Pre and Post Counting Approaches for Scalable Statistical-Relational Model Discovery

Richard Mar, Oliver Schulte
Simon Fraser University
richard_mar@sfu.ca, oschulte@cs.sfu.ca

Abstract

Statistical-Relational Model Discovery aims to find statistically relevant patterns in relational data. Relational patterns may be represented in first-order graph models and/or a set of clauses. For example, a relational dependency pattern may stipulate that a user's gender is associated with the gender of their friends. As with propositional graphical models, the major scalability bottleneck for model discovery is computing instantiation counts: the number of times a relational pattern is instantiated in a database. Previous work has utilized pre-counting or post-counting to solve this task. A pre-counting approach computes and caches instantiation counts for a large set of relational patterns *before* model search. A post-counting approach computes an instantiation count dynamically on-demand for each candidate pattern generated during the relational model search. This paper takes a detailed look at the memory and speed trade-offs between pre-counting and post-counting strategies for relational learning. We describe a novel hybrid approach that achieves a sweet spot with pre-counting for patterns involving positive relationships (e.g. pairs of users who are friends) and post-counting for patterns involving negative relationships (e.g. pairs of users who are not friends).

Introduction

Everyday the size of databases continues to grow and many applications from various areas such as business intelligence and healthcare extract information from them. Statistical-Relational Learning (SRL) aims to construct statistical models that extract knowledge from complex relational and network data. SRL models are often applied to datasets that are large (with many entities/nodes/rows) and complex (with many attributes and types of relationships, or predicates in logic syntax).

SRL model construction searches through a space of possible models, scoring different candidate models to find a (local) statistical score optimum. SRL models are typically composed of local dependency patterns, represented with edges in a model graph or clauses in logical syntax. The main computational burden in model scoring is *instance counting*, determining the number of times that a local pattern defined by the model occurs in the dataset. A key technique to scale counting for large and complex datasets is *caching* local instance counts and statistical scores. Caching trades memory for model search time and is standard in graphical model search for propositional (i.i.d.) data (e.g. (The Tetrad Group 2008)). There are two basic approaches to counts caching: Pre-counting builds up a large cache of local pattern counts before the model search phase. Post-counting computes local instance counts only for patterns generated during model search and then adds them (or the resulting scores) to a cache in case the pattern is revisited later during model search. (Lv, Xia, and Qian 2012) describe in detail the pros and cons of pre and post counting for propositional model search. This paper compares pre and post counting for relational model search.

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

To understand the differences, we briefly review why relational counting problems are fundamentally different from, and harder than, counting in propositional data. For propositional data, a conjunctive pattern specifies a set of attribute values for singletons of entities; for example, the set of female users over 40. Counting then requires only *filtering*, finding the subset of *existing* entities matching the pattern. Relational data specify linked pairs of entities, like users who are friends (assuming binary relationships), where two new counting problems arise:

1. Relational patterns may involve k-tuples of entities/nodes. For example, a triangle involves 3 nodes. A naive approach to counting would require enumerating n^k k-tuples of entities, which does not scale in the number of entities n even for small k. In an SQL table representation of relational data, counting for k-tuples of entities involves table JOINs; we therefore term it the *JOIN problem*. The JOIN problem has been extensively studied in the database community as part of SQL query evaluation (COUNT(*) queries).
2. Relational patterns may involve *negative relationships*, pairs of entities not listed in the data. For example, we may wish to find the number of women users who have *not* rated a horror movie. We term this the *negation problem*. The JOIN and negation problems both involve tuples that are not subsets of the given data tuples.

The main advantage of pre-counting is that the data is scanned only once to build up a cache. In particular, the number of table JOINs is minimized, as our experiments show. Also, the cache can be built incrementally using dy-

namic programming by computing counts for longer patterns from shorter ones (Lv, Xia, and Qian 2012). For example, in relational data, counts with a relationship chain length of l can be built up from relationship chains of length 1, 2, ..., l - 1 (Yin et al. 2004; Schulte and Qian 2019). The main disadvantage of pre-counting is building a large cache of counts for complex patterns, with only a fraction likely to be required for model selection. For example, pre-counted dependency patterns may involve 20 predicates, whereas model search rarely considers clauses with more than 5. The main advantage of post-counting is that counting is performed only as required for patterns generated by the model search. The disadvantage is that each pattern evaluation requires a new data access with potentially expensive table JOINs.

A high-level summary of our findings is that because of the JOIN problem, purely post-counting scales substantially worse than purely pre-counting. At the same time, the negation problem is challenging for the large patterns built up in pre-counting. Therefore, we introduce a hybrid method that uses pre-counting for the JOIN problem and post-counting for the negation problem. The hybrid method leverages the Möbius Join algorithm which computes instance counts for relational patterns that may involve positive and negative relationships (Qian, Schulte, and Sun 2014). The Möbius Join is an inclusion-exclusion technique that requires no further data accesses to the original data, and hence no further table JOINs. Our contributions may be summarized as follows:

- Describing pre and post counting strategies for building cache counts that support SRL model evaluation.
- Investigating the strengths and weaknesses of pre/post counting strategies for different dataset properties.
- Describing a new hybrid method that for many datasets combines the strengths of both pre and post counting.

Related Work

We selectively discuss the most relevant topics from previous SRL work.

Language Bias. We consider first-order relational patterns only that involve types of individuals, not specific individuals (e.g. $Friend(X, Y)$ not $Friend(joe, Y)$). While this is a common restriction in SRL (Ravkic, Ramon, and Davis 2015; Schulte et al. 2016; Kimmig, Mihalkova, and Getoor 2014), it limits the applicability of our results to methods that search for clauses involving individuals, such as Boostr (Natarajan et al. 2012). Our experiments investigate learning directed models (first-order Bayesian networks), but our observations about the fundamental JOIN and negation problems apply to other SRL models as well (e.g. undirected models).

Search Space: Predicates vs. Clauses Inspired by graphical models, a tradition in SRL is to search for dependencies between predicates, typically represented by links in a model graph. For example, an edge $intelligence(S) \leftarrow grade(S, C)$ may represent that the grades of a student predicts their intelligence. Another tradition, inspired by inductive logic programming, searches for dependencies among predicate values, typically represented by clauses (e.g. $intelligence(S) = high$:- $grade(S, C) = A$). Since the space of clauses is smaller, it supports a more efficient model search (Kersting and De Raedt 2007; Friedman et al. 1999; Khosravi et al. 2010) at the expense of expressive power (Khosravi et al. 2012). With regard to instance counting, a predicate dependency groups together a set of clauses (one for each combination of parent-child values). Model evaluation requires instance counting for the entire clause set, which is often more efficient than separate counting, if the data for the same predicates is stored together. For example if in an SQL table, grades are stored in the same table column, then counting instances for different grades requires only a single table JOIN. While our experiments evaluate only predicate-level search, we expect that the general trends apply also to clause-level search.

Approximate vs. Exact Counting. Several SRL approaches have increased scalability by aiming for approximate counting, which is often sufficient for model evaluation (e.g. (Das et al. 2019)). In this paper we use only exact counts, obtained by efficient SQL queries. Other SRL work has also shown the usefulness of SQL optimization for obtaining instance counts (e.g. (Niu et al. 2011)). Approximate counting speeds up the instance for a *single pattern*, which is orthogonal to our topic of counts caching; caches can be used for both approximate and exact counts. We discuss approximate counting further in the future work section. Another type of approximation is negative sampling of unlinked node pairs, which adds extra node pairs to the data (Nickel et al. 2016). Computationally, negative sampling solves the negation problems but increases the JOIN problem. Caching counts for negative sampling is an interesting direction for future work.

In the following sections, necessary background information will be provided followed by the possible methods for achieving scalable construction of contingency tables (ct-tables). Each method for computing ct-tables from multi-relational data is then analyzed and compared, showing the advantages of the HYBRID method for caching ct-tables from a relational database.

Graphical Model Background

We choose the FACTORBASE system (Schulte and Qian 2019) for experimentation and to illustrate computational counting challenges. FACTORBASE learns a first-order Bayesian network (BN) (Getoor and Taskar 2007)) for a relational dataset. FACTORBASE is an appropriate benchmark as it achieves SOTA scalability (Schulte et al. 2016). We focus on our description on fundamental issues associated with relational counting that we expect to generalize to other systems. In the Limitations section, we discuss other design options not explored in our experiments.

As an example model score, the BDeu score for a given BN B and dataset D is defined in Equation 1. Descriptions of the variables can be found in Table 1. (Schulte and Gholami 2017) showed how the BDeu score can be adapted for multi-relational data. The count operations required are essentially the same between the relational BDeu score and the propositional score shown, and similar for other BN scores. The most computationally expensive part of BDeu, and many other scoring metrics, is generating instantiation

$$BDeu(B,D) = \log\left(P(B)\right) + \sum_{i=1}^{n}\sum_{j=1}^{q_i}\left(\log\left(\frac{\Gamma\left(\frac{N'}{q_i}\right)}{\Gamma\left(N_{ij}+\frac{N'}{q_i}\right)}\right) + \sum_{k=1}^{r_i}\log\left(\frac{N_{ijk}+\frac{N'}{r_i q_i}}{\Gamma\left(\frac{N'}{r_i q_i}\right)}\right)\right) \quad (1)$$

Variable	Description
n	Number of nodes in the graph.
q_i	Number of parent configurations for node i.
r_i	Number of possible values for node i.
N'	Equivalent sample size.
N_{ij}	Number of occurrences where $node_i$ has its parents in the jth configuration.
N_{ijk}	Number of occurrences where $node_i$ is assigned its kth value with its parents in the jth configuration.

Table 1: Variables of the BDeu scoring metric.

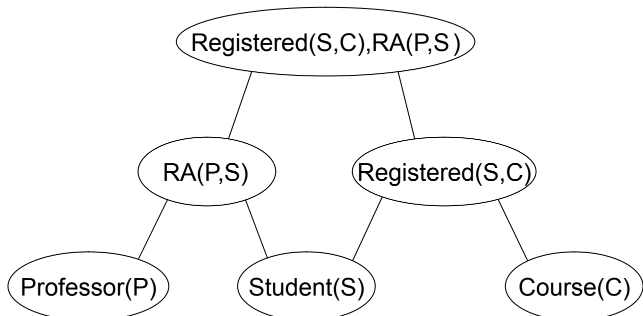

Figure 1: A relationship lattice for two relationships (students register in courses and work as RAs for professors) where each node in the graph is a lattice point.

Positive ct-table	Negative ct-table	Learning Method
Lattice Point	Lattice Point	PRECOUNT
Family	Family	ONDEMAND
Lattice Point	Family	HYBRID
Family	Lattice Point	IMPOSSIBLE

Table 2: Summary of possible methods to consider for computing ct-tables from a relational database based on the inputs for creating the positive and negative ct-tables.

Count	*Capa(P,S)*	*RA(P,S)*	*Salary(P,S)*
203	N/A	F	N/A
5	4	T	HIGH
4	5	T	HIGH
2	3	T	HIGH
1	3	T	LOW
2	2	T	LOW
2	1	T	LOW
2	2	T	MED
4	3	T	MED
3	1	T	MED

Table 3: A contingency table represents instantiation counts of a conjunctive condition in a database. For example, the last row shows that the number of instances (groundings) is 3 for the assignment *Capa(P,S)* = 1, *RA(P,S)* = T, *Salary(P,S)* = MED. Numbers are chosen for illustration.

or frequency counts to obtain the N_{ij} and N_{ijk} values. Instance counts can be represented in a **ct-table**: For a list of variables $V_1, \ldots, V_m$, the table contains a row for each value tuple $v_1, \ldots, v_m$, and records how many times this value combination occurs in the data set (Lv, Xia, and Qian 2012; Qian, Schulte, and Sun 2014). Following (Lv, Xia, and Qian 2012), *we use ct-tables as caches for instance counts.* We next describe a method for computing relational ct-tables.

Computing Relational Contingency Tables

We use the method of (Qian, Schulte, and Sun 2014), as implemented in FACTORBASE. As discussed under Related Work, other methods may be employed as well, such as approximate counting. In a relational ct-table, a key role is played by relationship indicator variables that indicate whether a relationship holds (e.g. $Registered(S,C) = T$) or not (e.g. $Registered(S,C) = F$ or $\neg Registered(S,C)$). In a **positive** ct-table, all relationship indicator variables are set to T (true). A complete ct-table contains values for both true and false relationships; see Table 3 for illustration. The last 9 rows form the positive ct-table. The predicate *RA(P,S)* represents that a student is an RA for a professor, and the attributes *Capa(P,S)* and *Salary(P,S)* represent a student's salary and capability when they are the RA for a given professor.

FACTORBASE uses a 2-stage approach to compute a relational ct-table, which is described as follows:

1. Input: A list of variables $V_1, \ldots, V_m$, and a relational dataset D.
2. Generate a positive ct-table $ct_+(\mathbf{R})$ for each relationship lattice point $\mathbf{R}$. FACTORBASE uses SQL INNER JOINs to compute a ct-table for existing relationships (a COUNT(*) query with a GROUP BY clause).
3. Extend $ct_+(\mathbf{R})$ to a complete ct-table $ct(\mathbf{R})$. FACTORBASE uses the Möbius Join for this step.

The Möbius Join is an inclusion-exclusion technique whose details will not be discussed as they are somewhat complex and not necessary for this work. The main points of importance for this research are as follows:

1. Given a positive ct-table, where all relationships are true, the Möbius Join returns a ct-table for both existing and non-existing relationships *without further access of the original data.*

2. The runtime cost of the Möbius Join as shown by (Qian, Schulte, and Sun 2014) is the following:

$$O\left(r \log\left(r\right)\right) \tag{2}$$

where r is the number of rows in the output ct-table.

The runtime analysis assumes that the JOINs executed during the Möbius Join algorithm use a sort merge join. Next we consider approaches for computing relational ct-tables. The basic choices are to compute a *global* ct-table for all variables or those associated with a chain of relationships. Or a smaller ct-table for those associated with a family (local pattern). Table 2 lays out the options, which we discuss in the next sections.

PRECOUNT Counts Caching Method

Chains of relationships form a lattice that can be used to structure relational model search (Schulte and Khosravi 2012; Yin et al. 2004; Friedman et al. 1999); see Figure 1. For instance, the learn-and-join model search (Schulte and Khosravi 2012) builds up BNs for each lattice point in a bottom-up fashion. The pre-count method computes a ct-table for each relationship chain, for all variables associated with the chain. For example, for the lattice point $\mathit{Registered}(S, C)$, the variables include all attributes of students, courses, and the registration link (e.g. grade). The high-level structure of pre-count caching is illustrated in Algorithm 1. Given a large ct-table, we can compute a smaller ct-table for a subset of columns by summing out the unwanted columns; this operation is called **projection** (Lv, Xia, and Qian 2012). For example, given a ct-table for the lattice point $\mathit{Registered}(S, C)$ with columns for all attributes of students, projection can be used to obtain a ct-table for just one student attribute. During structure search, pre-count computes ct-tables for local families from the applicable lattice point ct-table using projection.

PRECOUNT will generate ct-tables for each relationship chain that grow in size as we increase the chain length, which is a key factor for its computational performance.

Algorithm 1 The PRECOUNT method: pre-compute ct-tables for each lattice point.

1: **for each** $\mathit{latticePoint}\ LP \in \mathit{relationshipLattice}$ **do**
2: $\quad ct_{+}(LP) \leftarrow \text{INNERJOIN}(\text{TABLES}(LP))$
3: $\quad ct(LP) \leftarrow \text{MÖBIUSJOIN}(ct_{+}(LP))$
4: **end for**
5: **for each** $\mathit{family} \in \mathit{structureLearning}$ **do**
6: $\quad ct(\mathit{family}) \leftarrow \text{PROJECT}(ct(LP), \mathit{family})$
7: $\quad \mathit{score} \leftarrow \text{BDEU}(ct(\mathit{family}))$
8: **end for**

Contingency Table Growth Rate for PRECOUNT

Given a database table T with C columns where each column contains $\leq V$ values each, the size (number of rows) of T can be upper bound using the following expression:

$$\text{SIZE}(T) = O\left(V^{C}\right) \tag{3}$$

From Equation 3 we see that ct-tables grow exponentially with respect to the number of columns when using PRECOUNT. The alternative is to generate many small tables that are equivalent to the single large one, but grow in size at a slower rate. This is the approach used by ONDEMAND, which is described next.

ONDEMAND Counts Caching Method

This is one of two alternative methods to PRECOUNT for computing ct-tables from a relational database and is an adaptation of the single-table post-counting concept from (Lv, Xia, and Qian 2012). ONDEMAND works the same as PRECOUNT except it computes ct-tables for *families* not relationship chains.

Since ONDEMAND will generate a ct-table for each family of nodes that is scored, these ct-tables grow a lot slower than the ones generated with PRECOUNT.

Algorithm 2 The ONDEMAND method: compute ct-tables for each family during structure search.

1: **for each** $\mathit{family} \in \mathit{structureLearning}$ **do**
2: $\quad ct_{+}(\mathit{family}) \leftarrow \text{INNERJOIN}(\text{TABLES}(\mathit{family}))$
3: $\quad ct(\mathit{family}) \leftarrow \text{MÖBIUSJOIN}(ct_{+}(\mathit{family}))$
4: $\quad \mathit{score} \leftarrow \text{BDEU}(ct(\mathit{family}))$
5: **end for**

Contingency Table Growth Rate for ONDEMAND

Given a table T with C columns, with each column containing $\leq V$ values each and k parents to choose from, the total size of the equivalent multi-table version of the single large one can be upper bound using the following expression:

$$\text{SIZE}(T) = O\left(C\binom{C-1}{k}V^{k+1}\right) \tag{4}$$

From Equation 4 we see that *ct-tables grow exponentially with respect to the size of the family of nodes being scored when using ONDEMAND.* Therefore, ONDEMAND is advantageous only if the family sizes generated during search are small. Inspection of the BDeu equation seen in Equation 1 reveals that small families will be favoured as large families are highly unlikely since they will incur an exponential penalty resulting in a low metric score. Indeed the literature on BN learning states that the maximum number of parents is typically 4 (Lv, Xia, and Qian 2012; Tsamardinos, Brown, and Aliferis 2006). The general expectation that BN nodes have small indegrees also holds in this work's datasets. Table 4 shows the mean number of parents per node in the BN learned for the various databases used in the experiments and supports this statement.

Although the expected total size of the ct-tables generated using ONDEMAND is promising, it also generates many positive ct-tables, which can lead to many expensive table JOINs.

HYBRID Method

This is the new alternative method to PRECOUNT for computing ct-tables from a relational database. The pseudocode

for it is found in Algorithm 3. HYBRID combines the strengths from PRECOUNT and ONDEMAND and has the following characteristics:

- Like PRECOUNT, it generates a positive ct-table $ct_+(\mathbf{R})$ for each relationship lattice point $\mathbf{R}$.
- Like ONDEMAND, it generates a ct-table $ct(\mathbf{F})$ for each family of nodes $\mathbf{F}$ being scored.

Algorithm 3 Generating ct-tables using HYBRID.

1: **for each** $latticePoint\ LP \in relationshipLattice$ **do**
2: $\quad ct_+(LP) \leftarrow \textsc{InnerJoin}(\textsc{Tables}(LP))$
3: **end for**
4: **for each** $family \in structureLearning$ **do**
5: $\quad ct_+(family) \leftarrow \textsc{Project}(ct_+(LP), family)$
6: $\quad ct(family) \leftarrow \textsc{MöbiusJoin}(ct_+(family))$
7: $\quad score \leftarrow \textsc{BDeu}(ct(family))$
8: **end for**

The HYBRID method uses the $ct_+(\mathbf{R})$ as a cache to replace expensive JOINs with projections (line 5). Assuming that local families are small, extending positive ct-tables to complete ones is relatively fast; see Equation (2). Both HYBRID and PRECOUNT assume that the overall number of columns/relationships in the database is moderate so that computing the positive ct-table using all columns of a given lattice point is feasible. In other words, $ct_+(\mathbf{R})$ can be computed via a JOIN without exhausting the available resources. If the overall number of columns/relationships is too large to compute the $ct_+(\mathbf{R})$ tables, ONDEMAND must be used instead of PRECOUNT or HYBRID.

Hardware and Datasets

Jobs were submitted to Compute Canada's Cedar high performance cluster using Slurm version 20.11.4. Each job requested one 2.10GHz Intel Broadwell processor and enough RAM to process each dataset. In addition, the cluster required a time limit to be specified for each job and as a result each job was allotted a runtime of 100 minutes. This amount of time was allotted since it was more than enough time for FACTORBASE to finish processing most datasets while at the same time allowing the jobs to be scheduled to run on the cluster within 24 hours after job submission. Using MariaDB version 10.4 Community Edition, 8 real database sources of varying size and complexity were supplied as input to FACTORBASE for each method in Table 2. Of these 8 databases, 7 are benchmark databases that have been used in previous studies where the scalability of methods for constructing BNs from relational databases was studied (Schulte and Qian 2019; Schulte et al. 2016). Visual Genome is added to this list of benchmark databases due to the large amount of data and number of relationship tables it possesses compared to the other 7 databases; see Table 4. Visual Genome is the only dataset that required preprocessing where ternary relationships had to be converted into binary relationships. After the ternary to binary relationships conversion was done, entity tables with only a primary key column had a dummy attribute column added with the same value used for all rows.

File Size	Database	Relationship Table Count	MP/N
18KB	Mondial	2	1.3
20KB	UW	2	1.6
221KB	Mutagenesis	2	1.6
345KB	Hepatitis	3	1.7
1.1MB	MovieLens	1	1.4
5.7MB	Financial	3	1.9
23MB	IMDb	3	3.4
239MB	Visual Genome	8	0.5

Table 4: Databases used as input to FACTORBASE with their associated mysqldump file size, number of relationship tables they contain, and the mean number of parents per node (MP/N) in the BNs learned when using FACTORBASE and the BDeu score.

DB	Estimated $ct(family)$ Count	Total Row	$ct(database)$ Total Row Count
MovieLens	816		239
Mutagensis	6075		1631
UW	15318		2828
Visual Genome	2923968		20447
Mondial	55800		1738867
Financial	930468		3013006
Hepatitis	176220		12374892
IMDb	33040		15537457

Table 5: Size of the ct-tables generated for each family scored and the entire database. The $ct(family)$ column is for ONDEMAND and HYBRID while the $ct(database)$ column is for PRECOUNT. Databases are sorted in ascending order based on the size of the ct-table generated for the entire database.

These two modifications ensure that FACTORBASE could process the Visual Genome dataset.

Runtime Measurement Methodology

Since the focus of this research is reducing the time it takes to generate ct-tables for large relational databases, We break down ct-table construction runtime into the following 3 components to have a better understanding of the process:

- MetaData
- Positive ct-table (ct_+)
- Negative ct-table (ct_-)

MetaData This component consists of various metadata specifying the relational schema (syntax) that is extracted and generated from the input database. It includes the extraction of the first-order logic information related to the ct-tables being generated, generation of the relationship lattice,

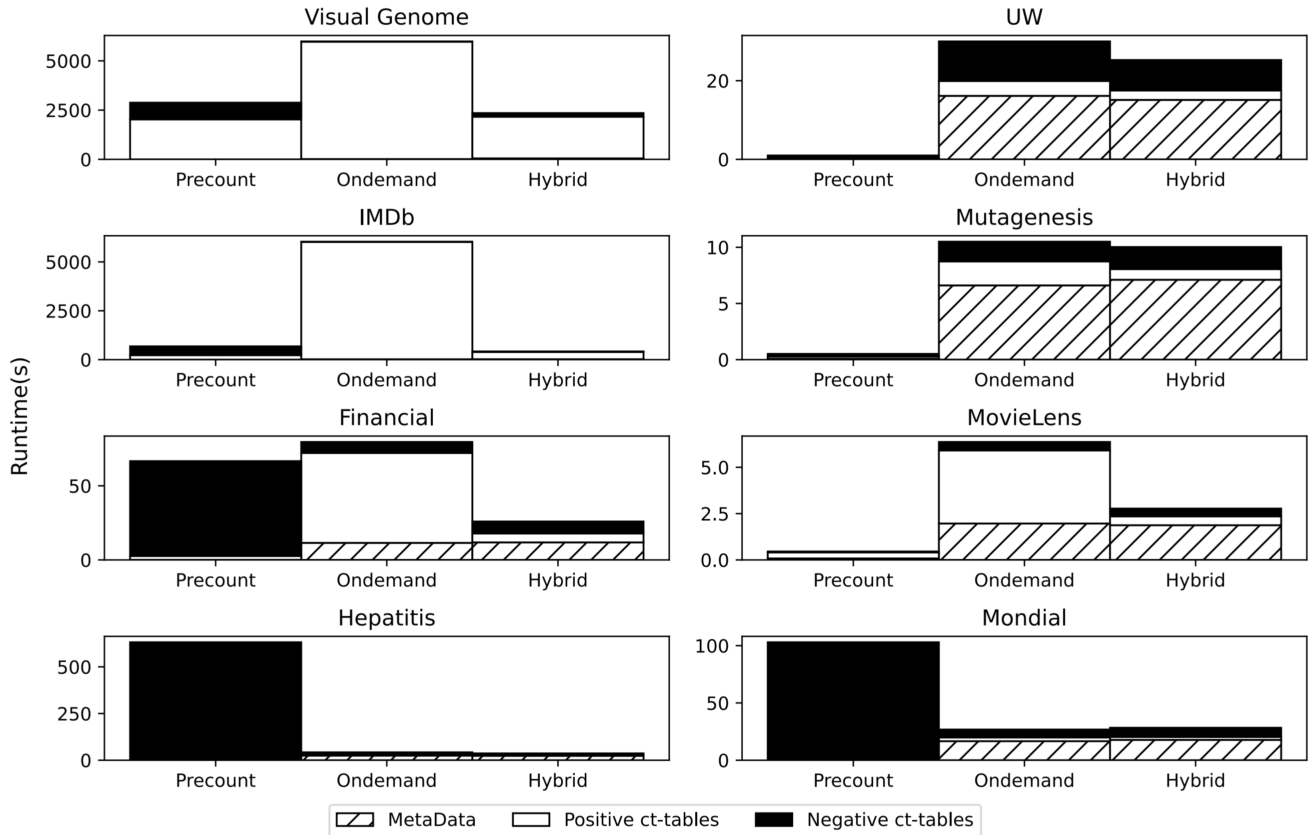

Figure 2: Comparison of the time it takes for ct-tables to be constructed for PRECOUNT, ONDEMAND, and HYBRID. For IMDb and Visual Genome, ONDEMAND failed to complete within the allotted time so only partial results have been included for those datasets in this figure.

and generation of the metaqueries, which are used to construct the dynamic SQL query statements for performing the necessary database queries.

Positive ct-table This component consists of the ct_+ counting information, which is made of two types of counting queries. One type of counting query generates the counts for entity tables where a simple query with a GROUP BY involving no JOINs is used. The other type of counting query is for generating the positive ct-tables that require JOINs.

Negative ct-table This component consists of the Möbius Join so it is made of the time it takes to generate the counts for all combinations of negative relationships.

Results

Figure 2 shows the cumulative time it takes to construct all the ct-tables for a given database. The times are broken down into the 3 components of MetaData, Positive ct-table (ct_+), and Negative ct-table (ct_-). For ONDEMAND, the execution of FACTORBASE exceeded the allotted runtime of 100 minutes when supplied IMDb and Visual Genome as input. Hence their results are omitted, or are incomplete if included in a figure. As expected, due to the large size of the ct-tables being generated, PRECOUNT spends a lot of time generating the negative ct-table compared to the other two methods. Overall ONDEMAND performs poorly in comparison to PRECOUNT, mostly due to the time it takes to generate the local positive ct-tables, especially for the large databases like IMDb and Visual Genome. HYBRID on the other hand performs better than PRECOUNT on several of the databases so we examine why next.

Negative ct-tables This is due to the reduced negative ct-table construction time because HYBRID generates smaller ct-tables (see Equations 2 and 4).

Some databases are exceptions to this trend (UW, Mutagenesis and MovieLens), where the Negative ct-table component for HYBRID takes longer than PRECOUNT. Here the local ct-tables are similar in size to the global one, negating the benefits of small ct-tables where their construction overhead becomes an issue. For further evidence, Table 5 shows that for the databases where HYBRID does worse than PRECOUNT, the total number of rows for the ct-tables generated when using HYBRID exceeds the number of rows for the global ct-table generated when using PRECOUNT.

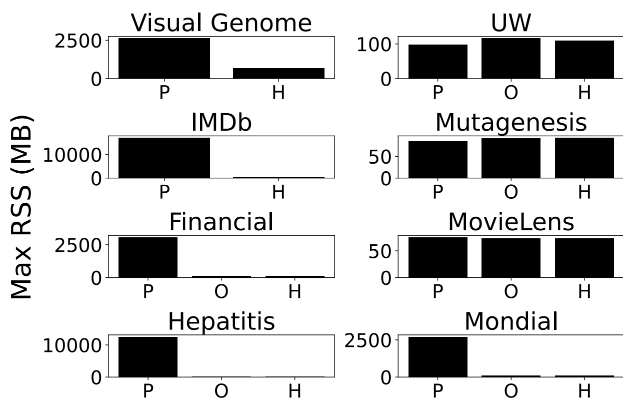

Figure 3: Comparison of the maximum resident set size used by the Java portion of FACTORBASE when constructing ct-tables for PRECOUNT (P), ONDEMAND (O), and HYBRID (H). Results have been omitted for ONDEMAND in the cases where it failed to complete within the allotted time.

Positive ct-tables The positive ct-tables for each family scored by HYBRID are generated by *projecting the information from the full positive ct-tables* for the lattice points. Therefore, scoring a family does not require further table JOINs. Although HYBRID performs better than PRECOUNT for most of the databases experimented with, there are a few cases where PRECOUNT does better. This is mainly a result of the MetaData component for HYBRID taking substantially longer due to overhead costs, and also due to the Negative ct-table component.

MetaData Figure 2 reveals that HYBRID inherited ONDEMAND's increased overhead to generate the metadata, which is shown to be negligible for PRECOUNT. Although the metadata overhead is a bottleneck only for ONDEMAND and HYBRID, it does not hurt scalability: The Metadata component is prominent for small databases like Mondial and UW, but essentially disappears for large databases like IMDb and Visual Genome.

Memory Profiling

Figure 3 compares the peak memory used when processing the various databases. It shows that *PRECOUNT is generally more memory intensive than ONDEMAND and HYBRID*. However, there are 3 databases where PRECOUNT uses similar amounts of memory. These 3 databases correspond to the ones in Figure 2 where PRECOUNT performs the best for the Negative ct-table component. In addition, from Table 5, these 3 databases are the cases where the total size of the ct-tables generated using ONDEMAND and HYBRID exceed the size of the global ct-table generated by PRECOUNT for the entire database. This observation suggests that the reason why PRECOUNT does better on these databases is that they are all generating negative ct-tables similar in size. Recall by Equation 2 the time to generate ct-tables depends on the size of the tables being generated. Also recall that ONDEMAND and HYBRID perform more operations to construct ct-tables than PRECOUNT, which explains their greater overhead for generating the negative ct-tables.

Limitations and Future Work

We discuss parts of the count-cache design space that we have not covered in our experiments.

Pre-Count Variants. Instead of storing instance counts for complete patterns, an intermediate approach is to store additional local information before model search that facilitates post-counting. An example is tuple ID propagation (Yin et al. 2004). The general idea of tuple ID propagation is that instead of performing an expensive JOIN operation, for each node and relationship type, we store the set of linked node IDs. A table JOIN is then replaced by ID propagation. Compared to the lattice pre-count approach, tuple ID propagation scales well in the number of data columns (predicates) but less well in the number of nodes (rows). Tuple ID propagation may be suitable for expanding the search space to include clauses with individuals since it expands the data with individual information.

Approximate Counting may make a purely post-counting method feasible even for large datasets. Our experiments suggest that it would have to be orders of magnitude more efficient than SQL JOINs for exact counts to make ONDEMAND competitive with PRECOUNT.

Conclusion

The key computational burden in learning SRL models from multi-relational data is obtaining the pattern instantiation counts that summarize the sufficient statistics (frequencies of conjunctions of relevant predicate values). Our research has investigated two approaches to caching relational counts, before and during SRL model search. Each approach has different implications for two counting challenges fundamental to relational learning: the JOIN problem (counting with k-tuples) and the negation problem (counting with non-existing relationships). We find that for many databases, the most scalable solution is a hybrid approach, which uses pre-counting to address the JOIN problem, and post-counting for the negation problem.

References

Das, M.; Dhami, D. S.; Kunapuli, G.; Kersting, K.; and Natarajan, S. 2019. Fast relational probabilistic inference and learning: Approximate counting via hypergraphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 7816–7824.

Friedman, N.; Getoor, L.; Koller, D.; and Pfeffer, A. 1999. Learning probabilistic relational models. In *IJCAI*, 1300–1309. Springer-Verlag.

Getoor, L.; and Taskar, B. 2007. *Introduction to Statistical Relational Learning*. MIT Press.

Kersting, K.; and De Raedt, L. 2007. Bayesian Logic Programming: Theory and Tool. In (Getoor and Taskar 2007), chapter 10, 291–318.

Khosravi, H.; Schulte, O.; Hu, J.; and Gao, T. 2012. Learning Compact Markov Logic Networks With Decision Trees. *Machine Learning* 89(3): 257–277.

Khosravi, H.; Schulte, O.; Man, T.; Xu, X.; and Bina, B. 2010. Structure Learning for Markov Logic Networks with Many Descriptive Attributes. In *AAAI*, 487–493.

Kimmig, A.; Mihalkova, L.; and Getoor, L. 2014. Lifted graphical models: a survey. *Machine Learning* 1–45.

Lv, Q.; Xia, X.; and Qian, P. 2012. A fast calculation of metric scores for learning Bayesian network. *International Journal of Automation and Computing* 9: 37–44. URL http://dx.doi.org/10.1007/s11633-012-0614-8.

Natarajan, S.; Khot, T.; Kersting, K.; Gutmann, B.; and Shavlik, J. W. 2012. Gradient-based boosting for statistical relational learning: The relational dependency network case. *Machine Learning* 86(1): 25–56.

Nickel, M.; Murphy, K.; Tresp, V.; and Gabrilovich, E. 2016. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE* 104(1): 11–33.

Niu, F.; Ré, C.; Doan, A.; and Shavlik, J. W. 2011. Tuffy: Scaling up Statistical Inference in Markov Logic Networks using an RDBMS. *PVLDB* 4(6): 373–384.

Qian, Z.; Schulte, O.; and Sun, Y. 2014. Computing Multi-Relational Sufficient Statistics for Large Databases. In Li, J.; Wang, X. S.; Garofalakis, M. N.; Soboroff, I.; Suel, T.; and Wang, M., eds., *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*, 1249–1258. ACM. doi:10.1145/2661829.2662010. URL https://doi.org/10.1145/2661829.2662010.

Ravkic, I.; Ramon, J.; and Davis, J. 2015. Learning relational dependency networks in hybrid domains. *Machine Learning* 100(2): 217–254.

Schulte, O.; and Gholami, S. 2017. Locally Consistent Bayesian Network Scores for Multi-Relational Data. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2693–2700. URL \url{https://www.ijcai.org/proceedings/2017/0375.pdf}.

Schulte, O.; and Khosravi, H. 2012. Learning graphical models for relational data via lattice search. *Mach. Learn.* 88(3): 331–368. doi:10.1007/s10994-012-5289-4. URL https://doi.org/10.1007/s10994-012-5289-4.

Schulte, O.; and Qian, Z. 2019. FACTORBASE: multi-relational structure learning with SQL all the way. *Int. J. Data Sci. Anal.* 7(4): 289–309. doi:10.1007/s41060-018-0130-1. URL https://doi.org/10.1007/s41060-018-0130-1.

Schulte, O.; Qian, Z.; Kirkpatrick, A. E.; Yin, X.; and Sun, Y. 2016. Fast learning of relational dependency networks. *Machine Learning* 1–30. ISSN 1573-0565. doi:10.1007/s10994-016-5557-9. URL http://dx.doi.org/10.1007/s10994-016-5557-9.

The Tetrad Group. 2008. The Tetrad Project. Http://www.phil.cmu.edu/projects/tetrad/.

Tsamardinos, I.; Brown, L. E.; and Aliferis, C. F. 2006. The Max-Min Hill-Climbing Bayesian Network Structure Learning Algorithm. *Machine Learning* 65(1): 31–78.

Yin, X.; Han, J.; Yang, J.; and Yu, P. S. 2004. CrossMine: Efficient Classification Across Multiple Database Relations. In *ICDE*.