# Computing Relational Sufficient Statistics for Large Databases

Zhensong Qian, Oliver Schulte, Yan Sun
Simon Fraser University, Canada
{zqian, oschulte, sunyans}@sfu.ca

## ABSTRACT

Fast access to sufficient statistics is a key factor for scalable data mining and machine learning. We present an efficient dynamic programming algorithm for computing relational sufficient statistics that combine information from different database tables. The statistics represent instantiation counts for conjunctive queries that specify the values of attributes, as well as the presence of any number of relationships, and the absence of any number of relationships. The statistics are stored in contingency tables. We introduce contingency table algebra, an extension of relational algebra, to compactly describe and efficiently implement the dynamic program. An empirical evaluation on seven benchmark databases demonstrates the scalability of our algorithm. As an application, we show how the precomputed sufficient statistics support learning a Bayes net model for the entire input database. The scalability of Bayes net structure learning depends mainly on fast access to sufficient statistics; our algorithm is the first to accomplish structure learning for databases with over 1 million records.

## 1. INTRODUCTION

In many applications of machine learning to big datasets with discrete variables, the dominant computational cost is due to counting how many times a pattern is instantiated in the data. An especially common pattern is a conjuntive query. These counts are referred to as *sufficient statistics* because they summarize the information in the data that a machine learning algorithm requires [17]. One of the most effective approaches to gathering sufficient statistics is to precompute a cache before executing the learning algorithm. This paper describes the first precounting approach for large *relational* datasets with sufficient statistics for queries that combine information from different database tables, and that may contain any number of negative relationships. Negative relationships concern the nonexistence of a relationship. Such statistics are important for learning correlations between different relationship types (e.g., if user $u$ performs a web search for item $i$, is it likely that $u$ watches a video about $i$ ?).

The precounting approach offers several advantages over on-demand counting. (1) Compute once, use often: Each sufficient statistic is computed only once, then looked up by the learning algorithm as required. Different learning algorithms for different tasks can access the same cache. (2) Dynamic programming: Computing a complete set of sufficient statistics at the same time facilitates the use of dynamic programming where counts for more complex queries are built up from results for simpler queries.

We apply precounting to learning a Bayes net structure for an entire relational database, including cross-table dependencies. Precounting makes it feasible to learn a Bayes net for tables with over 1 million records (drawn from the IMDB database). This is an order of magnitude larger than the databases for which graphical models have been learned previously [25, 13].

*The Materialization Challenge for Relational Learning.* In a single data table, sufficient statistics count how many table rows instantiate a query. A relational database contains multiple tables interrelated by foreign key constraints. For relational learning to discover cross-table correlations, it must compute counts for cross-table queries whose instantiations do *not* define a subset of rows in any of the input tables. Instead, they define a subset of rows in a *cross-product* of existing database tables. In principle, cross-table query counts can be computed by constructing or *materializing* the cross product and then applying single-table computations. In practice this is often infeasible because the size of the cross product grows exponentially. We refer to this fact as the *materialization challenge*.

*Virtual Join Approach.* An algorithm for computing sufficient counts without materializing a cross product is called a Virtual Join [34]. We present the first Virtual Join algorithm that finds sufficient statistics that involve negative relationships. A materialization approach for negative relationships does not scale because the unconstrained cross product of two or more different entity sets is far too large.

Sufficient statistics can be represented in contingency tables [17]. We introduce an extension of relational algebra with operations on contingency tables that generalize standard relational algebra operators. We establish a contingency table algebraic identity that reduces the computation of sufficient statistics with $k+1$ negative relationships to the computation of sufficient statistics with only $k$ negative relationships. A dynamic programming algorithm applies the identity to construct contingency tables that involve

$1, 2, \ldots, \ell$ relationships (positive and negative), until we obtain a joint contingency table for all tables in the database.

*Evaluation.* We evaluate our Virtual Join algorithm by computing joint contingency tables for 7 real-world databases. The computation times range from 2 seconds on the simpler database schemas to just over 2 hours for the most complex schema with over 1 million tuples from the IMDB database. Once the sufficient statistics have been computed, learning a Bayes net model that represents correlations across the entire database is relatively fast, and takes less time than computing the sufficient statistics.

*Contributions.* Our main contributions are as follows.

1. A dynamic program to compute a joint contingency table for sufficient statistics that combine several tables, and that may involve any number of positive and negative relationships.
2. An extension of relational algebra for contingency tables that supports the dynamic program conceptually and computationally.
3. An application to learning a Bayes net structure representing cross-table dependencies across the entire database.

*Paper Organization.* We review background for relational databases and statistical concepts. One of the inputs to the Virtual Join algorithm is a set of random variables for which sufficient statistics are required. We discuss how a default set of random variables can be generated using the schema information in the system catalog. The main part of the paper describes the dynamic programming algorithm for computing a joint contingency table for all random variables. We describe the contingency table algebra for dealing with negative relationships. A complexity analysis establishes feasible upper bounds on the number of contingency table operations required by the Virtual Join algorithm. We also investigate the scalability of the algorithm empirically. The final set of experiments examines how the cached sufficient statistics support learning a Bayes net model of cross-table dependencies.

# 2. BACKGROUND AND NOTATION

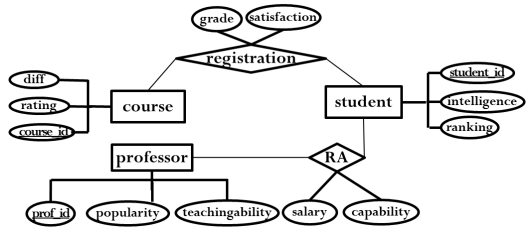


**Figure 1: A relational ER Design for a university domain.**

We assume a standard **relational schema** containing a set of tables, each with key fields, descriptive attributes, and possibly foreign key pointers. A **database instance** specifies the tuples contained in the tables of a given database schema. We assume that tables in the relational schema can be divided into *entity tables* and *relationship tables.* This is the case whenever a relational schema is derived from an entity-relationship model (ER model) [31, Ch.2.2].

## 2.1 Relational Random Variables

We adopt function-based notation from logic for combining statistical and relational concepts [23, 7]. A domain or **population** is a set of individuals. Individuals are denoted by lower case expressions (e.g., *bob*). A **functor** represents a mapping $f : \mathcal{P}_1, \ldots, \mathcal{P}_a \rightarrow V_f$ where $f$ is the name of the functor, each $\mathcal{P}_i$ is a population, and $V_f$ is the output type or **range** of the functor. In this paper we consider only functors with a finite range, disjoint from all populations. If $V_f = \{\mathrm{T}, \mathrm{F}\}$, the functor $f$ is a (Boolean) **predicate**. A predicate with more than one argument is called a **relationship**; other functors are called **attributes**. We use uppercase for predicates and lowercase for other functors.

A **Parametrized random variable** (PRV) is of the form $f(\mathbb{X}_1, \ldots, \mathbb{X}_a)$, where each $\mathbb{X}_i$ is a first-order variable [22]. Each first-order variable is associated with a population/type.

**Student**

| s_id | intelligence | ranking |
|---|---|---|
| jack | 3 | 1 |
| kim | 2 | 1 |
| paul | 1 | 2 |

(a)

**Course**

| c_id | rating | difficulty |
|---|---|---|
| 101 | 3 | 2 |
| 102 | 2 | 1 |
| 103 | 2 | 1 |

(b)

**Professor**

| p_id | popularity | teachingability |
|---|---|---|
| jim | 2 | 1 |
| oliver | 3 | 1 |
| david | 2 | 2 |

(c)

**RA**

| s_id | p_id | salary | capability |
|---|---|---|---|
| jack | oliver | High | 3 |
| kim | oliver | Low | 1 |
| paul | jim | Med | 2 |
| kim | david | High | 2 |

(d)

**Registration**

| s_id | c_id | grade | satisfaction |
|---|---|---|---|
| jack | 101 | 1 | 1 |
| jack | 102 | 2 | 2 |
| kim | 102 | 3 | 1 |
| paul | 101 | 2 | 1 |

(e)

**Figure 2: Database Table Instances based on university schema.**

## 2.2 Contingency Tables

Sufficient statistics can be represented in *contingency tables* as follows [17]. Consider a fixed list of random variables. A **query** is a set of ($variable = value$) pairs where each value is of a valid type for the random variable. The **result set** of a query in a database $\mathcal{D}$ is the set of instantiations of the first-order variables such that the query evaluates as true in $\mathcal{D}$. For example, in the database of Figure 2 the result set for the query ($intelligence(\mathbb{S}) = 2$, $rank(\mathbb{S}) = 1$, $popularity(\mathbb{P}) = 3$, $teachingability(\mathbb{P}) = 1$, $RA(\mathbb{P}, \mathbb{S}) = T$) is the singleton $\{\langle kim, oliver\rangle\}$. The **count** of a query is the cardinality of its result set.

For every set of variables $\mathbf{V} = \{V_1, \ldots, V_n\}$ there is a **contingency table** $ct(\mathbf{V})$. This is a table with a row for each of the possible assignments of values to the variables in $\mathbf{V}$, and a special integer column called *count*. The value of the *count* column in a row corresponding to $V_1 = v_1, \ldots, V_n = v_n$ records the count of the corresponding query. Figure 3 shows the contingency table for the university database. The value of a relationship attribute is undefined for entities that are not related. Following [16], we indicate this by writing $capability(\mathbb{P}, \mathbb{S}) = n/a$ for a reserved constant $n/a$. The assertion $capability(\mathbb{P}, \mathbb{S}) = \mathrm{n/a}$ is therefore equivalent to the assertion that $RA(\mathbb{P}, \mathbb{S}) = \mathrm{F}$. A **conditional contingency table**, written

$$ct(V_1, \ldots, V_k | V_{k+1} = v_{k+1}, \ldots, V_{k+m} = v_{k+m})$$

is the contingency table whose column headers are $V_1, \ldots, V_k$ and whose counts are defined by subset of instantiations that match the condition to the right of the | symbol. For

| Count | Diff. | Rat. | Pop. | Teach. | Intel. | Rank. | Cap. | Sal. | Grade | Sat. | RA | Reg. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 2 | 3 | 1 | 3 | High | 1 | 1 | T | T |
| 1 | 1 | 2 | 1 | 2 | 2 | 2 | n/a | n/a | 2 | 2 | F | T |
| 3 | 1 | 2 | 1 | 2 | 2 | 2 | 1 | Med | n/a | n/a | T | F |
| 24 | 2 | 1 | 1 | 2 | 1 | 5 | n/a | n/a | n/a | n/a | F | F |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 1 | 2 | 1 | 2 | 2 | 1 | 4 | n/a | n/a | 3 | 2 | F | T |

**Figure 3: Excerpt from the joint contingency table for the university database of Figure 2. Each row shows a query and its instantiation count in the database.**

compactness, our algorithms work with contingency tables that omit rows with count 0.

## 2.3 Bayes Nets for Relational Data

Poole introduced the Parametrized Bayes net (PBN) formalism that combines Bayes nets with logical syntax for expressing relational concepts [22]. A **Bayes Net (BN)** is a directed acyclic graph (DAG) whose nodes comprise a set of random variables and conditional probability parameters. For each assignment of values to the nodes, the joint probability is specified by the product of the conditional probabilities, $P(child|parent\_values)$. A **Parametrized Bayes Net** (PBN) is a Bayes net whose nodes are Parametrized random variables [22]. Since in our machine learning application, PRVs appears in Bayes nets, we often refer to PRVs simply as **nodes**. Figure 4 shows a PBN for the University database.

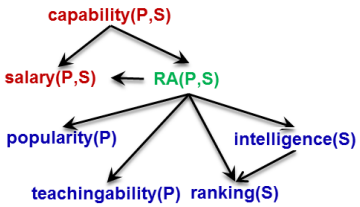


**Figure 4: A sample Parametrized Bayes Net based on the university schema.**

# 3. SPECIFICATION OF PARAMETRIZED RANDOM VARIABLES

The Virtual Join algorithm takes as input a relational database and a set of PRVs or nodes and returns a contingency table for the set. In this section we discuss how to represent a set of PRVs.

## 3.1 The Random Variable Database

The information about the random variables is stored in the **random variable database**. The Virtual Join algorithm distinguishes three types of PRVs: 1Nodes, RNodes, and 2Nodes. For each attribute of an entity, there is a corresponding attribute node called a **1Node** for short. The possible values for a 1Node are the same as for the corresponding attribute. There is a Boolean random variable for each relationship set, called a **relationship node** or **RNode** for short. Throughout this paper we assume that all relationships are binary, though this is not essential for our algorithm. For each attribute of an relationship, there is a corresponding attribute node called a **2Node** for short. The possible values for a 2Node are the same as for the corresponding attribute. Table 1 illustrates the logical functor notation and its relationship to the database ER diagram.

| ER Design | Type | Functor | PRV |
|---|---|---|---|
| Relation Tables | RNodes | RA | RA(P,S) |
| Entity Attributes | 1Nodes | intelligence, ranking | {intelligence(S), ranking(S)} =1Nodes(S) |
| Relationship Attributes | 2Nodes | teachingability, salary | {teachingability(P,S), salary(P,S)} = 2Nodes(RA(P,S)) |

**Table 1: Translation from ER Diagram to Parametrized Random Variables. Population variables are derived from entity tables as described in the text.**

The definitions of the three types of PRVs are listed in three tables in the Random Variable Database. This database also provides information about the random variables that is required for learning, such as their domains and the sizes of their domains. We refer to this information about the random variables as **metainformation**.

## 3.2 Generating Default PRVs

Users may lack the expertise for translating domain knowledge into the functor representation and metainformation. An alternative is to exploit the database schema catalog to generate default random variable database. We use this default in the experiments reported below. Because the system metadata is always available for any relational database, by exploiting it, default random variables can be generated in a completely general way. This means that the Random Variable Database can be set up with no information from the user beyond the SQL system catalog. However, a user or an application can change the random variable metainformation, by editing the tables in the Random Variable Database.

Given an ER-diagram, it is straightforward to generate functors as illustrated in Table 1. Given a database information schema, we can compute an implicit ER diagram by reversing the standard translation from ER-diagrams to SQL [31]. For instance, if a table contains a single primary key column and no foreign key pointers, we treat it as representing an entity set and associate a first-order variable with it. Since the database information schema is itself stored in table form, the default random variable database can be created using an SQL script. We omit further details due to space constraints.

The main complication arises when the database contains a self-relationship [10] that relates two entities of the same type. If the schema contains a self-relationship, we add one first-order variable for each role in the relationship, and one set of 1Nodes for each first-order variable. For example, the Mondial database contains a self-relationship *Borders* that relates two countries. The corresponding relationship node is $Borders(\mathbb{C}_1, \mathbb{C}_2)$. The different positions of the first-order variables can represent different roles in the self-relationship. In the Mondial example, there are two 1Nodes $continent(\mathbb{C}_1)$ and $continent(\mathbb{C}_2)$.

# 4. RELATIONAL CONTINGENCY TABLES

Many relational learning algorithms take an iterative deepening approach: explore correlations along a single relationship, then along relationship chains of length 2, 3, etc. Chains of relationships form a natural lattice structure, where iterative deepening corresponds to moving from the bottom to the top. The Virtual Join algorithm computes contingency tables by using the results for smaller relationships for larger relationship chains, which leads to an efficient dynamic program. Another benefit of level-wise lattice learning is that if the learning algorithm requires sufficient statistics only up to a certain depth in the lattice, the Virtual Join algorithm can be terminated at this depth.

A relationship set is a **chain** if it can be ordered as a list $[R_1(\boldsymbol{\tau}_1), \ldots, R_k(\boldsymbol{\tau}_k)]$ such that each functor $R_{i+1}(\boldsymbol{\tau}_{i+1})$ shares at least one first-order variable with the preceding terms $R_1(\boldsymbol{\tau}_1), \ldots, R_i(\boldsymbol{\tau}_i)$. All sets in the lattice are constrained to form a chain. For instance, in the University schema of Figure 1, a chain is formed by the two relationship nodes

$$\mathit{Registration}(\mathbb{S}, \mathbb{C}), \mathit{RA}(\mathbb{P}, \mathbb{S}).$$

If relationship node $\mathit{Teaches}(\mathbb{P}, \mathbb{C})$ is added, we may have a three-element chain

$$\mathit{Registration}(\mathbb{S}, \mathbb{C}), \mathit{RA}(\mathbb{P}, \mathbb{S}), \mathit{Teaches}(\mathbb{P}, \mathbb{C}).$$

The subset relation defines a lattice on relationship sets/chains. Figure 5 illustrates the lattice for the relationship nodes in the university schema. For reasons that we explain below, entity tables are also included in the lattice and linked to relationships that involve the entity in question.

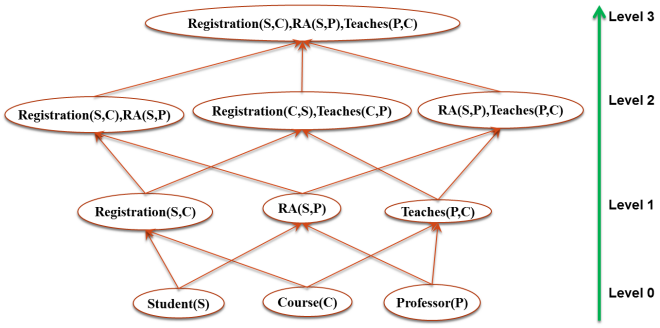


**Figure 5: A lattice of relationship sets for the university schema of Figure 1. Links from entity tables to relationship tables correspond to foreign key pointers.**

With each relationship chain $\mathbf{R}$ is associated a *ct*-table $ct_{\mathbf{R}}$. For a relationship chain $\mathbf{R}$ the nodes in the *ct*-table $ct_{\mathbf{R}}$ comprise: the Rnodes in the set $\mathbf{R}$, the 1Nodes and 2Nodes associated with each of the Rnodes. To describe these, we introduce the following notation.

- $\mathit{1Nodes}(R)$ denotes the set of entity attribute nodes for the first-order variables involved in the relationship $R$.
- $\mathit{1Nodes}(\mathbf{R})$ is the union of the entity attributes for each relationship $R \in \mathbf{R}$.
- $\mathit{2Nodes}(R)$ denotes the set of relationship attribute nodes for the relationship $R$.
- $\mathit{2Nodes}(\mathbf{R})$ is the union of the relationship attributes for each relationship $R \in \mathbf{R}$.
- $1\mathit{Nodes}(\mathbb{A})$ denotes the attributes of a first-order variable $\mathbb{A}$ collectively.

In this notation, the nodes in the *ct*-table $ct_{\mathbf{R}}$ are denoted as $\mathbf{R} \cup \mathit{ANodes}(\mathbf{R})$.

The goal of the Virtual Join algorithm is to compute a contingency table for each point in the lattice. In the example of Figure 5, the algorithm computes 10 contingency tables. The *ct*-table for the top element of the lattice is the **joint** *ct*-**table** for the entire database. It is useful to distinguish two types of rows within each *ct*-table: rows where all the relationship nodes are assigned the value T—positive relationships only—and rows where one or more relationship nodes are assigned the value F—some negative relationships. We begin with the case of positive relationships only.

# 5. COMPUTING CONTINGENCY TABLES FOR POSITIVE RELATIONSHIPS

If a conjunctive query involves only positive relationships, then it can be computed using SQL's count aggregate function applied to a table join. The SQL count query represents the positive part of the *ct*-table much like a view definition represents a relation. To construct the SQL count query without advance knowledge of the database schema, we utilize the metainformation in a **metaquery**.

An SQL metaquery is an SQL query that takes as input schema information as recorded in the random variable database, and produces four kinds of tables: the Select, From, Where and Group By tables. The Select table lists the entries in the Select clause of the target query, the From table lists the entries in the From clause, and similar for Where and GROUP BY tables. Given the four query tables, the corresponding SQL query can be easily executed in an application or stored procedure to produce the *ct*-table entries. Figure 6 shows an example of metaqueries for the university database. The metaquery accesses tables in the Random Variable Database *RV*. The entries of the Group By table (not shown) are the same as the Select table without the *count* column. The @*database*@ is a string placeholder for the input database name. We omit further details due to space constraints.

| Metaqueries | Entries for rnid = RA(S,P) |
|---|---|
| CREATE TABLE Select_List AS<br>SELECT rnid, CONCAT('COUNT(*)',' as "count"') AS Entries<br>FROM RV.RNodes<br>UNION DISTINCT<br>SELECT<br>rnid, CONCAT(pvid,'.',COLUMN_NAME,' AS ',1nid) AS Entries<br>FROM RV.RNodes_1Nodes | COUNT(*) as "count"<br>prof0.popularity AS `popularity(prof0)`<br>prof0.teachingability AS `teachingability(prof0)`<br>student0.intelligence AS `intelligence(student0)`<br>student0.ranking AS `ranking(student0)` |
| CREATE TABLE From_List AS<br>SELECT<br>rnid, CONCAT('@database@.',TABLE_NAME) AS Entries<br>FROM RV.RNodes_pvars<br>UNION DISTINCT<br>SELECT<br>rnid, CONCAT('@database@.',TABLE_NAME) AS Entries<br>FROM RV.RNodes | @database@.prof AS prof0<br>@database@.student AS student0<br>@database@.RA AS `RA` |
| CREATE TABLE Where_List AS<br>SELECT<br>rnid, CONCAT(rnid,'.',COLUMN_NAME,' = ',<br>pvid,'.',REFERENCED_COLUMN_NAME) AS Entries<br>FROM RV.RNodes_pvars; | `RA`.prof_id = prof0.prof_id<br>`RA`.student_id =student0.student_id |

**Figure 6: Example of metaqueries and metaquery results for the relationship table *RA* based on university database.**

# 6. COMPUTING CONTINGENCY TABLES FOR NEGATIVE RELATIONSHIPS

Computing sufficient statistics that involve negative relationships is infeasible using standard table joins. Standard join tables materialize new data tables that enumerate tuples of objects that are *not* related. However, the number of unrelated tuples is too large to make this scalable (consider the number of user pairs who are *not* friends on Facebook). We describe a Virtual Join algorithm that computes the required data tables without the quadratic cost of materializing a cross-product. Our experiments below compare the virtual joins with materializing table joins. First, we introduce an extension of relational algebra that we term **contingency table algebra**. The purpose of this extension is to show that query counts using $k+1$ negative relationships can be computed from two query counts that each involve at most $k$ relationships. Second, a dynamic programming algorithm applies the algebraic identify repeatedly to build up a complete contingency table from partial tables.

## 6.1 Contingency Table Algebra

We introduce relational algebra style operations defined on contingency tables.

### 6.1.1 Unary Operators

**Selection** $\sigma_\phi ct$ selects a subset of the rows in the $ct$-table that satisfy condition $\phi$. This is the standard relational algebra operation except that the selection condition $\phi$ may not involve the *count* column.

**Projection** $\pi_{V_1,\ldots,V_k} ct$ selects a subset of the columns in the $ct$-table, excluding the count column. The counts in the projected subtable are the sum of counts of rows that satisfy the query in the subtable. The $ct$-table projection $\pi_{V_1,\ldots,V_k} ct$ can be defined by the following SQL code template:

SELECT SUM(COUNT) AS COUNT, $V_1, \ldots, V_k$
FROM $ct$
GROUP BY $V_1, \ldots, V_k$

**Conditioning** $\chi_\phi ct$ returns a conditional contingency table. Ordering the columns as $(V_1, \ldots, V_k, \ldots, V_{k+j})$, suppose that the selection condition is a conjunction of values of the form $C = (V_{k+1} = v_{k+1}, \ldots, V_{k+j} = v_{k+j})$. Conditioning can be defined in terms of selection and projection by the equation:

$$\chi_\phi ct = \pi_{V_1,\ldots,V_k}(\sigma_\phi ct)$$

### 6.1.2 Binary Operators

With some abuse of notation, we use $\mathbf{V}$, $\mathbf{U}$ in SQL templates to denote a list of column names in arbitrary order. The notation $ct_1.\mathbf{V} = ct_2.\mathbf{V}$ indicates an equijoin condition: the contingency tables $ct_1$ and $ct_2$ have the same column set $\mathbf{V}$ and matching columns from the different tables have the same values.

**Cross Product** The **cross-product** of $ct_1(\mathbf{U}), ct_2(\mathbf{V})$ is the Cartesian product of the rows, where the product counts are the products of count. The cross-product can be defined by the following pseudo-SQL code:

SELECT
$(ct_1.count * ct_2.count)$ AS *count*, $\mathbf{U}$, $\mathbf{V}$
FROM $ct_1, ct_2$

**Addition** The **count addition** $ct_1(\mathbf{V}) + ct_2(\mathbf{V})$ adds the counts of matching rows, as in the following SQL template.

SELECT $ct_1.count + ct_2.count$ AS *count*, $\mathbf{V}$
FROM $ct_1, ct_2$
WHERE $ct_1.\mathbf{V} = ct_2.\mathbf{V}$

If a row appears in one $ct$-table but not the other, we include the row with the count of the table that contains the row.

**Subtraction** The **count difference** $ct_1(\mathbf{V}) - ct_2(\mathbf{V})$ equals $ct_1(\mathbf{V}) + (-ct_2(\mathbf{V}))$ where $-ct_2(\mathbf{V})$ is the same as $ct_2(\mathbf{V})$ where the counts are negative. Table subtraction is defined only if (i) without the *count* column, the rows in $ct_1$ are a superset of those in $ct_2$, and (ii) for each row that appears in both tables, the count in $ct_1$ is at least as great as the count in $ct_2$.

### 6.1.3 Implementation

The selection operator can be implemented using SQL as with standard relational algebra. Projection with $ct$-tables requires use of the GROUP BY construct as shown in Section 6.1.1.

For addition/subtraction, simply executing the SQL query shown above may lead to a quadratic cost if a nested-loops join is used, and therefore does not scale to large datasets. A more efficient algorithm is a sort-merge join [31]. Given two union-compatible $ct$-tables, each row in one matches at most one other on the non-count columns. As is well known, with unique matches, the cost of a sort-merge join is $size(table1) + size(table2)+$ the cost of sorting both tables. Our experiments below are based on a sort-merge join (our own implementation in Java).

The cross product is easily implemented in SQL as shown in Section 6.1.2. The cross product size is quadratic in the size of the input tables, so a quadratic cost is unavoidable.

## 6.2 Lattice Computation of Contingency Tables

This section describes a method for computing the contingency tables level-wise in the relationship chain lattice. We start with a contingency table algebra equivalence that allows us to compute counts for rows with negative relationships from rows with positive relations. Following [17], we use a "don't care" value $*$ to indicate that a query does not specify the value of a node. For instance, the query $R_1 = \mathrm{T}, R_2 = *$ is equivalent to the query $R_1 = \mathrm{T}$.

PROPOSITION 1. *Let $R$ be a relationship node and let $\mathbf{R}$ be a set of relationship nodes. Let Nodes be a set of nodes that does not contain $R$ nor any of the 2Nodes of $R$. Let $\mathbb{X}_1, \ldots, \mathbb{X}_l$ be the first-order variables that appear in $R$ but not in Nodes, where $l$ is possibly zero. Then we have*

$$ct(Nodes \cup 1Nodes(R)|\mathbf{R} = \mathrm{T}, R = F) = \quad (1)$$
$$ct(Nodes|\mathbf{R} = \mathrm{T}, R = *) \times ct(\mathbb{X}_1) \times \cdots \times ct(\mathbb{X}_l)$$
$$- ct(Nodes \cup 1Nodes(R)|\mathbf{R} = \mathrm{T}, R = T).$$

*If $l = 0$, the equation holds without the cross-product term.*

Figure 7 illustrates using this equation (to build the $ct_{\mathrm{F}}$ table).

PROOF. The equation

$$ct(Nodes \cup 1Nodes(R)|\mathbf{R} = \mathrm{T}, R = *) = \quad (2)$$
$$ct(Nodes \cup 1Nodes(R)|\mathbf{R} = \mathrm{T}, R = T)+$$
$$ct(Nodes \cup 1Nodes(R)|\mathbf{R} = \mathrm{T}, R = F)$$

holds because the set $Nodes \cup 1Nodes(R)$ contains all first-order variables in $R$.[1]

Equation (2) implies

$$ct(Nodes \cup 1Nodes(R)|\mathbf{R} = \mathrm{T}, R = \mathrm{F}) = \quad (3)$$
$$ct(Nodes \cup 1Nodes(R)|\mathbf{R} = \mathrm{T}, R = *)-$$
$$ct(Nodes \cup 1Nodes(R)|\mathbf{R} = \mathrm{T}, R = \mathrm{T}).$$

To compute the $ct$-table conditional on the relationship $R$ being unspecified, we use the equation

$$ct(Nodes \cup 1Nodes(R)|\mathbf{R} = \mathrm{T}, R = *) = \quad (4)$$
$$ct(Nodes|\mathbf{R} = \mathrm{T}, R = *) \times ct(\mathbb{X}_1) \times \cdots \times ct(\mathbb{X}_l)$$

which holds because if the set $Nodes$ does not contain a first-order variable of $R$, then the counts of the associated $1Nodes(R)$ are independent of the counts for $Nodes$. If $l = 0$, there is no new first-order variable, and Equation (4) holds without the cross-product term. Together Equations (3) and (4) establish the proposition. □

**Algorithm 1:** The Pivot function returns a conditional contingency table for a set of attribute nodes and all possible values of the relationship $R_{pivot}$, including $R_{pivot} = \mathrm{F}$. The set of conditional relationships $\vec{R} = (R_{pivot}, \ldots, R_\ell)$ may be empty in which case the Pivot computes an unconditional ct-table.

**Input**: Two conditional contingency tables $ct_{\mathrm{T}} := ct(Nodes, 2Nodes(R_{pivot})|R_{pivot} = \mathrm{T}, \vec{R} = \mathrm{T})$ and $ct_* := ct(Nodes|R_{pivot} = *, \vec{R} = \mathrm{T})$ .
Precondition: The set $Nodes$ does not contain the relationship node $R_{pivot}$ nor any of its descriptive attributes $2Nodes(R_{pivot})$.;
**Output**: The conditional contingency table $ct(Nodes, 2Nodes(R_{pivot}), R_{pivot}|\vec{R} = \mathrm{T})$ .

1: $ct_{\mathrm{F}} := ct_* - \pi_{Nodes} ct_{\mathrm{T}}$.
{Implements the algebra Equation 1 in proposition 1.}
2: $ct^+_{\mathrm{F}}$ := extend $ct_{\mathrm{F}}$ with columns $R_{pivot}$ everywhere false and $2Nodes(R_{pivot})$ everywhere $n/a$.
3: $ct^+_{\mathrm{T}}$ := extend $ct_{\mathrm{T}}$ with columns $R_{pivot}$ everywhere true.
4: **return** $ct^+_{\mathrm{F}} \cup ct^+_{\mathrm{T}}$

Algorithm 1 provides pseudo-code for applying Equation (2) to compute a complete $ct$-table given a partial table where a specified relationship node $R$ is true and another partial table that does not contain the relationship node. We refer to $R$ as the **pivot** node. For extra generality, we apply Equation (2) with a condition that lists a set of relationship nodes fixed to be true. Algorithm 2 shows how the Pivot operation can be applied repeatedly to find all contingency tables in the relationship lattice. Figure 7 illustrates the computation for the case of only one relationship. The computation proceeds as follows.

*Initialization.* Compute $ct$-tables for entity tables. Compute $ct$-tables for each single relationship node $R$ , conditional on $R = \mathrm{T}$. If $R = *$, then no link is specified between the first-order variables involved in the relation $R$. Therefore the individual counts for each first-order variable are independent of each other and the joint counts can be obtained by the cross product operation. Apply the Pivot function to construct the complete $ct$-table for relationship node $R$.

*Lattice Computation.* The goal is to compute $ct$-tables for all relationship chains of length $> 1$. For each relationship chain, order the relationship nodes in the chain arbitrarily. Make each relationship node in order the Pivot node $R_i$. For the current Pivot node $R_i$, find the conditional $ct$-table where $R_i$ is unspecified, and the subsequent relations $R_j$ with $j > i$ are true. This $ct$-table can be computed from a $ct$-table for a shorter chain that has been constructed already. The conditional $ct$-table has been constructed already, where $R_i$ is true, and the subsequent relations are true (see loop invariant). Apply the Pivot function to construct the complete $ct$-table, for any value of the Pivot node $R_i$, conditional on the subsequent relations being true.

### 6.2.1 Complexity Analysis

The key point about the Virtual Join algorithm is that it avoids the materializing the cross product of entity tuples. *The algorithm accesses only **existing** tuples, never constructs nonexisting tuples.*

To analyze the computational cost, we examine the total number of $ct$-algebra operations performed by the Virtual Join algorithm. We provide upper bounds in terms of two parameters: the number of relationship nodes $m$, and the number of rows $r$ in the $ct$-table that is the output of the algorithm. For these parameters we establish that

$$\#ct\_ops = O(r \cdot \log_2 r) = O(m \cdot 2^m).$$

This shows the efficiency of our algorithm for the following reasons. (i) Since the time cost of any algorithm must be at least as great as the time for writing the output, which is as least as great as $r$, the Virtual Join algorithm adds at most a logarithmic factor to this lower bound. (ii) The second upper bound means that the number of $ct$-algebra operations is fixed-parameter tractable with respect to $m$.[2] In practice the number $m$ is on the order of the number of tables in the database, which is very small compared to the number of tuples in the tables.

*Derivation of Upper Bounds.* For a given relationship chain of length $\ell$, the Virtual Join algorithm goes through the chain linearly (Algorithm 2 inner for loop line 12). At each iteration, it computes a $ct_*$ table with a single cross product, then performs a single Pivot operation. Each Pivot operation requires three $ct$-algebra operations. Thus overall, the number of $ct$-algebra operations for a relationship chain of length $\ell$ is $6 \cdot \ell = O(\ell)$. For a fixed length $\ell$, there are at

[1]We assume here that for each first-order variable, there is at least one *1Node*, i.e., descriptive attribute.

[2]For arbitrary $m$, the problem of computing a $ct$ table in a relational structure is #P-complete [4, Prop.12.4].

**Algorithm 2:** Virtual Join algorithm for Computing the Contingency Table for Input Database

**Input**: A relational database $\mathcal{D}$; a set of functor nodes
**Output**: A contingency table that lists the count in the database $D$ for each possible assignment of values to each functor node.

**for all** first-order variables $\mathbb{X}$ **do**
  compute $ct(\mathit{1Nodes}(\mathbb{X}))$ using SQL queries.
**end for**
**for all** relation node $R$ **do**
  $ct_* := ct(\mathbb{X}) \times ct(\mathbb{Y})$ where $\mathbb{X}$,$\mathbb{Y}$ are the first-order variables in $R$.
  $ct_{\mathrm{T}} := ct(\mathit{1Nodes}(R)|R = \mathrm{T})$ using SQL joins.
  Call $\mathit{Pivot}(ct_{\mathrm{T}}, ct_*)$ to compute $ct(\mathit{1Nodes}(R), \mathit{2Nodes}(R), R)$.
**end for**
**for** Rchain length $\ell = 2$ to $m$ **do**
  **for all** Rchain $R_1, \ldots, R_\ell$ **do**
    $\mathit{Current\_ct} := ct(\mathit{1Nodes}(R_1, \ldots, R_\ell)\mathit{2Nodes}(R_1, \ldots, R_\ell)|R_1 = \mathrm{T}, \ldots, R_\ell = \mathrm{T})$ using SQL joins.
    **for** $i = 1$ to $\ell$ **do**
      **if** $i$ equals 1 **then**
        $ct_* := ct(\mathit{1Nodes}(R_2, \ldots, R_\ell), \mathit{2Nodes}(R_2, \ldots, R_\ell)|R_1 = *, R_2 = \mathrm{T}, \ldots, R_\ell = \mathrm{T}) \times ct(\mathbb{X})$ where $\mathbb{X}$ is the first-order variable in $R_1$, if any, that does not appear in $R_2, \ldots, R_\ell$ {$ct_*$ can be computed from a $ct$-table for a Rchain of length $\ell - 1$.}
      **else**
        $\mathit{1Nodes}_{\bar{i}} := \mathit{1Nodes}(R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_\ell)$.
        $\mathit{2Nodes}_{\bar{i}} := \mathit{2Nodes}(R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_\ell)$.
        $ct_* := ct(\mathit{1Nodes}_{\bar{i}}, \mathit{2Nodes}_{\bar{i}}, R_1, \ldots, R_{i-1})|R_i = *, R_{i+1} = \mathrm{T}, \ldots, R_\ell = \mathrm{T}) \times ct(\mathbb{Y})$ where $\mathbb{Y}$ is the first-order variable in $R_i$, if any, that does not appear in $\vec{R}$.
      **end if**
      $\mathit{Current\_ct} := \mathit{Pivot}(\mathit{Current\_ct}, ct_*)$.
    **end for**{Loop Invariant: After iteration $i$, the table $\mathit{Current\_ct}$ equals $ct(\mathit{1Nodes}(R_1, \ldots, R_\ell), \mathit{2Nodes}(R_1, \ldots, R_\ell), R_1, \ldots, R_i|R_{i+1} = \mathrm{T}, \ldots, R_\ell = \mathrm{T})$}
  **end for**{Loop Invariant: The $ct$-tables for all Rchains of length $\ell$ have been computed.}
**end for**
**return** the $ct$-table for the Rchain involves all the relationship nodes.

most $\binom{m}{\ell}$ relationship chains. Using the known identity[3]

$$\sum_{\ell=1}^{m} \binom{m}{\ell} \cdot \ell = m \cdot 2^{m-1} \tag{5}$$

we obtain the $O(m \cdot 2^{m-1}) = O(m \cdot 2^m)$ upper bound.

For the upper bound in terms of $ct$-table rows $r$, we note that the output $ct$-table can be decomposed into $2^m$ subtables, one for each assignment of values to the $m$ relationship nodes. Each of these subtables contains the same number of rows $d$ , one for each possible assignment of values to the attribute nodes. Thus the total number of rows is given by $r = d \cdot 2^m$. Therefore we have $m \cdot 2^m = \log_2(r/d) \cdot r/d \leq \log_2(r) \cdot r$. Thus the total number of $ct$-algebra operations is $O(r \cdot \log_2(r))$.

From this analysis we see that both upper bounds are overestimates. (1) Because relationship chains must be linked by foreign key constraints, the number of valid relationship chains of length $\ell$ is usually much smaller than the number of all possible subsets $\binom{m}{\ell}$. (2) The constant factor $d$ grows exponentially with the number of attribute nodes, so $\log_2(r) \cdot r$ is a loose upper bound on $\log_2(r/d) \cdot r/d$. We conclude that the number of $ct$-algebra operations is not the critical factor for scalability, but rather the cost of carrying out a single $ct$-algebra operation. In Section 6.1 we discussed how to perform these operations in a scalable manner.

[3]math.wikia.com/wiki/Binomial_coefficient, Equation 6a

# 7. APPLICATION: BAYES NET LEARNING

This section describes an application of our contingency table algorithm to a challenging problem in machine learning: constructing a Bayes net structure for a relational database. Bayes net learning has been considered as an application for caching sufficient statistics for single table data [17, 15]. For data exploration, a Bayes net model provides a succinct graphical representation of complex statistical-relational correlations. The model also supports probabilistic reasoning for answering "what-if" queries about the probabilities of uncertain outcomes conditional on observed events.

## 7.1 The Learn-and-Join Algorithm

This Learn-and-Join Algorithm (LAJ) takes as input a relational database and constructs a Bayes net for each relationship chain. The final Bayes net is the model associated with the largest chain. The LAJ algorithm was previously applied only conditional on all relationship nodes being true. Because our virtual join algorithm provides sufficient statistics about negative relationships, we can extend the LAJ algorithm for learning a Bayes net model of dependencies among relationship nodes. We provide a brief summary; for a detailed description please see [24]. The algorithm applies a standard single-table Bayes net learner, which can be chosen by the user, to the contingency table for each chain.

The algorithm proceeds level-wise by considering relationship chains of length $\ell = 0, 1, 2, \ldots$. After Bayes nets have been learned for length $\ell$, the learned edges are propagated

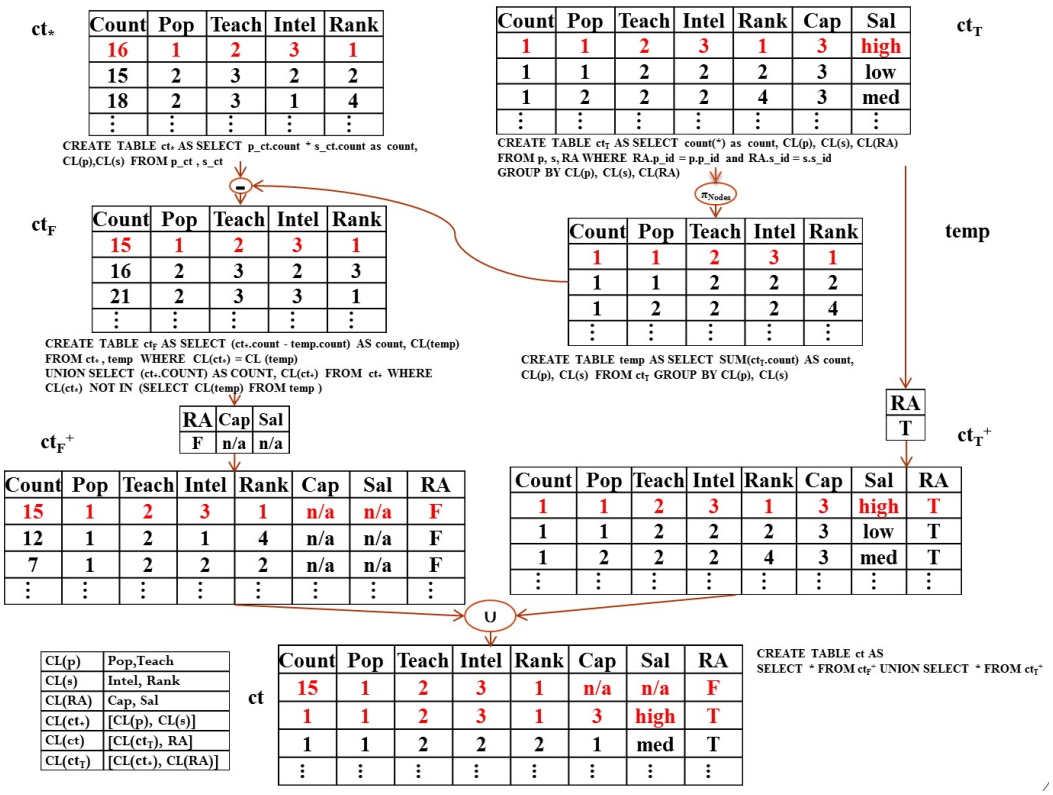


**Figure 7: Computation of the *ct*-table for the Rnode $RA(\mathbb{S},\mathbb{P})$ using Algorithm 1. The operations are implemented using dynamic SQL queries as shown. Lists of column names are abbreviated as shown and as follows. $CL(ct_*) = CL(temp) = CL(ct_F)$, $CL(ct) = CL(ct_F^+) = CL(ct_T^+)$.**

to chains of length $\ell + 1$. In the initial case of single relationship tables where $\ell = 0$, the edges are propagated from Bayes nets learned for entity tables. Propagated edge constraints are of two types: Required and forbidden edges.

Suppose that $\boldsymbol{R}$ at level $s$ is a parent of $\boldsymbol{R}'$ at level $s+1$. (A parent contains exactly one less relationship node, see Figure 5.) For required edges, if an edge $X \rightarrow X'$ is present in the Bayes net $\mathbb{B}_{\boldsymbol{R}}$ for the parent chain $\boldsymbol{R}$, the edge must also be contained in the Bayes net $\mathbb{B}_{\boldsymbol{R}'}$ for the child chain $\boldsymbol{R}'$. In other words, edges present in smaller relationship chains are inherited by larger relationship chains. For forbidden edges, if an edge $X \rightarrow X'$ is absent from the Bayes nets of all parent chains of $\boldsymbol{R}'$, it must also be absent from the Bayes net of $\boldsymbol{R}'$. In other words, the absence of edges in smaller relationship chains is inherited by larger relationship chains.

*Example.* If the *Student* Bayes net contains an edge

$$intelligence(\mathbb{S}) \rightarrow ranking(\mathbb{S}),$$

then the Bayes net associated with the relationship $RA(\mathbb{S},\mathbb{C})$ must also contain this edge (see Figure 2). If the *Professor* Bayes net does not contain an edge

$$popularity(\mathbb{P}) \rightarrow teachingability(\mathbb{P}),$$

it must also be absent in the Bayes net associated with the relationship node $RA(\mathbb{P},\mathbb{S})$.

## 7.2 Implementation in SQL

The Bayes multinet can be stored in a master table called *ChainBayesNet*, as shown in Figure 8. An entry such as

$$\langle [Registration(\mathbb{S},\mathbb{C}), RA(\mathbb{S},\mathbb{P})], ranking(\mathbb{S}), intelligence(\mathbb{S})\rangle$$

means that the Bayes net graph for the relationship chain $Registration(\mathbb{S},\mathbb{C}), RA(\mathbb{S},\mathbb{P})$ contains an edge $ranking(\mathbb{S}) \leftarrow intelligence(\mathbb{S})$. The Bayes net learner receives a *ct*-table and edge constraints as input, and returns a Bayes net. The learned edges are exported to the database tables. An SQL view defines tables that store the forbidden and required edges: each time that new edges are inserted in the *ChainBayesNet* table, the view mechanism propagates the results as constraints of learning to larger relationship chains. The view mechanism provides a compact and flexible update procedure (only 20 lines of SQL code). Figure 8 shows the computation flow for our Bayes net learning system. After structure learning is complete, the Bayes net parameters can be easily computed from the contingency tables using maximum likelihood estimation, and stored in conditional probability tables in the BN database together with the *ChainBayesNet* table.

# 8. EVALUATION

Code was written in Java, JRE 1.7.0. and executed with 8GB of RAM and a single Intel Core 2 QUAD Processor Q6700 with a clock speed of 2.66GHz (there is no hyperthreading on this chip). The operating system was Linux Centos 2.6.32. The MySQL Server version 5.5.34 was run

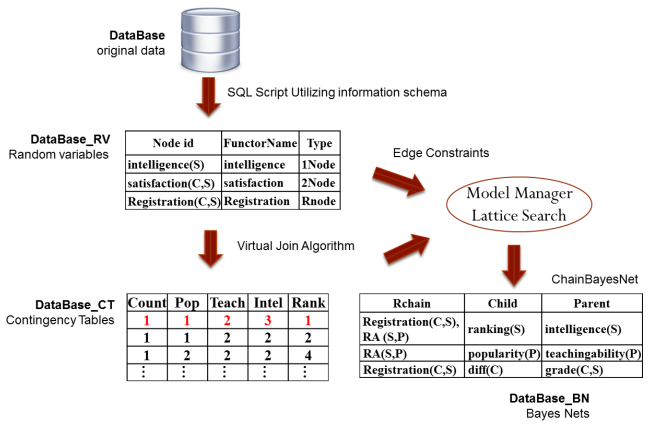


**Figure 8: Bayes net structure learning with SQL.**

with 8GB of RAM and a single core processor of 2.2GHz. All code and datasets are available [11]. We made use of the following single-table Bayes Net search implementation: GES search [2] with the BDeu score as implemented in version 4.3.9-0 of CMU's Tetrad package (structure prior uniform, ESS=10; [30]).

| Dataset | #Relationship Tables | #Self Relationships | #Tuples | #Attributes |
|---|---|---|---|---|
| Movielens | 1 | 0 | 1,010,051 | 7 |
| Mutagenesis | 2 | 0 | 14,540 | 11 |
| Financial | 3 | 0 | 225,932 | 15 |
| Hepatitis | 3 | 0 | 12,927 | 19 |
| IMDB | 3 | 0 | 1,354,134 | 17 |
| Mondial | 2 | **1** | 870 | 18 |
| UW-CSE | 2 | **2** | 712 | 14 |

**Table 2: Datasets characteristics. #Tuples = total number of tuples over all tables.**

*Datasets.* We describe the datasets in terms of their representation as databases with tables. The databases follow an ER design [31]. We used 7 benchmark real-world databases, with the modifications described by Schulte and Khosravi [25]. See that article for more details.

**MovieLens Database** A dataset from the GroupLens[4]. The data are organized in 3 tables (2 entity tables, 1 relationship table, and 7 descriptive attributes). The Rated table contains Rating as descriptive attribute; more than 1 million ratings are recorded.

**Mutagenesis Database.** This dataset is widely used in Inductive Logic Programming research [29]. It contains two entity tables and two relationships.

**Financial** This dataset is a modified version of the financial dataset from the discovery challenge that was organized at PKDD'99. We adapted the database design to fit the ER model by following the modification from CrossMine [34] and Graph-NB [1]. The data are organized in 7 tables (4 entity tables, 3 relationship tables with 15 descriptive attributes in total).

**Hepatitis Database.** This dataset is a modified version of the PKDD02 Discovery Challenge database [5]. The data are organized in 7 tables (4 entity tables, 3 relationship tables) with 16 descriptive attributes.

[4]www.grouplens.org

| Dataset | VJ-time | SQL-time | Sum(counts) | #Tuples | Compress Ratio |
|---|---|---|---|---|---|
| Movielens | 2.70 | 703.99 | 23M | 252 | 93,053.32 |
| Mutagenesis | 1.67 | 1096.00 | 1M | 1,631 | 555.00 |
| Financial | 1421.87 | N.T. | 149,046,585M | 3,013,011 | 49,467,653.90 |
| Hepatitis | 3536.76 | N.T. | 17,846M | 12,374,892 | 1,442.19 |
| IMDB | 7467.85 | N.T. | 5,030,412,758M | 15,538,430 | 323,740,092.05 |
| Mondial | 1112.84 | 132.13 | 5M | 1,746,870 | 2.67 |
| UW-CSE | 3.84 | 350.30 | 10M | 2,828 | 3,607.32 |

**Table 3: Constructing the full contingency table for each database. Computation times are given in seconds. M = million. N.T. = nontermination.**

**IMDB Database.** This is the most challenge dataset in terms of number of total tuples (more than 1.3M) and schema complexity. The dataset combines the MovieLens database with data from the Internet Movie Database (IMDB)[5] [21]. The schema contains four entity tables, and three relationship tables.

**Mondial Database.** This dataset is a geography database, contains data from multiple geographical web data sources. We follow the modification of She[27], and use a subset of the tables and disretized features.

**UW-CSE Database.** This dataset lists facts about the Department of Computer Science and Engineering at the University of Washington[4]. There are two entity tables (i.e., $Person$, $Course$) and two relationship tables ($AdvisedBy$, $TaughtBy$); both are person-person self-relationships.

## 9. CONTINGENCY TABLE COMPUTATION

We report measurements about our Virtual Join (VJ) algorithm for constructing the full contingency table for each database. We compare the VJ algorithm with using an SQL query to construct the table. The SQL query materializes the cross product of the entity tables for each first-order variable (primary keys). The size of this cross product is identical to the sum of counts in the *ct*-table, reported in Table 3. The ratio of this sum to the number of tuples in the *ct*-table measures how much compression the *ct*-table provides compared to enumerating instantiations (cases). For faster processing, both methods used an extra B+tree index built on each column in the original dataset. The VJ method also utilized B+ indexes on the *ct*-tables; we include the cost of building these indexes in the reported time.

The Virtual Join algorithm returned a result in feasible time. On the biggest dataset IMDB with 1.3 million tuples, it took just over 2 hours. The SQL query did not always terminate, crashing after around 4, 5, and 10 hours on Financial, IMDB and Hepatitis respectively. When the SQL query did terminate, it took orders of magnitude longer than the VJ method except for the Mondial dataset. Generally the higher the compression ratio, the higher the time savings. On Mondial the compression ratio is especially low, so materializing the cross-product was faster. We also tried a more complex SQL query that combines the cross-product (in the FROM clause) with the Count aggregate function and Group By on the nodes. This ran even more slowly because it requires counting in addition to processing the cross-product.

## 10. STATISTICAL MODEL SELECTION

We compared the following structure learning methods.

[5]www.imdb.com, July 2013

| Dataset | LAJ | Flat |
|---|---|---|
| Movielens | 1.53 | 0.96 |
| Mutagenesis | 1.78 | 1.86 |
| Financial | 96.31 | 1,241.07 |
| Hepatitis | 416.70 | N.T. |
| IMDB | 551.64 | N.T. |
| Mondial | 190.16 | 1,289.53 |
| UW-CSE | 2.89 | 2.36 |

**Table 4: Model Structure Learning Time in seconds. N.T. = nontermination.**

**Learn-and-Join** The learn-and-join algorithm with contingency tables representing link uncertainty (Sec. 7.1).

**Flat Search** Applies the single-table Bayes net learner to the full database contingency table.

**Complete Graph** Fully connected DAG. The complete graph has the maximum number of parameters.

Table 4 provides the model search time for each of the link analysis methods. On the smaller and simpler datasets, all search strategies are fast, but on the medium-size and more complex datasets (Hepatitis, MovieLens), LAJ is much faster due to its use of constraints. For hierarchical search, the runtime for building the contingency tables (see Table 3) dominates the structure learning cost.

*Statistical Scores.* We report learning time, log-likelihood, Bayes Information Criterion (BIC), and the Akaike Information Criterion (AIC). BIC and AIC are standard scores for Bayes nets [2], defined as follows. We write

$$L(\hat{G}, \mathbf{d})$$

for the log-likelihood score, where $\hat{G}$ is the BN $G$ with its parameters instantiated to be the maximum likelihood estimates given the dataset $\mathbf{d}$, and the quantity $L(\hat{G}, \mathbf{d})$ is the log-likelihood of $\hat{G}$ on $\mathbf{d}$. We use the relational log-likelihood score defined in [24], which differs from the standard single-table likelihood only by replacing counts by frequencies so that scores are comparable across different nodes.

The BIC score is defined as follows [2, 24]

$$BIC(G, \mathbf{d}) = L(\hat{G}, \mathbf{d}) - par(G)/2 \times ln(m)$$

where the data table size is denoted by $m$, and $par(G)$ is the number of free parameters in the structure $G$. The AIC score is given by

$$AIC(G, \mathbf{d}) = L(\hat{G}, \mathbf{d}) - par(G).$$

AIC is asympotically equivalent to selection by cross-validation.

Fat search tends to introduce many more parameters than necessary. The likelihood score of flat search is essentially the same as that for the completely connected graph. The LAJ lattice search achieves the best scores on all datasets. The more complex the schema, the greater the improvement. This is evidence that the lattice edge constraints help avoid overfitting.

| **Movielens** | BIC | AIC | log-likelihood | # Para. |
|---|---|---|---|---|
| LAJ | **-4927.84** | **-295.44** | **-3.44** | **292** |
| Flat | -5168.08 | -315.44 | -3.44 | 312 |
| Complete | -11461.86 | -678.44 | -3.44 | 675 |

| **Mutagenesis** | BIC | AIC | log-likelihood | # Para. |
|---|---|---|---|---|
| LAJ | **-9083.88** | **-726.96** | -5.96 | **721** |
| Flat | -9224.25 | -1120.59 | **-5.59** | 1115 |
| Complete | -5806905.08 | -423374.59 | -5.59 | 423369 |

| **Financial** | BIC | AIC | log-likelihood | # Para. |
|---|---|---|---|---|
| LAJ | **-68562.16** | **-2443.74** | -10.74 | **2433** |
| Flat | -188908818.94 | -7206670.64 | **-10.66** | 7206660 |
| Complete | -12218648361.31 | -374400009.12 | -10.67 | 374399999 |

| **Hepatitis** | BIC | AIC | log-likelihood | # Para. |
|---|---|---|---|---|
| LAJ | **-10364.93** | **-585.58** | **-16.58** | **569** |
| Flat | N.T. | N.T. | N.T. | N.T. |
| Complete | N.T. | N.T. | N.T. | 25804799999 |

| **IMDB** | BIC | AIC | log-likelihood | # Para. |
|---|---|---|---|---|
| LAJ | **-2072673.77** | **-60070.39** | **-11.39** | **60059** |
| Flat | N.T. | N.T. | N.T. | N.T. |
| Complete | N.T. | N.T. | N.T. | 967680291 |

| **Mondial** | BIC | AIC | log-likelihood | # Para. |
|---|---|---|---|---|
| LAJ | **-3980.41** | **-357.20** | -18.20 | **339** |
| Flat | -8886625.98 | -865255.07 | **-14.07** | 865241 |
| Complete | N.T. | N.T. | N.T. | 23437499999 |

| **UW-CSE** | BIC | AIC | log-likelihood | # Para. |
|---|---|---|---|---|
| LAJ | **-3187.86** | **-248.1** | -8.10 | **240** |
| Flat | -42038.65 | -3939.01 | **-6.01** | 3933 |
| Complete | -356947710.61 | -22118404.95 | -6.02 | 22118399 |

**Table 5: Statistical Performance of different Searching Algorithms by dataset.**

## 11. RELATED WORK

*Sufficient Statistics for Single Data Tables.* Several data structures have been proposed for storing sufficient statistics defined on a *single* data table. One of the most well-known are ADtrees [17]. The branches in an ADtrees are labeled with variable values, so a path defines a conjunctive query. A node stores the count of the query that corresponds to the path from the root to the node. The ADtree provides a memory-efficient data structure for *storing* and retrieving sufficient statistics once they have been computed. In this paper, we focus on the problem of *computing* the sufficient statistics, especially for the case where the relevant rows have not been materialized. For computing the sufficient statistics, storing them in contingency tables has the advantage that blocks of sufficient statistics can be processed all at once as a table algebra operation. Thus ADtrees and contingency tables are complementary representations for different purposes: contingency tables support a computationally efficient block access to sufficient statistics, whereas ADtrees provide a memory efficient compression of the sufficient statistics. An interesting direction for future work is to build an ADtree for the joint contingency table once that table has been computed. Similar points apply to other data structures for storing sufficient statistics with minimal memory, such as KDtrees, data cubes and frequent item sets; for discussion, see [17].

There are many proposals for utilizing relational database management systems to speed up traditional single-table machine learning (e.g., with User-Defined Functions [19]). The Unpivot operator aims for efficiently computing sufficient statistics from data using a relational database [9]. In contrast to our work, Unpivot gathers sufficient statistics only from a single table, and only for two random variables at a time (an attribute and the class label). Our work aims

to support cross-table machine learning for the entire relational database.

*Virtual Joins for Multiple Data Tables.* Tuple ID propagation provides a Virtual Join method for computing query counts based on joins of existing database tables [34]. It expands the original data records so that query counts can be computed quickly. Within our dynamic program, Tuple ID propagation can be used to compute counts for sufficient statistics that involve positive relationships only. Since methods such as Tuple ID propagation address the case of positive relationships only, this paper focuses on computing query counts with a combination of positive and negative relationships.

Getoor *et al.* provided a subtraction method for the special case of estimating counts with only a single negative relationship [6, Sec.5.8.4.2]. Schulte *et al.* show that in principle, the fast Möbius transform can be used to extend the subtraction method to the case of multiple negative relationships [26]. They used the Möbius transform for Bayes net parameter learning, not structure learning. Parameter learning requires sufficient statistics only for a child node and its parents. In their experiments the child + parent families were fairly small (at most 6 nodes). Their evaluation did not involve sufficient statistics for more than one relationship. In contrast, our method can be used to support Bayes net structure learning, which requires joint sufficient statistics over the entire database. Other novel aspects are the *ct*-table operations and the use of the relationship chain lattice to facilitate dynamic programming.

*Learning Graphical Models for Relational Databases.* Several researchers have noted the usefulness of constructing a graphical statistical model for a relational database [3, 28, 33], for instance for exploratory data analysis and dealing with uncertainty. Computing sufficient statistics is a fundamental requirement for learning graphical models. Singh and Graepel propose an algorithm for compiling a database schema into a set of random variables for a Bayes net model [28]. This is like our approach to generating a default set of random variables from the database schema. The main difference is that in our application we learn the Bayes net structure from the data rather than generating a fixed structure as in [28]. The fixed Bayes net structure is based on latent variables, whereas in this paper we do not address learning with latent variables. Latent variable methods such as matrix and tensor factorization [20] have been successfully applied to learning graphical models for relational data. An interesting direction for future work is to extend our Virtual Join algorithm to support latent variable learning.

Inductive Logic Programming (ILP) is an approach to discriminative learning for relational data based on logical clauses [14]. For a comparison of ILP with graphical model learning please see [4, 12]. Most ILP systems do not utilize a relational database management system (RDBMS). The user provides metainformation about the data and possible patterns through mode declarations, which requires significant expertise [32]. Our approach of storing metainformation in database tables is like the BayesStore system [33], where all components of a statistical model are treated as first-class citizens in the RDBMS. That is, structured random variables, structured models, model parameter values and sufficient statistics are all stored in database tables. The BayesStore system focuses on relational probabilistic *inference* with a Bayes net model. Our work is complementary in that it focuses on *learning* the Bayes net model; BayesStore can be used to perform inference on the model once it is learned. The Tuffy system also performs statistical-relational inference utilizing an RDBMS [18].

# 12. CONCLUSION AND FUTURE WORK

A key factor for scalable machine learning is fast access to sufficient statistics, which can be stored in contingency tables. Sufficient statistics for negative relationships support learning correlations among relationships, which can be used for probabilistic reasoning about links in the relational structure. We presented an dynamic algorithm for computing the joint contingency table for an entire relational database. The contingency table includes sufficient statistics for conjunctive queries that combine information from different database tables, and may contain any number of negative relationships. The basis of the program is a subtraction method for obtaining query counts with $k+1$ negative relationships, from counts with only $k$ negative relationships. To derive the subtraction method, we introduced *ct*-table algebra, an extension of relational algebra designed for computation with contingency tables. *ct*-table algebra facilitates the efficient implementation of the dynamic program.

The joint contingency table can be applied to learning a Bayes net structure that represents probabilistic dependencies across the entire database. The efficient computation of sufficient statistics allows the system to scale to large datasets (over 1M tuples) with complex schemas. We found that a hierarchical lattice search strategy [25] performs better than simply using the joint contingency table, in terms of both learning time and model selection scores.

In addition to computational efficiency, an attractive property of our system is that it applies to SQL-based databases in general, and requires almost no configuration input from the user to prepare a target database for analysis. We achieve this generality by extracting from the system metadata information about the random variables that are the target of statististical analysis.

*Future Work.* Our dynamic program scales well with the number of rows, but not with the number of columns in the data tables. This limitation stems from the fact that the joint contingency table grows exponentially with the number of random variables in the table. One approach would be to apply the dynamic programming algorithm only up to a relatively small relationship chain length. A sufficient chain length could be specified by the user or determined by a learning algorithm. Another approach is to use the Virtual Join algorithm with postcounting [15]: Rather than precompute a large contingency table for a large set of random variables prior to learning, compute many small contingency tables for small subsets of variables, as needed by the learning program.

A cache of sufficient statistics can be applied to many learning problems, in addition to learning Bayes net structure. Examples include relational classification and link prediction methods. Another potential application is in probabilistic first-order inference [22]. Such inferences often require sufficient statistics (parfactors) defined with respect to one or more specified individuals (e.g., the number of user $u$'s male friends). It is easy to extend our Virtual Join algorithm to compute sufficient statistics for specified individuals.

In sum, our Virtual Join algorithm efficiently computes query counts which may involve any number of negative relationships. These sufficient statistics supports a scalable statistical analysis of dependencies among both relationships and attributes in a relational database.

## 13. REFERENCES


[1] H. Chen, H. Liu, J. Han, and X. Yin. Exploring optimization of semantic relationship graph for multi-relational Bayesian classification. *Decision Support Systems*, 48(1):112–121, 2009.

[2] D. Chickering. Optimal structure identification with greedy search. *Journal of Machine Learning Research*, 3:507–554, 2003.

[3] A. Deshpande and S. Sarawagi. Probabilistic graphical models and their role in databases. In *VLDB '07*, pages 1435–1436, 2007.

[4] P. Domingos and M. Richardson. Markov logic: A unifying framework for statistical relational learning. In *Introduction to Statistical Relational Learning* [8].

[5] R. Frank, F. Moser, and M. Ester. A method for multi-relational classification using single and multi-feature aggregation functions. In *PKDD*, pages 430–437, 2007.

[6] L. Getoor, N. Friedman, D. Koller, A. Pfeffer, and B. Taskar. Probabilistic relational models. In *Introduction to Statistical Relational Learning* [8], chapter 5, pages 129–173.

[7] L. Getoor and J. Grant. Prl: A probabilistic relational language. *Machine Learning*, 62(1-2):7–31, 2006.

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

[9] G. Graefe, U. M. Fayyad, and S. Chaudhuri. On the efficient gathering of sufficient statistics for classification from large sql databases. In *KDD*, pages 204–208, 1998.

[10] D. Heckerman, C. Meek, and D. Koller. Probabilistic entity-relationship models, PRMs, and plate models. In *Introduction to Statistical Relational Learning* [8].

[11] H. Khosravi, T. Man, J. Hu, E. Gao, and O. Schulte. Learn and join algorithm code. http://www.cs.sfu.ca/ oschulte/jbn/.

[12] H. Khosravi, O. Schulte, J. Hu, and T. Gao. Learning compact markov logic networks with decision trees. *Machine Learning*, 89(3):257–277, 2012.

[13] T. Khot, S. Natarajan, K. Kersting, and J. W. Shavlik. Learning Markov logic networks via functional gradient boosting. In *ICDM*, pages 320–329, 2011.

[14] N. Lavravc and S. Dvzeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.

[15] Q. Lv, X. Xia, and P. Qian. A fast calculation of metric scores for learning bayesian network. *International Journal of Automation and Computing*, 9:37–44, 2012.

[16] B. Milch, B. Marthi, S. Russell, D. Sontag, D. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *Statistical Relational Learning*, pages 373–395. MIT Press, 2007.

[17] A. W. Moore and M. S. Lee. Cached sufficient statistics for efficient machine learning with large datasets. *J. Artif. Intell. Res. (JAIR)*, 8:67–91, 1998.

[18] F. Niu, C. Ré, A. Doan, and J. W. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *PVLDB*, 4(6):373–384, 2011.

[19] C. Ordonez and S. K. Pitchaimalai. Fast UDFs to compute sufficient statistics on large data sets exploiting caching and sampling. *Data Knowl. Eng.*, 69:383–398, 2010.

[20] E. E. Papalexakis, U. Kang, C. Faloutsos, N. D. Sidiropoulos, and A. Harpale. Large scale tensor decompositions: Algorithmic developments and applications. *IEEE Data Eng. Bull.*, 36:59–66, 2013.

[21] V. Peralta. Extraction and integration of movielens and imdb data. Technical report, Laboratoire PRiSM, Universit de Versailles, 2007.

[22] D. Poole. First-order probabilistic inference. In *IJCAI*, pages 985–991, 2003.

[23] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[24] O. Schulte. A tractable pseudo-likelihood function for Bayes nets applied to relational data. In *SIAM SDM*, pages 462–473, 2011.

[25] O. Schulte and H. Khosravi. Learning graphical models for relational data via lattice search. *Machine Learning*, 88(3):331–368, 2012.

[26] O. Schulte, H. Khosravi, A. Kirkpatrick, T. Gao, and Y. Zhu. Modelling relational statistics with bayes nets. *Machine Learning*, 94:105–125, 2014.

[27] R. She, K. Wang, Y. Xu, and P. S. Yu. Pushing feature selection ahead of join. In *SIAM SDM*, pages 536–540, 2005.

[28] S. Singh and T. Graepel. Automated probabilistic modelling for relational data. In *CIKM*, pages 1497–1500, 2013.

[29] A. Srinivasan, S. Muggleton, M. Sternberg, and R. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1-2):277–299, 1996.

[30] The Tetrad Group. The Tetrad project, 2008. http://www.phil.cmu.edu/ projects/tetrad/.

[31] J. D. Ullman. *Principles of Database Sjystems*. W. H. Freeman & Co., 2 edition, 1982.

[32] T. Walker, C. O'Reilly, G. Kunapuli, S. Natarajan, R. Maclin, D. Page, and J. W. Shavlik. Automating the ilp setup task: Converting user advice about specific examples into general background knowledge. In *ILP*, pages 253–268, 2010.

[33] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. pages 340–351, 2008.

[34] X. Yin, J. Han, J. Yang, and P. S. Yu. Crossmine: Efficient classification across multiple database relations. In *ICDE*, pages 399–410. IEEE Computer Society, 2004.