QuickFOIL: Scalable Inductive Logic Programming

Qiang Zeng
University of Wisconsin–Madison
qzeng@cs.wisc.edu

Jignesh M. Patel
University of Wisconsin–Madison
jignesh@cs.wisc.edu

David Page
University of Wisconsin–Madison
page@biostat.wisc.edu

ABSTRACT

Inductive Logic Programming (ILP) is a classic machine learning technique that learns first-order rules from relational-structured data. However, to-date most ILP systems can only be applied to small datasets (tens of thousands of examples). A long-standing challenge in the field is to scale ILP methods to larger data sets. This paper presents a method called QuickFOIL that addresses this limitation. QuickFOIL employs a new scoring function and a novel pruning strategy that enables the algorithm to find high-quality rules. QuickFOIL can also be implemented as an in-RDBMS algorithm. Such an implementation presents a host of query processing and optimization challenges that we address in this paper. Our empirical evaluation shows that QuickFOIL can scale to large datasets consisting of hundreds of millions tuples, and is often more than order of magnitude more efficient than other existing approaches.

1 Introduction

A key promise of the on-going big data revolution is to extract deep knowledge from large datasets. Naturally, there has been substantial interest in integrating machine learning (ML) inside database management systems (DBMSs) for ease-of-use, scalability, and manageability. In the past few years, a number of ML methods such as classification, clustering, stochastic gradient descent, and information extraction have been integrated into the DBMS (e.g. [13, 16, 21, 25]).

Most ML algorithms studied so far for DBMS integration assume a *propositional* representation of data. In other words, each data instance is represented by a single feature vector, where each feature characterizes a property of the instance. However, feature representation-based approaches cannot directly capture the relationships between different features, and the relationships between different data instances. Feature-based learning methods either assume that both features and instances are independent, or use correlation in terms of probability to represent latent relationships. Explicit semantic relationships, such as structural, spatial and temporal relationships, are not modeled. These inadequacies not only limit how the datasets are described, but also limit the ability to learn over the relationships.

(a) Training examples		(b) Background knowledge		
U(Daniel, Jacob)	U(John, Jason)	B(Andrew, Jacob)	P(Daniel, Andrew)	S(Daniel, June)
U(Jason, Andrew)	U(Noah, John)	B(Jason, Noah)	P(Jason, Jacob)	S(Daniel, Jennifer)
U(Noah, Andrew)	U(Jason, Justin)	B(Jacob, Andrew)	P(Noah, Jacob)	S(Daniel, Rachel)
U(Daniel, William)	U(Noah, Justin)	B(Noah, Jacob)	P(Noah, Justin)	S(Daniel, Jason)
		B(Owen, William)	P(Jimmy, Jason)	S(John, William)
				S(Noah, Gwen)
				S(Jason, Sara)

Figure 1: The first-order logic form of the Uncle relationship concept learning problem. The negative examples are underlined. For simplicity, we denote a predicate by its initial (e.g. B for Brother and P for Parent). A fact $L(X, Y)$ can be read as "Y is a L of X".

Inductive Logic Programming (ILP), a subfield of machine learning, can mine data with complex relational structures. Instead of using features, it represents both the data and the learning problem in first-order logic. Furthermore, the output of ILP is a set of Horn clauses, which makes it easier for humans to interpret and use the ILP prediction.

Definition 1 (ILP) *Given the background knowledge B, positive examples E_+, and negative examples E_-, all being a set of Horn clauses, the ILP task is to find a hypothesis H, composed of a set of Horn clauses, such that*

- *$\forall p \in E_+, H \wedge B \models p$ (completeness)*
- *$\forall n \in E_-, H \wedge B \not\models n$ (consistency)*

In Definition 1, $\models$ denotes the classical logical consequence. The positive examples and the negative examples are usually only ground literals (i.e. clauses without free variable and with an empty body). The background knowledge is composed of a finite set of facts, and a finite set of rules that can be transformed to its extensional representation. Completeness requires that the learnt hypothesis with the background knowledge can prove all positive examples, while consistency requires that it cannot derive any negative examples. These two criteria can be relaxed to tolerate noisy data.

Example 1 Consider a simple task of learning a concept $Uncle(X, Y)$, defined on pairs of people X, Y. The value of $Uncle(X, Y)$ is true if Y is an uncle of X. Figure 1 shows an instance to illustrate this task, where the data is represented in first-order logic form. The training examples consist of four positive examples and four negative examples. Each positive example is a fact of the predicate $Uncle$ that is known to be true, while each negative example indicates a pair of people are not connected by the uncle relationship. The background knowledge includes the brother, the parent

Proceedings of the VLDB Endowment, Vol. 8, No. 3
Copyright 2014 VLDB Endowment 2150-8097/14/11.

and the sister relationships, which are represented by three predicates: *Brother*, *Parent* and *Sister*, respectively. The output of an ILP algorithm could be the following "learnt" hypothesis/rule expressed as a Horn clause:

$$Uncle(X, Y) \text{ :- } Brother(Z, Y), Parent(X, Z)$$

The declarative rule simply states that if a person Y is a brother of a parent of X, then Y is an uncle of X.□

There are a number of applications of ILP [37], including concept-learning (as illustrated above). Interestingly, ILP can also be used as a method to augment existing traditional machine learning methods, for example for feature selection [34].

Unfortunately, most ILP systems cannot handle datasets beyond thousands of training examples [43]. In fact, the available testbeds for ILP usually contain only dozens of positive and negative examples. However, many enterprise and scientific datasets increasingly have larger and more complex datasets. Applying ILP to such data is very promising, as ILP is able to handle data with rich relationships. Thus, it is crucial to find ILP methods that scale with increasing data volumes. This paper proposes a new ILP method called QuickFOIL that addresses this need.

QuickFOIL is a top-down approach that constructs one (Horn) clause at a time. In constructing a clause, it starts with the most general one with an empty body, and then adds new literals, one at a time, using a greedy heuristic.

In each iteration, ILP algorithms have to make crucial decisions about a) the scoring function that it uses to drive the heuristic search process, and b) how to prune portions of the "search" space. QuickFOIL innovates on both these aspects and uses a novel scoring function, and a new technique to *prune redundant literals*. Another aspect of QuickFOIL is its mapping to relational operations. With this mapping, the task of computing the score of a literal translates into four aggregate binary join operations. The space of candidate literals for a predicate is $O(n^k)$, where n is the number of variables in the clause and k is the arity of the predicate. In other words, a naïve mapping hinders efficiency and scalability. We develop a novel query processing method to reduce the number of join operations from $O(n^k)$ to $O(nk)$ single-predicate joins that also share accesses to the inner relation. We then develop multi-query optimization and caching techniques to further speed up the processing of these queries. The collective sum of these contributions produces an ILP method that is efficient and scalable to much larger datasets than previous methods.

The remainder of this paper is organized as follows: Section 2 describes the underlying algorithm that is used in QuickFOIL. Section 3 describes QuickFOIL's scoring function and pruning approach, where as Section 4 describes the mapping to database operations. Empirical results are presented in Section 5, and Section 6 discusses related work. Finally, Section 7 contains our concluding remarks.

2 Preliminaries

In this section, we first describe some basic terminology, and then describe a generic top-down greedy ILP algorithm that we use as the skeleton for QuickFOIL.

2.1 Definitions

An *atom* is of the form $L(X_1, \ldots, X_k)$, where L is a predicate symbol and X_i is a variable, a constant, or a function. A *literal* is either an atom or its negation. A *clause* is a disjunction of literals: $\forall X_1 \forall X_2 \ldots \forall X_n (L_1 \vee L_2 \vee \ldots \vee L_n)$. A *Horn clause* is a clause that has at most one positive (i.e. unnegated) literal. In this paper, a clause with $n+1$ literals is written in the form: L_h :- $L_1, \ldots, L_n$, where L_h is the positive literal called the head of the clause, and the right side is the body. A clause with an empty body and no variables is a *ground fact* of the head predicate.

The *hypothesis language* for an ILP system is composed of clauses that the ILP system is able to build for the specified ILP problem. The hypothesis language that we study in this paper is *function-free Horn clauses*.

For an ILP learning problem, the predicate corresponding to the training examples is called the *target predicate*, the predicates that appear in the background knowledge are *background predicates*. In Example 1, the target predicate is *Uncle*, and the background predicates include *Brother*, *Parent* and *Sister*. There should be only a single target predicate for an ILP learning task. The target predicate can also be a background predicate if recursion is allowed. Unless otherwise specified, whenever we refer to a clause, the head predicate is the target predicate.

A *background relation* is the set of ground facts (a.k.a. *extensions*) for a background predicate, and the *target relation* is the set of ground facts (extensions) for the target predicate. For a predicate/literal L, we use the bold-faced symbol $\mathbf{L}$ to denote the corresponding relation.

2.2 Generic top-down, greedy ILP algorithm

The ILP learning problem can be viewed as a search problem for clauses that deduce the training examples. The search can be performed bottom-up or top-down. A bottom-up approach builds most-specific clauses from the training examples, and searches the hypothesis space by using generalization. It is best suited for incremental learning from a few examples. In contrast, a top-down approach starts with the most general clauses and then specializes them. It is better suited for large-scale datasets with noise, since the search can be easily guided by heuristics.

Algorithm 1 sketches a generic top-down ILP algorithm using a hill-climbing heuristic. It contains two loops: i) a clause construction loop that builds a clause at a time, and ii) a nested literal search loop that specializes the current clause that is being constructed. The literal search procedure begins with the most general clause (with an empty body) and incrementally adds literals that maximize a specified scoring function until the *necessary stopping criterion* is met (Line 4–12). After building one clause, the algorithm updates the training examples by removing the covered positive examples before constructing a new rule (Line 14). A *sufficient stopping criterion* is used to decide when to stop adding clauses to the hypothesis. For noise-free data, the sufficient stopping criterion requires that the hypothesis covers all the positive examples and none of the negative examples; the necessary stopping criterion requires that the building clause has no negative tuple (i.e. $T_{-} = \emptyset$).

Next we describe three key steps in more detail.

Creating the binding set. The *binding set*[1] of a clause is the set of bindings for every variable in the clause. We call a tuple in the binding set a *binding tuple*. A binding tuple has an arity equal to the number of variables in the

[1] The binding set is called the training set in prior work [36]. We use the training set to refer specifically to the training examples.

Algorithm 1: Generic top-down, greedy ILP algorithm

```
Input: Background knowledge $B$, target predicate $R$,
       positive examples $E_+$, negative examples $E_-$
Output: a set of Horn clauses $H$
$H \leftarrow \emptyset$; $U \leftarrow E_+$
while sufficient stopping criterion is not satisfied do
    $T_+ \leftarrow U$; $T_- \leftarrow E_-$
    $C \leftarrow R$ :- true
    while necessary stopping criterion is not satisfied do
        generate new candidate literals
        for each candidate literal do
            compute the score using a scoring function
        select a literal $L$ with the maximum score
        add $l$ to the body of $C$
        Create the new positive binding set $T_+$ for $C$
        Create the new negative binding set $T_-$ for $C$
    $H \leftarrow H \cup C$
    $U \leftarrow U - \{c \in U | H \wedge B \models c\}$
```

corresponding clause. The binding set of a clause is the basis on which we compute a score to evaluate the quality of the clause. Given a set of training examples, it can be computed recursively as follows. (We provide an example of the binding set below in Example 2.)

- The binding set for the most general clause is the set of training examples. A binding tuple is positive (negative) if it corresponds to a positive (negative) example.
- For a clause C and its binding set T, the binding set T' for the clause C' created by expanding C with L is the result of the join $T \bowtie \mathbf{L}$, where there is an equality-based join predicate between two columns if they correspond to the same variable in L and C. A tuple in T' is positive (negative) if it is an extension of a positive (negative) tuple in T. For a tuple t in T, t is *covered* by the expanded clause C' if t has an extension in T'; in this case, we simply say that the *literal L covers a binding tuple t* without referring to C' and T. A training example is covered by a clause if it appears as (part of) a binding tuple of the clause.

Generating candidate literals. The hypothesis space of a logic program is potentially infinite. To restrict the search space, a valid literal that can be added to a clause can be any background predicate that is constrained to having at least one existing variable in the current clause. If the current clause has n variables, then we can approximate the total number of candidate literals for a k-ary predicate as the number of assignments from $(n+k-1)$ variables (n variables and $k-1$ new variables) to the k inputs[2], which is $(n+k-1)^k$.

There are two ways to further reduce the size of the set of candidate literals. First, we can utilize *constraints* to remove those literals that are not allowed in a valid clause. The most popular constraint that is used in practice is type restriction on input arguments in the predicates. Second, we can prune the search space by eliminating redundant literals. As an example, a literal identical to an existing one in the current constructed clause can be excluded from consideration. We study such pruning strategies in more detail in Section 3.2.

Symbols	**Description**
$L, \mathbf{L}$	L denotes a candidate literal. $\mathbf{L}$ represents the relation corresponding to the predicate L, which is composed of ground facts for this predicate.
X, Y, Z	These represent argument variables for literals.
C, $C'(L)$, C'	C denotes the current clause that we have built so far, and want to specialize further. $C'(L)$ is a specialization of C that is generated by adding a new candidate literal L. When it is clear what L is from the context, or when the specific candidate literal is irrelevant, we use C' for short.
T	The binding set for the current clause C.
T'	The binding set for C'.
T_+, T_-	T_+ and T_- are the set of positive binding tuples and the set of negative tuples in T, respectively.
T'_+, T'_-	Similarly, T'_+ and T'_- represents the positive and negative binding tuples in T', respectively.
d, d_+, d_-	d is the number of tuples in T that have an extension in T'. Similarly, d_+ is the number of positive tuples in T that have an extension in T', and d_- is the number of negative tuples in T that have an extension in T'.
p, p'	p is the fraction of positive tuples (i.e. precision) in T. We use p' to denote the precision in T'.
$\|S\|$	$\|S\|$ is the cardinality of a set S.

Table 1: A summary of symbols used in this paper.

Choosing the best literal. Next, using the symbols summarized in Table 1, we introduce heuristics to choose a literal to specialize a clause.

Generally speaking, we want to find a clause that maximizes the coverage of positive training examples and minimizes the negative examples. A scoring function of a literal L measures the utility of adding L to the current clause C. It is based on the difference in clause quality before and after the literal L is added.

A common measure of the quality of a clause C includes the precision function $P(C) = |T_+|/|T|$ and the information function $I(C) = -\log_2 P(C)$. The corresponding scoring functions are the precision gain $f_p(L) = P(C') - P(C)$ and the information gain $f_i(L) = I(C') - I(C)$. Since a clause that has a high precision does not necessarily cover a large number of positive binding tuples, the well-known ILP algorithm FOIL [36] proposes the following gain function that takes into account the number of covered positive tuples:

$$f_g(L) = d_+ \cdot f_i(L) \qquad (1)$$

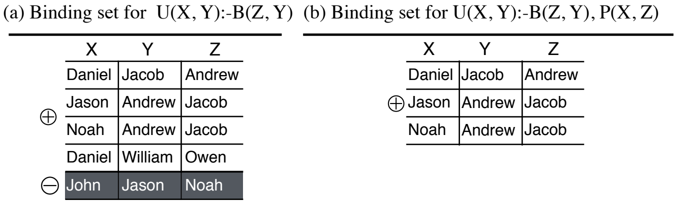
We can interpret the FOIL gain as the total amount of information that is reduced to encode the classification of all positive tuples in T due to L. An alternative weighted gain function [29] replaces d_+ with the relative frequency of positive tuples $\frac{|T'_+|}{|T_+|}$:

$$f_r(L) = \frac{|T'_+|}{|T_+|} \cdot f_i(L) \qquad (2)$$

Example 2 We show how the top-down, greedy algorithm can be applied to construct rules to model the uncle relationship. In this example, we assume that the scoring function is the FOIL gain function described above in Equation 1.

We start to learn the first rule with the most general clause $U(X, Y)$:-. The binding set for this clause consists of all

[2] The exact number is $n^k + \sum_{i=1}^{k-1} \sum_{j=i}^{k-1} \binom{k}{j} \left\{ {j \atop i} \right\} n^{k-j}$, where $\left\{ {j \atop i} \right\}$ is a Stirling number of the second kind.



(a) Binding set for U(X, Y):-B(Z, Y)

	X	Y	Z
	Daniel	Jacob	Andrew
⊕	Jason	Andrew	Jacob
	Noah	Andrew	Jacob
	Daniel	William	Owen
⊖	John	Jason	Noah

(b) Binding set for U(X, Y):-B(Z, Y), P(X, Z)

	X	Y	Z
	Daniel	Jacob	Andrew
⊕	Jason	Andrew	Jacob
	Noah	Andrew	Jacob

Figure 2: Learning Uncle using ILP.

the training examples. We first generate candidate literals for every background predicate. For example, for the predicate B, we enumerate every assignment from three variables (two existing variables X and Y and a new variable Z) to the inputs. This step results in the following eight candidate literals: $B(X,Y)$, $B(Y,X)$, $B(X,Z)$, $B(Z,X)$, $B(Y,Z)$, $B(Z,Y)$, $B(X,X)$, $B(Y,Y)$. Similarly, we have eight literals for the predicate P, and eight for the predicate S.

Next, we compute the gain for each candidate literal and choose the one with the largest gain. Consider the candidate literal $B(Z,Y)$. The binding set for the expanded clause $U(X,Y)$:- $B(Z,Y)$ is shown in Figure 2(a). This binding set is the result of a join between the current binding relation (i.e. the training examples) and the relation **B** of the new literal on an equality predicate between their second columns. The set contains four positive tuples and one negative tuple, of which each positive one is extended from an original positive binding tuple in T_+ and the negative one is from T_-. Therefore, we get $|T'_+| = 4$, $|T'| = 5$, $d_+ = 4$. Thus the gain is: $4 \times \left(\log_2(\frac{4}{5}) - \log_2(\frac{4}{8})\right) = 2.7$.

Suppose $B(Z,Y)$ is selected as the addition literal, resulting in a more specific rule $U(X,Y)$:-$B(Z,Y)$. Since the binding set has one negative tuple, we continue to find a more specialized clause. The current binding set is then replaced by the set shown in Figure 2(a). Note that the arity of the binding tuples increases as a new variable is introduced.

A new search process is started to further specialize the new extended clause. With three variables in the current clause, we next generate 15 candidate literals for the predicate P, 15 for the predicate S, and 14 for the predicate B (with the identical one removed). As another example of the gain calculation, adding $P(X,Z)$ produces a binding set consisting of three positive tuples (Figure 2(b)). The gain of the literal $P(X,Z)$ is thus $3 \times (-log_2(4/5)) = 0.97$.

Suppose $P(X,Z)$ has the largest gain. Then, the new specialized rule is $U(X,Y)$:- $B(Z,Y), P(X,Z)$. Since the new binding set has no negative tuples, we can terminate the search for further specialization, and add this clause into the hypothesis. Next, we remove the covered positive training examples, leaving the last positive example as "uncovered." Finally, we begin a new search for an additional rule with the initial binding set consisting of the uncovered positive training example and the entire set of negative examples. □

3 QuickFOIL

The basic algorithm employed by QuickFOIL is the top-down, greedy ILP search shown in Algorithm 1. In this section, we describe the QuickFOIL scoring function and QuickFOIL's literal pruning strategy.

3.1 Scoring function

The score of a literal is computed based on the cardinality differences between the binding sets before and after the literal is added. The two weighted functions f_g and f_r (cf. Equations 1 and 2) are sensitive to the *join skew* in computing a binding set: The number of matching tuples for a positive binding tuple differs significantly from the number for a negative tuple. More precisely, we define the join selectivity as the ratio of the join output cardinality to the cardinality of the input binding set. The skew for a candidate literal is the ratio of the join selectivity for the positive binding set to the join selectivity for the negative binding set: $(|T'_+| \cdot d_-)/(|T'_-| \cdot d_+)$. The skew can be caused by the non-uniform data distribution in the background relation, which is not uncommon in real datasets. The information gain $f_i(L)$ becomes misleading in the presence of high skew, because a candidate literal can receive a high precision value $|T'_+|/|T'|$ but exclude few negative tuples (i.e. having a large value of d_-), or vice versa. In the two functions f_g and f_r, neither d_+ nor the relative frequency $|T'_+|/|T_+|$ measures the changes in the negative coverage d_-, and hence these functions fail to balance the (biased) weight attributed to the term f_i. As an example, assume that the predicate $Sister$ in Example 2 is skewed as shown in Figure 1(b), where Daniel has many more sisters than other people. Consider the first literal search iteration to specialize the clause $U(X,Y)$:-. The candidate literal $S(X,Z)$ does not eliminate any negative binding tuples, but still has the greatest FOIL gain: $f_g = 4 \times \left(\log_2(\frac{10}{12}) - \log_2(\frac{4}{8})\right) = 2.9$. In addition to reducing the quality of the rule, choosing a "bad" literal (i.e. a skewed predicate) can considerably slow down performance, as (a) it prolongs the time that it takes to reach the stopping criterion; and (b) the corresponding join skew generally increases the size of the result binding set (join output), making the subsequent literal search computationally expensive.

To overcome this skew problem, we develop a new scoring function. We view the process of choosing literals for a clause as performing a sequence of binary classification tasks. A clause expanded with a candidate literal can be considered as a binary classifier that classifies the tuples in the current binding set into two groups on the basis of whether they are positive or negative. Specifically, the data in the classification problem is the binding set. A tuple in the set is predicted to be positive if it has an extension in the new binding set; otherwise, it is considered to be negative. Accordingly, we can derive the following confusion matrix:

		True labels	
		P	N
Clause coverage	P	$TP = d_+$	$FP = d_-$
	N	$FN = \lvert T_+ \rvert - d_+$	$TN = \lvert T_- \rvert - d_-$

Our greedy search chooses the literal that gives rise to the "best" classifier, using as measure the quality of a binary classifier to evaluate each candidate literal. The measure that we use is the Matthews Correlation Coefficient (MCC) [32]. The MCC is calculated as:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (3)$$

This measure takes into account all the four terms TP, TN, FP and TF in the confusion matrix, and is able to fairly assess the quality of classification even when the ratio of positive tuples to the negative tuples is not close to 1. The MCC values range from -1 to $+1$. A coefficient of $+1$ represents a perfect classification, 0 represents one no better than a random classifier, and -1 indicates total disagreement between the predicted and the actual labels. Note that the MCC measure considers both d_+ and d_-.

There is one subtle yet important distinction between scoring for classification and scoring for ILP. The ILP problem is to build clauses that cover as many positive training examples as possible while preferably excluding negative ones, rather than maximally discriminating the positive examples from the negative examples. We use a new measure AUE (Area Under the Entropy) to evaluate the effect of positive coverage. Given $0 \leq p \leq 1$, the area under the entropy curve between 0 and p without scaling is:

$$\begin{aligned}&\int_0^p -p\log_2(p) - (1-p)\log_2(1-p)\mathrm{d}p \\ =&\frac{1}{2}\Big((p-1)^2\log_2(1-p) - p^2\log_2(p) + \frac{p}{\ln 2}\Big)\end{aligned} \tag{4}$$

AUE(p) is Equation (4) divided by $1/(2\ln 2)$, the total area under a entropy curve, to scale the value. As the derivative of Equation (4) is the entropy function, AUE(p) has a small slope when p is close to 0 or 1, and a relatively larger slope of up to 1 when p is around the middle point.

Our new scoring function is a F_β-measure of the MCC score and the AUE gain. This scoring function is:

$$f(L) = \frac{1+\beta^2}{\beta^2 \cdot \frac{1}{MCC+1} + \frac{1}{\mathrm{AUE}(p')-\mathrm{AUE}(p)+1}} \tag{5}$$

where $p' = |T'_+|/|T'|$, $p = |T_+|/|T|$ and β is a weight parameter that defines the relative importance of MCC and AUE. Incrementing both MCC and the difference of the AUE values by 1 guarantees that both the denominator terms are non-negative. As MCC is a more balanced measure than AUE, we empirically find $\beta = 2$ works well for datasets with skew, and use it as the default value.

3.2 Pruning candidate literals

The method that is used to generate candidate literals is crucial to the overall performance of ILP algorithms. On one hand, since the coverage tests for candidate literals dominate the running time, we do not want to explore the entire space of candidate literals, but instead employ some pruning strategy. On the other hand, excluding "good" literals may exclude certain crucial rules (lowering the quality of the ILP output), and/or take a lot longer to find a rule that eliminates enough negative examples.

A common pruning method is to remove those literals that are considered duplicates when building a clause. In this work, we only consider syntactic duplicates, as they are the most common type of duplicates and can be applied to any learning task. Exploiting background knowledge is an interesting direction for future work.

Besides the trivial identical duplicates, prior work [10,18] has used another type of syntactic duplicates, which we call *renaming duplicates*. Formally, a literal L_i is a renaming duplicate if there is another literal L_j in the clause such that (i) there is a renaming of the free variables in L_i such that L_i and L_j are identical after renaming, and (ii) all renamed variables only occur in L_i or L_j. For example, assume the current partially constructed clause $L_h(X)$:- $L_1(X,Y)$. The literal $L_1(X,Z)$ is duplicate to the literal $L_1(X,Y)$ under the renaming $\rho_{Z\rightarrow Y}$, where Z and Y do not occur in other literals. However, consider another clause $L_h(X)$:- $L_1(X,Y), L_2(Y), L_1(X,Z)$. Since Y appears in L_1, $L_2(Z)$ is not a renaming duplicate to $L_2(Y)$, but it is reasonable to leave it out of consideration to avoid increasing the cardinality of the training set until we have included other predicates on either Y or Z.

To overcome the problem with the renaming duplicates, we use the the notion of *equivalence* of two clauses: Two clauses C_1 and C_2 are equivalent if and only if $C_1 \models C_2$ and $C_2 \models C_1$. QuickFOIL prunes redundant literals using a new type of duplicate, called the *replaceable duplicates*.

Definition 2 (Replaceable duplicate) *A literal L is a replaceable duplicate with respect to a clause C if there is a literal L' in the body of the clause such that the new clause that replaces L' in C with L is equivalent to C.*

A renaming duplicate is clearly also a replaceable duplicate. Intuitively, a replaceable duplicate can be *replaced* by an existing literal in a clause. For example, the literal $L_1(X,Z)$ is a replaceable duplicate w.r.t. the clause $L_h(X)$:- $L_1(X,Y)$, because replacing $L_1(X,Y)$ with $L_1(X,Z)$ produces an equivalent clause $L_h(X)$:- $L_1(X,Z)$. Similarly, one can verify that $L_2(Z)$ is a replaceable duplicate w.r.t. the clause $L_h(X)$:- $L_1(X,Y), L_2(Y), L_1(X,Z)$, and $L_1(X,Z)$ is not w.r.t. the clause $L_h(X)$:- $L_1(X,Y), L_2(Y)$.

By Definition 2, determining the replaceable redundancy of a literal is to test the equivalence of two clauses, which can be reduced to the equivalence problem of conjunctive database queries under set semantics. Specifically, for a clause C, we can construct a Datalog program Q that contains C as the single Datalog rule. Two clauses C_1 and C_2 are equivalent if and only if their corresponding Datalog queries Q_1 and Q_2 are equivalent. QuickFOIL tests the query equivalence by checking their mutual containment, which is done by evaluating one query on the *canonical database* of the other query (using the algorithm proposed in [38] for the conjunctive query containment problem).

Besides reducing the search space, pruning candidate literals has two additional advantages. First, most of the redundant literals do not eliminate any negative binding tuples, but they can still be amongst the literals with high scores due to the high coverage of positive literals. It is important to exclude them from the body of the clause being constructed since they can increase the cardinality of the positive binding set significantly. Second, the pruning helps keep the rule concise, thereby improving the ability of an end-user to interpret the output of the ILP method.

4 QuickFOIL implementation

In this section, we describe our methods to run and optimize QuickFOIL as an in-RDBMS algorithm.

4.1 Relational operations

From the logic programming point of view, a database relation can be seen as a specification of a predicate. The background knowledge and the training examples represented in logic programs in ILP can be naturally stored in a database. Specifically, we create a relation per predicate, where 1) the relation name is the predicate symbol, 2) there is an attribute per predicate argument that has the same type as the predicate's argument, and 3) each tuple in the relation corresponds to a (predicate) fact. For example, the SQL schema of the relation for the predicate *Brother* in our running example (cf. Figure 1 and Example 2) is:

```
CREATE TABLE Brother (person VARCHAR(50),
                      parent VARCHAR(50))
```

With the relational representation of data, the QuickFOIL algorithm can be expressed in (extended) relational algebra using the mapping shown in Table 2. The line numbers in this table refer to Algorithm 1. Note, $\ltimes$ is the left semi-join operation, $\bowtie$ is the natural join operation, $\rhd$ represents the antijoin operation.

In a literal search iteration, to compute the score of a candidate literal L (Equation 5), the following aggregate values are needed: d_+, d_-, $|T_+|$, $|T_-|$, $|T'_+|$ and $|T'_-|$, of which $|T_+|$ and $|T_-|$ are common across different literals, and can be found in the previous iteration. Recall that the binding set T' of the clause resulting from adding L can be computed by $T \bowtie \mathbf{L}$, where there is an equality-based join predicate between two columns if they correspond to the same variable in C and L. This expansion leads to four queries q_1^+, q_1^-, q_2^+ and q_2^- for every candidate literal to compute d_+, d_-, $|T'_+|$ and $|T'_-|$, respectively. After finding a complete rule, the positive training examples can be updated by performing an antijoin operation, $U \rhd T$, to remove the examples that are covered by the new rule, where U represents a relation composed of the uncovered positive training examples as shown in Algorithm 1.

Compute the scoring components (Line 8)	For every candidate literal L, $q_1^+ : d_+ = \text{COUNT}(T_+ \ltimes \mathbf{L})$, $q_1^- : d_- = \text{COUNT}(T_- \ltimes \mathbf{L})$, $q_2^+ : \lvert T'_+ \rvert = \text{COUNT}(T_+ \bowtie \mathbf{L})$, $q_2^- : \lvert T'_- \rvert = \text{COUNT}(T_- \bowtie \mathbf{L})$
Create a new binding set (Line 11, 12)	For the best literal L, $q_3^+ : T_+ = T_+ \bowtie \mathbf{L}$, $q_3^- : T_- = T_- \bowtie \mathbf{L}$
Update the uncovered positive training examples (Line 14)	$q_4 : U = U \rhd T$

Table 2: Relational operations in QuickFOIL

4.2 In-database implementation

This section describes our in-database implementation of QuickFOIL, which leverages database query processing techniques to achieve high performance and scalability.

As discussed in Section 2.2, the total number of candidate literals to be explored (in each round of constructing a clause) can quickly increase to a significant number, thus leading to a large number of join queries when computing the scores for every candidate literal. In addition, we observe that in real-life problems, the binding relation cardinality can increase considerably. A key challenge that is associated with building a scalable in-RDBMS version of QuickFOIL is to optimize the performance of a large number of join queries on potentially large relations when searching for the best literals. Our approach to this problem is to maximize the sharing amongst the queries while minimizing the materialization cost that is incurred by sharing.

Our implementation is built in the main-memory RDBMS, Quickstep [9]. We assume partitioned hash join algorithms in developing the performance optimization techniques. QuickFOIL uses the radix hash join algorithm [31].

4.2.1 *Combining positive and negative tuples*

Since the binding tuples are labeled as positive or negative, there are two natural relational models for the binding set: an *integrated model* and an *independent model.* The integrated model represents the entire set of binding tuples as a single relation, whereas the independent model separates them into two relations, one for the positive tuples and the other for the negative ones.

The conventional choice is to use the independent model. It might appear that compared to the independent model, using the integrated model has no benefits, but it introduces the overhead of performing extra selection operations to pick the positive/negative tuples. However, we find that the integrated model can be exploited to improve the join performance. By putting the positive and the negative tuples in a single relation, we observe that for each literal, the four join queries q_1^+, q_2^+, q_1^-, and q_2^- (see Table 2) have a common join operation between the two relations T and $\mathbf{L}$. This common join operation can be "shared" if we push the join operation *below* the selection operation. In Section 4.2.4, we introduce a continuous query execution model that enables "operator sharing", and allows a single query operator to produce multiple query outputs. In contrast to the independent model where we need to perform two joins $T_+ \bowtie \mathbf{L}$ and $T_- \bowtie \mathbf{L}$ separately, using the integrated model only requires one join $T \bowtie \mathbf{L}$, followed by selection operations.

QuickFOIL implements a join operation using a join operator on a single join predicate, where the binding relation is picked as the inner relation. A selection operator is used to evaluate any remaining join predicates (more details regarding this issue are described below in Section 4.2.2).

For our implementation we choose the integrated model that reduces the join operations for each literal from two to one, as it has two key advantages. First, eliminating a join operation reduces one pass/scan over the common relation $\mathbf{L}$. Second, merging the positive and the negative binding tuples yields good cache locality for the subsequent selection operation. Consider a tuple t in $\mathbf{L}$ that has multiple matching binding tuples in the single-predicate join. After an attribute value of t is read from memory for a matching binding tuple, it likely resides in the CPU cache when it is accessed again for other matching tuples, since these matching tuples are processed in succession. Hence, we can improve the sharing for tuples with both positive and negative matchings.

4.2.2 *Caching partitions and hash tables*

We have the following two observations about the join workload generated by QuickFOIL.

Observation 1 For every literal search iteration, all joins in the four queries q_1 to q_4 have one common relation, namely the binding relation.

Observation 2 For the entire execution, the background relations do not change, and are repeatedly used a large number of times.

Thus, there is an opportunity to share intermediate results on common relations across the queries. Now, recall that a partitioned hash join algorithm has three phases: partition, build and probe. The *partition phase* divides both relations into partitions. The *build phase* builds a hash table on each partition of the inner relation, which is then probed in the *probe phase.* The partition and the build phases often consumes a combined of 50%–90% of the (radix hash) join execution time, when ignoring the cost to materialize the join results [3]. Since the partition phase and the build phase are performed on individual relations, we can reduce the partition and build costs by keeping a cache of the partitioned results and the hash tables for each relation once they are built. In particular, the partitions for a background relation are shared throughout the entire execution, as that relation never changes. In order to maximize the benefit

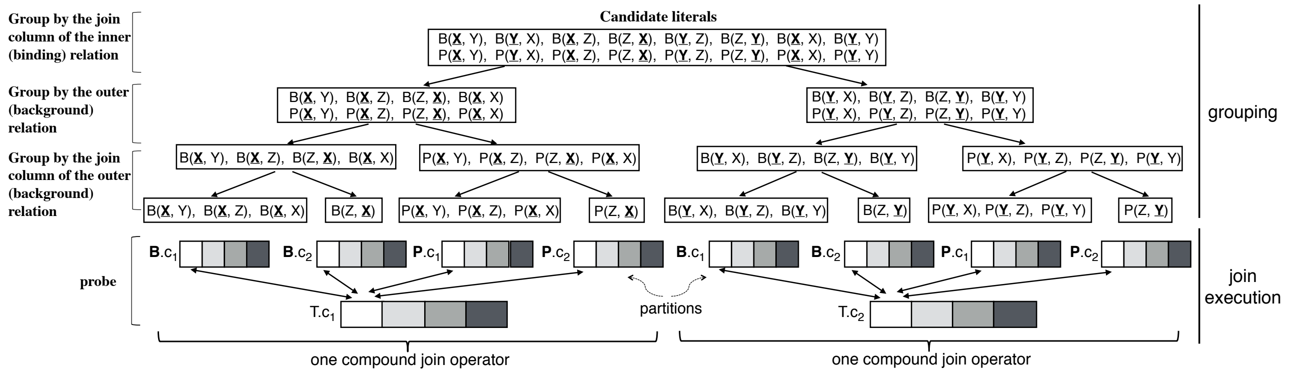

Group by the join column of the inner (binding) relation
Group by the outer (background) relation
Group by the join column of the outer (background) relation
probe
Candidate literals
B(X, Y), B(Y, X), B(X, Z), B(Z, X), B(Y, Z), B(Z, Y), B(X, X), B(Y, Y)
P(X, Y), P(Y, X), P(X, Z), P(Z, X), P(Y, Z), P(Z, Y), P(X, X), P(Y, Y)
B(X, Y), B(X, Z), B(Z, X), B(X, X)
P(X, Y), P(X, Z), P(Z, X), P(X, X)
B(Y, X), B(Y, Z), B(Z, Y), B(Y, Y)
P(Y, X), P(Y, Z), P(Z, Y), P(Y, Y)
B(X, Y), B(X, Z), B(Z, X), B(X, X)
P(X, Y), P(X, Z), P(Z, X), P(X, X)
B(Y, X), B(Y, Z), B(Z, Y), B(Y, Y)
P(Y, X), P(Y, Z), P(Z, Y), P(Y, Y)
B(X, Y), B(X, Z), B(X, X)
B(Z, X)
P(X, Y), P(X, Z), P(X, X)
P(Z, X)
B(Y, X), B(Y, Z), B(Y, Y)
B(Z, Y)
P(Y, X), P(Y, Z), P(Y, Y)
P(Z, Y)
B.c1
B.c2
P.c1
P.c2
T.c1
T.c2
partitions
one compound join operator
one compound join operator
grouping
join execution

Figure 3: Illustration of the grouping process when finding a literal to specialize the clause $U(X,Y)$:-. The i-th column of relation R is indicated by $R.c_i$. The join columns are underlined and in boldface. For the purpose of illustration, we simply choose c_1 as the join column when there are join predicates on both c_1 and c_2.

of caching, next, we study two issues: 1) implementing the multi-predicate join algorithm, and 2) determining the inner relation (i.e. the build relation).

The join operator in QuickFOIL only evaluates a single join predicate. For a multi-predicate join (i.e. the join is on more than one column in the two join relations), we evaluate one join predicate in the join operator, and the remaining predicates are evaluated in a subsequent selection operator.

For example, consider the candidate literal $B(X,Y)$ in the first iteration of the literal search process, where the current clause is $U(X,Y)$:-. We need to perform a join between $\mathbf{B}$ and T with a conjunction of two join predicates, i.e. $\mathbf{B}.c1 = T.c1$ and $\mathbf{B}.c2 = T.c2$, where we denote the i-th column as ci. The join operator evaluates one of these conjuncts, and the other conjunct is evaluated in the selection operator.

For a k-ary predicate and a binding relation with n-columns, the $O(n^k)$ join queries share $O(nk)$ single-predicate joins in a literal search, suggesting potential for work sharing. The simple multi-predicate join implementation avoids building hash tables for every combination of columns for the binding relation. It also reduces the number of partitions that are built for a relation from being potentially exponential to being linear with the number of columns.

Finally we note that QuickFOIL currently uses a simple heuristic to choose the join predicate that is evaluated in the join operator. It picks the join predicate on the column of the background relation that has the maximum estimated number of distinct values. This strategy aims to pick the join column that helps differentiate the tuples the most, and thus minimizes the number of calls that are made to the subsequent selection operators. As part of future work, we plan on exploring other alternatives for this design choice.

A canonical query optimizer selects the inner relation according to a cost function, and generally picks the smaller relation as the inner/build relation. While adaptively selecting the inner relation is reasonable for individual queries, it is not necessarily a good design for multiple query optimization when caching is enabled.

QuickFOIL restricts the inner relation to be the binding relation. The benefits of this approach are two-fold. First, it can considerably reduce the memory footprint, as only the hash tables for the binding relation is cached. The maximum number of hash tables that need to be built and cached is equal to the number of columns of the binding relation. In contrast, using an adaptive approach can require building a hash table for every column for every background relation, because the approach can choose either of the two join relations to build the hash table, and the join predicates can be on any pair of columns. Second, it also maximizes the potential opportunities for sharing amongst the queries. As long as the join key is identical across the queries, the queries access the same hash table. By carefully grouping these join queries in a way such that the hash table is probed sequentially, we can increase the data locality, and hence reduce the cost of memory accesses. This grouping strategy is described further in Section 4.2.3.

Because partitioning breaks the tuple ordering, a performance issue is that the selection operation followed by the single-predicate join could incur a large number of random accesses spanning the entire table. To improve the cache performance, we can perform the partitioning on a background relation *holistically.* That is, not only is the join column partitioned, but the columns on which there are predicates for the selection are also further divided into partitions based on the join key. After holistic partitioning, the selection operators that follow the join operators then access the background relation on the partitions, rather than on the original columns.

4.2.3 Grouping join operations

For a candidate literal L, the four queries q_1^+, q_1^-, q_2^+ and q_2^- share the same join kernel $T \bowtie \mathbf{L}$. For candidate literals $L_1, \ldots, L_n$, the join workload is $\mathbb{W} = \{T \bowtie \mathbf{L_1}, \ldots, T \bowtie \mathbf{L_n}\}$; i.e. it consists of binary join operations having a common inner relation but with different join conditions. This section studies how to optimize the performance of this "workload." Our basic idea is to divide the join operations (each corresponding to a literal) into groups, and organize the sequence of single-predicate join executions in a way that increases the shared working set (the common relation).

Next, we use the first search iteration of Example 2 to illustrate the grouping technique. Recall that the task in that example is to learn the Uncle predicate (abbreviated as U). Here we have 24 candidate literals to specialize the clause $U(X,Y)$:-. We have the following eight candidate literals: $B(X,Y)$, $B(Y,X)$, $B(X,Z)$, $B(Z,X)$, $B(Y,Z)$, $B(Z,Y)$, $B(X,X)$, and $B(Y,Y)$, where B denotes the *Brother* predicate. Similarly, we have eight literals for the predicate *Parent*, and eight for the predicate *Sister*. For simplicity in illustration, we ignore the *Sister* predicate below. The grouping for this iteration is shown in Figure 3. We use literals to represent the associated joins for ease of presentation. The join columns are underlined and in boldface. For example, a literal $B(\underline{\mathbf{X}}, Z)$ represents the join operation $\mathbf{B} \bowtie_{B.c_1=T.c_1} T$, where c_i refers to the i-th column.

The grouping technique first partitions the join operations by the join column on the inner binding relation T. Consequently, every outer relation $\mathbf{L_i}$ in a group is joined with the inner relation on the same column. As shown in Figure 3, the 16 candidate literals are divided into two groups according to the common variables that they have with the current clause, which correspond to the join columns on the binding relation. Next, the join operations in each group are further distributed into smaller groups based on the outer relation, and then on its join column. As a result, the join operations in each final group have the same join predicate. For example, the literals $B(X, Y), B(X, Z)$ and $B(X, X)$ are in the same group, because they involve a common single-predicate join $\mathbf{B} \bowtie_{\mathbf{B}.c1=T.c1} T$. During the join execution, for each inner join column, we use a *compound join* operator to perform the probe phases of all the associated single-predicate join operations collectively as follows:

```
for i=1 to n  /* n is the number of partitions */
do
    for each outer join column c, ordered by the
    relation to which c belongs do
        scan the i-th partition of c, probe the hash table
        on the i-th partition of the inner join column
```

In our example, the single-predicate join $\mathbf{B} \bowtie_{\mathbf{B}.c1=T.c1} T$ is shared by three join operations and is performed only once by a compound join operator that simultaneously executes three other joins with the same inner join column. Since the initial binding relations have two columns, we can see that the total number of compound join operators is two.

4.2.4 Merging queries with shared operators

A key question that stills needs to be answered is how to connect the compound join operator with the remaining operators. Queries for different literals in a group share the same single-predicate join, but they have distinct join conditions evaluated by multiple selection operators. Thus, we need to find a way to feed the same join output to multiple operators. In addition, for every literal, the four queries (q_1^+, q_1^-, q_1^+ and q_1^-) return four different aggregate values, but have the same selection operation. The last problem that we need to tackle is how to share a common selection operation, in addition to sharing the join operation.

In a standard database query processing model [23], each operator reads one tuple or a batch of tuples by calling a next() function in an iterator interface. A crucial drawback of these standard iterators is that they are *one-directional*. Once an operator consumes a tuple, the iterator cannot roll back, and thus other operators cannot read it again.

To address this limitation, we develop a new inter-operator communication model, which we call the *continuous query model*. The basic idea behind the continuous model is that the output tuples of an operator (producer) is pushed, instead of being pulled, to its consumers (operators that takes the tuples as input). In the continuous model, each operator is a subclass of a *notifier* interface that has two basic functions: register and update. The function register allows a consumer to be registered in the producer as an output consumer. When output tuples are available, the producer notifies each consumer by calling the function update, with the output tuples as arguments. Clearly, a producer can have more than one changeconsumer. Therefore, we can merge multiple queries with shared operators into one graph-structured query execution plan.

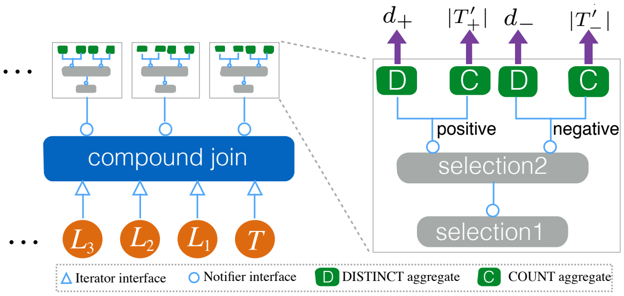


Figure 4: Query execution plan. The DISTINCT aggregation operators count the number of distinct tuples, and the COUNT aggregation operators count all the input tuples.

It is important to note that an operator can open multiple registration points. Not only can the final output of an operator be shared with other operators, but any intermediate results can also be shared using the continuous model. For example, a join operator can share four types of outputs: inner join result, semi-join result, (left/right) outer join result and (left/right) anti-join result.

The continuous query model can be mixed along with the traditional iterator model. The only restriction is that the query plan should not have a path that uses the iterator interface after the notifier interface (to prevent breaking the data flow). Note that if no operator is registered in more than one operators, no extra materialization cost is incurred; otherwise, materialization may be needed since the inputs from multiple producers are not synchronized. This materialization can be completely avoided in QuickFOIL.

Figure 4 shows the unified execution plan to compute the aggregates for all the candidate literals that have a common variable in the current clause. For each candidate literal, the lower selection operator (selection1) evaluates the join predicates that have not been evaluated in the single-predicate join, and the upper selection operator (selection2) determines the positive/negative label of the input binding tuples. All the single-predicate join operations are executed by a single compound join operator. As this join operator is shared, we use the notifier interface between the compound join operator and the selection1 operator. Since the selection1 operator subscribes to the compound join operator's notifier interface, it is connected to the selection2 using the notifier interface, even though the selection1 operator is not shared. The selection2 operator has two registration points for the case when the selection predicate is true and the case when it is false. An aggregate operator is registered in one of the two points based on whether it counts the positive or the negative tuples. Next, for each literal, four aggregate values (i.e. d_+, $|T'_+|$, d_- and $|T'_-|$) are returned, based on which the final score is then computed using Equation 5.

5 Experiments

In this section, we evaluate the QuickFOIL approach.

5.1 Experiment setup

Datasets. For our experiments, we use two real-life datasets (WebKB and HIV), and synthetic datasets (Bongard), which are summarized in Table 3.

Name	#P	#N	#T	#Pred.
WebKB-Student	418	2.7K	278K	935
WebKB-Department	561	225K	280K	942
HIV-Large	5.3K	33K	10M	80
HIV-Small	45	90	39K	33
Bongard-TH1	270K	270K	53M	9
Bongard-TH2	18K	18K	36M	9
Bongard-TH3	900K	900K	178M	9

Table 3: The number of positive (#P) and negative (#N) examples in the training set, and the total number of tuples (#T) and the number of predicates (#Pred.) in the background knowledge for each type of training datasets.

WebKB [11] is a standard dataset for link classification and discovery, and is available at `http://www.biostat.wisc.edu/~craven/webkb/`. It contains web pages of faculty members, research projects and students, as well as hyperlinks between them, crawled from web sites of four computer science departments. The background knowledge was constructed from the link structure, word occurrences, and positions on the web pages. We performed two learning tasks on this dataset. The first task, *WebKB-Student*, constructs rules to identifying student pages. The second task, *WebKB-Department*, determines the "department-of" relationship between a pair of pages. We used *leave-one-university-out* cross validation to evaluate the learning model. Specifically, we partition all tuples into four partitions according to which university they are from, and perform four rounds of validation. Each round leaves one partition out for testing and keeps the remaining for training.

HIV-Large is a collection of over 42,000 compounds for the National Cancer Institute's AIDS antiviral screen.[3] This dataset contains 62 background predicates for atom elements, 1 for the compound-atom relationship, 1 for the atom-bond relationship and 15 for the properties of atoms and bonds. We also created a small sample of data, **HIV-Small**. We assign positive labels to compounds that have a substructure C-C-O-C=O (in the SMILES format), which is a frequent substructure in compounds capable of inhibiting the HIV virus [28]. We used 10-fold cross validation for this task.

Bongard datasets consist of labeled pictures, each having a few simple geometrical objects (circle, rectangle, triangle). The learning problem is to find theories that distinguish the positive pictures from the negative ones given the shapes and the positions relative to each other. We have implemented a program[4] that randomly generated a corpus of 100 million pictures, each having 8–10 objects. The three target theories shown in Table 4 were used to label the pictures, resulting in three sets of positive pictures which bound the number of positive training examples shown in Table 3. TH1 is a very simple concept that has only three objects and two correlated relations. TH2 and TH3 represent two different hard problems. TH2 contains a large number of objects but the pairwise relations are independent, whereas the relations in TH3 are all correlated. We created four different datasets for each of the three problems and report the averaged execution time in this paper. Each dataset has an equal number of positive tuples and negative tuples, with ten percent for testing. The training set has five percent of positive tuples and the same percent of negative tuples that are assigned wrong labels to introduce noise.

Name	Definition
TH1	positive(P) :- inside(C, T), east(T, R)
TH2	positive(P) :- inside(C1, TD), east(T1, R1), east(T2, T3), north(C2, C3), inside(TU, R2)
TH3	positive(P) :- inside(O1, O2), east(O2, O3), east(O3, O4), north(O4, O5), inside(O5, O6), north(O1, O5)

Notations: P: picture; C: circle; T: triangle; R: rectangle; TU/TD: point-up/point-down triangle; O: any object.

Table 4: Labeling theories for the Bongard datasets. To avoid literal overloading, predicates other than *east*, *north* and *inside* are omitted and the variable name is used to indicate omitted predicates on the variable.

Comparisons. We compare QuickFOIL with an in-memory implementation in Quickstep with the following two systems, which are also in-memory systems.

1) **FOIL** is the FOIL program written by J. R. Quinlan. We use the latest version, namely FOIL 6.4[5].

2) **Aleph** [42] is a popular ILP system that has been widely used in prior work. To find a rule, Aleph starts by building the most specific clause, which is called the "bottom clause", that entails a seed example. Then, it uses a branch-and-bound algorithm to perform a general-to-specific heuristic search for a subset of literals from the bottom clause to form a more general rule. We set Aleph to use the heuristic enumeration strategy, and the maximum number of branch nodes to be explored in a branch-and-bound search to 500K.

Metric. We use the standard metrics precision, recall and F-1 score to measure the quality of the results.

All experiments were run in a single thread on a 2.67GHz Intel Xeon X5650 processor. The server had 24GB of main memory and was running Scientific Linux 5.3.

5.2 Results with real-life datasets

5.2.1 WebKB learning tasks

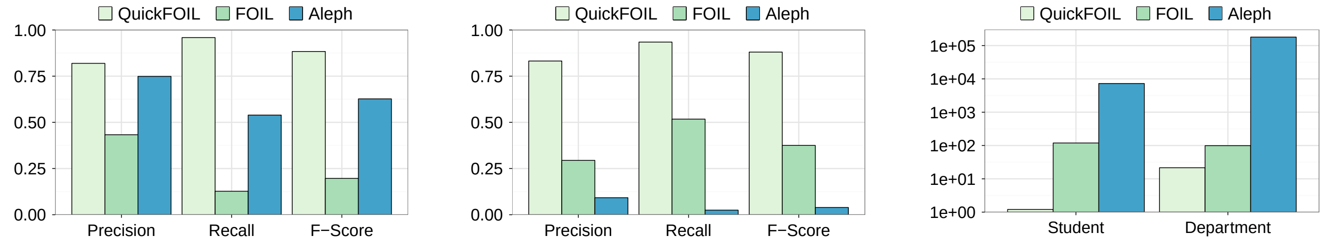
There are two important features in the WebKB datasets. First, it is well known that the link structure in the web follows a power-law distribution [4], which is highly skewed. For the WebkKB datasets, the largest in-degree of a page is 179, while 60% of pages have only one in-link. As discussed in Section 3.1, such skew raises performance issues. Second, although the WebKB datasets has nearly 1000 predicates, the hypothesis search space is not very large because only one of these predicates has an arity greater than one. However, the large number of predicates makes it harder to produce high-quality rules because they add uncertainty into the learning process, and increase the probability and penalty of making a mistake.

Figures 5(a) and 5(b) compares the quality of the results produced by QuickFOIL, FOIL and Aleph. QuickFOIL achieves the best F-1 scores on both tasks. For the WebKB-Student task, FOIL has the worst performance with a low recall of 12.7%. For the WebKB-Department task, Aleph failed to finish on two of the four-fold cross validation rounds as it exceeded the amount of available main memory. The quality measures shown in Figure 5 only includes the

[3] `https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data`

[4] Available at `http://www.cs.wisc.edu/~qzeng/quickfoil/bongard-generator.html`

[5] Available at `http://www.rulequest.com/Personal/`



(a) on WebKB-Student

(b) on WebKB-Department

(c) Execution time (s)

Figure 5: Results with the WebKB datasets

two successful runs. QuickFOIL also produces the smallest number of rules. For example, QuickFOIL outputs one rule in three of the four validation rounds for the WebKB-Student task, while Aleph and FOIL produce more than 10 output rules. Next, we describe the reasons why Aleph and FOIL produce (relatively) lower quality rules.

FOIL is not good at incorporating predicates with an arity greater than one, which restricts its ability to mine richly structured data. The FOIL gain function is biased towards literals that introduce new variables and cover a large number of positive tuples, yet not necessarily reducing the coverage of the negatives tuples. Most of these literals dramatically increase the size of the binding set. To reduce the impact of the problem, FOIL drops these previously added literals that contain free variables when it finds a new literal with exclusively bound variables, which in most cases is an unary literal. On one hand, the FOIL scoring function prevents it from finding good literals due to this bias. On the other hand, FOIL cannot fully utilize the relationship presented in the background knowledge as it tends to drop literals of high arity and to construct clauses with mostly unary literals. Specifically, for the WebKB datasets with hundreds of unary predicates, FOIL typically first adds two *link-to* literals and then replaces them with an unary literal. Notice that the unary literal is a greedy selection for the current clause containing the two *link-to* literals, but is kept as the new addition literal to the clause after the drop action. Therefore, it is unable to find a latent link between the student/department page with other pages. Compared to FOIL, QuickFOIL does not use the growing strategy and the scoring function is able to alleviate the problem that is associated with the FOIL gain as it takes into account the coverage of the negative tuples.

Aleph uses a classic branch-and-bound algorithm to guide its search. It discards a constructed clause only when the upper bound on the scores of the clause and all the expanded clauses from it is smaller than the score of another constructed clause. The search is terminated when there is only one available clause left, or when the limit on the maximum number of constructed clauses is reached. As it is usually infeasible to have a tight upper bound, this pruning strategy is not effective. In fact, Aleph usually ends up evaluating the maximum number of clauses, creating a big memory footprint that is required to maintain the large set of constructed clauses.

HIV-Small				HIV-Large	
FOIL		QuickFOIL		QuickFOIL	
F-Score	Time (s)	F-Score	Time (s)	F-Score	Time (s)
0.36	261	0.81	4.2	0.84	783

Table 5: F-Score and the execution time on the HIV tasks.

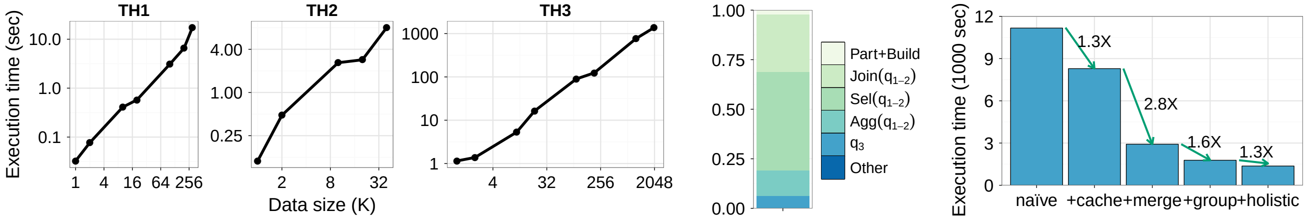
Aleph suffers from this problem starkly on the WebKB-Department task. Due to rich structure in the link graph of the dataset, the length of the bottom clause to cover an example of a binary Department predicate is 10 times the length for the unary Student predicate. The search space for the WebKB-Department task is thus far larger than the space for the WebkKB-Student task. Aleph is unable to find a good clause even after evaluating the maximum 500K clauses, thus resulting in relatively worse performance on the WebKB-Department task than the WebKB-Student task. In contrast to the search in Aleph, the greedy algorithm used by QuickFOIL abandons previously constructed clauses and keeps only one clause for future expansion. Aleph explores orders of magnitude more clauses than QuickFOIL in each single-clause search. For the WebKB task, QuickFOIL explored on average 28K literals, whereas Aleph constructed more than 10M clauses.

From Figure 5(c), we observe that QuickFOIL is 100X faster than FOIL, and more than 6000X faster than Aleph on the WebKB-Student task. The key reason behind this behavior is that QuickFOIL needs to evaluate a smaller number of literals, and constructs fewer rules than FOIL and Aleph. As an example, FOIL searches 17 times as many literals as QuickFOIL on the WebKB-Student task. Database query processing techniques also help improve the performance, but they are not significant factors in this experiment, as most background relations in the WebKB datasets only have dozens of tuples.

5.2.2 HIV learning tasks

The HIV task is to find a structure pattern in the positive compounds. The problem is challenging, because each compound has hundreds of atoms that are connected to one another, and the hypothesis space of target patterns is large.

Table 5 reports the result on the HIV tasks. QuickFOIL is the only program that can process the HIV-Large dataset. Aleph did not terminate after 24 hours on both the HIV datasets, and FOIL started thrashing on the HIV-Large dataset. The reasons for the poor performance of Aleph and FOIL here are the same as those discussed in Section 5.2.1. However, another important reason why they do not work well on the HIV datasets is that they are lack of an effective pruning strategy. For example, an intermediate clause built by FOIL contains the following consecutive literals: $bond(L, B, M)$, $bond(N, O, G)$, $bond(P, Q, M)$, $bond(R, B, S)$. Likewise, Aleph expands the built clause starting with the following four literals: $atom(A, B)$, $atom(A, C)$, $atom(A, D)$, $atom(A, E)$. These literals are not "interesting," as the constraint that they enforce is subsumed by



(a) Scalability (b) Time breakdown (c) Incremental improvement

Figure 6: Performance of QuickFOIL with the Bongard datasets

other existing literals. Adding these literal dramatically increases the size of binding sets, since each of them introduces new variable(s). The pruning criterion based on the replaceable duplicates can effectively eliminate these literals.

Drilling down further into the pruning component, the average number of replaceable duplicates encountered during the ten-fold execution on the HIV-Large dataset is 9 times the number of renaming duplicates. We ran another set of experiments where we measured the performance of two cases: 1) the case when we only eliminate the queries for the renaming duplicates, and 2) the case when we only prune literals based on type constraints. For both cases, we ensured that no replaceable duplicates can be added to the built clause to guarantee that they have the same output result. This experiment revealed that 1) performance slows down by 34% when only eliminating the renaming duplicates, and 2) although the replaceable duplicates make up only 1.3% of the total literals that are explored, the performance speedup is 1.54X when removing them from the step that computes the scores. Thus, these experiments validate the effectiveness of our pruning strategy. Additional discussion on the WebKB and the HIV datasets can be found in the appendix.

5.3 Microbenchmarks on synthetic datasets

This section studies the performance of QuickFOIL using the Bongard datasets. Except for the scalability experiment, all other experiments use the TH3 datasets.

Scalability. In this experiment, we vary the size of the training sets for the three Bongard learning problems. The results plotted in Figure 6(a) demonstrate that QuickFOIL is able to tackle large datasets efficiently. QuickFOIL takes less than half an hour to finish processing the dataset with 2 million examples and with 178 million tuples in the background relations.

Execution breakdown. Figure 6(b) presents the detailed breakdown of the execution time on the TH3 datasets with two million examples. This figure breaks down the execution time into six categories: the time spent on partitioning relations and building hash tables (Part+Build); the time spent within the compound join operator (Join), the selection1 operators (Sel), and the aggregation operators (Agg) in executing the merged plan for q_1 and q_2 (cf. Figure 4); the execution time for q_3; and the time spent on all other components. The selection operators take ~50% of the total time, as it incurs a large number of random memory accesses. The next largest component is the join operator, which takes up ~30% of the total time. Overall, over 90% of the total time is spent in executing q_1 and q_2 (to calculate the literal scores).

Figure 6(c) shows the performance improvement on the TH3 dataset by incrementally adding the four optimization techniques that were described in Section 4 in the following order: 1) caching the intermediate partitioning results and hash tables (cache), 2) grouping queries that share the same inner join column (group), 3) merging the four queries q_1^+, q_1^-, q_2^+ and q_2^- for every candidate literal (merge) and 4) radix partitioning the background relations holistically on the join column values (holistic). The first three techniques are discussed in Section 4.2.2, 4.2.3 and 4.2.4, respectively. The last technique is introduced at the end of Section 4.2.2. We can see that the merging and the grouping techniques yield the biggest performance improvements. Collectively, the four techniques improve performance by a factor of 7X. Additional empirical results for these techniques are in in Appendix A.

6 Related Work

A survey of ILP work can be found in [35]. Past ILP systems have used a number of heuristic scoring functions, including accuracy [33], weighted relative accuracy [30], information [7], gini index [42] and m-estimates [15]. Other similar scoring functions, include the two weighted gain functions f_g [36] and f_r [29] (cf. Equation 1 and 2). Fossil [20] employs MCC as its scoring function to utilize the uniform scale property as its stopping criterion. The major problem with MCC is that it does not capture an important optimization goal of ILP, namely to maximize the coverage of the positive tuples. QuickFOIL achieves a more robust measure by combining MCC and the AUE gain, which can be considered as a variant of information gain but with a bounded scale.

The problem of conjunctive query containment was first studied in [8], and is related to the θ-subsumption test [26]. In prior work, θ-subsumption is used to specialize clauses, determine the coverage of examples, and for post-pruning redundant literals from complete learnt rules [33], but not for pruning search space when building rules. Note that removing replaceable duplicates is not equivalent to eliminating extended clauses that are subsumed by the base clauses.

There has been a long interest in scaling ILP to large datasets. FOIL-D [5] is an earlier SQL implementation of a simplified version of FOIL. To compute the scores for each candidate literal, it estimates the query results based on single-column histograms on relations, rather than actually executing the queries. While that strategy improves the performance, it relies on having accurate estimates, which can be challenging especially for multi-predicate join queries. CrossMine [46] is another work that aims to optimize the join queries that results from running FOIL. Instead of materializing the binding sets, CrossMine does not stitch tuples from the joined relations together but assigns the IDs of joined training examples to each tuple. When computing the coverage of examples for a candidate literal, CrossMine propagates the tuple IDs to the relation of the candidate literal from another joined relation. This tuple ID propagation approach can only be applied to a join on exactly two relations, one of which cannot correspond to an existing literal in the current rule. Therefore, the hypothesis language space of CrossMine is smaller than that of QuickFOIL. In particular, CrossMine generally cannot add a candidate literal that has common variables with multiple existing literals, which requires a join on more than two relations. Another fundamental distinction between CrossMine and QuickFOIL is that CrossMine counts the coverage of examples in the scoring function, while the scoring function in QuickFOIL is based on the coverage of the bindings. Counting examples is more expensive than counting bindings, because it needs an extra semi-join to find distinct examples in the bindings. PILP [45] scales ILP by compressing training examples. It groups multiple related examples into a single example and gives the resulting example a probabilistic label based on the proportion of each label in the original example set. There are also research efforts in parallel or distributed ILP systems [1, 43]. These are orthogonal to our work that focuses on improving the single-core performance.

We build on a rich body of work on sharing work across and within queries (e.g. [2, 6, 24, 48]). For example, QuickFOIL combines positive and negative tuples into a single table for query processing, resembling the input merging techniques in CJoin [6] and DataPath [2]. QuickFOIL also shares hash tables, partitioning results, and the intermediate results of common single-predicate join operators amongst different queries, which are classic computation sharing techniques. However, there are several key differences from existing work. First, as a main-memory ILP engine, QuickFOIL favors computation sharing over scan sharing. As a result, QuickFOIL pushes the join operation below the selection operation to improve the opportunity for join sharing, in contrast to the traditional selection push-down strategy. Second, the join in QuickFOIL is not a simple primary-key foreign-key join, but has multiple join predicates. In fact, there is no common join operation amongst queries for different literals in a literal search iteration. QuickFOIL exposes work sharing by executing a multi-predicate join as a single-predicate join with a subsequent selection operation, and then groups the joins to find overlapping computations. Third, QuickFOIL simultaneously performs multiple join operations that have different join predicates in a special join operator – i.e. the compound join operator. This operator carefully coordinates a partitioned hash join technique to achieve good cache locality.

7 Conclusions and Future Work

To scale ILP, we need two key components: a good learning algorithm that can produce high-quality results in a small number of search iterations without exhausting a large number of clauses, and an efficient and scalable implementation. This paper proposes a new ILP algorithm QuickFOIL that uses a top-down, greedy search with a novel scoring function and a new pruning strategy to meet these challenges. We have also developed query processing techniques for an efficient in-database implementation of QuickFOIL, and empirically demonstrated the effectiveness of our techniques.

There are a number of directions for future work, including expanding QuickFOIL to distributed environments, exploring the proposed query processing methods for general database workloads, and most importantly building on the connection between ILP and query processing articulated here to the broader field of Relational Learning.

8 Acknowledgements

This work was support in part by the National Science Foundation under grants III-0963993 and IIS-1250886, and by a gift donation from Google.

9 References

[1] Parallel ilp for distributed-memory architectures. *Machine Learning*, 74(3):257–279, 2009.

[2] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The datapath system: A data-centric analytic processing engine for large data warehouses. SIGMOD, pages 519–530, 2010.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[4] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[5] J. Bockhorst and I. Ong. FOIL-D: Efficiently scaling foil for multi-relational data mining of large datasets. In *ILP*, volume 3194, pages 63–79. 2004.

[6] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *VLDB*, 2(1):277–288, 2009.

[7] J. Cendrowska. Prism: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27(4):349–370, 1987.

[8] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. STOC '77, pages 77–90, 1977.

[9] C. Chasseur and J. M. Patel. Design and evaluation of storage organizations for read-optimized main memory databases. *VLDB*, 6(13):1474–1485, 2013.

[10] V. S. Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. V. Laer. Query transformations for improving the efficiency of ILP systems. *JMLR*, 4:465–491, 2003.

[11] M. Craven and S. Slattery. Relational learning with statistical predicate invention: Better models for hypertext. *Machine Learning*, 43(1-2):97–119, 2001.

[12] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. PODS, pages 59–70. ACM, 2001.

[13] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: integrating R and hadoop. In *SIGMOD*, pages 987–998, 2010.

[14] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Min. Knowl. Discov.*, 3(1):7–36, 1999.

[15] S. Džeroski. Handling imperfect data in inductive logic programming. SCAI, pages 111–125, 1993.

[16] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-RDBMS analytics. In *SIGMOD*, pages 325–336, 2012.

[17] P. M. Fernandez. Red brick warehouse: A read-mostly rdbms for open smp platforms. SIGMOD, page 492, 1994.

[18] N. Fonseca, V. Costa, F. Silva, and R. Camacho. On avoiding redundancy in inductive logic programming. In *ILP*, volume 3194, pages 132–146. 2004.

[19] N. Fonseca, F. Silva, and R. Camacho. April – an inductive logic programming system. In *Logics in Artificial Intelligence*, volume 4160, pages 481–484. Springer Berlin Heidelberg, 2006.

[20] J. Fürnkranz. Fossil: A robust relational learner. ECML, pages 122–137, 1994.

[21] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *ICDE*, pages 231–242, 2011.

[22] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: Killing one thousand queries with one stone. *VLDB*, 5(6):526–537, 2012.

[23] G. Graefe. Volcano: An extensible and parallel query evaluation system. *TKDE*, 6(1):120–135, 1994.

[24] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. SIGMOD, pages 383–394, 2005.

[25] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib analytics library: Or MAD skills, the SQL. *VLDB*, 5(12):1700–1711, 2012.

[26] J.-U. Kietz and M. LÃijbbe. An efficient subsumption algorithm for inductive logic programming. In *ICML*, pages 130–138, 1994.

[27] A. Klug. On conjunctive queries containing inequalities. *J. ACM*, 35(1):146–160, 1988.

[28] S. Kramer, L. De Raedt, and C. Helma. Molecular feature mining in hiv data. In *KDD*, pages 136–143, 2001.

[29] N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Routledge, 1993.

[30] N. Lavrac, P. A. Flach, and B. Zupan. Rule evaluation measures: A unifying view. In *Proceedings of the 9th International Workshop on Inductive Logic Programming*, ILP '99, pages 174–185. Springer-Verlag, London, UK, UK, 1999.

[31] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *TKDE*, 14(4):709–730, 2002.

[32] B. W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975.

[33] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13(3-4):245–286, 1995.

[34] S. Muggleton and W. L. Buntine. Machine invention of first order predicates by inverting resolution. In *ML*, pages 339–352, 1988.

[35] S. Muggleton, L. Raedt, D. Poole, I. Bratko, P. Flach, K. Inoue, and A. Srinivasan. ILP turns 20. *Machine Learning*, 86(1):3–23, 2012.

[36] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[37] L. D. Raedt. Inductive logic programming. In *Encyclopedia of Machine Learning*, pages 529–537. 2010.

[38] R. Ramakrishnan, Y. Sagiv, J. D. Ullman, and M. Y. Vardi. Proof-tree transformation theorems and their applications. PODS, pages 172–181, 1989.

[39] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.

[40] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[41] T. Scheffer, R. Herbrich, and F. Wysotzki. Efficient θ-subsumption based on graph algorithms. In *Inductive Logic Programming*, volume 1314, pages 212–228. 1997.

[42] A. Srinivasan. The Aleph manual. http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html.

[43] A. Srinivasan, T. A. Faruquie, and S. Joshi. Data and task parallelism in ILP using MapReduce. *Mach. Learn.*, 86(1):141–168, 2012.

[44] S. N. Subramanian and S. Venkataraman. Cost-based optimization of decision support queries using transient-views. *SIGMOD Rec.*, 27(2):319–330, 1998.

[45] H. Watanabe and S. Muggleton. Can ilp be applied to large datasets? In *ILP*, pages 249–256, 2010.

[46] X. Yin, J. Han, J. Yang, and S. Philip. Crossmine: efficient classification across multiple database relations. In *ICDE*, pages 399–410, 2004.

[47] Y. Zhao, P. M. Deshpande, J. F. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *SIGMOD*, pages 271–282, 1998.

[48] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a dbms. VLDB, pages 723–734, 2007.

APPENDIX

A Impact of Caching, Grouping and Merging

In this section, we examine the individual impact of the three techniques: cache, group and merge (cf. Section 4). We compare the performance with and without employing each technique.

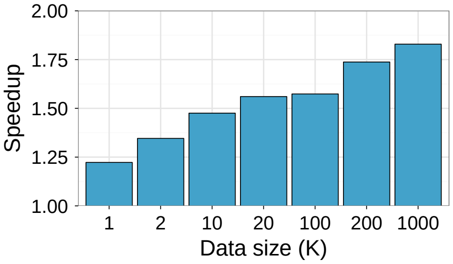


Figure 7: Effect of caching

A.1 Caching

Figure 7 plots the speedup from using the caching technique as the training size is varied. We see that the performance improvement is far higher than what Figure 6(c) shows, highlighting the benefit of using caching in conjunction with other optimizations. The performance gain increases as the training data grows, because the partitioning and the building time increases nearly linearly with the relation size. The use of caching makes the partitioning and the building time marginal, which can be seen in Figure 6(b).

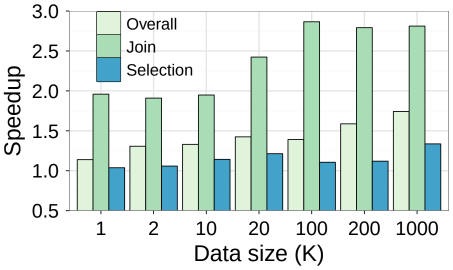


Figure 8: Effect of grouping

A.2 Multi-query optimization techniques

Figure 8 and 9 illustrate the overall improvement, and the performance gains associated the two operators: join and selection – the most significant two operators as shown in Figure 6(b).

Grouping and merging considerably speed up the single-predicate join operation, mainly because the shared join amongst the queries only needs to be performed once. The figures show that the merging technique reduces the joining time by up to two times, and the grouping technique leads to a speedup of two. The join ordering enforced by grouping and merging also improves the data locality in accessing the two join relations for the selection operators. In particular, merging reduces the number of selection operators by half, and thus we can see the selection time is twice as fast when merging is enabled. Thus, these results demonstrate the effectiveness of the techniques described in Section 4.2.3 and 4.2.4.

We have also conducted experiments that show the benefit of the integrated model over the independent model (∼20% performance improvement), and the impact of setting the binding relation as the inner relation (∼50% performance improvement). We omit these results in the interest of space.

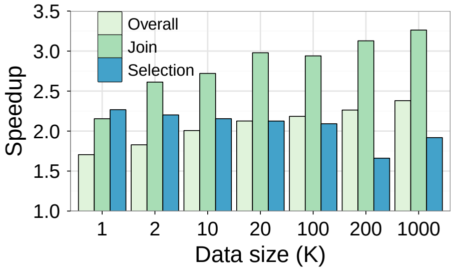


Figure 9: Effect of merging

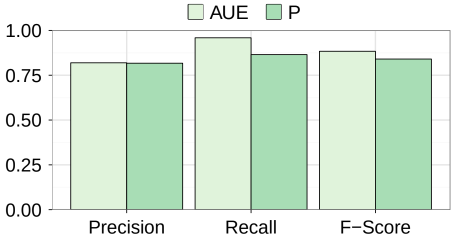
B Physical storage for the integrated model

The integrated model raises an important implementation issue. Recall that the initial binding set for building a new rule consists of those positive training examples that are not covered by previous rules and the entire set of negative examples. The two sets of training examples do not originally belong to one relation: The negative examples are part of the target relation, whereas the uncovered positive examples are a temporary relation generated at the end of the last iteration. Therefore, using the integrated model requires combining data from different relations into a new relation, and the original relations must be retained. Because physically copying data in every iteration can be expensive, we design the notion of *virtual blocks* to allow a relation to "borrow" data from other relations. A virtual block is a read-only block that does not itself own any data but simply refers to other blocks. The access to virtual blocks is transparent to operators by a unified iterator interface. We can incorporate the negative examples from the target relation into the binding relation by simply adding a virtual block pointing to the storage blocks of the target relation that contain the negative examples. We can consider the initial binding relation as a virtual non-materialized view on two relations.

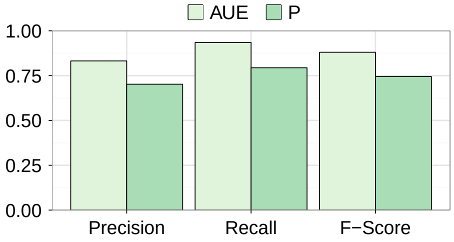
C On the QuickFoil scoring function

The QuickFoil scoring function combines the MCC measure and the AUE gain. The MCC measure is computed based on the coverage of the current binding set, and the AUE gain is the difference of AUE on the current binding set and the binding set of the extended rule with a new candidate literal. MCC measures how well the candidate literal can distinguish the positive bindings from the negative ones. Since MCC is symmetric with respect to false positives and false negatives, MCC give similar scores to the following two clauses: one that covers a lot of positive bindings with some negative bindings, one that excludes a large number of negative bindings and has few positive bindings. AUE is introduced to penalize clauses with few positive bindings.

The scoring function used in QuickFOIL counts bindings as the FOIL gain function. It is easy to give an excessive score to clauses with a high ratio of positive bindings, which do not necessarily have a large coverage of positive examples due to the skew problem. The AUE is able to mitigate the effect[6]. The S-shaped AUE does not assign a score to a clause in a manner that is linearly correlated with the ratio of the positive bindings (precision), but has a relatively smaller score increase as the precision becomes large. We replaced AUE with the precision (p) and measured the performance on the WebKB datasets. The results are shown in Figure 10. As can be seen in this figure, using the AUE scoring function outperforms using p, especially on the WebKB-Department task which is more vulnerable to the skew as it is a binary relationship learning problem on a skewed web graph and it has more variables and literals in the learnt rules compared to the WebKB-Student task.



(a) on WebKB-Student



(b) on WebKB-Department

Figure 10: Comparisons AUE and the precision scoring functions

D Related work on ILP scoring functions

In this section, we describe previous work on scoring functions used in past ILP methods.

Most ILP systems follow a separate-and-conquer approach, i.e. they select one rule at a time, remove the positive examples covered by the rule, and then iterate. This approach needs a metric or a heuristic scoring function to drive the process of specializing or choosing a rule. We can classify the search heuristics into two categories. The first category measures the quality of a candidate rule based on the coverage of the input training examples. The scoring functions in this category essentially treat every rule as complete. These scoring functions use measures such as accuracy [33], weighted relative accuracy [30], information [7], gini index [42] and m-estimates [15]. The second category, which has not been extensively studied in literature, evaluates the effect of adding a new condition (eg. a new literal) to specialize a rule. The quality of a rule is defined not by itself, but with respect to a generalized rule. The existing scoring functions that fit in the second category have the two weighted gain functions f_g [36] and f_r [29] (cf. Equation 1 and 2), and MCC used in Fossil [20]. The two weighted information gain functions are computed based on the information difference between the two binding sets before and after a literal is added. As discussed in Section 3.1, they result in poor performance in the presence of skewed data that leads to imbalanced growth of the positive and negative binding tuples. Fossil [20] employs MCC as the scoring function to utilize the uniform scale property for a cutoff stopping criterion. The major problem with MCC is that it does not capture an important

[6] Both the scoring function and the pruning strategy are designed to overcome the skew. MCC is a balanced measure that takes into account the coverage of negative binding sets. The pruning techniques that can remove literals with high ratios of positive bindings, but without excluding negative bindings.

optimization goal of ILP to maximize the coverage of positive tuples. QuickFOIL achieves a more robust measure by combining MCC and the AUE gain, which can be considered as a variant of information gain with a bounded scale.

E Connection to previous work related to the QuickFOIL pruning technique

In this section, we make a more detailed connection to previous work that is related to the pruning technique described in Section 3.2.

The problem of conjunctive query containment (on which our pruning method discussed in Section 3.2 is based) was first studied in [8]. Subsequent work has studied the containment problem for various extensions of conjunctive queries in the database community (e.g. [27,40]). The corresponding problem in logic programming is called the θ-subsumption test [26, 41]. However, In prior work, θ-subsumption is used to specialize clauses, determine the coverage of examples, and for post-pruning redundant literals from complete learnt rules [33], not for pruning search space when building rules. In Warmr [14], pruning is performed after the learning process to remove clauses that are θ-subsumed (i.e. contained) by others. To the best of our knowledge, the only syntax redundancy introduced in literature to eliminate literals when specializing a clause is the renaming duplicate [10, 18]. We have shown that renaming duplicates are only a subset of replaceable duplicates and replaceable duplicates can be determined practically using the canonical database test [38]. Aleph [42] provides declarative language support for user-custom pruning. Users can specify the type constrains for each predicate argument, the syntax form of valid clauses, and provide predicates that determine whether an input literal is redundant in a given clause. April [19] introduces language extensions to describe the properties of predicates such as tautology and commutativity as expert knowledge to restrict the search space. However, they cannot express replaceable duplicates. Also, these language extensions can be integrated into our work.

F Shared workload optimization

In this section, we describe previous work on shared workload optimization in more details.

Exploiting shared work among different queries to speed up the performance has been well-studied in the database community. Earlier work [39, 44, 47] on traditional disk-based database management systems that used the iterator execution model mainly employed costly materialization to share intermediate results for batch query optimization. Dalvi et al. [12] introduces a theoretical framework for deciding whether intermediate results can be pipelined instead of materialized, and proposes query optimization algorithms to maximize the opportunities for pipelining. However, their work does not provide a practical mechanism to enable pipelining for shared operators.

There is a rich body of recent work targeting parallel query evaluation, where there is one thread per query or one thread per operator. Pipelining the shared intermediate results across different queries is naturally supported in parallel or distributed environment. To mitigate the cost of (expensive) disk I/Os, scan sharing is a common technique that has been proposed in previous research. A number of systems including RedBrick [17], QPipe [24], DataPath [2], CJoin [6] use circular scan operators that constantly stream data from disk and feed all queries requesting the data when available. Cooperative scan [48] improves the sharing by dynamically scheduling the scan operations based on the demands from concurrent queries. In addition to I/O sharing, QPipe detects overlap computations amongst queries and pipelines the results to each sharing query. CJoin and DataPath merges multiple scan operators, or multiple join operators with the same predicate, on overlapping inputs into a single "big" operator on a union set of all inputs, using bits to indicate the ownership of input tuples and output results, so that the tuple access and computation costs are amortized. ShardDB [22] generalizes this technique to other relational operators including sorting and grouping for more general workloads with response time guarantees.

G Comparisons with CrossMine

CrossMine uses a top-down, greedy learning strategy and applies the FOIL gain to evaluate candidate literals. When building a rule, for each tuple t in every relation (a.k.a. active relation) that participates in the body of the rule currently being built, it maintains a set of IDs of examples (i.e. target tuples) that are covered by the rule and can be joined with t. To assess a new rule by augmenting the current rule with a new predicate for which the corresponding relation can be joined with an active relation, it computes the union of the tuple IDs in those tuples of the active relation that can be joined with the relation of the new predicate to get the number of examples covered in the join result. It also proposes a sampling approach to facilitate the learning: If negative examples are more than the positive examples by a threshold, a subset of the negative examples is created by sampling, and the algorithm is applied to the sampled negative examples combined with the positive ones.

CrossMine and QuickFOIL share some common ideas.

- Both use the top-down, greedy rule learning strategy.
- One optimization in both approaches is avoiding redundant computations. In each literal search strategy, CrossMine performs the join between two relations only once and applies multiple value constraints (corresponding to multiple predicates) on the join results.

However, they differ in the following ways.

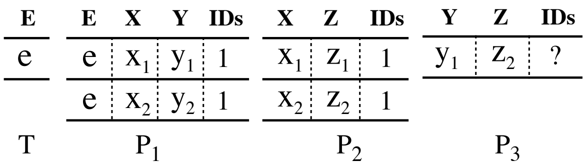
First, CrossMine uses the FOIL gain scoring function, but it deviates from FOIL in that it counts the coverage of examples rather than the coverage of bindings (a.k.a. substitutions). QuickFOIL instead counts bindings to compute its scoring function. The tuple ID propagation approach cannot be applied if we count bindings. More details comparisons between counting examples and counting bindings is given later.

Second, CrossMine has a more restricted literal (or predicate) search space, and is less expressive in terms of rules that can be learnt compared to QuickFOIL.

1. CrossMine assumes that there is only one way to join two relations, and does not generate a literal that requires a join between two relations that involve multiple primary-key-foreign-key/foreign-key-foreign-key pairs, or a join that is not on a primary-key-foreign-key/foreign-key-foreign-key pair. For example, consider learning nodes that are in a cycle of length two in a graph:

$$\mathit{positive}(N_1) \text{ :- } \mathit{link}(N_1, N_2), \mathit{link}(N_2, N_1)$$

To compute the FOIL gain of the second literal $\mathit{link}(N_2, N_1)$, we need to perform a self-join on *two*



E
e

T

E	X	Y	IDs
e	x_1	y_1	1
e	x_2	y_2	1

P_1

X	Z	IDs
x_1	z_1	1
x_2	z_2	1

P_2

Y	Z	IDs
y_1	z_2	?

P_3

Figure 11: Tuple ID propagation cannot be applied to generic multi-relation joins.

equality predicates (i.e. two foreign-key-foreign-key pairs), and thus CrossMine cannot learn the target rule. Note that it is *impossible* for CrossMine to learn the second literal by two steps (each using a join on a single-column join predicate), because the second step involves a join between two active relations. As discussed later, tuple ID propagation cannot be applied to a join with two active relations. Nevertheless, we can still overcome this limitation by considering every possible join predicate for each pair of relations in generating complex predicates. We then evaluate each join predicate, and propagates tuple IDs to a new distinct instance of relation *link*, one per predicate.

2. CrossMine cannot add a literal that has common variables with multiple existing literals, in which case we need a join across more than one active relation. Because CrossMine does not materialize the binding set for the current rule and we cannot in general determine whether two tuples in two relations can be joined from whether they can be joined with some tuples in a third relation, we cannot derive the covered examples from the tuple IDs stored in the active relations.
   In general[7],
   - We cannot propagate the IDs of target tuples from active relations to a non-active one when the non-active relation is joined with more than one active relation.
   - We cannot update the IDs of an active relation when we add a new constraint that joins it with another active relation.

   Consider the example shown in Figure 11. Assume that the target relation is T and we have added two non-target active relations P_1 and P_2, which form the current built rule $T(E)$:- $P_1(E, X, Y), P_2(X, Z)$. The tuple IDs for each tuple are shown in the figure. Suppose we are considering adding a new literal $P_3(Y, Z)$ and need to compute its FOIL gain. Although every tuple in P_1 and P_2 has a non-empty equivalent tuple ID set ($\{1\}$) and P_2 can be joined with both P_1 and P_2, it is easy to see that the join result for $P_1 \bowtie P_2 \bowtie P_3$ (with join predicates determined by common variables) is empty, and hence the tuple ID set for the tuple in P_3 should be empty. We cannot reach this conclusion by performing set operations on the tuple IDs. Instead, we need to perform *the entire join for all non-target active relations*, not only for relations that are directly connected to the non-active relation[8].

The above two classes of literals, in particular the second class for which the tuple ID propagation is inapplicable, limit the applications of CrossMine. These literals that cannot be learnt by CrossMine is not uncommon in real learning tasks. In fact, the target concepts for the learning problems in our experiments all require them.

H Discussion on the WebKB and the HIV experiments

An important rule found in the WebKB learning tasks reveals an interesting link pattern between student pages and the student directory page: a student page usually has a link from the student directory page on the department website. Specifically, a result Horn clause learnt from the WebKB dataset is:

$$student(X) \text{ :- } link\_to(W, Y, X), has\_jame(Y), has\_paul(Y),$$
$$link\_to(V, Z, Y), has\_student(Z)$$

This rule is the single rule that QuickFOIL found in one of the four-fold validation rounds. This clause indicates that a page X is a homepage of a student if X is linked from another page Y which has two words *jame* and *paul*, and which is in turn is linked from a page Z containing a word *student*. Recall that the background knowledge only includes links between pages, the bag-of-words representation for pages as well as the anchoring text of links. It does not provide the type of pages. The rule identifies a directory page by two names. Note that although the two names *jame* and *paul* are common, it is not common that they appears on the same page and the page links to another page with a keyword student.

Table 6 summarizes the average number of learned rules per validation round (#R) and the average number of literals per rule (#L). For HIV-Small, the reason why FOIL outputs a small number of rules is that it fails to find a rule that has a minimum accuracy (0.8) in most clause search iterations.

	Student		Department		HIV-Small		HIV-Large	
	#R	#L	#R	#L	#R	#L	#R	#L
QuickFOIL	11	5	12	5	14	8	38	16
FOIL	33	5	32	6	3	6	–	–
Aleph	13	5	191	6	–	–	–	–

Table 6: Summary of learned rules on the WebKB and HIV datasets

[7] Even if CrossMine only considers a join between a active relation and a non-active relation, the problems still happen when CrossMine updates the tuple IDs after it selects a literal. The Tuple IDs in each active relation cannot be updated in place. CrossMine has to perform a complete join for the entire rule.

[8] If we have another active relation $P_4(E, X)$, it also needs to be included in the join in addition to P_1, P_2 and P_3, in order to compute the coverage of examples, although P_4 is not directly joined with P_3.