called exactly once. Observe that a context can be dequeued from $L$ at most once and hence, the number of calls to Algorithm 2 are at most polynomial in the total number of possible contexts. As any subset of the input tuples $I$ can be considered as a context, the end-to-end algorithm has EXPTIME complexity.

### 3.1 Example-Guided Enumeration of Projection and Joins

An enumeration context is a non-empty subset of the input tuples, $C \subseteq I$. The shaded part of Figure 5 corresponds to the context $C = \{\text{Alice, Comp.}, 201\}, \text{Comp.,Engineering}\}$. An enumeration
context \( C \subseteq I \) is said to explain a tuple \( t \in O^+ \) when for each column \( c \) of \( t \), there is a tuple \( t_c \in C \) such that for some column \( c' \) in \( t_c \), we have \( t.c = t_c.c' \).

Given a tuple \( t \in O^+ \), we construct the initial set of enumeration contexts for step 2 of Algorithm 1 by considering enumeration contexts:

\[
C_t = \{ \{ t_c : \text{for each column } c \text{ of } t, \\
\text{there exists a column } c' \text{ of } t_c \text{ such that } t.c = t_c.c' \} \}
\] (1)

In step 4d we extend a context \( C \) by adding a tuple \( t' \) such that for some \( t \in C \), there is an edge \( t \rightarrow t' \in E \) with appropriate labels. One can translate a context \( C = \{ t_1, \ldots, t_n \} \) and an output tuple \( t \) into a partial query with projection and joins.

Consider the output ‘Alice’ that is explained by the context

\[ C = \{(\text{Alice, Comp., 201}), (\text{Comp., Engineering})\} \]

This can be translated to the query:

\[
\text{SELECT registration.studentID} \\
\text{FROM registration JOIN department} \\
\text{ON registration.deptCode = department.deptCode}
\] (2)

This is because the constant ‘Alice’ occurs in the column studentID of registration, and the tuples from the tables registration and department share a constant for the columns department.deptCode and registration.deptCode. In general, given an output tuple \( t \) and a context \( C \subseteq I \), we first consider the sequence of columns \( (T_{i_1}, c_{i_1}, \ldots, T_{i_k}, c_{i_k}) \) from where we get the constants in \( t \). These will correspond to the columns for the projection operator. For each tuple \( t_i \in C \), we consider the table \( T_i \). These will correspond to the tables to be joined. In order to construct the parameters for the join, for each \( t_i \in C \), let \( E_t \) be the set of predicates of the form \( (t_i.c = t_i'.c') \) where \( t_i' \in \{ t_1, \ldots, t_i-1 \} \). Then, we can construct the queries of the form:

\[
\text{SELECT } (T_{i_1}, c_{i_1}, \ldots, T_{i_k}, c_{i_k}) \\
\text{FROM } (\ldots (T_1 \text{ JOIN } T_2 \text{ ON } \theta_2) \ldots \text{ JOIN } T_n \text{ ON } \theta_n)
\] (3)

Where each \( \theta_i \) is a conjunction of comparison predicates that label edges in \( E_t \). We consider all possible non-empty subsets of the labels in \( E_t \), and therefore, for each pair \( (t, C) \) of output tuple and context, there may be multiple candidate joins.

### 3.2 Supervised Learning of Comparisons for Selection

We now turn to decision trees to add a selection operator to the query in Equation 3. Our approach is motivated by the Iterative Dichotomiser 3 (ID3) algorithm for learning decision trees [30]. We first need to convert the tables \( T_1, \ldots, T_n \) in context \( C \) into one single table \( T_C \). This is achieved by implementing the join in Equation 3, that is we consider the output of the SQL query:

\[
\text{SELECT } * \text{ FROM } (\ldots (T_1 \text{ JOIN } T_2 \text{ ON } \theta_2) \ldots \text{ JOIN } T_n \text{ ON } \theta_n)
\] (4)

This join produces a single table. As before, each context corresponds to multiple joins and hence there are multiple candidates for \( T_C \). We consider all of them in our search.

We start by introducing some notation. Let \( \gamma \) be the schema of the output tuples, that is the types of the columns from which we draw output tuples. Let \( \pi_\gamma(T_C) \) represent the projection of \( T_C \) to
the columns in \( y \). Then, the entropy of a node \( N \) is defined as:

\[
    p = P(O^+| \pi_y(T_C), O^+ \cup O^-) = \frac{|\pi_y(T_C) \cap O^+|}{|\pi_y(T_C) \cap (O^+ \cup O^-)|}
\]

\[
    n = P(O^-| \pi_y(T_C), O^+ \cup O^-) = \frac{|\pi_y(T_C) \cap O^-|}{|\pi_y(T_C) \cap (O^+ \cup O^-)|}
\]

\[
    S(N) = -\left( p \log_2 p + n \log_2 n \right)
\]

Here, we restrict our analysis to output tuples that occur in \( O^+ \) or \( O^- \) only. Consider the joined table in Figure 2a. We can compute the entropy of the node with label (school = Engineering):

\[
    p = P(\{Alice, Bob\}|(Alice, Bob, Charlie, David)) = \frac{1}{2}
\]

\[
    n = P(\{Charlie, David\}|(Alice, Bob, Charlie, David)) = \frac{1}{2}
\]

\[
    S(N) = -\left( \frac{1}{2} \log_2 \frac{1}{2} + \frac{1}{2} \log_2 \frac{1}{2} \right) = 1
\]

Here, we do not consider 'Erin.' This is our first concrete modification to the decision tree learning algorithm.

A comparison predicate \( a \) splits the table \( T_C \) into two: \( \sigma_a(T_C) \) which comprises of rows that satisfy \( a \) and \( \sigma_{\neg a}(T_C) \) which comprises of rows that do not satisfy \( a \). Let \( \sigma_a(T_C) \) correspond to a node \( L \) and \( \sigma_{\neg a}(T_C) \) correspond to a node \( R \). Then we can compute their entropies \( S(L) \) and \( S(R) \) just as above. Consider the predicate (school = Engineering) which splits the joined table in Figure 2a into two tables as shown in Figure 6. Let \( \sigma_{\text{school} = \text{Engineering}}(T_C) \) form node \( L \) and \( \sigma_{\neg (\text{school} = \text{Engineering})}(T_C) \) form node \( R \). The entropies \( S(L) \) and \( S(R) \) are 0.918 and 1 respectively.

We can now compute the information gain. Information gain is defined as the difference between the entropy of the node and the weighted sum of the entropy of its children. That is, the information gain at node \( N \) is of the form:

\[
    IG(N) = S(N) - (\alpha S(L) + \beta S(R))
\]

where \( \alpha + \beta = 1 \). In a classical setting, the coefficients \( \alpha \) and \( \beta \) are the ratio of the number of rows corresponding to the child nodes \( L \) and \( R \). In our study, we focus on projection, and only the tuples in \( O^+ \) and \( O^- \). For ease of notation, let \( \pi_y(\sigma_a(T_C)) \cap (O^+ \cup O^-) \), the number of rows in \( \sigma_a(T) \), projected to columns \( y \), that occur in either \( O^+ \) or \( O^- \) be \( \lambda_a \) (and analogously for \( \sigma_{\neg a}(T) \) be \( \lambda_{\neg a} \)). Then information gain at Node \( N \) with comparison predicate \( a \) is defined as:

\[
    IG(N, a) = S(N) - \left( \frac{\lambda_a}{\lambda_a + \lambda_{\neg a}} S(L) + \frac{\lambda_{\neg a}}{\lambda_a + \lambda_{\neg a}} S(R) \right)
\]

In our running example, \( \lambda_a = 3 \) and \( \lambda_{\neg a} = 2 \). This gives us an information gain of 0.0328. The change in the weighted sum is our second concrete modification to decision tree learning.

The decision tree learning algorithm as described in Algorithm 2 starts with the table \( T \) and node \( N \) as an input. We introduce node \( N \) so we can call this procedure recursively. If \( O^+ \) or \( O^- \) is empty, we return the trivial tree with \( N \) as the only node. Otherwise, we construct a set of comparison predicates of the form \( (T.e \sim k) \), where \( T.e \) is a column of the table \( T \), \( k \) is a constant that occurs in the column \( T.e \), and \( \sim \) is a comparison operator (in our case either \( = \), \( < \), or \( \leq \)). Then, similar to the classical algorithm, we pick a comparison predicate \( a \) that maximizes the information gain \( IG(N, a) \). If no comparison predicate can maximize the information gain beyond 0, we return the trivial tree with \( N \) as the only node, labeled with '"?' and terminate the process. This is the case where there is no classifier for the given input data.

Otherwise, we split the table \( T \) on predicate \( a \) as tables \( \sigma_a(T) \) and \( \sigma_{\neg a}(T) \), introduce child nodes \( L \) and \( R \) corresponding to them, and call the DTL process recursively on the children of \( N \). When we call DTL on the \( L \) and \( R \) nodes, we ensure that the \( O^+ \) and \( O^- \) are updated to the output tuples that occur in \( \sigma_a(T) \) and \( \sigma_{\neg a}(T) \).

On executing the DTL procedure on our running algorithm, we get a tree as in Figure 7. Observe that it has a redundant right subtree, and one of the leaves is labeled '"?' instead of the desired tree.

The problem of finding a minimal decision tree, or even approximating it, is NP-complete [36]. Therefore we opt for a greedy search that is computationally efficient. Instead of considering all possible Boolean combinations of comparison predicates, Algorithm 2 makes locally optimal decisions, enabling it to handle large data-sets efficiently and produces satisfactory results in practice. While it is possible that locally optimal choices may not lead to the smallest decision tree, it most often leads to a good enough solution that is succinct and general, as observed in Section 4.

The soundness check of Algorithm 1 also ensures that while DTL may generate a larger tree, the synthesized query will always be consistent with
### Figure 6: In order to compute the information gain of a comparison predicate at a given node, we split the rows at the node into two parts, those that satisfy the predicate and the others that don’t. Here, we have split the joined table \( T_C \) (from Figure 2a) on the predicate \( \text{school} = \text{Engineering} \).

<table>
<thead>
<tr>
<th>studentID</th>
<th>deptCode</th>
<th>courseID</th>
<th>school</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Comp.</td>
<td>201</td>
<td>Engineering</td>
</tr>
<tr>
<td>Bob</td>
<td>Mech.</td>
<td>320</td>
<td>Engineering</td>
</tr>
<tr>
<td>Alice</td>
<td>Mech.</td>
<td>550</td>
<td>Engineering</td>
</tr>
<tr>
<td>Bob</td>
<td>Mech.</td>
<td>550</td>
<td>Engineering</td>
</tr>
<tr>
<td>David</td>
<td>Comp.</td>
<td>500</td>
<td>Engineering</td>
</tr>
<tr>
<td>David</td>
<td>Mech.</td>
<td>502</td>
<td>Engineering</td>
</tr>
</tbody>
</table>

(a) Table with rows of \( T_C \) that satisfy the predicate (\( \text{school} = \text{Engineering} \)).

<table>
<thead>
<tr>
<th>studentID</th>
<th>deptCode</th>
<th>courseID</th>
<th>school</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Chem.</td>
<td>310</td>
<td>Arts and Science</td>
</tr>
<tr>
<td>Charlie</td>
<td>Chem.</td>
<td>310</td>
<td>Arts and Science</td>
</tr>
<tr>
<td>Erin</td>
<td>Chem.</td>
<td>310</td>
<td>Arts and Science</td>
</tr>
</tbody>
</table>

(b) Table with rows of \( T_C \) that do not satisfy the predicate (\( \text{school} = \text{Engineering} \)).

---

While there exist any unexplained tuples, we dequeue a node from the queue and branch it out as described in Algorithm 2. Instead of calling the process recursively, we enqueue the children and then eventually get to them only when there are unexplained tuples. This lazy evaluation allows us to generate smaller trees with fewer redundancies. With this modification, we get the desired tree depicted in Figure 2b.

In summary, we make three modifications to adapt the classical decision tree learning algorithm to our setting: first we modify the entropy formula in order to support partial labeling, then we change the weights in the information gain formula to overcome the ambiguity that results from projections and partial labeling, and finally, we implement a lazy version of DTL to generate small decision trees that act as partial separators between \( O^+ \) and \( O^- \) to avoid producing redundancies. Note that these modifications do not compromise any guarantees about termination of the procedure or size of the learned decision tree [29].

### 3.3 Interleaving Decision Tree Learning with Example-Guided Search for Joins

A decision tree \( \Delta \) can then be converted to a boolean formula \( \sigma_\Delta \) in disjunctive normal form. For each leaf of the tree that is marked \( \checkmark \), we consider a clause that is composed of the conjunction of the predicate at its parent node (if the node is a left child, and the negation of the predicate otherwise). And then, we construct the disjunction of each of these clauses. For example, we can translate the tree in Figure 2b to the formula \((\text{courseID} < 500) \land (\text{school} = \text{Engineering})\). The negations, if any, can be removed by considering the negated comparison operators (\( \neq \), \( > \), and \( \geq \)).

Therefore we can convert a joined table \( T_C \) and decision tree \( \Delta \) into a query \( Q(T_C, \Delta) \) by using the boolean formula \( \sigma_\Delta \) to complete the query in Equation 3. This gives us the query:

\[
\text{SELECT } T.c_1, \ldots, T.c_k \\
\text{FROM } (\ldots (T_1 \text{ JOIN } T_2 \text{ ON } \theta_1) \ldots \text{ JOIN } T_n \text{ ON } \theta_{n-1}) \text{ WHERE } \sigma_\Delta
\]

(6)

Figure 4 summarizes Libra. The end-to-end algorithm guarantees completeness:
Theorem 3.1 (Completeness). If there exists a relational query consistent with the input-output example \( E = (I, O^+, O^-) \), then \textsc{Libra} produces a query \( Q \) consistent with \( E \).

The proof of this theorem relies on the completeness of the example-guided enumeration and the completeness of decision tree learning. We assume the reader is familiar with the analogous guarantees for example-guided synthesis of conjunctive queries [40], and those for classical decision trees. Observe that if a context \( C \) explains a tuple \( t \), then all contexts \( C' \supseteq C \), also explain \( t \). We can consider the largest context \( C = I \), that is, the set of all input tuples, to prove the following lemma:

Lemma 3.2. If there exists a relational query consistent with the input-output example \( E = (I, O^+, O^-) \), then there exist a decision tree \( \Delta \) with predicates of the form \((T.c > k)\) where \( T \) is an input table, \( c \) is a column of \( T \), and \( k \) is a constant in the column \( c \), such that the query \( Q(I; \Delta) \) is consistent with \( E \).

It follows from the completeness of example-guided enumeration that a decision tree must exist, however, it remains to show that the predicates for the decision tree must be of the said form. Without loss of generality, suppose the comparison predicate is of the form \((T.c > k_1)\), where \( k_1 \) does not occur in \( c \). The arguments for other comparison operators are analogous. Observe that must exist the greatest lower bound \( k_2 \) of \( k_1 \) in \( c \) (that is, \( k_2 = \max(k \in c : k < k_1) \)). Replacing the predicate by \((T.c > k_2)\) does not change the semantics of the query with respect to input \( I \), as there are no constants in between \( k_1 \) and \( k_2 \). By systematically replacing the predicates in a query consistent with \( E \), we can prove that there must exist a query where the selection operator corresponds to a decision tree of the said form. As we exhaustively enumerate all possible predicates, we can guarantee:

Lemma 3.3. Given a table \( T \), a node \( N \) and output tuples partitioned as \( O^+ \) and \( O^- \), if there exists a decision tree that separates \( O^+ \) from \( O^- \), then \( \text{DTL}(T, N, O^+, O^-) \) will return such a tree.

Together, Lemma 3.2 and Lemma 3.3 can prove Theorem 3.1.

Additionally, observe that at each step of the algorithm, we maintain the constant \( N \) that tracks the size of the query. As the contexts are maintained in increasing order of size, the number of joins in the enumerated queries is always increasing.

4 Evaluation

We have implemented the \textsc{Libra} algorithm in Scala. In this section, we evaluate it on a large-scale benchmark suite. First, we measure the performance of our algorithm compared to state-of-the-art synthesis tools. We do so by comparing the number of instances solved by each tool and the time taken by each tool to do so. Next, we evaluate the generality of the solutions generated by each tool. To do so at scale, we leverage Occam’s razor to use the succinctness of a query as a proxy of how specific a query is to the training data. We then test the sensitivity of \textsc{Libra} to partial labels. We do so by dropping a fraction of labels and evaluating the number of instances solved by \textsc{Libra}. As such, we propose to answer three main research questions:

Q1. Performance: How effective is \textsc{Libra} on synthesis tasks from different domains in terms of synthesis time?

Q2. Succinctness: How large are the programs synthesized by \textsc{Libra} compared to the reference solution?

Q3. Sensitivity to Partial Labels: How is the performance of \textsc{Libra} affected by partial labels?

We discuss our benchmark suite in Section 4.1 and the baselines against which we compare \textsc{Libra} in Section 4.2 along with the setup for each. We present our findings in Sections 4.3, 4.4, and 4.5. We performed all experiments on a Linux server with an 18-core, 36-thread, 3 GHz Xeon Gold 6154 CPU and 394 GB of RAM.

4.1 Benchmarks

In order to effectively evaluate \textsc{Libra}, we put together a benchmark suite of SPJ instances with the following requirements based on our problem statement: first, the queries must be over a variety of different databases; second, some of them must query more than one table; and last, some of them must contain at least a comparison with a constant. Upon exploring various query datasets, we find the \textsc{Spider} [46] and \textsc{Geography} [12, 21, 47] datasets to contain queries that satisfy our requirements. We, therefore, evaluate \textsc{Libra} on the set of all SPJ instances from the \textsc{Spider} [46] and \textsc{Geography} [12, 21, 47] datasets. \textsc{Spider} is an open-access large-scale manually annotated dataset. There are 1,203 SPJ instances in the \textsc{Spider} dataset over 159 databases. On the other hand, \textsc{Geography} is a dataset of SQL queries about US geography. We use version 4 of the modified \textsc{SQL} dataset for \textsc{Geography} from [12, 21, 47]. Additional statistics for each benchmark are provided in Table 1. Upon deduplication of the queries, we extract 272 SPJ instances, all over the same database, giving us a total of 1,475 instances over both datasets. For each benchmark, we consider the tables from its corresponding database as the input tables and the result of running the ground truth query over that database as the output table.

Each benchmark has 2 to 26 input tables (with a median of 8), each with 1 to 352 columns (with a median of 30), containing 8 to around 553k tuples in the input tables (with a median of 937). Additionally, each benchmark is labeled with a ground truth query that serves as a reference solution. This reference solution is used to obtain the output examples for the corresponding benchmark. Overall, the reference solutions feature a join of at most 6 tables and the use of at most 3 predicates.

4.2 Baselines and Setup

We compare \textsc{Libra} against baselines that are at least as expressive as \textsc{Libra} itself. We therefore compare \textsc{Libra} against the following baselines in Sections 4.3 and 4.4: \textsc{Scythe} [43], which synthesizes SQL queries using enumerative search, and \textsc{PatSQL} [39], which uses
We run PatSQL on all 1,475 benchmarks with a timeout of 10 minutes and summarize the performance of each tool in a cactus plot in Figure 8. From this figure, we see that Libra solves the most number of benchmarks, solving 1,361 out of 1,475 in an average of 58.9 seconds, and solves 1,097 of those within 10 seconds. Of the 1,361 solved benchmarks, 1,090 are SPJ instances from the SPIDER dataset, while 271 are from the GEOGRAPHY dataset.

The plot for Libra plateaus at 600 seconds since it searches for a minimal solution to a benchmark, but returns the best solution found so far when it times out. PatSQL is outperformed by Libra, solving 673 benchmarks in an average of 23.13 seconds, and 548 in 10 seconds. SCYTHE solves only 195 benchmarks, in an average of 139.50 seconds, and only 15 in 10 seconds. All of the benchmarks solved by PatSQL and SCYTHE are instances from the SPIDER dataset; neither tool solves a single instance from the GEOGRAPHY dataset. Also, PatSQL solves 2 benchmarks unsolved by Libra, while all benchmarks solved by SCYTHE are solved by Libra and PatSQL.

Among the benchmarks that Libra uniquely solves, a significant portion of the benchmarks have ground truths involving many joins, but with a few shared constants between tables, leading to a sparse tuple co-occurrence graph while there are syntactically many possible joins. The following generated query which Libra is the only one to produce (and which happens to match the reference solution) shows how the example-guided technique allows for learning very large queries and combined with decision tree learning allows for learning complex SPJ queries:

```sql
SELECT employee.emp_fname, class.class_room
FROM ((class JOIN employee ON class.prof_num = employee.emp_num)
JOIN professor ON employee.emp_num = professor.emp_num)
JOIN department ON department.dept_code = professor.dept_code)
WHERE department.dept_name = "Accounting"
```

The example-guided strategy used by Libra allows it to explore solutions of a larger size more quickly than syntax-guided strategies since the smaller joins that are syntactically valid but don’t explain any output tuple are skipped. This results in Libra solving benchmarks with reference solutions of a larger size where other baselines would require a longer time to search through the hypothesis space despite the additional supervision provided.

However, it is difficult to scale Libra over larger input databases. The largest benchmark solved by Libra consists of 5303 tuples in its input, in 4 tables and 26 columns, and 20 tuples in its output. The size of its target query is 5. On the other hand, for the 114 benchmarks unsolved by Libra, over 70% have more than 5,000 tuples, and all have tables with over 20 columns, with a median of 64 columns. Libra faces two main issues when solving these benchmarks. First, it may struggle to build the tuple co-occurrence graph that it uses to enumerate contexts, and second, frequently occurring constants can result in a large number of contexts being enumerated. The second case is true for the 2 benchmarks that PatSQL solved which were unsolved by Libra, since they contained 43 and 20 columns, with 103 and 577 rows respectively. However, the ground truth solutions for those benchmarks could be easily explored by syntax-guided processes, with one of the benchmarks consisting only of joins, and so PatSQL was able to synthesize them.

```
SELECT campusfee FROM campuses
JOIN csu_fees ON campuses.id = csu_fees.campus
WHERE (campuses.campus = "San Francisco State University")
AND (csu_fees.year = 1996)
```
Instead of this solution, Libra generates the query:

```sql
SELECT campusfee FROM csu_fees
WHERE (csu_fees.campus = "18")
```

This is because the campus name "San Francisco State University" occurs only once in campuses with id "18", and the only row with campus "18" in csu_fees has year of 1996. Therefore, the conjunction on both the campus name and year is unnecessary and there is also no longer a need for the join of campuses with csu_fees since selecting campus "18" directly from csu_fees is sufficient.

There are 22 benchmarks where the size of the query generated by Libra is larger than the reference solution. On manual inspection of these benchmarks, we observe that the larger size is due to the sub-optimal size of the decision tree generated by DTL. As discussed before, the problem of finding a minimal decision tree is intractable and hence we adopt a greedy heuristic-based search. Therefore, any minimality guarantee will be subject to the performance of the decision tree, but we quantitatively observe that 99% of the synthesized programs are minimal.

For the baselines, we observe that the size of the synthesized programs is usually large. In contrast to Libra, only 115 of the 673 (17%) programs synthesized by PatSQL are minimal, and only 99 of the 195 (51%) programs synthesized by Scythe are minimal.

4.5 Sensitivity to Partial Labels

We conduct experiments in order to evaluate the ability of Libra to solve problems with partial labels. Since neither PatSQL nor Scythe support partial labels, we do not compare Libra with them.

We run sensitivity experiments by dropping a fraction of both positive and negative labels. To do this, we choose the desired fraction X% of labels to be dropped. We then choose with uniform randomness X% of the positive and X% of the negative tuples, and drop them from the instance. Dropping these tuples effectively simulates a partially labeled instance, since these tuples are now labeled as unknown. We then ask Libra to solve these instances. This process is repeated for each benchmark with increasingly
larger fractions of dropped tuples, from 10% to 70%. We conduct this experiment three times for each fraction to account for randomness.

We measure both the average F1 score of the solved benchmarks (shown in blue in Figure 10) and the number of perfectly solved benchmarks, i.e. solved with an F1 score of 1.0 (shown in red in Figure 10). We see from Figure 10 that the number of perfectly solved examples drops from 1090 by around 7.8% when 10% of the tuples are removed, and drops by around 46.8% when 70% of those tuples are removed. The average F1 score remains relatively high, dropping by only 0.12 points when 70% of the labels are removed.

This shows that even after removing a significant proportion of labels, positive or negative, Libra can synthesize queries that come close to representing user intent, even if the synthesized query is not the target solution itself.

5 RELATED WORK

We discuss related work on the synthesis of relational queries, inductive logic programming (ILP), decision tree learning, and example-guided search.

5.1 Synthesis of Relational Queries

Program synthesis and Inductive Logic Programming have made significant progress in the synthesis of rules from input-output examples. Each of these techniques specializes to support specific features (such as aggregation, invented predicates, recursive predicates, etc.) and require additional instance-specific supervision in the form of templates such as predicate signatures, mode declarations, or candidate rules.

GenSynth [28] is an evolutionary search-based approach for synthesizing Datalog programs. The evolutionary strategy maintains a population of candidate programs that are incrementally mutated to optimize the fitness score for those programs. GenSynth can synthesize recursive predicates when the signatures of invented predicates are specified (unless they coincide with that of an input or output relation). Scythe [43] and PatSQL [39] are enumerative techniques that require the exhaustive list of constants that may occur in the selection operator and the list of aggregation operators that may be used by the query. Scythe uses a two-phase approach where it first searches for partial queries that can potentially explain the training data, and then instantiates them and ranks the results. PatSQL also adopts a two-phase approach, but it additionally uses properties of relational algebra to rewrite the partial queries so that a combinatorial blow-up (due to the projection operator) can be avoided.

While Scythe and PatSQL are both two-phase approaches, they differ from Libra in two concrete ways. Both the phases for Libra are data-driven (the first phase is built on EGS while the second is a modified decision tree learning algorithm) while Scythe and PatSQL use enumerative techniques that do not take the input-output data into consideration. And secondly, the interaction between the two phases in Libra are interleaved so as to minimize the number of joins and the number of comparison predicates, while PatSQL and Scythe run sequentially.

Additionally, the problem of synthesis of relational queries has been studied in an interactive setting [2, 42], and from natural language [11, 23, 45] and other specifications [5, 49].

5.2 Inductive Logic Programming

ILASP [27], a constraint solving-based approach, can synthesize recursive Answer Set Programs producing constraints over the hypothesis space. The hypothesis space for ILASP is defined using mode declarations that bound the number of joins and variables in the synthesized rules. Extensions of ILASP can support noisy data.

Popper [9] is a constraint solving ILP technique that implements the ‘learning from failures’ strategy. It supports Answer Set Programs and Prolog syntax and can reason about lists, numbers, and textual data. Similar to ILASP, Popper uses mode declarations to restrict the expressiveness of the queries, and requires instance-specific supervision from the user in form of these modes.

First-Order Inductive Learner (FOIL) [32] is an iterative algorithm that searches for a first-order clause that maximally covers a set of positive examples while minimizing the number of negative examples covered. QuickFOIL [48] is a parallel and scalable implementation of the FOIL algorithm and supports scalable and robust learning using parallelization. While FOIL and QuickFOIL do not restrict the hypothesis space using mode declarations, they employ greedy heuristics that do not explore the space of all possible rules and hence cannot guarantee the completeness of the search.

These techniques also do not support numerical reasoning as required for the synthesis of comparison predicates. In order to support such predicates, the user must augment the input database with additional tables corresponding to each comparison constant, causing a blow-up in space and significant increases in run-time.

5.3 Decision Tree Learning

In this paper, we implement a straightforward decision tree learning algorithm based on the Iterative Dichotomiser 3 (ID3) algorithm [30]. Several modifications and extensions of this algorithm have been studied [34, 35, 38]. The applications of decision trees for invariant synthesis [14, 24] as well as to support noisy and uncertain data [15, 41] have been studied. Ideas and algorithms from these works can be integrated into query synthesis.

MRDTL is a decision tree learning algorithm that supports the multi-table setting [18]. However, it assumes a primary table whose
rows are labeled as positive and negative examples. In that respect, it supports joins (by drawing information from other tables in the database) and comparison predicates (through the decision tree), but not projection. It differs from our setting, especially in the case where projection may draw columns from different tables.

The problem of summarizing a multi-table input database in a single table is studied in the context of supervised learning. Hamlet [25] is a system of handcrafted decision rules to predict joins across the input database so analysts balance between performance and accuracy. ARDA [7] proposes a framework for automated relational data augmentation which can discover joins of input tables efficiently and improve the performance of predictive models in a multi-table setting.

Applications of decision tree learning in program synthesis have been studied in the context of syntax-guided search for programs with conditional operators. EUSolver [3] uses a divide-and-conquer technique that first enumerates expressions that are consistent with a subset of input-output examples and then constructs a decision tree to combine these expressions using conditional predicates in order to scalably synthesize complex programs. DivandSynth [19] is a synthesis tool for conditional linear integer arithmetic (CLIA) programs that uses decision trees to represent conditional programs and uses a combination of enumerative and symbolic reasoning techniques to synthesize them. ESolver [22], a unification-based solver for programming-by-example, enumerates an expression for each example and then incrementally identifies unification strategies for them using decision trees. While these tools combine search-based synthesis and decision trees, they do not tackle the challenge of learning relational queries over multiple tables.

5.4 Example-Guided Search

Example-guided and data-driven search techniques have previously been used for synthesizing relational queries, regular expressions, string transformations, and spreadsheet operators. EGS [40] is an end-to-end synthesis engine for conjunctive queries that feature only projection and equi-joins. As discussed in Section 3.1, the search technique for joins in Libra is motivated by this work. However, we differ from it primarily in the way in which we summarize the data and explore the input tables. In particular, EGS constructs a constant co-occurrence graph and then uses the EXPLAINCELL procedure that explains an output tuple in a column-by-column fashion. Instead, we do not need any such construction and explain all columns of an output tuple at once based on our construction of the initial context.

FlashFill [16], a tool available in Microsoft Excel which synthesizes string transformations using input-output pairs to generate trace expressions and then uses these expressions to construct a program that is consistent with the training data. This is analogous to our construction of partial queries (with only join and projection) from input-output examples, and then using a modified decision tree to add a selection operator.

Beyond program synthesis, example-guided search techniques have also been used for domains such as graph search and grammatical inference [4, 6].

6 LIMITATIONS AND FUTURE WORK

We discuss some limitations of Libra in this section. First, Libra does not support aggregation operators and nested queries. While it is certainly a desirable feature, including support for such operators can be extremely computationally expensive, and may require sacrificing other features to be supported. For instance, while Scythe and PatSQL do support aggregation, they do so at the cost of additional supervision. Second, the underlying algorithm of Libra is not incremental, and as such does not support incremental updates to the input database. However, since for most cases, the turnaround time of Libra is relatively low, Libra can be rerun upon every update to the database. Third, as discussed in Section 4.3, Libra struggles to scale up as the sizes of the input database and the number of constants increase. Syntax-guided approaches face similar difficulties as highlighted by the performance of PatSQL and Scythe, especially when synthesizing target queries that involve large numbers of joins. Works such as [26] explore joinable tables in a database, but require additional heuristics. In the future, we intend to further explore the issue of finding optimal joins to address problems with scalability.

7 CONCLUSION

We have presented a novel approach to the problem of synthesizing select-project-join queries from input-output examples. Our insight is to view this problem as a two-dimensional search for synthesizing joins and learning comparison predicates. We designed an example-guided enumerator to synthesize joins and modify the classical decision tree learning technique to learn predicates (while maintaining the guarantees about the termination of the procedure and size of the learned tree). We then propose a way to interleave the two to synthesize relational queries with categorical and numerical comparison predicates. We show that our algorithm strongly encourages minimalism of the synthesized program and we prove the completeness of its search. We implement the algorithm in a tool named Libra and evaluate it to show that it outperforms state-of-the-art approaches on a variety of tasks in terms of synthesis time as well as the quality of the programs.

We outline three lines of future research for this work. Firstly, we would like to extend Libra to support the full SQL syntax including disjunction (union), aggregation operators, and nested queries. Secondly, the short synthesis time of Libra creates an opportunity to develop an interactive interface that allows the user to provide real-time feedback on the synthesized query. This allows the user to start with a small input-output example and progressively add more complexity to cover all features of the desired query. Finally, the insight of example-guided search for candidate joins can be developed agnostic to the downstream supervised learning task (as done for Hamlet and ARDA [7, 25]). This can allow us to develop an end-to-end system that can translate a multi-table database to a single augmented table that can balance performance with generalizability.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful suggestions that significantly improved the paper. This research was supported by NSF grant #2107429.