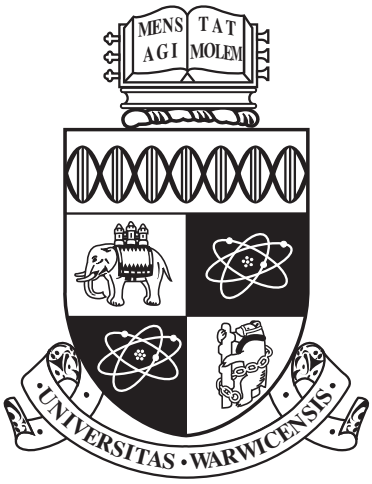


Learned Joins with Probabilistic Graphical Models

by

Ali Mohammadi Shanghooshabad

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy in Computer Science

Department of Computer Science

October 2022

THE UNIVERSITY OF
WARWICK

Contents

List of Tables

List of Figures

Acknowledgments

I normalized the acknowledgments into three tables. Please, execute the following join query.

```
Select Subject, Con_Type as Contribution
From Subject S, Subject_Contribution SC, Contribution C
Where S.Sub_Id = SC.Sub_Id AND SC.Con_ID = C.Con_Id;
```

Subject

Sub_Id	Subject
1	*Prof. Peter Triantafillou*
2	*My Parents*
3	*Prof. Graham Cormode*
4	*Prof. Yulan He*
5	*My Siblings*
6	*Meghdad*
7	*Gabriele*
8	*Vasan*
9	*Mohammad*
10	*Mehrdad*
11	*Qingzhi*
12	*Ming-wei*
13	*Technical Staff*
14	*Iranian Gov.*
15	*EPSRC*
16	*Chance*

⋈

Subject_Contribution

Id	Sub_Id	Con_Id
1	1	1
2	1	2
3	1	3
4	1	4
5	1	5
6	2	3
7	2	4
8	2	7
9	3	4
10	3	5
11	4	4
12	4	5
13	5	4
14	5	7
15	6	4
16	6	7
17	7	4
18	7	7
19	8	4
20	8	7
21	9	4
22	9	7
23	10	7
24	11	7
25	12	7
26	13	6
27	14	8
28	15	9
29	16	10

⋈

Contribution

Con_Id	Con_Type
1	*Supervision*
2	*Guidance*
3	*Support*
4	*Encouragement*
5	*Advice*
6	*Technical Assistance*
7	*Joy Friendship*
8	*Sabotage*
9	*Money*
10	*NULL*

Declarations

This thesis is submitted in support of my application to the University of Warwick and is presented in compliance with the regulations for the degree of Doctor of Philosophy. It was written entirely by myself and was never submitted in any previous applications for any degree. This thesis was completed by myself under the supervision of Prof. Peter Triantafillou. Parts of this thesis have previously been published by the author:

Publications:

- Ali Mohammadi Shanghooshabad, Meghdad Kurmanji, Qingzhi Ma, Michael Shekelyan, Mehrdad Almasi, and Peter Triantafillou. 2021. PGMJoins: Random Join Sampling with Graphical Models. In Proceedings of the 2021 International Conference on Management of Data. p1610–1622 (Explained in Chapter 3)

  `https://dl.acm.org/doi/abs/10.1145/3448016.3457302`

- Ali Mohammadi Shanghooshabad. 2021. XLJoins. In Proceedings of the 2021 International Conference on Management of Data. p2902–2904 (Explained in Chapter 3 and Chapter 5).

  `https://dl.acm.org/doi/abs/10.1145/3448016.3450582`

The papers listed below are available on arxiv but have not yet been published.

- Ali Mohammadi Shanghooshabad, and Peter Triantafillou. "Revisiting Join

Algorithms with Probabilistic Graphical Models", arXiv preprint arXiv:2206.10435, 2022 (Explained in Chapter 4).

`https://arxiv.org/abs/2206.10435`

- Ali Mohammadi Shanghooshabad, and Peter Triantafillou. "Model Join: Enabling Knowledge Discovery Over Joins of Absent Big Datasets", arXiv preprint arXiv:2206.10434, 2022 (Explained in Chapter 5).

  `https://arxiv.org/abs/2206.10434`

The following publications, of which I am not the first author, were prepared in collaboration with other lab members.

- Q. Ma, A. M. Shanghooshabad, M. Kurmanji, M. Almasi, and P. Triantafillou. "Learned Approximate Query Processing: Make it Light, Accurate and Fast". In Proceedings of CIDR 2021

  `https://www.cidrdb.org/cidr2021/papers/cidr2021_paper15.pdf`

- Michael Shekelyan, Graham Cormode, Qingzhi Ma, A. M. Shanghooshabad, Peter Triantafillou. "Weighted Random Sampling over Joins". EDBT 2023.

  `https://arxiv.org/abs/2201.02670`

Sponsorship and Grants:

This work was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) grant EP/R513374/1 for the University of Warwick.

Abstract

Because of the normalization in relational databases, joins are ever-present in analytical environments. However, joining multiple large tables raises critical challenges for modern databases as the operation is both time- and resource-consuming.

In this work, we investigate the use of Machine Learning (ML) and statistical models to reduce join costs at multiple levels. More specifically, we look at how learning the distribution of the join result might help us deal with the costs of joins. The distribution should be learned without generating the join result; otherwise it defeats the point. To do so, we map the join problem into Probabilistic Graphical Models (PGMs) in order to derive the factorized distribution of the join result. This is accomplished simply by scanning the tables once. Then, using the factorized distribution and tweaked versions of PGMs' algorithms that are efficient, principled, and easy-to-understand, we propose three different PGM-based solutions to three different problems with joins.

First, we present the *PGM-Join* sampler, a PGM-based algorithm for generating a uniform and independent sample of the join result without producing the join result, and we demonstrate that our solution is up to 28 times faster than the state-of-the-art join sampling methods.

Second, we propose Graphical Join (GJ), a new physical join algorithm. GJ generates a frequency-based summary of the join result, which is subsequently de-summarized to retrieve the exact join result (after optionally storing and loading the summary from disk). GJ significantly reduces the time and storage cost of joins.

Furthermore, while doing our research on joins, we found out that a new challenge in ML-driven modern relational databases is emerging, called the *model join* problem. For several reasons, sometimes the tables may not be accessible, but models of the tables are. So, how can one join the models rather than the tables? We define the model join problem for the first time and suggest the first solution for that which is PGM-based.

Abbreviations

AQP:	Approximate Query Processing
BN:	Bayesian Network
FDB:	Factorized Databases
JT:	Junction Tree
LKD:	Learning and Knowledge Discovery
MDNs	Mixture Density Networks
MRF:	Markov Random Field
NN:	Neural Network
NDVs:	Number of Distinct Values
PGMs:	Probabilistic Graphical Models
PJA:	Physical Join Algorithm
RDBMSs:	Relational DataBase Management Systems
R.I.P.:	Running Intersection Property
UIRs	Unneeded Intermediate Results
VEA:	Variable Elimination Algorithm
WOJAs:	Worst-case Optimal Join Algorithms

Chapter 1

Introduction

The overarching goal of this work is to investigate how machine learning (ML) and statistical models might assist data management systems in fulfilling their responsibilities more effectively. More specifically, in this thesis, the use of ML and statistical models to reduce the costs associated with joins, as well as the emergence of new challenges (e.g. joining models), are explored, and, of course, solutions are provided as well. In particular, Probabilistic Graphical Models (PGMs) serve as our central idea for tackling join challenges.

This section begins with an explanation of what a join operation is, followed by a discussion of the challenges and costs associated with join operations. Next, the thesis's research questions are listed. Then, our core idea, which is used to answer the majority of the questions, is briefly explained. After that, the contributions of the thesis are listed, and then the structure of the thesis is given.

1.1 The Join and its Challenges

To preserve the data's integrity and save time and space in transactional systems, the data is typically normalized into many tables. However, when it comes to analyzing the data in analytical environments, the normalized tables need to be joined, hence the join operations are ever-present.

The join operation merges data from multiple tables into a single one. Informally, a join is the process of combining multiple tables by inserting records from each table into the same row if and only if their fields match.

The following example equi-join serves as a case study for the entirety of this thesis.

Running Example: Assume the three normalized tables D_1, D_2 and D_3

D_1

...	**A**	**B**	...
...	a_0	b_0	...
...	a_0	b_0	...
...	a_0	b_0	...
...	a_1	b_1	...
...	a_1	b_1	...
...	a_2	b_1	...
...	a_3	b_3	...
...	a_3	b_3	...
...	a_3	b_4	...
...	a_3	b_4	...
...	a_3	b_4	...
...	a_3	b_4	...

D_2

...	**B**	**C**	...
...	b_0	c_0	...
...	b_0	c_0	...
...	b_1	c_0	...
...	b_1	c_0	...
...	b_1	c_0	...
...	b_2	c_1	...
...	b_2	c_1	...
...	b_2	c_1	...
...	b_3	c_2	...
...	b_4	c_3	...
...	b_4	c_3	...
...	b_4	c_4	...

D_3

...	**C**	**D**	...
...	c_1	d_0	...
...	c_1	d_0	...
...	c_1	d_0	...
...	c_1	d_0	...
...	c_2	d_2	...
...	c_2	d_2	...
...	c_2	d_2	...
...	c_2	d_2	...
...	c_3	d_3	...
...	c_3	d_3	...
...	c_4	d_4	...
...	c_4	d_4	...

Figure 1.1: Example with three tables

Join Result

id	**A**	**B**	**C**	**D**
0	a_3	b_3	c_2	d_2
...	...	...	...	...
7	a_3	b_3	c_2	d_2
8	a_3	b_4	c_3	d_3
...	...	...	...	...
23	a_3	b_4	c_3	d_3
24	a_3	b_4	c_4	d_4
...	...	...	...	...
31	a_3	b_4	c_4	d_4

Figure 1.2: The join result for the running example

shown in Figure 1.1 and the join query Q as below:

```
SELECT A,B,C,D FROM D1, D2, D3
WHERE D1.B = D2.B and D2.C=D3.C
```

Figure 1.2 depicts the join result of Q with 32 entries.

Nowadays, data is expanding at a rapid rate, and join operations in this era of big data can become excessively expensive in terms of execution times, resource consumption (memory and disk), and monetary charges (in cloud environments). This cost is considerably greater when joining (many) very big tables, especially when they involve many-to-many relationships.

When the join size is too large, sampling the join result is a promising alternative to computing the actual join, as it can lower the time and space costs of joins. If one wants to analyze the join result and perform various learning and knowledge discovery tasks, uniform and independent samples of the actual join result are great to avoid the above issues while facilitating such analytics, due to their well-understood theoretical guarantees. However, computing the join first and then sampling is obviously defeating the point. One option would be to just uniformly sample each table first and then join the uniform samples. Unfortunately, it is well-known that this would produce a join-result sample of poor quality. In Chapter 3,

we present our solution, called *PGM-Join* sampler, to the join sampling problem to generate a uniform and independent sample of the join result without generating the join result. The *PGM-Join* sampler is based on PGM principles.

However, sampling is not always sufficient; even when the join size is too large, it is occasionally necessary to find the full join result and store/retrieve it. This is especially true for situations in which we want exact answers to queries. Or when the result of a join query is input for another join query. This problem is referred to as the PJA problem (problems related to the Physical Join Algorithms) in this thesis.

A good solution for the PJA problem is the one that can produce the join result as quickly as possible while avoiding extra costs for redundancy in the join result and Unneeded Intermediate Results (UIRs) for multi-way joins. It should also be able to represent the result in a way that takes up less space and is easy to store and retrieve. A possible method is to generate the result of the join, then summarize it (e.g., using Run-Length Encoding RLE), and then store it on disks. Although the RLE summary over joins is easily stored/retrieved and de-summarized (particularly in columnar databases), generating it after the join result generation defeats the point. In fact, the goal is to skip generating the full join results. In Chapter 4, we discuss how a summary like RLE can be produced without generating the join result. One exciting achievement of Chapter 4 is that we demonstrate for the first time that the idea of producing the RLE without generating the join result and de-summarizing it can be considered as a novel PJA capable of outperforming state-of-the-art PJAs on queries with UIRs and redundancy. We demonstrate that our new PJA (called *Graphical Join* or GJ for short) is a Worst-case Optimal Join Algorithm (WOJA). A WOJA is a join algorithm that efficiently handles UIRs (to be explained in Chapter 2).

Our recent efforts to solve join problems have revealed a need to run joins even when the tables are inaccessible. Sometimes we have access to models over tables, but we do not have access to the tables, particularly in Approximate Query Processing (AQP) where the models are trained over data and queries are answered by using the models. The models are updated when the new data comes in, and it seems impractical to maintain the massive underlying data. Hence, the data can be forgotten (deleted) [Kersten and Sidirourgos, 2017; Milo, 2019]. In the case of AQP, methods like DeepDB [Hilprecht et al., 2020], DBEst [Ma and Triantafillou, 2019] and DBEst++ [Ma et al., 2021] are some of the successful models over relational data. Furthermore, sometimes the tables exist but because of privacy concerns or in federated learning settings, we may not have access to them. For any reasons,

we sometimes have models rather than tables, and we have to still perform a join query on them. So, how can we join models? In Chapter 5, for the first time, we define the *model join* problem as a new challenge in today's ML-driven relational databases. We also introduce a framework to deal with the model join problem, called the Model-Join framework. The Model-Join framework generates approximately uniform samples of the join results. The approximation stems from the models, and the framework does not add any additional error to the uniformity.

Thus, in this thesis, we focus on three main problems: (i) the table-join sampling problem, in which all of the tables are available for joining; (ii) the PJA problem, when an exact-full join result is needed, but where redundancy and UIRs would be a burden; and (iii) the model join problem, in which ML models substitute the tables to be joined.

For all the aforementioned challenges, we have a single core idea (explained in Section 1.3); Learn the distribution of the join result with PGMs, then generate the samples or summaries.

1.2 Research Questions

Following is a list of the specific research questions that this thesis attempts to answer:

- How are table-join sampling and model-join sampling problems, as well as the PJA problem, formulated using PGMs?
- How may PGM algorithms be adjusted to handle the three join problems mentioned?
- How to efficiently generate a uniform and independent sample of the join without generating the joint result?
- How can a PJA deal with UIRs and redundancy?
- How can the RLE-based summary over joins be generated without generating the join result?
- Is summarization-then-desummarization idea a suitable alternative to standard PJAs that directly generate join tuples? If so, in what scenarios?
- How to enumerate the join result in a columnar way?

Joint Distribution

id	**A**	**B**	**C**	**D**	***Freq***
0	a_3	b_3	c_2	d_2	8
1	a_3	b_4	c_3	d_3	16
2	a_3	b_4	c_4	d_4	8

Figure 1.3: The joint distribution of the join result of the running example

- How to perform model joins more accurately and generate a uniform and independent sample of the join result where the tables are absent.
- How can models be joined with other available tables?
- What are the main obstacles to learning ML models on tabular data? And what are the solutions for those obstacles?
- What are other open challenges with the model join problem?

1.3 The Core Idea

The key to resolving most of these challenges with joins is in learning the distribution of the join result (the joint distribution). For example, a uniform sample of the join result may be generated, **if** we already had the joint distribution. For instance, the joint distribution illustrated in Figure 1.3 contains the distinct tuples of the join result of Q (the running example) and their frequencies, and the uniform sample can be generated using this information. Note, it is straightforward to convert frequencies to probabilities by dividing the frequencies by the join size. Or, if we need to produce the full join result regarding the PJA problem, we can save time by producing the join tuples from the joint distribution. In other words, the joint distribution provides the frequencies of the join tuples, so we can replicate the tuples based on their frequencies (and without repeatedly accessing the indices of different tables). Additionally, the joint distribution is smaller than the join result, so the space consumed is likewise less. These are only a few examples of how joint distributions could be used to address our problems.

However, the question that needs to be answered is how the joint distribution may be determined. The obvious answer would be to first construct the join result and then learn the joint distribution; however, this would defeat the point! We intend to avoid generating the join result since joins are costly in terms of time, space and money (in clouds). The good news is that we can quickly discover another kind of the joint distribution (namely, factorized joint distribution) from a normalized database by scanning each to-be-joined table once. Factorized distributions are

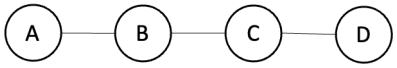


Figure 1.4: PGM graph for the running example

Factor on D_1

A	**B**	***Freq***
a_0	b_0	3
a_1	b_1	2
a_2	b_1	1
a_3	b_3	2
a_3	b_4	4

Factor on D_2

B	**C**	***Freq***
b_0	c_0	2
b_1	c_0	3
b_2	c_1	3
b_3	c_2	1
b_4	c_3	2
b_4	c_4	1

Factor on D_3

C	**D**	***Freq***
c_1	d_0	4
c_2	d_2	4
c_3	d_3	2
c_4	d_4	2

Figure 1.5: The factorized distribution for the running example

represented by PGMs. Essentially, a PGM model consists of a number of factors (potential functions) whose product is proportional to the joint distribution without factorization. In relational databases, normalization is analogous to distribution factorization. Since the data is already normalized in a relational database [1], the distributions learnt from individual tables can be utilized as factors for a factorized joint distribution, and the product of those factors will give the factorized joint distribution. Therefore, all that remains is to scan the tables, calculate the exact frequencies for the involved attributes, and generate the table-specific factors. Add every factor to a PGM graph to obtain the factorized joint distribution.

However, dealing with the factorized distribution differs from working with the unfactorized joint distribution. Effective inference techniques are required when dealing with factorized distributions, and this is exactly what PGMs give us.

We explain PGMs in Chapter 2, but for now, assume PGMs have nodes (in our case, table attributes) and edges among them (the dependencies among attributes). Having two attributes in the same table indicates that they are dependent. For example, Figure 1.4 depicts the PGM graph for our running example. The dependencies among attributes are exactly determined by scanning the tables once. Note, all attributes that are not involved in the query are simply ignored. Figure 1.5 shows the dependencies (the factors) in the edges of the graph in Figure 1.4.

In this thesis, we intend to formulate our three main join problems with PGMs. PGMs are principled machinery that provide a variety of efficient tools and standard algorithms for examining factorized distributions and performing statistical inference (e.g. calculating the marginals). PGMs are both time- and space-efficient (will be explained in Chapter 2). We first find the factorized distribution for a given query by scanning the tables once, and then employ PGMs' principled algorithms

[1] Certainly, if we are asked to run a join query, it means that the database has already been normalized to some extent; otherwise, we would not have more than one table to join.

to overcome our join sampling problems and the PJA problem. However, the known standard algorithms of PGMs cannot be applied directly to solve our problems. We shall modify the algorithms to make them specific to our problems.

1.4 Contributions

Contributions are categorized into three groups: those that address the join sampling problem, those that address the PJA problem, and those that address the model join problem.

1.4.1 PGM-Join Sampler: A New PGM-Based Join Sampling Method

- The adaptation of PGMs for uniform join sampling. The PGM formulation makes the problem and solutions easier to understand and prove. We show how to construct PGMs for the join sampling problem and the optimizations we put forth, leveraging the PGM Markov properties and PGM inference algorithms. These optimizations refer to (i) the PGM structure itself and (ii) to the inference process and algorithms. Notably, the PGM-Join sampler performs exact inference, producing uniform samples of the true join result.
- A solution that relies only on the number of distinct values per attribute (NDVs) and not on all tuples. As NDVs are typically much fewer, this has significant performance advantages.
- A new, more efficient method to deal with cyclic join sampling.
- A new (adapted to uniform join sampling) Variable Elimination Algorithm (VEA) to build a uniform sample generator.
- Detailed performance evaluation comparing the PGM-Join sampler against two state-of-the-art methods using queries and data from TPC-H, JOB, TPC-DS and Twitter.

  The code can be found at `hhttps://github.com/shanghoosh1/PGMJoins`

1.4.2 GJ: A Novel Worst-Case Optimal Physical Join Algorithm with PGMs

- A mapping from the physical n-way equi-join problem to PGMs.
- A different way of thinking about physical n-way equi-join algorithms: instead of building indexes and using binary-join algorithms or WOJAs over raw data

tables, first produce a (PGM-based) join summary, followed by optionally storing it to disk and retrieving it from disk when called to (re)produce the join result, and finally desummarizing it. The summary can be stored on and retrieved from a disc simply; hence it is compatible with relational databases and very compact.

- A novel algorithm to generate an RLE-style summary over the grouped join result *without* executing the expensive join operation and *without* paying the costs related to UIR. This is based on leveraging PGM principles and on tweaked PGM inference algorithms for statistics calculation. (This summary has other important applications of its own).

- An algorithm for efficiently producing the tuples from the factorized distribution (columnarly).

- A complexity analysis showing worst-case optimality.

- A detailed performance evaluation using JOB, lastFM and TPCH data and queries, comparing GJ against join processing in PostgreSQL [Drake and Worsley, 2002], MonetDB [Idreos et al., 2012]) and the WOJA in Umbra [Freitag et al., 2020]. We also compare GJ against FDB [Olteanu and Schleich, 2016]. This study showcases GJ's large gains in space and time and highlights (less) favorable use cases for GJ. The code for GJ and its parallel version are available at `https://github.com/shanghoosh1/Graphical_join` and at `https://github.com/shanghoosh1/Parallel_Graphical_Join`.

1.4.3 Model-Join Sampler: Join Models, Forget Tables

- A definition of the model join problem.

- The first framework (coined Model-Join) which essentially can join the models (which have replaced tables) without adding any extra error. The framework is based on the PGM principles.

- An altered inference algorithm that, instead of calculating marginals, calculates all of the statistics needed to construct a sample generator. The end result is a generative model, able to generate an arbitrary number of high-quality approximations of the data tuples in the join result.

- If we need to conduct a model join query involving multiple models and tables, Model-Join learns all of the necessary models on those available tables with

a new efficient and accurate method, then joins the models. Each per-table model entails a novel blending of embeddings, clusterings, and feed-forward Neural Networks (NN).

- A detailed experimental evaluation, analyzing this new problem and quantifying the quality of the produced sample, the efficiency of Model-Join, and its appropriateness for downstream analytics tasks. The code is available at: `https://github.com/shanghoosh1/ModelJoin`.
- A list of open challenges to be addressed en route.

1.5 Thesis Structure

Six chapters make up this thesis.

Chapter 2 is devoted to discussions of background, in which we explain PGMs and PJAs, as well as several analytical engines over joins. Chapter 3 describes and evaluates the PGM-Join sampler. In Chapter 4, a novel PJA is presented and assessed. In Chapter 5, we define the model join problem and present our proposed framework. Finally, Chapter 6 outlines the conclusions and future work.

Chapter 2

Background

This chapter includes an overview of all the necessary concepts, algorithms, and techniques required to understand the remainder of the thesis, as well as an explanation of some related work required for all chapters.

Since PGMs form the basis for our solutions, we begin by providing an overview of these factorized models and the inference procedures that can be applied to them. Moreover, as a new PJA is introduced in Chapter 4, we study the existing PJAs here, including binary PJAs (Nested-Loop, Sort-Merge, and Hash join algorithms with their derivatives) that join tables two-by-two and multi-way PJAs that join all tables simultaneously (a.k.a. WOJAs). Then we describe the methods for analytics over joins. These analytics are performed without generating the join result.

2.1 Probabilistic Graphical Models

A PGM factorizes a distribution by using a graph $G(V, E)$ and certain rules $\mathcal{M}$, where V is a collection of vertices (also known as variables or nodes) and E is a set of edges between the nodes. The graph and $\mathcal{M}$ determine the model type. The model is called a Bayesian Network (BN) [Pearl, 1985] if the graph has directed edges, and a Markov Random Field (MRF) [Kindermann and Snell, 1980] if the graph has undirected edges. Also, the $\mathcal{M}$ for MRFs and BNs varies. Because BNs are outside the scope of our work, we will solely discuss MRFs.

An MRF is a type of PGM having an undirected graph G and three (Pairwise, Local and Global) Markov properties as $\mathcal{M}$ which are defined by the concept of *conditional independence*. Two variables A and B are conditionally independent given C if $P(A, B|C) = P(A|C) \times P(B|C)$.

Given $G(V, E)$, the Pairwise Markov property expresses that two non-adjacent variables X_u and X_v are conditionally independent if all other variables are observed: $X_u \perp\!\!\!\perp X_v | X_{V \setminus \{u,v\}}$.

The Local Markov property says if the neighboring variables of a variable are fully observed, that variable is conditionally independent from all other variables: $X_u \perp\!\!\!\perp X_{V \setminus N[v]} | X_{N(v)}$, where $N(v)$ is the set of neighbors of variable v, and $N[v]$ is $\{v\} \cup N(v)$ is called the closed neighborhood of v.

Finally, the Global Markov property states that two sets of variables are mutually independent if a separating set of variables are observed: $X_A \perp\!\!\!\perp X_B | X_S$ where X_A and X_B are separated by X_S; in other words, all the paths from X_A to X_B go through variables in X_S.

A PGM consists of two components: qualitative (a.k.a. structure learning) and quantitative (a.k.a. parameter learning). Qualitative learning entails learning the structure of the PGMs (the vertices and edges), whereas quantitative learning entails learning the factors (a.k.a. potential functions). The factors should be learned over the joint distribution in such a way that the product of the factors yields the same joint distribution. In most PGMs, learning the structure and factors is a time-consuming process that requires human understanding of the structure as well as approximate learning approaches such as Maximum likelihood. However, as will be explained in Chapter 3, due to normalization in relational databases, the structure and factors of PGMs for join queries are already available. This is the primary reason why we map the join problem to PGMs to obtain the factorized distribution of joins.

To show the effect of the factorization with PGMs, imagine a distribution $p(A, B, C)$ in which we know A and C are conditionally independent given B. Instead of keeping a large probability function holding the full joint distribution $p(A, B, C)$, we can learn and keep a small factorized version of p like $\psi(A, B) \times \psi(B, C)$ with lower storage cost, where ψ is called a factor (a.k.a. as a potential or potential function). Note that not only can PGMs reduce the storage cost by keeping smaller local factors, but also they can make the probabilistic inference (e.g. marginalization) more efficient. For example, to calculate the marginal of A in the factorized distribution $\psi(A, B) \times \psi(B, C)$ with an elimination order $O = \{C, B\}$, first, we can sum out C from $\psi(B, C)$ and calculate a factor for B as $\psi(B)$ then we can sum out B from $\psi(A, B) \times \psi(B)$. This dynamic programming-based paradigm to calculate the marginals is known as the Variable Elimination Algorithm (VEA). Summing B and C out from the factorized distribution by VEA is faster than summing them out from the larger unfactorized distribution $p(A, B, C)$. Section 2.1.1 explains VEA in detail.

Definition 1 (Cliques C(G)). *A clique c is a set of fully connected nodes in a graph G(V,E). C(G) contains all the cliques that exist in G. In other words, if* $c \in C(G)$, *and* $u, v \in c$ *then the edge* $(u, v) \in E(G)$

Given a graph $G(V, E)$ with cliques $C(G)$, the probability distribution $p(x_V)$ of an MRF can be represented as follows

$$p(x_V) = \frac{1}{Z} \prod_{c \in C(G)} \psi_c(x_c) \tag{2.1}$$

where ψ is a factor that is a non-negative function over the joint domain of a set of variables. x_c is the set of variables in clique c. All x_c variables form x_V which is the set of all variables in the model. $p(x_V)$ is the product of all factors of cliques in the MRF. Z is called the partition function which is a normalization constant:

$$Z = \sum_{x_V} \prod_{c \in C(G)} \psi_c(x_c) \tag{2.2}$$

In our case, Z is the join size. Since in our work we just need the frequencies of distinct values (not the probabilities), we can omit Z simply from the Equation 2.1.

Definition 2 (Maximal Cliques/ maxcliques $\mathcal{C}(G)$). *A maximal clique (or maxclique) is a clique which by adding any* $v \in V$ *in that clique, makes it no longer a clique.*

Any clique can be absorbed by its maxclique. So, we can replace C with $\mathcal{C}$ in Equation 2.1. That absorption is accomplished by calculating the product of the factors of cliques inside a maxclique.

2.1.1 VEA

At its heart, this thesis relies on probabilistic inference. Probabilistic inference is essentially a task of calculating quantities (e.g., marginals) of some variables over a distribution [Koller and Friedman, 2009]. VEA is one of the exact algorithms introduced to perform probabilistic inference.

To calculate a marginal over the distribution presented in Equation 2.1, VEA eliminates the variables one by one based on an elimination order O by using a sum-product operation presented in Equation 2.3 per variable. For example, let us assume we have n variables in $V = \{x_1, x_2, ..., x_n\}$. To calculate the marginal of a variable x_1, all other variables should be eliminated (summed-out) one-by-one. The formula

for the sum-product operation for eliminating one of them (x_i) is as follows:

$$\sum_{x_2,\ldots,x_{i-1},x_{i+1},\ldots,x_n} \prod_{c\in\mathcal{O}_{x_i}} \psi_c(x_c) \sum_{x_i} \prod_{c\in\mathcal{I}_{x_i}} \psi_c(x_c) \tag{2.3}$$

where $\mathcal{I}_{x_i}$ and $\mathcal{O}_{x_i}$ contain the cliques that do and do not involve x_i, and ψ is the potential function. In essence, a sum-product operation involves calculating the product of all the factors that include the variable x_i and then summing the variable x_i out from the product result. The result of the summing out will be a new factor for the x_i's neighbors.

Thus, we can say that the complexity of calculating the marginals is specified by the size of the largest $\mathcal{I}$ (how many variables it contains). Assume $\mathcal{I}$ has m variables and each variable has r distinct items in its domain, the complexity of the VEA to eliminate all variables is $O(r^m)$.

Note that after eliminating a variable, all its neighbors should make a single clique, and if the neighbors are not fully-connected in the corresponding graph, new edges are added. The new edges are called fill-in edges. These edges are added because there exist no functions like $g(x)$ and $h(y)$ in summing out of z:

$$g(x)h(y) = \sum_z f_1(x,z).f_2(z,y) \tag{2.4}$$

and the output of the summing out should be a function of both x and y, $f(x,y)$. In other words, x and y are dependent on each other via z, and if z is removed, we should maintain the dependency via a function that includes both x and y. These new edges make the inference inefficient and we want to avoid them because they make larger cliques and the larger cliques make larger $\mathcal{I}$ in the sum-product operation. Hence, the elimination order should be chosen carefully so that it creates the fewest new fill-in edges.

In trees, each node has just one parent, meaning that there is at least one elimination order O that does not introduce any new fill-in edges. We call that O a perfect elimination order. An elimination order is perfect when, by eliminating the variables, no fill-in edges are added to the graph. Eliminating the leaves of the trees results in no new fill-in edges, and after the leaves are eliminated, new leaves appear, and eliminating the new leaves results in no new edges too. Trees thus have a perfect elimination order starting from the leaves to the root (note, not all eliminations are perfect in trees), and VEA can calculate the marginals optimally. But what about graphs with cycles?

2.1.2 VEA on Graphs

Generally, to run VEA on graphs, people translate the graph into a structure that has the same characteristics as trees. In other words, a tree of maxcliques is made from the graph. So, every node may represent more than one variable. The Running Intersection Property (R.I.P) is one of the characteristics that the tree of maxcliques should hold, as it holds for trees. R.I.P is identical to Cluster Intersection Property (C.I.P). Both are explained below.

R.I.P of maxcliques: Let $C_1, C_2, ..., C_l$ be an ordered sequence of maxcliques in graph G where l is the number of maxcliques. We say the ordering obeys the R.I.P if for all i>1, there exists $j < i$ such that $C_i \cap (\cup_{k<i} C_k) = C_i \cap C_j$

In Figure 2.1 (e), assume the order is $C1, C2$ and $C3$ then the intersection between $C2$ and $C3$ (which is the node (C, D)) is equal to $C3 \cap (C2 \cup C1)$. The union $(C2 \cup C1)$ is called the history of $C3$.

C.I.P of maxcliques Suppose we have a tree of maxcliques $\mathcal{T}$. For any pair of maxcliques C_i and C_j in $\mathcal{T}$, there should be a unique path between C_i and C_j, and $C_i \cap C_j$ should appear in all the maxcliques along the path from C_i to C_j. For example, in Figure 2.1 (e), the node C is the intersection of maxcliques $C1$ and $C3$ and it appears in the maxclique $C2$ as well.

Interestingly, a tree of maxcliques that possesses C.I.P also possesses R.I.P and vice versa.

Junction Tree (JT) of maxcliques: JT of maxcliques is a tree of the maxcliques which possesses the R.I.P and C.I.P properties. There have exist many algorithms to translate graphs to JTs [Koller and Friedman, 2009; Cowell et al., 2007; Jensen and Nielsen, 2007]. Here, we just provide a brief explanation of JT creation. JT creation is used in Chapter 4 where a new PJA is introduced.

For a graph, there may be more than one JT. We need to find the best JT where the size of the largest maxclique is smaller than the size of largest maxcliques in other possible JTs from the same graph. Recall that the complexity of finding a marginal is dependent on the size of the largest clique in the graph. For example, we could take all the variables as one maxclique by adding fill-in edges, but in this case, the size of the largest maxclique would be equal to the size of V and the complexity of the inference (VEA) on that tree will be exponentially high. Extracting the best JT is NP-Complete.

A triangulated graph is a graph that the number of nodes in any cycle is not greater than 3. It has already been proved that if a graph is triangulated, it has a JT of maxcliques which possesses R.I.P. [Wainwright and Jordan, 2008; Blake et al., 2011]. The triangulated graphs have a perfect elimination order and eliminating any

node will not introduce any new fill-in edges. However, not all PGM graphs have a perfect elimination order. For example, in Figure 2.1 (a), there is a cycle with 4 nodes and eliminating each of them will introduce a fill-in edge. If the graph is not triangulated (e.g. Figure 2.1 (a)), finding the best elimination order and adding new fill-in edges can lead us to the best triangulated graph, thus we can have the best JT as well. One should check all the combinations to find the best elimination order, hence finding the best elimination order is NP-Complete as well. The good news is that, in our join problem, the number of nodes is small and hence one can find the best elimination order with a small number of fill-in edges, exhaustively. Nonetheless, greedy heuristic algorithms, such as min fill-in, work well even on the graphs with thousands of nodes. We will use this algorithm in our solutions.

Moreover, other efficient greedy ways in [Abo Khamis et al., 2016] can be utilized to convert a cyclic graph to an acyclic one.

The Min Fill-in Heuristic: In each step, the min fill-in heuristic adds one node in the elimination order O, and that node is the node which introduces the minimum number of new fill-in edges. If there is more than one node with minimum fill-in edges, it breaks the ties arbitrarily. In Figure 2.1 (a), A should be eliminated first then among B, C, D and E, one is chosen randomly, and so on and so forth.

The min fill-in heuristic can provide the triangulated graph. For a given graph $G(V, E)$, we find the new fill-in edge set E' with the min fill-in heuristic and make the triangulated graph as $G'(V, E \cup E')$. Figure 2.1 (b) is one of the possible triangulated graphs for the graph Figure 2.1 (a). Note that when we add a fill-in edge, it is as if we add a potential equal to 1 that can join with any other potentials. The main potentials for fill-in edges are calculated during inference. The min fill-in heuristic can also output all the maxcliques in the graph during triangulation since after elimination of a node, all its neighbors should be a clique. Any pair of maxcliques that have some shared variables are connected to each other so that they make a graph of the maxcliques (e.g. Figure 2.1 (c)).

So, the output of the min fill-in heuristic contains an elimination order O, a triangulated graph, and a graph of the maxcliques. The issue is how to derive a JT of maxcliques from the graph of maxcliques. There are several ways, and the easiest way is to apply the maximal spanning tree algorithm. This algorithm finds all the separator sets among maxcliques and chooses the edges with maximum separator sizes one-by-one to span the graph of maxcliques. A separator set is a set of nodes in the graph G that if we remove it, G is divided into disconnected sub-graphs.

Figure 2.1 shows all the steps of translating the graphs to JTs of maxcliques. (a) shows the original graph G, (b) shows the triangulated G, (c) contains the graph

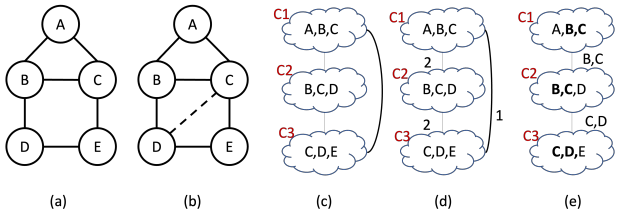


Figure 2.1: Example non-tree graph translation to a tree

of maxcliques. (d) shows the weights (separator sizes) and finally (e) is the JT of maxcliques after applying the maximal spanning tree algorithm.

Once the graph is translated to a JT, the same VEA can be used to calculate the marginals.

For more information about translating graphs to JTs, please refer to [Koller and Friedman, 2009; Cowell et al., 2007; Jensen and Nielsen, 2007; Wainwright and Jordan, 2008; Blake et al., 2011].

2.1.3 Applications of PGMs

PGMs have a wide range of potential uses; some examples include image processing (de-noising, in-painting, and generation), natural language processing (generation, and translating), audio (super-resolution, speech synthesis, and speech recognition), and healthcare. We are also not the first to use PGMs in relational databases. PGMs have a long history of application in database research for cardinality estimation, aggregation computation, and regression learning over join result without creating the join result [Wang et al., 2008; Deshpande and Sarawagi, 2007; Singh and Graepel, 2012; Tzoumas et al., 2011, 2013; Garrouch and Omri, 2017; Abo Khamis et al., 2016; Olteanu and Schleich, 2016]; nonetheless, our use of PGMs in this thesis is different as we attempt to solve different challenges (the join sampling, performing the physical joins, and the model join sampling).

2.2 Physical Join Algorithms (PJAs)

PJAs are joins that are not used in SQL queries by users. Rather, query processing engines implement these as operators or algorithms to accomplish logical joins (e.g. inner and outer joins). Query processing engines typically apply all filters and predicates to single tables before employing a PJA (or a mix of PJAs) to perform the actual join. In other words, PJAs are concerned with generating (enumerating) any tuples in the join result that fulfill the join predicates; other predicates and operations (e.g., aggregations, group-by, etc.) are not of interest.

The PJAs that join the tables two-by-two are known as binary PJAs. Those that join all the involved tables together simultanously are known as multi-way PJAs.

2.2.1 Binary Physical Join Algorithms

Binary PJAs are the most common and widely deployed PJAs in commercial relational databases. The three main binary PJAs used in relational databases are hash joins, nested-loop, and sort-merge algorithms. When running the join queries, the optimizer selects one (or a combination of them), each of which has advantages and downsides of its own. For full information about those algorithms please refer to [Mishra and Eich, 1992; DeWitt et al., 1993; Dittrich et al., 2002; Graefe, 1994; Kitsuregawa et al., 1983; DeWitt et al., 1984], however, we quickly review each of them and discuss their advantages and disadvantages.

Nested-Loop Join

A simple nested-loop join is implemented using two level nested for-loops. The outer loop reads rows one at a time from the first (outer) table, passing each row to a nested loop that processes the next (inner) table in the join. The inner table is fully scanned for each tuple in the outer table. The matched tuples are returned as the join results. This simple algorithm works best with cross joins.

Block Nested-Loop Join is a version of nested-loop join that pairs each inner relation block with each outer relation block. In order to accomplish this, the relations are processed per block rather than per tuple. Blocks are stored in buffers. When the buffer size is sufficient to store the complete relation in memory, the block nested-loop join reduces block access significantly.

Indexed nested-loop join builds an efficient index on the inner table, which significantly improves efficiency.

Despite being costly with large tables, nested-loop join can be used with any type of joins, making it suited for any join condition. Nested-loop will operate even if there is no index and the tables are not sorted. This approach is superior to other binary PJAs when the tables to be joined are small. Last but certainly not least, this algorithm suffers from UIRs because of its binary nature.

Sort-merge Join

With a sort-merge join, both relations are sorted based on their join attributes, then scanned in the order of their join attributes. The resulting relation is formed by

merging tuples that meet the join requirement. In other words, it merges two sorted tables in a zipper-like fashion.

Sort-merge has the advantage that each relation is only scanned once if it is sorted, and the join condition can be any condition. Moreover, the memory usage is usually low with sort-merge join. The issue with the sort-merge join is the requirement that relations must be sorted on the join attribute prior to merging. This algorithm also suffers from UIRs.

Hash Join

Hash join creates an in-memory hash table on the join column of the inner table, then scans the outer table for matches to the hash table and joins the corresponding data from the two tables.

If the complete hash table fits in memory, a hash join is less expensive than any other binary PJAs. However, hash join cannot be utilised for inequality joins (a.k.a. theta joins), and this approach undoubtedly suffers from UIRs the same as other binary PJAs.

2.2.2 Worst-case Optimal Physical Join Algorithms

WOJAs were invented to deal with UIRs as much as possible in the multi-way many-to-many joins (where UIRs exist). Any join algorithm that has a complexity of $O(N^{\rho})$ is a WOJA. ρ is the edge cover number of the fractional edge cover and N is the size of largest table in the join. In the following, we first explain the edge cover and then the fractional edge cover. Then, we will overview the state-of-the-art WOJAs and present a general pseudo-code for all WOJAs.

Edge Cover

Assume a hyper-graph (V,E) is constructed over the involved tables; V is the set of all nodes (a node per attribute) and E is the set of the hyper edges among them, which are inserted as follows: For each table, there is a hyper-edge including all the nodes coming from the same table. Edge cover is a subset $C \subseteq E$ of edges such that each node appears in at least one edge. Finding the edge cover can be formulated as an integer programming problem by assigning to each edge $e_i \in E$ a weight λ_i, with $\lambda_i = 1$ when $e_i \in C$ and $\lambda_i = 0$ when e $\notin C$. For example, for a "triangle" join TQ of tables $D_1(A, B), D_2(B, C), D_3(C, A)$ with the sizes $|D_1|, |D_2|$ and $|D_3|$ respectively, the integer program can be defined as below:

Minimize $\lambda_{D_1} + \lambda_{D_2} + \lambda_{D_3}$, subject to

$$\text{A: } \lambda_{D_1} + \lambda_{D_3} \geqslant 1$$
$$\text{B: } \lambda_{D_1} + \lambda_{D_2} \geqslant 1$$
$$\text{C: } \lambda_{D_2} + \lambda_{D_3} \geqslant 1$$

For query TQ, the edge cover is 2 and the upper bound for the join size is $O(N^2)$ where N is the max of $|D_1|, |D_2|, |D_3|$.

The Fractional Edge Cover ρ

Atserias et al. in [Atserias et al., 2008] proved a tight bound based on the "fractional edge cover" for the maximum join size for full conjunctive queries. ρ is computed by relaxing the integer program to a linear programming problem, allowing the edge weights to range between 0 and 1. For example, choosing $\lambda_{D_1} = \lambda_{D_2} = \lambda_{D_3} = 1/2$ for TQ yields a valid fractional cover and the upper bound is $O(N^{3/2})$.

General WOJA

WOJAs have enjoyed significant attention in the last decade and several WOJAs have been introduced, e.g., NPRR in [Ngo et al., 2012], Leapfrog TrieJoin in [Veldhuizen, 2012] and the versions of WOJAs with new data structures to build the tries including [Freitag et al., 2020], [Arroyuelo et al., 2021] and [Fekete et al., 2019]. The WOJA presented in [Freitag et al., 2020] offers the state-of-the-art WOJA for RDBMSs.

Algorithm 1 describes a general algorithm for WOJAs. This algorithm resembles Algorithm 1 from [Freitag et al., 2020]. All to-be-joined tables should be sorted with all WOJAs, and they are attribute-based (not table-based) and require specific attribute generation order. The attributes are generated sequentially in accordance with this order.

The idea is that WOJAs simultaneously process the related tables with the same join attributes (let us call those tables as $\mathcal{D}_{join}$). They efficiently discover the matches in those tables $\mathcal{D}_{join}$ by using structures like tries and trees. Finding the matches is similar to a sort-merge join algorithm, however it is performed on many tables with the identical join attribute (refer to [Veldhuizen, 2012]). For each tuple k_i in the result of $\mathcal{D}_{join}$ (of the first join attribute), the same approach is used to identify the matching tuples in the $\mathcal{D}_{join}$ of the succeeding join attribute. π and σ are the projection and selection operators, respectively. This procedure will continue until the algorithm has covered all of the join attributes and enumerated all of the linked tuples. To enumerate all join tuples, the procedure is recursive and backtracks numerous times.

Algorithm 1 General Algorithm for WOJAs

For a given hyper-graph $\mathcal{H}_Q(V, \mathcal{E})$ of query Q where $V = \{v_1, v_2, ..., v_n\}$ is the set of all involved attributes and $\mathcal{E} = \{e_1, e_2, ..., e_m\}$ is the set of hyper-edges over V, the recursive function ($WOJA_{rec}$) will return all the tuples in the join result of m tables.
i is the index for join attributes, and $\mathcal{D}$ is the set of tables involved in the join query.

1: **procedure** $WOJA_{rec}(i, \mathcal{D})$
2: **if** $i <= n$ **then** ▷ If any attribute remains
3: $\mathcal{D}_{join} \leftarrow \{\mathcal{D}_j \in \mathcal{D} | v_i \in \mathcal{D}_{E_{\mathcal{D}_j}}\}$ ▷ $\mathcal{D}_{join}$ contains all tables that include v_i
4: $\mathcal{D}_{other} \leftarrow \{\mathcal{D}_j \in \mathcal{D} | v_i \notin \mathcal{D}_{E_{\mathcal{D}_j}}\}$ ▷ $\mathcal{D}_{other}$ contains all tables that exclude v_i
5: **for each** $k_i \in \bigcap_{\mathcal{D}_j \in \mathcal{D}_{join}} \pi_{v_i}(\mathcal{D}_j)$ **do** ▷ k_i is the shared value among tables in $\mathcal{D}_{join}$
6: $\mathcal{D}_{next} \leftarrow \{\sigma_{v_i=k_i}(\mathcal{D}_j) | \mathcal{D}_j \in \mathcal{D}_{join}\}$ ▷ $\mathcal{D}_{next}$ contains the matching tuples
7: $WOJA_{rec}\ (i+1, \mathcal{D}_{next} \bigcup \mathcal{D}_{other})$ ▷ The recursive part for the next attribute
8: **end for**
9: **else**
10: product $(\bigtimes_{\mathcal{D}_j \in \mathcal{D}} \mathcal{D}_j)$ ▷ Enumerates all the tuples in the result of the product
11: **end if**
12: **end procedure**

2.3 Analytics over Joins

The traditional way of calculating aggregations or performing any other kind of analytics over joins is to first find the join result, and then perform the analytics over it. However, these days, several new solutions have been proposed, and their goal is to perform analytics over joins without generating the results. Hence, they are not physical join algorithms.

Sampling over joins (join sampling) (see Chapter Chapter 3 and 5), learning regressions or any other machine learning models over joins without generating the join result (e.g. [Olteanu and Schleich, 2016]), calculating the aggregations over joins without generating the join result (e.g. [Olteanu and Závodnỳ, 2015; Abo Khamis et al., 2016]) and generating a compressed version of the join result (e.g Run-Length Encoding in Chapter 4) without generating the join result are some of examples for analytics over joins. In this section, we give a high-level overview of the existing solutions for analytics over joins, and in subsequent chapters, we introduce our own solutions for some of the applications involving analytics over joins.

2.3.1 Uniform Join Sampling Methods

Join sampling (without first computing the join) has a long history as a topic of study within our community. This thesis focuses specifically on uniform and independent

samples due to their extensive applicability in AQP, ML and any other data analytic tasks.

Olken's method in [Olken, 1993] is based on rejection sampling and gives uniform and independent samples over 2-way joins. Assume two tables R_0 and R_1. His approach first selects a tuple t_1 from R_0 randomly, and next finds $(t_1 \bowtie R_1)$ and randomly selects t_2 from it, and then accepts $(t_1 \bowtie t_2)$ with the probability of the frequency of t_2 over the maximum frequency in R_1, otherwise rejects it. This approach has been found to be inefficient because of the high number of rejections and is only applicable on two-way joins.

Chadhuri et al's method in [Chaudhuri et al., 1999] generates uniform and independent samples for 2-way joins. If $t_1.ja$ is the value of the join attribute in tuple t_1, their approach selects a tuple t_1 from R_0 with the probability of the frequency of $t_1.ja$ in R_1 over the size of R_1. Once t_1 is selected, t_2 is then selected randomly from $(t_1 \bowtie R_1)$.

Both Chadhuri et al. and Olken's approaches need indices in the second table to be efficient, and they only work on 2-way joins.

Going beyond two-way joins, [Acharya et al., 1999] proposed join synopses for computing uniform n-way join samples, albeit only for foreign key joins. In general, when using foreign key joins, joining a uniform sample of the first table with other tables in the join query yields a uniform sample of the join result.

Zhao et al. in [Zhao et al., 2018b] introduced an extended framework for Olken's and Chadhuri et al's approaches for arbitrary n-way joins. This is currently the state of the art. It requires all tables (and related metadata like indexes) to be brought into memory. Their algorithm calculates the weights per *tuple* in a table over the join result in a dynamic programming way. Processing cyclic joins is not very efficient with their method as they need to perform the actual join on some part of the query. Authors in [Zhao et al., 2018b] introduced several methods, but *Exact Weight* (EW) and *Online Exploration* (OE) were the most efficient ones. We compare our method to these two methods in Chapter 3. Both of them are multi-way join sampling methods and have two phases: preparation and generation.

Exact Weight (EW)

EW uses a dynamic programming approach in the preparation phase that computes the exact weight for every tuple in a join. This is a generalization of the original Chaudhuri el al.'s algorithm [Chaudhuri et al., 1999]. Then it starts to generate the samples based on the exact weights.

EW has a slower preparation phase than OE, but it has a faster generation

phase.

Online Exploration (OE)

The OE algorithm employs Wander join [Li et al., 2016] and rejection sampling to generate uniform and independent samples. The preparation phase for OE is faster than for EW, but the generation time is longer because of the rejection sampling.

Both EW and OE require the tables and indices over the tables to be in memory during the preparation and generation phases.

The generated samples by these approaches can be used for AQP, learning models, visualization, or any other downstream tasks.

2.3.2 Factorized Databases (FDB)

The primary goal of FDB [Olteanu and Schleich, 2016] is to represent the join result as succinctly as possible so that analytics can be performed on it at a lower time and space cost. In other words, FDB can calculate aggregations and train regressions more efficiently without generating the join result. Therefore, FDB attempts to maximally factorize the join result. Please note that the factorized join result (data) is different than the factorized join distribution.

Assume the query is to calculate the natural join of two tables $R(A, B)$ and $S(A, C)$, and that both tables contain A values as $\{a_1, a_2, ..., a_n\}$ for a large n. Attributes B and C of the two tables are partitioned such that there is one partition per A-value for $i \in \{1, 2, ..., n\}$.

$R_i = \pi_B \sigma_{A=a_i}(R)$ and $S_i = \pi_C \sigma_{A=a_i}(S)$.

The join result can be obtained by the Cartesian product of those partitions in R and S. The expression includes product and union operations.

$R \bowtie S = \{(a_1)\} \times R_1 \times S_1 \cup ... \cup \{(a_n)\} \times R_n \times S_n$

However, FDB is not intended to calculate all partition products; rather, it maintains an unmaterialized version of partitions (implemented by views) in a tree-like structure with sum and product operations (considered as factorized result). Therefore, analytics (like aggregations) can be performed on this tree rather than on the actual join results. The FDB community calls this *f-representation* [Olteanu and Závodnỳ, 2012]. Also, keep in mind that FDB cannot perform analytics without the tables since *f-representation* just consists of unmaterialized views.

The authors of FDB also added two more join result representations. When more than two tables are to be joined, some partitions are recurred numerous times in their tree, thus they give a name (pointers) to each partition and simply reuse

the names in the *f-representation* to lower the size of the factorized result. The term for this representation is *d-representations* [Olteanu and Závodnỳ, 2015]. In addition, they give *cover* representation [Kara and Olteanu, 2017], which is a listing of the minimal edge cover of a bipartite graph generated from the tables. The tuples of the tables are nodes in the bipartite graph, and the edges are between the tuples that can join.

However, there are times when the enumeration cannot be avoided. The authors of [Olteanu and Závodnỳ, 2015] also propose a constant-delay enumeration approach that allows them to enumerate all join tuples from factorized representations. It is a Depth-First Search (DFS) algorithm with a backtracking-heavy approach for graph tracing. Even FDB can be deemed PJA with this enumeration algorithm. However, they do not implement it and compare it to other PJAs, as they do not aim to generate the join result. In fact, FDB strives to keep the join result representation as compact as possible and to avoid the join tuple enumeration. The enumeration algorithm generates join tuples row-by-row, necessitating access to indices for each value in each tuple. Furthermore, recall that FDB only works with factorized data and not factorized distribution, and thus it does not know the column values' frequencies in advance. Consequently, FDB cannot enumerate the join tuples columnarly, and hence the join result enumeration by FDB is not efficient. We believe columnar join result generation may be more efficient than row-by-row enumeration because columnar generation requires fewer index accesses than row-by-row enumeration. This will be experimentally demonstrated in Chapter 4.

Given that FDB is an in-memory engine, it is not clear how the factorized representation of the join result can be stored on a disk and retrieved into the memory.

For more information, refer to the main page for FDB [1].

2.3.3 FAQ problem

The authors in [Abo Khamis et al., 2016] define and investigate the Functional Aggregate Query problem (FAQ). They recognized that many frequently asked questions in constraint satisfaction, databases, matrix operations, probabilistic graphical models, and logic had similar properties, therefore they grouped them all together as a FAQ problem and proposed some efficient solutions for them all.

FAQ is related to our work because the FAQ solutions are based on PGM's principles, and the authors have modified the basic algorithms (such as the hypertree decomposition algorithm and the variable elimination algorithm) to address the

[1] `https://fdbresearch.github.io/index.html`.

frequently asked questions in a variety of settings (which is very useful, especially in PGMs and FDB).

The FAQ solution can be used to calculate aggregations over joins without generating the result of the join. FAQ solution works with factorized distributions. On this factorized joint distribution of the input tables, aggregations across a join query can be calculated. Their approach is not to enumerate the join result, however they claim that the enumeration algorithms from FDB may be utilized to enumerate the join results from the factorized joint distribution of a join query. Thus, the FAQ approach shares the same disadvantages as FDB in that it cannot enumerate the join tuples columnarly.

The main technical contribution in [Abo Khamis et al., 2016] is to propose heuristic algorithms to find the hyper-tree decomposition, which is very useful in PGMs. These heuristic algorithms can also be utilized in our join algorithms.

2.3.4 Approximate Query Processing (AQP)

In this section, we cover AQP techniques, applications, and new issues that have evolved in this field, such as the AQP over joins.

There are multiple well-established approaches.

For instance, online aggregation [Hellerstein et al., 1997] is a technique whereby the user receives online estimates of an aggregate query as soon as the query is issued. For example, if the final answer is 500, the user initially receives estimates such as 480 (or 520), and the estimate error continues to decrease as the AQP engine processes more samples.

AQP can also be performed with data sketching [Cormode and Muthukrishnan, 2005] and sample-based methods [Park et al., 2018a; Agarwal et al., 2013]. Using the sample-based method, samples are generated based on the workload, and when a query is received, an approximation of the answer is derived using the samples.

Nonetheless, AQP research has made significant progress in recent years. ML/statistical models that are accurate, efficient, and lightweight have altered the other AQP techniques. For example, DeepDB [Hilprecht et al., 2020], DBEst [Ma and Triantafillou, 2019; Ma et al., 2021], deep generative models [Thirumuruganathan et al., 2020], etc. are the most popular ones.

DeepDB: DeepDB applies Sum-Product Networks (SPNs) [Poon and Domingos, 2011] to tabular data and discovers the table's full distribution. Fundamentally, DeepDB attempts to cluster the table to the greatest extent possible until the attributes become independent. Then, for each attribute per cluster, a histogram is constructed. The model is a tree in which the histograms are the leaves and the

sum and product operations are intermediate nodes. The aggregate calculation in DeepDB is a bottom-up approach. Therefore, an aggregation computation begins at the leaves by examining the frequencies in the histograms, then ascends by performing the sum and product operations, and at the root, the answer to the query is obtained. DeepDB is trained on a sample, hence its estimated statistics are approximations.

DBEst: Based on a specific workload, DBEst trains density estimators and regression models to approximate query template answers. Assume a regression model $y = R(x)$ and density estimator $D(x)$, where x is the range predicate and y is the select attribute for which aggregations are to be calculated. This formula is used to calculate an aggregate SUM (y).

$$SUM(y) = S \cdot \int_{lb}^{ub} D(x)R(x)dx \tag{2.5}$$

where S represents the scaling factor and lb and ub represent the lower limit and upper bound of the range predicate, respectively. S depends on the sample size that the DBEst models have been trained.

For a group-by query, DBEst keeps a density estimator and a regression model for each group of a grouping attribute. If the number of groups is quite high, the size of models may swell. To solve this issue, in DBEst++ paper [Ma et al., 2021], we replaced all those models with a single conditional regressor called Mixure Density Networks (MDNs) [Bishop, 1994] and Embeddings (Skip-Gram [Mikolov et al., 2013]). DBEst++ is more accurate, lighter, and faster than DBEst.

These models are also capable of being updated. Therefore, when an exact response is not required, the underlying data can be forgotten. In this thesis, we refer to the forgotten tables as absent tables. There are further reasons why some tables are absent, such as privacy concerns. We will discuss those reasons in Chapter 5. The challenge with the absence of tables is that the learnt models cannot be used to perform AQP over the join of tables. Hence, this is a newly-emerging challenge in AQP, which is the subject of our Chapter 5. We will introduce the first model join framework to join AQP models so that queries on the joint distribution of the models can be answered. The framework is based on PGMs.

2.4 Embedding Learning: Skip_Gram with Negative Sampling

Learning embeddings is important for us because we are going to learn real-valued vector representations for each distinct value in the categorical attributes (e.g. join attributes) in Chapter 5.

Word embeddings are a type of word representation that bridges the gap between human and computer language comprehension. These are necessary for solving the majority of NLP problems. With word representation, words are represented as real-valued vectors, and the proximity of two word vectors indicates their relationship to one another. There are many embedding learning approaches; embeddings like Continuous Bag-of-Words (CBOW) and Skip_Gram [Mikolov et al., 2013] have been highly successful for NLP tasks because of the deep linguistic theory behind them (coined distributional hypothesis). In this work we use Skip_Gram with negative sampling [Mikolov et al., 2013] which is faster than the naive Skip_Gram.

Skip_Gram is a simple Neural Network (NN) with a single hidden layer with n dimensions as embedding vectors. The goal is to learn embedding vectors. Formally, given a sequence of training words $w_1, w_2, w_3, ..., w_K$, the objective of the Skip_Gram model is to maximize the average log probability

$$\frac{1}{K}\sum_{k=1}^{K}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{k+j} \mid w_k) \tag{2.6}$$

where c is the size of the window in the context, and K is the number of words in the vocabulary.

In the last layer of the Skip-Gram NN, a Softmax function is used to turn the logits to probabilities. To find $p(w_O|w_I)$ the following formula is used:

$$p(w_O|w_I) = \frac{\mathbf{exp}({v'_{w_O}}^{\top} \times v_{w_I})}{\sum_{w=1}^{M}\mathbf{exp}({v'_w}^{\top} \times v_{w_I})} \tag{2.7}$$

where v_w and v'_w are the input and output vector representations of the word w.

Negative sampling

The Skip_Gram has a large number of weights and all of them should be updated according to (in our case, maybe millions of) training data instances. Instead of updating all weights, negative sampling [Mikolov et al., 2013] helps to change the problem to a binary classification problem and randomly select just a small number

of *negative* words and then try to distinguish between randomly chosen negative words and the current word, so that there is no need to compute the similarity of one word with all other words in the corpus. The following equation is replaced with every $\log p(w_O|w_I)$ in the Skip-gram loss function Equation 2.6.

$$\mathbf{log}\ p({v'_{w_O}}^{\top} \times v_{w_I}) + \sum_{i=1}^{n} \mathbb{E}_{w_i \sim P_n(w)}[\sigma(-{v'_{w_i}}^{\top} \times v_{w_I})] \tag{2.8}$$

where there are n negative samples, and where $\sigma(x) = 1/(1+\mathbf{exp}(-x))$ and $P_n(w)$ is the noise distribution. (For more information please refer to [Mikolov et al., 2013].)

Chapter 3

A New Join Sampling Algorithm with PGMs

3.1 Motivation

Join operations are ever-present. However, they are expensive operations, in terms of execution times, resource consumption, and monetary costs, especially when joining (several) very large tables and when they entail many-to-many relationships. In the era of big data and of the high importance of fast big data analytics, the join operation can become prohibitive (e.g., taking days to complete [Zhao et al., 2018b]). As explained in the introduction section, sampling of the join result presents a promising alternative to computing the actual join: If one wants to analyze the join result and perform various learning and knowledge discovery (LKD) tasks (e.g., AQP, regression, clustering/classification, build advanced ML models, etc.) uniform and independent samples of the true join result are great to avoid the above issues while facilitating such analytics, due to their well-understood theoretical guarantees. For example, the state-of-the-art AQP methods are either based on such samples (VerdictDB [Park et al., 2018b]) or are based on ML models built from such samples (DBEst [Ma and Triantafillou, 2019; Ma et al., 2021], DeepDB [Hilprecht et al., 2020]). Likewise for selectivity estimation tasks [Yang et al., 2020].

However, computing the join first and then sampling is obviously defeating the point. One option would be to just uniformly sample each table first and then join the uniform samples. Unfortunately, it is well-known (Acharya et al [Acharya et al., 1999] and Chadhuri et al. [Chaudhuri et al., 1999]) that this would produce a join-result sample of poor quality. High quality samples in this setting are defined as those observing the qualities of *uniformity* and *independence*. In essence, the goal is

two-fold: First, to derive a sample of a certain size, by repeatedly and independently selecting a tuple at a time from the join-result with the same probability; and second to generate such a uniform and independent sample without the cost of computing the join result first. Although other types of join samples, such as hash sampling [Hadjieleftheriou et al., 2008] and the sampling techniques used in wander join [Li et al., 2016] and ripple join [Haas and Hellerstein, 1999; Haas, 1997; Jermaine et al., 2008] are adequate for certain applications, clearly these fail to ensure uniformity and/or independence.

Recently, [Zhao et al., 2018b] made a significant step forward generalizing previous efforts in [Chaudhuri et al., 1999; Olken, 1993] and being able to handle general n-way joins while computing uniform samples of joins without computing the join first.

However, the current state of the art [Zhao et al., 2018b] leaves plenty of opportunity for improvement, and the overall approach is not only inefficient but also complex and difficult to follow. We contribute a principled solution, coined PGM-Join sampler. The PGM-Join sampler adapts Probabilistic Graphical Models to deriving provably *uniform samples of the join result* for (n-way) key-joins, many-to-many joins, and cyclic and acyclic joins. The PGM-Join sampler contributes optimizations both for deriving the structure of the graph (especially with cyclic joins) and for PGM inference (with VEA). It also contributes a novel inference algorithm to build a uniform sample generator which from it generates the uniform sample of the joint distribution (join result) efficiently and a novel way to deal with cyclic joins. Despite the use of PGMs, the learned joint distribution is not approximated and the uniform samples are drawn from the true distribution.

The key idea is that [Zhao et al., 2018b] uses the real data tuples and creates indices over the data. However, the PGM-Join sampler concentrates on the distribution of the join query and does not access the underlying data when it is processing the join query. Additionally, the PGM-Join sampler is based on the established principles underlying PGMs, making it easier to understand, prove correct and come up with new solutions.

In this chapter, first, an overview of the PGM-Join sampler is provided. The creation of MRF graphs is then discussed. It is then described how the PGM-Join sampler processes (acyclic and cyclic) joins, which involves two phases: inference and sample generation. Furthermore, the uniformity of the generated samples is examined and proven. At the end, the experimental evaluation using queries and datasets from TPC-H, JOB, TPC-DS, and Twitter is shown.

3.2 Related Work

Join sampling. We have already discussed join sampling approaches in the background section and section 3.1, so we will refrain from repeating ourselves and instead concentrate on other related works.

Machine Learning for Data Analytics. ML models are being increasingly used in the data management community. An apparent good fit for this are analytical queries in approximate query processing (AQP) engines. [Anagnostopoulos and Triantafillou, 2015, 2017b] and [Park et al., 2017] showed how to build models which exploit previous data system answers to predict answers for future queries. [Anagnostopoulos and Triantafillou, 2017a] extended the above to predict answers for regression queries. More recently, [Ma and Triantafillou, 2019; Ma et al., 2021; Hilprecht et al., 2020; Thirumuruganathan et al., 2019; Kumar et al., 2015] all utilized various types of ML models to improve accuracy and efficiency for AQP engines. Also, [Yang et al., 2020] uses models for cardinality estimation over joins. All these works rely strongly on having random join samples when dealing with joins. Beyond AQP, ML has been playing an increasingly significant role in DB research. SageDB [Kraska et al., 2019] argues for a new DB design where deep ML models are first class citizens, able to learn base data distributions and guide the tasks of indexing, join processing and query optimization. Deep ML models have also been applied to learn index structures [Kraska et al., 2018], estimate join cardinality [Kipf et al., 2018], [Ortiz et al., 2018], and for analytical query forecasting [Ma et al., 2018] and performance prediction [Venkataraman et al., 2016].

PGMs. The majority of PGM learning algorithms are utilized when the joint distribution is known and we wish to factorize it. In join sampling problem, however, we already possess the factorized data, so we only learn the distribution of the factorized tables. Existing approaches like Monte Carlo methods [Hastings, 1970] approximate the distribution, leading to an *approximate uniform* sample. Markov chain approaches [Kass et al., 1998] also generate *dependent* uniform samples. The most common general Markov Chain Monte Carlo algorithm is called Gibbs Sampling [Gilks and Wild, 1992] and also Metropolis-Hastings Algorithm [Robert and Casella, 1999] which both generate dependent sample rows and approximate the uniformity.

In join sampling, we have a small graph with just a few nodes (a node per attribute) in the graph and cycles are easy to break. In this case, instead of executing VEA twice (as it is usual in PGMs) to find the exact factors for the joint distribution and then sample from it, we show that with one VEA execution the samples can be derived correctly and efficiently.

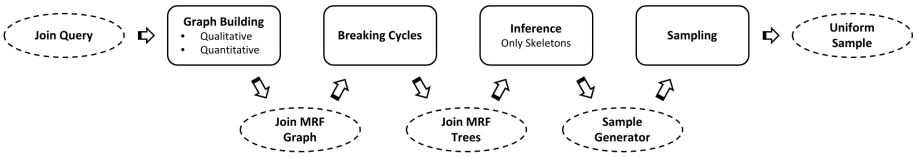


Figure 3.1: An overview of PGM-Join sampler

3.3 A Brief Overview of PGM-Join Sampler

Figure 3.1 depicts an overview of how the PGM-Join sampler generates uniform samples for given join queries.

For a given query, the PGM-Join sampler builds an MRF graph. There are qualitative and quantitative aspects to graph construction. In qualitative learning, a node is added to the graph for each involved attribute in the query, and an edge is added between attributes if they are in the same table. In quantitative learning, the tables are scanned and the exact dependency among the nodes is captured in the form of frequency tables. In this thesis, these frequency tables are called factors, potentials, and potential functions interchangeably.

Cycles may be present in the resulting MRF graph. The graph's cycle-forming elements are selected out as remainder components. If the remainder components also contain cycles, a portion of them is likewise pulled out once more to get rid of all the cycles. Several trees might appear in the end. However, most of the time there is just a single remainder component with a singe edge. Section 3.4 has further information.

Inference is performed on all of the resulting MRF trees in order to build sample generators for each tree. The PGM-Join sampler begins the inference process over the MRF trees with Algorithm 2 based on an elimination order that begins with the leaves and progresses to the root. Only the join attributes are used in inference. More information can be found in Section 3.5.

Once the sample generator for each tree is constructed, the PGM-Join sampler begins to generate samples (as explained in Algorithm 3.6.2) from the root to the leaves (the reverse order of the elimination order). The PGM-Join sampler generates samples for join attributes first, then adds non-join attributes to the sample. When there is more than one tree (indicating that the query is cyclic), the PGM-Join sampler employs a rejection mechanism to ensure that the samples are uniform. More information can be found in Section 3.6.

3.4 Modelling Join Queries with MRFs

PGMs are heavy machinery and learning the structure of the PGMs and their distribution factors is a time-intensive process. At first glance, it may appear that mapping joins to PGMs is not a good idea, but in normalized databases, the factors are already present and we do not need to learn or approximate them (as explained in Section 1.3). Any normalization in relational databases is analogous to distribution factorization. Basically, when a table D is normalized to many tables $D_0, D_1, ..., D_m$, the product (join) of the normalized tables will yield the exact D. That means if we have the exact distribution of the normalized tables and calculate the product of the distributions, we can obtain the exact joint distribution. In other words, the factors from the normalized tables can be integrated and form a PGM. An equivalence for the conditional independence of the PGMs in relational databases could be the fact that non-join attributes from different tables are conditionally independent from each other if the shared join attributes are observed. It is worth mentioning that factorizing a distribution does not necessarily imply that the underlying data has been properly normalized.

To be able to obtain uniform samples without generating join results, we desired to learn the joint distribution; hence, we formulate our solutions for the join sampling problem utilizing PGM principles. As a general example, assume all attributes in the normalized table D_i as a clique c_i in the query graph, and ψ_i as the distribution (potential function/factor) of the normalized table D_i. The factorized distribution of the join result of $D_1 \bowtie D_2 \bowtie ... \bowtie D_m$ based on Equation 2.1 is like:

$$p_D(x_V) = \frac{1}{Z} \prod_{i=1}^{m} \psi_i(c_i) \tag{3.1}$$

p_D is the joint distribution and x_V are the attributes/variables in the join result. Z is the normalization constant and can be ignored in the inference process. In our case, Z is the join size.

Here, we discuss how we map a join query to an MRF (a kind of PGM), and in the next sections, we explain how uniform samples are efficiently derived from the MRF.

Prior to initiating the join operation for a query, all conditions other than join conditions can be applied. In other words, filters and predicates can be pushed down and executed before anything else. Therefore, for the sake of simplicity, we will assume that the queries contain only join conditions. It goes without saying that our queries should lack aggregations as well, given that we wish to generate samples

of the join result.

For a given join query, the PGM-Join sampler learns an MRF. As explained in the background section, learning any kind of PGM has two components: qualitative and quantitative. For the qualitative component, the PGM-Join sampler adds a node in the MRF graph per join attribute. The same join attributes in different tables have a single node in the MRF graph. The attributes not involved in the join query (neither as a join attribute nor a projection attribute) are ignored. All non-join attributes are considered as a single node (a maxclique) in the graph. If the corresponding attributes of the two nodes are present in the same table, an edge between those nodes is added, showing that their variables are dependent on each other. We use undirected edges. It should be noted that only join attributes have an impact on sample uniformity. The MRF's leaves are all non-join attributes, while their parents are all join attributes from the same table. In other words, non-join attributes are only linked to join attributes in the same table. These parents act as separators, separating the leaves from the other nodes. In accordance with the Global Markov property, the non-join attributes are dependent exclusively on their parents (the join attributes from the same table) and independent of all other attributes once their parents have been observed.

$$non - JA_i \perp\!\!\!\perp A_{V \setminus JA_i} \quad | \quad JA_i \tag{3.2}$$

$non - JA_i$ is the set of non-join attribute of the i_{th} table, which is involved in the query. JA_i is a set of join attributes of the i_{th} table involved in the query, and $A_{V \setminus JA_i}$ is the set of all other attributes involved in the query. This fact makes the inference simpler and more efficient since we can first build a uniform sample of all join attributes and then add values for the desired non-join attributes to each data point in the sample. To accomplish so, we need to keep $\psi(non - JAs_i | JAs_i)$ for each table and use them for adding the non-join attributes' values in the sample of the join attributes.

We call the potential functions related to the join attributes as *skeleton*, and *non-skeleton* for the potential functions related to the non-join attributes. The non-skeleton potentials are added to the sample generator directly without performing any kind of inference over them.

As previously stated, learning the potentials in PGMs is a time-consuming process, but in our case, the factors are already ready due to normalization. To identify the potential functions per clique in the factorized distribution, we just need to scan the tables. These potential functions return the frequency of the distinct

tuples (only the involved attributes in the join). Each edge or node in the graph may have more than one potential. In that case, we just multiply the potentials. The complexity of the building potential is $O(N)$, where N is the size of the largest table in the join. Note that competitors require the tables and indices over the tables beforehand in order to execute their algorithms. The complexity of creating indexes is the same, and at least a full scan is needed. Consequently, as the competitors pre-build the indices, in our case, the graph can also be offline pre-built. Pre-built potentials, like pre-built indices, can be reused for several queries. In addition, they can be utilized for a variety of other applications, such as cardinality estimation and aggregation calculations.

As explained in the background section, trees have a perfect elimination order and running the inference from the leaves to the root of the tree is efficient. Hence, we want to make trees from any cyclic MRF graphs. Two ways are possible: either eliminate the edges that make the graph cyclic and then use the eliminated edges in a rejection mechanism (similar to the rejection mechanism explained in [Zhao et al., 2018b]), or apply a hyper-tree decomposition algorithm like the junction-tree creation algorithm to the cyclic graph to turn it into an acyclic one. With both, we will end up with a tree structure. Here, we will use the first option that we pull-out some parts of the graph to make it a tree. Our reason is that making the junction trees and the potentials for the maxcliques are time-consuming, however in the join sampling problem (especially with online AQP), the response time is very important. We will talk about the complexity of the junction tree creation in Chapter 4. So, instead of searching for the best junction tree and calculating the product of clique factors in maxcliques, here, we just pull out the edges that make cycles.

The MRF graph G is simple with a small number of nodes. Any cycles in G can easily be broken by pulling out some part of the graph (usually just an edge). This operation divides the graph G into two graphs G^0 and G^1. G^0 is referred to as the main component, while G^1 is referred to as the remainder component. To remove an edge from G, we need to add two related nodes into G^1 as well. So, G^0 and G^1 have nodes in common. Note, in rare cases G^1 is also cyclic and its cycles need to be broken as well. This will create another graph G^2. G^1 and G^2 also have some shared nodes. This continues until all the components are acyclic. Hence, in the end, several MRF graphs for a query may emerge. PGM-Join sampler uses G^0 as the main graph and generates a sample point for G^0 nodes, then it uses other remainder components with its rejection mechanism to reject the generated sample point. If not rejected, the attributes in the remainder components are also added into the sample point.

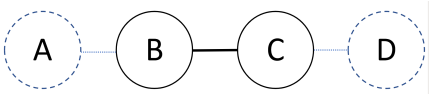


Figure 3.2: The MRF for the running example

$\psi_{D_1}(B)$

B	freq
b_0	3
b_1	3
b_3	2
b_4	4

$\psi_{D_2}(B, C)$

B	C	freq
b_0	c_0	2
b_1	c_0	3
b_2	c_1	3
b_3	c_2	1
b_4	c_3	2
b_4	c_4	1

$\psi_{D_3}(C)$

C	freq
c_1	4
c_2	4
c_3	2
c_4	2

Figure 3.3: Potentials for the skeleton of the graph in Figure 3.2

Authors in [Zhao et al., 2018b] showed that splitting the join into two parts and using a rejection mechanism can still yield a uniform sample. Our mechanism follows the same concepts but at an attribute level. Another difference is that they always make two parts and run the actual join on the pulled-out part of the query (which defeats the point), but we may build several parts and we do not run the actual join at all.

Figure 3.2 shows the MRF graph for our running example. A and D are the non-join attributes, and B and C form the skeleton. The factorized joint distribution for the skeleton of the MRF is proportionally equal to:

$$p(B, C) \propto \psi_{D_1}(B) \times \psi_{D_2}(B, C) \times \psi_{D3}(C) \tag{3.3}$$

here, the subscript for ψ indicates the table's name. Note that ψ is a potential function that includes frequencies, not a probability distribution.

The potentials for the skeleton after scanning the tables are shown in Figure 3.3.

For the non-join attributes, we build conditional potentials conditioned on the join attributes of the same table. These conditional potentials maintain the cumulative distribution of the dependent attributes (non-join attributes) for each distinct value in the independent attributes (join attributes). We use the cumulative distribution to make the sampling easier. The potentials for the non-skeleton attributes are shown in Figure 3.4. For example, in Table D_1, the frequencies of a_1 and a_2 related to b_1 in the join attribute B are 2 and 1, respectively. Therefore, the cumulative frequencies for a_1 and a_2 are 2 and 3, respectively. For other B values, there is just one A value, so they remain as they are.

The potentials depicted in Figure 3.3 will be utilized to conduct the inference

$\psi_{D_1}(A|B)$

B	A	freq
b_0	a_0	3
b_1	a_1	2
	a_2	3
b_3	a_3	2
b_4	a_3	4

$\psi_{D3}(D|C)$

C	D	freq
c_1	d_0	4
c_2	d_2	4
c_3	d_3	2
c_4	d_4	2

Figure 3.4: Potentials for the non-skeleton attributes in the graph in Figure 3.2

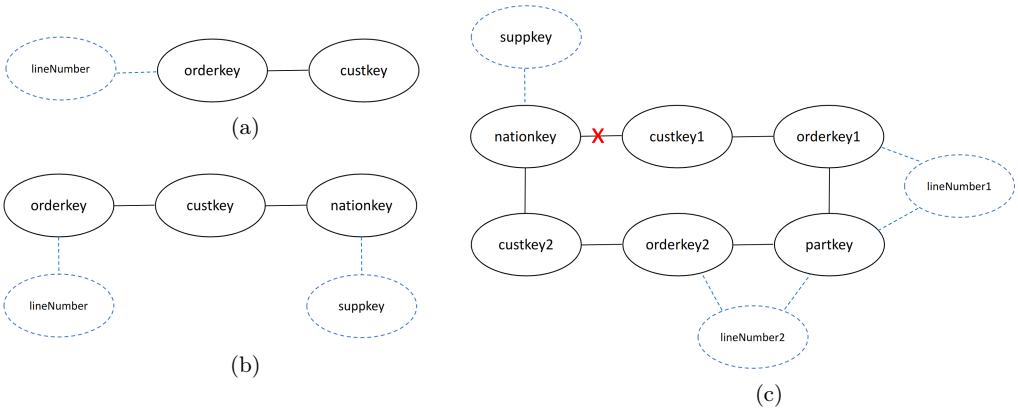


Figure 3.5: MRF graphs for TPC-H queries: a. MRF for Q3 b. MRF for QX and c. MRF for QY

and construct a uniform sample of the skeleton. The potentials in Figure 3.4 are used to add non-join attributes to the sample once the skeleton has been produced.

Figure 3.5 shows the MRFs for TPC-H queries Q3, QX and QY (cf. the experimental section for query specifics). The factorized distributions for the skeleton of Q3, QX and QY (p_{q3}, p_{qx} and p_{qy}, respectively) are listed below.

$$p_{q3}(\text{custkey,orderkey}) \propto \psi_c(\text{custkey}) \times \psi_o(\text{custkey, orderkey}) \times \psi_l(\text{orderkey}) \quad (3.4)$$

For non-join attributes $\psi_l(\text{lineNumber}|\text{orderkey}))$ is also needed.

$$p_{qx}(\text{nationkey,custkey,Orderkey}) \propto \psi_n(\text{nationkey}) \times \psi_s(\text{nationkey}) \times \psi_c(\text{custkey,nationkey}) \times \psi_o(\text{orderkey, custkey}) \times \psi_l(\text{orderkey}) \quad (3.5)$$

for non-join attributes $\psi_s(\text{suppkey}|\text{nationkey})$ and $\psi_l(\text{lineNumber}|\text{orderkey})$ are also needed.

$$p_{qy}(\text{nationkey,custkey1,custkey2,orderkey1, orderkey2,partkey}) \propto$$
$$\psi_{c1}(\text{nationkey,custkey1}) \times \psi_{c2}(\text{nationkey,custkey2}) \times$$
$$\psi_{o1}(\text{custkey1,orderkey1}) \times \psi_{o2}(\text{custkey2,orderkey2}) \times$$
$$\psi_{l1}(\text{orderkey1,partkey}) \times \psi_{l2}(\text{orderkey2,partkey}) \quad (3.6)$$

QY is a cyclic query and the skeleton should be cyclic. In this case, one of the edges should be pulled out to make it acyclic (to be explained later). Also ψ_{l1}(lineNumber1|orderkey1,partkey) and ψ_{l2}(lineNumber2|orderkey2,partkey) are required to handle non-join attributes.

Complexity of the Graph Building To build each of the potentials we need just to scan each table once. Assuming N is the largest table in the join, the complexity of building the graphs for a join query is $O(N)$.

Once the MRF model for the provided join query is ready, the inference step begins preparing the necessary statistics (the sample generator) to generate uniform samples.

3.5 Inference Phase

Typically, the inference (e.g. VEA) is performed in PGMs to calculate a quantity (e.g. a marginal) over the factorized distribution in a dynamic programming manner, by eliminating the variables one-by-one from the leaves to the root, utilizing the sum-product operation.

We also utilize VEA to perform inference on the factorized joint distribution of our join queries; however, our goal is not merely to calculate a marginal, but rather to construct a uniform sample generator. The sample generator is then employed to produce the sample points. The inference is performed only on the skeleton, while the non-skeleton potentials are added directly to the sample generator. After the sample for join attributes is generated, the non-skeleton potentials are used to add the non-join attributes in the sample too.

If the MRF graph for a join query is not cyclic then we have a single G, otherwise we may have several remainder components $G^0, G^1, \ldots$. PGM-Join sampler runs inference on all the graphs of a join query. Recall that with cyclic queries, G^0 has some shared nodes with G^1; those shared nodes should be the root of G^1 because adding the nodes of G^1 is dependent on the all the shared nodes in G^0. G^1 and G^2 also have some shared nodes which should be assumed as the root for G^2, so on so

$\psi(B, C) \times \psi(C)$

B	C	freq
b_2	c_1	12
b_3	c_2	4
b_4	c_3	4
b_4	c_4	2

Figure 3.6: The result of the product in sum-product operation

$\phi_{\not C}(B)$

B	freq
b_2	12
b_3	4
b_4	6

Figure 3.7: The result of the sum-product operation to eliminate the variable C

forth.

For each graph, PGM-Join sampler begins eliminating nodes from the leaves to the root, one by one. This elimination order is denoted by the letter O and is determined in the same manner for all graphs of a join query. A sum-product operation is used in each variable elimination that is being performed. Our running example is a nice example of how the sum-product of the PGM-Join sampler may be used to remove a single variable from a factorized distribution. Assume the graph in Figure 3.2 with the factorized distribution for its skeleton in the Equation 3.3 and the actual exact potentials for the skeleton in Figure 3.3. Let's assume we want to eliminate C first from the skeleton. The sum-product operation has two steps. It first calculates the product of the all potentials that include C and then it sums out C from the result of the product and builds a new factor for B (the parent of C). Based on the potentials in Equation 3.3, the potentials that include C are $\psi_{D_2}(B, C)$ and $\psi_{D3}(C)$. The result of the product operation is shown in Figure 3.6.

Note that the entries with zero frequencies are ignored.

And after summing out C from the result of the product in Figure 3.6, the factor $\phi_{\not C}(B)$ (shown in Figure 3.7) is created.

The new potential created for B is notated with ϕ to make it more readable and to indicate that the factor is temporary and will not be kept after finishing the join processing. We usually refer to ϕs as factors, not potentials, but in essence both potentials and the factors are just frequency tables with random access to the entries.

During performing the sum-product operation, the PGM-Join sampler also calculates a conditional potential ($\psi(C|B)$). $\psi(C|B)$ is calculated based on the result of the product operation (shown in Figure 3.6). It is a conditional potential

that gives the frequencies of C given B over the distribution of all B's descendants' eliminated variables. The returned frequencies are cumulative per distinct value. For example, $\psi(C|B)$ is shown in Figure 3.8. The frequencies for c_3 and c_4 given b_4 are 4 and 2 respectively based on Figure 3.6, but the accumulative frequencies in $\psi(C|B)$ (shown in Figure 3.8) are 4 and 6, respectively. This helps to uniformly choose either c_3 or c_4 given that b_4 has already been selected. All conditional potentials resulting from each sum-product operation are added to the sample generator, and later they are used to generate the samples. Recall that the conditional potentials related to non-skeleton attributes have already been added into the sample generator.

$\psi(C|B)$

B	C	freq
b_2	c_1	12
b_3	c_2	4
b_4	c_3	4
	c_4	6

Figure 3.8: The conditional and cumulative potential added in the sample generator after eliminating C.

Algorithm 2 Building the uniform sample generator

For a given query graph G:
$V = \{v_1, v_2, ..., v_n\}$ is the set of variables in G.
O is a dictionary containing all $v \in V$ with the elimination order index.
P is a dictionary includes the parent of each variable $v \in V$.
$E = \{e_1, e_2, ..., e_{n-1}\}$ is the set of edges in G. Each e contains two nodes.
$\Psi = \{\psi_1, \psi_2, ..., \psi_m\}$ is the set of potentials in G per edge.
φ is the sample generator.

1: **procedure** BUILD_SAMPLE_GENERATOR
2: **for** $i = 1$ to $n - 1$ **do** ▷ The loop is run per variable
3: $v_i \leftarrow O[i]$ ▷ The current variable to eliminate
4: $\Psi' \leftarrow \{\psi_j \in \Psi | v_i \in e_j\}$ ▷ All the potentials which include v_i
5: $\Psi'' \leftarrow \Psi - \Psi'$ ▷ All the potentials which exclude v_i
6: $\psi_\alpha \leftarrow \prod_{\psi \in \Psi'} \psi$ ▷ Product of all the related potentials
7: $\psi(v_i|P[v_i]) \leftarrow conditionalize\_cumulative(\psi_\alpha, P[v_i])$
8: $\varphi \leftarrow \varphi \bigcup \psi(v_i|P[v_i])$
9: $\phi_\beta \leftarrow \sum_{v_i} \psi_\alpha$ ▷ Summing out v_i from the product
10: $\Psi \leftarrow \Psi'' \bigcup \{\phi_\beta\}$ ▷ The new Ψ includes potentials/factors without v_i
11: **end for**
12: $\phi_{root} = cumulative(\prod_{\psi \in \Psi} \psi)$ ▷ Unconditional-cumulative potential for the root(s)
13: $\varphi \leftarrow \varphi \bigcup \phi_{root}$
14: **end procedure**

A pseudo-code for the PGM-Join inference is provided in Algorithm 2. The algorithm eliminates all variables one by one (except the root) inside a for-loop statement (lines 2-11). In each loop, based on the elimination order O, one variable

v_i is chosen to be eliminated in line 3. Line 4 collects all potentials that include v_i. Ψ contains all the potentials initially, and Line 5 removes the potentials that include v_i from Ψ. Line 6 calculates the product of the potentials that include v_i and line 9 runs the sum-out operation on the product result. Line 7 builds the conditional potentials to be added in the sample generator φ at line 8. Line 10 adds Ψ'' and the new factor for the parent of v_i (generated by the sum-product operation) into Ψ. So, Ψ has all correct potentials with the exception of those associated with the eliminated variable.

After the for-loop statement, what remains in Ψ are the potentials for the root. If many potentials exist for the root, it is necessary to determine a single unconditional potential for the root. Because we wish to efficiently generate uniform samples, this potential should also be accumulative. After calculating the product of the root potentials and making it cumulative, the result is then added into the sample generator. The final potential of the root is not conditional and serves as the starting point for sample generation. The join size is determined by summing the root out. The join size is exact because the sum-product operations are applied on the exact potentials.

Example: Consider the query QX's skeleton in Figure 3.5 and its factorized distribution in Equation 3.5. The graph has 3 nodes in its skeleton. Assuming that *orderkey* is the root, the elimination order is $O = \{(1, \text{nationkey}), (2, \text{custkey}), (3, \text{orderkey})\}$. Equation 3.5 can be represented as below in order to eliminate the variables. The subscripts for the potentials show the table names.

$$\sum_{\text{orderkey}} \sum_{\text{custkey}} \sum_{\text{nationkey}} \psi_l(\text{orderkey}) \times \psi_o(\text{orderkey, custkey}) \times \psi_c(\text{custkey,nationkey}) \times \psi_n(\text{nationkey}) \times \psi_s(\text{nationkey}) \quad (3.7)$$

If we are to eliminate *nationkey* first, Equation 3.7 can be changed to Equation 3.8 based on distributive law. This is an important feature of VEA: instead of running sum-product on all potentials at once to eliminate *nationkey*, it considers only the potentials related to *nationkey* (the last three potentials).

$$\sum_{\text{orderkey}} \sum_{\text{custkey}} \psi_l(\text{orderkey}) \times \psi_o(\text{orderkey, custkey}) \times \sum_{\text{nationkey}} \psi_c(\text{custkey,nationkey}) \times \psi_n(\text{nationkey}) \times \psi_s(\text{nationkey}) \quad (3.8)$$

As shown in Equation 3.9, the result of the sum-product operation to eliminate the *nationkey* is a factor for *custkey*, and all three potentials related to *nationkey* are eliminated. The subscript for the factors that resulted from the variable eliminations shows the eliminated variables. In addition, a conditional potential ψ(nationkey|custkey) with cumulative distributions per distinct value of *custkey* is generated and added into the potential set of the sample generator.

$$\sum_{\text{orderkey}} \sum_{\text{custkey}} \psi_l(\text{orderkey}) \quad \times \quad \psi_o(\text{orderkey, custkey}) \quad \times \quad \phi_{\cancel{\text{nationkey}}}(\text{custkey}) \tag{3.9}$$

Now it is time to eliminate *custkey*. The product of the last two potentials that include *custkey* is calculated, and the *custkey* variable is then summed from the product result, as shown in Equation 3.10.

$$\sum_{\text{orderkey}} \psi_l(\text{orderkey}) \times \phi_{\cancel{\text{nationkey,custkey}}}(\text{orderkey}) \tag{3.10}$$

$\phi_{\cancel{\text{nationkey,custkey}}}$(orderkey) is the factor of *orderkey* after eliminating *nationkey* and *custkey*. It contains the frequencies of the *orderkey* values over the eliminated variables. In addition, a conditional potential ψ(custkey|orderkey) is added to the sample generator. ψ(custkey|orderkey) is a potential function that returns the frequency of *custkey* values given *orderkey* values over the eliminated part of the distribution starting from *custkey*.

Now, *orderkey* is the only variable remaining. The product of the potentials related to *orderkey* is calculated first, then the accumulative frequencies per distinct value is calculated. The result of the product is a factor $\phi_{\cancel{\text{nationkey,custkey}}}$(orderkey). Assume ϕ(orderkey) is the accumulative version of $\phi_{\cancel{\text{nationkey,custkey}}}$(orderkey) that returns the accumulative frequencies of *orderkey* over whole distribution (in the join result). The sum of the frequencies in the last factor will result in the exact join size.

Figure 3.9 depicts the sample generator for query QX. Note that the non-join attribute potentials ψ_s(suppkey|nationkey) and ψ_l(lineNumber|orderkey) are already in the sample generator. ϕ(orderkey) is the last factor resulted from the inference and is considered as the starting point for generating the samples.

This inference is run for all the graphs related to the join query. However, the roots of the remainder components should be those nodes that are shared among the graphs. The roots may have more than a node.

Complexity of the Inference Since we never encounter a cycle in the graphs when we run inference (all cycles are broken during the MRF construction

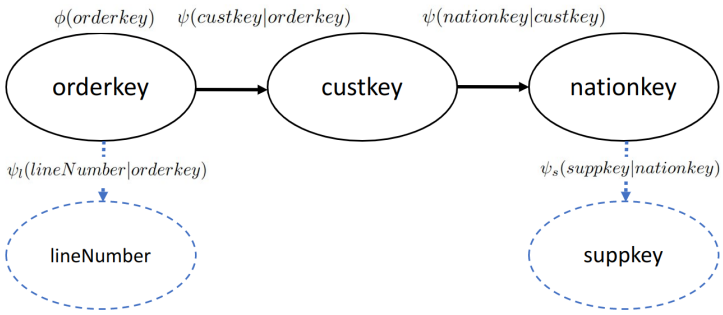


Figure 3.9: The sample generator for QX after inference

phase), the inference complexity for a join query is always $O(M)$, where M is the size of the largest potential in the MRF.

3.5.1 The Altered VEA vs. Standard VEA

By tweaking the VEA, a new VEA was created for the purpose of generating uniform samples of a factorized distribution. Hence, our goal with the altered VEA is not to identify a single marginal, but to construct a sample generator. The conditional-cumulative distributions per variable after eliminations form the sample generator.

Another difference is that to deal with cyclic queries, we do not create junction trees; rather, we extract some edges from the graph to make it acyclic and then use those pulled-out edges in our rejection process. This is because response time is crucial in AQP applications, particularly when sampling is involved. Therefore, it is unreasonable to devote a great deal of time to constructing the junction trees before beginning the sampling process. By breaking the cycles, the process of sample generation begins earlier and the AQP engines can offer the initial replies more quickly.

Furthermore, in our VEA, all non-join attributes are also treated as single nodes. These nodes are omitted during inference because they have no influence on the sample uniformity. This also helps us deal with cyclic graphs.

3.6 Sample Generation Phase

After inference, once the sample generator is prepared, the phase of sample generation begins. First, we describe how sample points for an acyclic query are constructed, and then we describe how we handle cyclic queries.

The MRF graph for an acyclic join query will always be acyclic after separating the non-join attributes as the non-skeleton from the graph. The acyclic graph

will be a tree if one of the nodes is assumed to be the root. As a result, we can create samples for any acyclic query without employing the rejection mechanism. Once the sample for the skeleton is generated, the non-join attributes are added to the sample too.

The PGM-Join sampler takes the sample generator produced by the inference and begins to generate samples from the root to the leaves in the reverse order of the variable elimination order.

Here, we describe how a single point sample is obtained. To obtain a sample of the size n, this process is repeated n times independently.

Upon completion of the inference phase, the root's unconditional-cumulative factor and the size of the join are available. A random number is generated between zero and the join size, and the associated distinct value s_0 in the root potential is selected using a binary search algorithm (very similar to inversion sampling method). s_0 is the uniformly chosen value for the root. Given s_0, s_1 for the next variable is chosen based on the corresponding conditional-cumulative potential in the sample generator. The process is continued until all the variables are added into the sample. It is known as ancestral sampling. Calculating the cumulative frequencies during the inference phase increases the sampling's efficiency since the cumulative frequencies are computed once and used several times, and the PGM-Join sampler does not need to calculate them for each sample point generation. Once the values for the join attributes (the skeleton) have been produced, the non-skeleton attributes are generated based on the potentials derived from scanning the tables.

In our example sample generator, as shown in Figure 3.9, a value for *orderkey* is chosen using ϕ(orderkey), then a value for *custkey* is chosen using ψ(custkey|orderkey), and finally a value for *nationkey* is added using ψ(nationkey|orderkey). Once the skeleton sample is generated, the values for the $lineNumber$ and *suppkey* variables are added based on their corresponding join attributes *orderkey* and *nationkey*, respectively.

So far, we have described how the sample points for a single acyclic MRF are generated. Next, we prove that the samples generated are uniform. In Section 3.6.2, the generation of samples for cyclic queries with multiple MRF graphs is described.

3.6.1 Proof of Sample Uniformity

Theorem 1. *Using the sample generator, ancestral sampling produces uniform samples.*

Proof. The frequencies in the root factor of the sample generator and the join size

are exact due to use of an exact inference algorithm and the fact that all primary potentials contain the exact frequencies after scanning the tables. Thus, the sample points generated for the root (based on the frequency of the root values over the join size) are uniform. Given the root value, subsequently, the other values are chosen *uniformly* by using the intermediate conditional factors in the sample generator. This uniformity is proved by the Global Markov Property (GMP).

Recall, GMP states that two sets of variables are mutually independent if a separating set of variables are observed. $x_A \perp\!\!\!\perp x_B \mid x_S$ where x_A and x_B are separated by x_S, in other words all the paths from x_A to x_B go through x_S.

Note that our query graph is always a tree, and each node separates its children from its parent. This means that based on GMP, the probabilities for any intermediate variables in the tree are only dependent on their children to the leaves if the parent of that intermediate variable is observed. The observed parent here has the separator role. In our running example, if *custkey* is observed then generating *nationkey* values is solely dependent on the frequencies of *nationkey* values conditioned on the *custkey* values across the sub-tree from *custkey* to the leaves (the corresponding factor in the sample generator is ψ(nationkey|custkey)), and is unaffected by *orderkey* values or any other variables. And since we have already calculated the exact ψ(nationkey|custkey) in our sample generator, it proves that PGM-Join sampler has (uses) the correct and exact probabilities to generate the uniform samples for the intermediate variables as well.

NB1: Based on GMP, if a node x_i has more than one child in a tree, the children are independent from each other once x_i is observed. Thus, each of the children can be generated independently.

NB2: According to GMP, all non-skeleton (non-join) attributes are dependent only on the join attributes from the same table and are independent of all other attributes. As a result, after the values for their parents (join attributes from the same table) are observed, the non-join attributes can be uniformly added to samples.

□

3.6.2 Sample Generation for Cyclic Queries

For a cyclic join query, more than one MRF is built. One of them is the main graph, and the others are the remainder components, which are used in the rejection mechanism to make the samples uniform. These MRF graphs have shared nodes, and these shared nodes are the roots for the remainder components. Each of the graphs has an elimination order, and the inference is done on all the MRFs according to that elimination order. During inference, a sample generator is built per MRF. The

Algorithm 3 Generating a sample point uniformly and independently

$i = \{0, ..., m\}$ where m is the number of MRF graphs for the given join query
G^i is the ith graph. G^0 is the main graph and others are the remainder components.
φ^i is the sample generator for G^i
γ^i contains the root nodes for the ith graph. For $i > 0$, γ^i contains the shared nodes between G^{i-1} and G^i
Υ^i is the potential for the root (γ^i) of G^i in φ^i
ω^i is the largest frequency in Υ^i
$\mathcal{S}$ is a sample point and $\mathcal{S}[\gamma^i]$ is the chosen value for the root of G^i

procedure UNIFORM_SAMPLER
  while (true) **do** ▷ keeps on until an unrejected sample point is reached
    $\mathcal{S} \leftarrow$ generate a sample point for G^0 nodes by using the ancestral sampling on φ^0
    i $\leftarrow 1$
    rejected $\leftarrow$ false
    while ($i < m$ and rejected=false) **do**
      if ($\mathcal{S}[\gamma^i] \in \Upsilon^i$ and $\omega^i > 0$) **then**
        $r \leftarrow rnd(0, 1)$
        if ($r < \frac{\Upsilon^i[\mathcal{S}[\gamma^i]]}{\omega^i}$) **then** ▷ Rejects with the probability of $1 - \frac{\Upsilon^i[\mathcal{S}[\gamma^i]]}{\omega^i}$
          add nodes of G^i into $\mathcal{S}$ using ancestral sampling on φ^i
          i++
        else
          rejected $\leftarrow$ true
        end if
      else
        rejected $\leftarrow$ true
      end if
    end while
    if rejected <> true **then**
      return $\mathcal{S}$ and break the While statement
    end if
  end while
end procedure

PGM-Join sampler starts with the main graph and generates the samples by using the ancestral sampling method (explained in Section 3.6) with the reverse order of the elimination order. Once the sample point for the main MRF is generated, the root potentials of the remainder components are used to reject the sample point. If the sample point is not rejected, the values for the other nodes in the remainder components are added to the sample point with the same ancestral sampling method. Algorithm 3 uniformly generates a single sample point for a cyclic join query. The next sample point is drawn independently of all previously obtained samples. The outer $While$ statement is a loop that is not terminated until a sample point that was not rejected is reached. Line 3 generates a sample point for the nodes in the main graph in a way that is explained in Section 3.6. In the inner $While$ statement, the generated sample point is examined according to all the remainder components to check whether the sample point should be rejected or not. Most of the time, there is just a single remainder component with a small number of nodes because the number of involved attributes in join queries is typically not excessively large. It is however feasible to have many remainder components. For now, assume there is one remainder component and the graphs are G^0 and G^1. The sample point generated for the main graph G^0 is rejected if the values for the shared nodes do not exist in the unconditional factor of the root in the sampler generator of the remainder component (G^1). If not rejected yet, in line 9, the sample point is checked again to be rejected with the probability of $1-\frac{\Upsilon[\mathcal{S}[\gamma]]}{\omega}$ where Υ is the root factor in the sample generator of the remainder component, and $\mathcal{S}[\gamma]]$ returns the values in the generated sample point for the shared nodes between the main graph and the remainder component. ω is the largest frequency in Υ. Authors in [Zhao et al., 2018b] showed that this rejection mechanism makes the random samples uniform. If the sample point is not rejected in line 9, the nodes in the remainder component are added to the sample point (in line 10). If there is more than one remainder component, the same process is followed in the inner $While$ statement for other remainder components. It may be easier to think of this as, G^1 is the main graph and G^2 is the remainder component. In lines 19-21, it is determined whether or not the values for all MRF graphs are generated. If not, the outer $While$ statement continues the procedure until a complete sample point is reached.

3.7 Experimental Evaluation

We study the PGM-Join sampler's performance against the Exact Weights (EW) and Online Exploration (OE) methods in [Zhao et al., 2018b]. In the context of acyclic

queries, we refer to the PGM-Join sampler as $PGMJoins$, while in the context of cyclic queries, we refer to it as rej_PGMs. We use the queries in [Zhao et al., 2018b] plus two many-to-many join querys on TPC-DS, and a join query from JOB.

We report separately the building, inference and sample generation times. All times are reported in seconds.

All algorithms require a "preparation" phase. $PGMJoins$ builds the graph. EW and OE build indices per join attribute. All of these require $O(N)$ time, where N is the size of the largest table. As the preparation times can be shared among several queries, both we and [Zhao et al., 2018b] separate it from query response time.

Query processing time in $PGMJoins$ is inference time plus sample generation time. EW and OE spend time on dynamic programming (DP) and sample generation. For ease of comparison, here, we call DP time as inference time.

The code and use instructions are publicly available in [1]. Our code operates directly on the raw data from CSV files.

Queries: The tested join queries do not include any selection operators (filters).

TPC-H Benchmark. The TPC-H benchmark data with different scaling factors (1 and 10) is used. We use three queries Q3, QX and QY. The queries have been taken from [Zhao et al., 2018b].

Q3: A foreign key join.

```
SELECT c.custkey, o.orderkey, l.linenumber
FROM customer c, orders o, lineitem l
WHERE c.custkey = o.custkey AND l.orderkey = o.orderkey;
```

QX: An acyclic join.

```
SELECT n.nationkey,s.suppkey,c.custkey,o.orderkey,l.lnumber
FROM nation n,supplier s,customer c,orders o,lineitem l
WHERE n.nationkey=s.nationkey AND s.nationkey=c.nationkey
AND c.custkey = o.custkey AND o.orderkey = l.orderkey;
```

QY: A cyclic join.

```
SELECT l1.linenumber,o1.orderkey,c1.custkey,l2.linenumber,
o2.orderkey,s.suppkey,c2.custkey
FROM lineitem l1, orders o1, customer c1, lineitem l2,
orders o2, customer c2, supplier s
WHERE l1.orderkey = o1.orderkey AND o1.custkey = c1.custkey
AND l1.partkey = l2.partkey AND l2.orderkey = o2.orderkey
AND o2.custkey = c2.custkey AND c1.nationkey = s.nationkey
AND s.nationkey = c2.nationkey;
```

[1] https://github.com/shanghoosh1/PGMJoins

Join Order Benchmark (JOB) queries. The data comes from IMDB. Queries are taken from [Leis et al., 2015]. Here, we use query Q16b.

```
SELECT an.person_id, t.id
FROM an, ci, cn, k, mc, mk, n, t
WHERE an.person_id = n.id AND n.id = ci.person_id
AND ci.movie_id = t.id AND t.id = mk.movie_id
AND mk.keyword_id = k.id AND t.id = mc.movie_id
AND mc.company_id = cn.id AND an.person_id = ci.person_id
AND ci.movie_id = mc.movie_id AND ci.movie_id = mk.movie_id
AND mc.movie_id = mk.movie_id;
```

Twitter Data and Queries. This data refers to follower links and profiles of twitter users – also used in [Cha et al., 2010; Zhao et al., 2018b]. A tuple in the data represents a connection with source and destination. We scale up the twitter data (the users dataset) 10 times then uniformly sample from it with different sizes 1x, 2x, 4x, 6x, 8x and 10x. Here, the number of distinct values increase slightly with different scaling factors. The popular-users dataset remains unchanged. We use queries QT and QS on this data as done in [Zhao et al., 2018b].

QT: A cyclic join.

```
SELECT * FROM pop-user A, twitter-user B, twitter-user C
WHERE A.dst = B.src AND B.dst = C.src AND C.dst = A.src;
```

QS: A cyclic join.

```
SELECT * FROM pop-user A,twitter-user B, twitter-user C
,twitter-user D
WHERE A.dst = B.src AND B.dst = C.src AND C.dst = D.src
AND D.dst = A.src;
```

TPC-DS Benchmark To study many-to-many joins we use the TPC-DS database and create meaningful fact-table join queries. The queries are as follows:

QDS1: An acyclic query.

```
SELECT SS.promo_sk, SR.sr_reason_sk ,WR.wr_reason_sk,
CR.cr_raeson_sk ,CS.c_page_sk, WS.web_page_sk ,SS.store_sk
FROM SS, SR, WS, WR, CS, CR
WHERE SS.promo_sk = WS.promo_sk AND WS.promo_sk =
CS.promo_sk AND SS.store_sk=SR.store_sk AND WS.web_page_sk
= WR.web_page_sk AND CS.c_page_sk=CR.c_page_sk;
```

QDS2: A cyclic query.

```
SELECT CS.c_page_sk ,WS.web_page_sk ,WS.promo_sk ,
WR.wr_reason_sk
FROM WS, WR, CS, CR
WHERE WS.promo_sk = CS.promo_sk AND WS.web_page_sk =
```

	Q3	QX	QY	QDS1	QDS2	Q16b	QT	QS
SF1 (1X)	15.6	15.6	37.9	4.1	1.1	20.8	169.9	192.2
SF10 (10X)	224.2	243.4	575.1	25.4	10.7	168.9	430.2	634.1

Table 3.1: Building time in seconds for $PGMJoins$

```
WR.web_page_sk AND CS.c_page_sk = CR.c_page_sk AND
WR.wr_reason_sk = CR.cr_reason_sk;
```

Setup We use one core of an Intel(R) Xeon(R) CPU E5-2660 with 2.60GHz on a server with 64GB RAM. Graphs are memory-resident; there is no I/O during inference or sample generation.

The building times of the PGM-Join sampler for all queries with SF1 and SF10 are shown in Table 3.1.

3.7.1 TPC-H Experiments

Figure 3.10 shows the results for TPC-H queries Q3, QX and QY with different scaling factors to generate a sample of one million. Blue parts relate to inference, and orange to generation times. $PGMJoins$ is faster than OE and EW by 2X and 3X for Q3 and QX and 30-40% faster for QY, for SF=10. Since OE uses random walks, the result of the preparing step is not the exact, and that is why it is fast in its DP step, but slow in the generation phase because it needs a lot of rejections. EW and $PGMJoins$ provide the exact weights at the end of the inference step, making them faster in the generation phase.

QY is cyclic and more challenging. Our $rejPGMs$ randomly selects one of the edges to pull out and make the graph acyclic, but OE and EW apply an optimization process to find out which one of the *tables* should be pulled out. If there is more than one table to be pulled out, the exact join is carried out over the pulled-out tables which destroys the goals of avoiding exact joins. The superiority of $rejPGMs$ for cyclic queries is showcased with QY. The generating time of $rejPGMs$ for QY is smaller owing to the lower rejection rate as it pulls out an edge instead of a whole table. When entire tables are extracted by [Zhao et al., 2018b] from a cyclic query, the actual join is conducted on the tables, which defeats the point. We do not want to conduct the actual join. Also, the more attributes you pull out, the more rejections you encounter.

Figure 3.11 shows the overall time to generate different-size samples for Q3, QX, and QY, with SF = 10. $PGMJoins$ is much faster across the board. The efficiency of OE becomes worse when the sample size is increased because it requires more rejections.

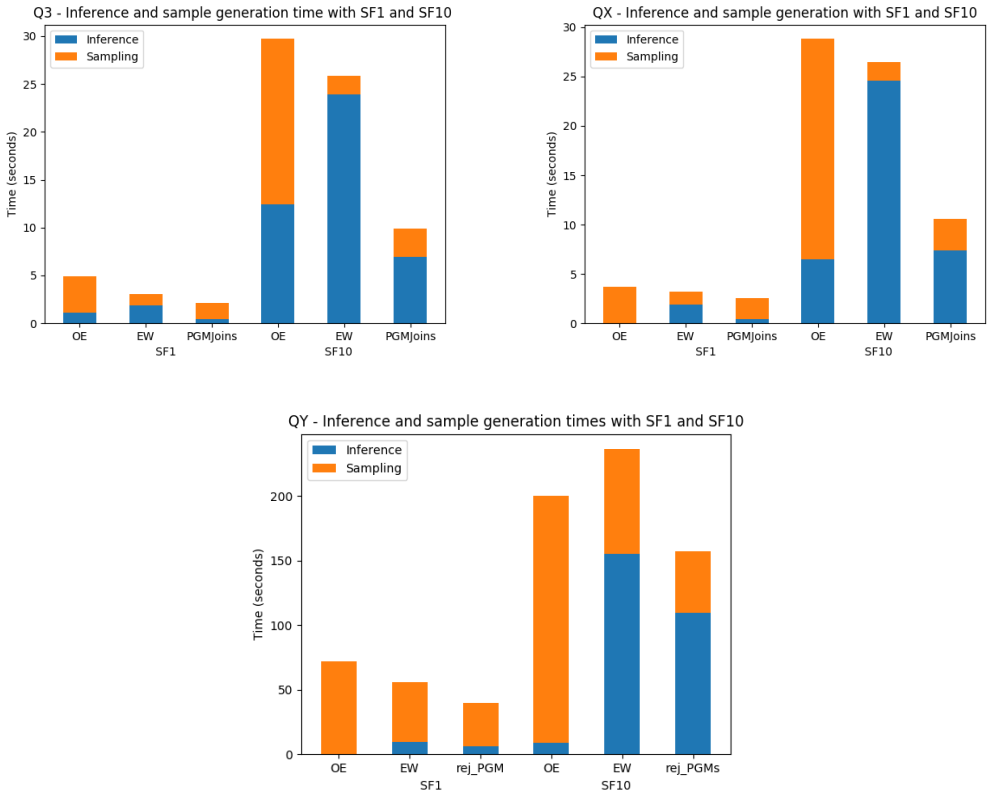


Figure 3.10: Query processing time to generate a sample of 1 million for Q3, QX and QY (SF1 and SF10)

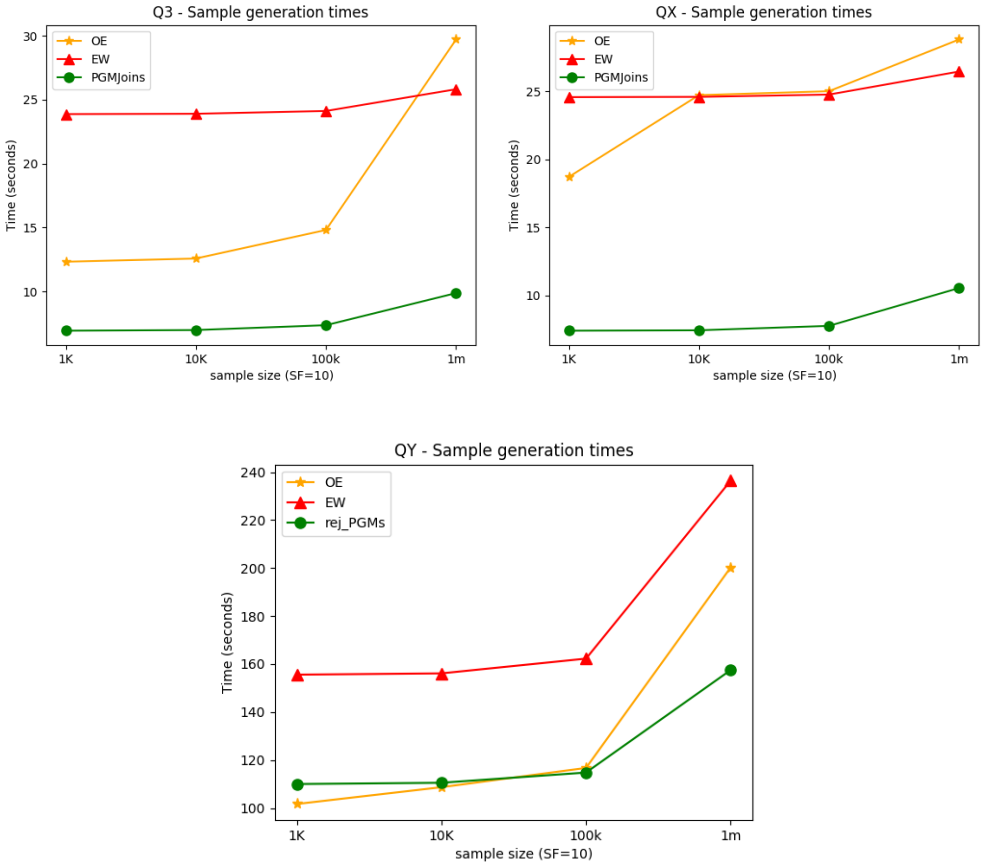


Figure 3.11: Query processing time to generate 1K, 10K, 100K, and 1M sample for Q3, QX, and QY on SF10

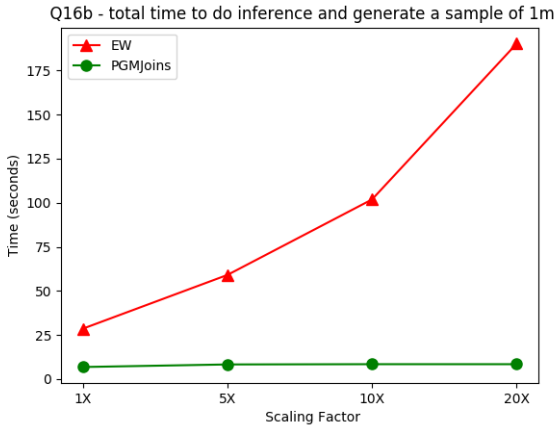


Figure 3.12: Query processing time to generate a sample of 1 million for Q16b of JOB

3.7.2 JOB Experiments

We tested *EW* and *PGMJoins* on 1X, 5X, 10X, and 20X scaled-up (replicated) JOB data without raising the number of distinct values. The code for *OE* did not run on this query and the authors claimed that the performance of *OE* would be very similar to *EW*.

Figure 3.12 depicts the time required to infer and construct a sample of size 1 million using different SFs. It also demonstrates that scaling the data has little effect on the performance of *PGMJoins*, but has a significant impact on *EW*. This is due to the fact that *PGMJoins* concentrates on the frequency of distinct attribute values, but not tuples. As data amounts increase, the gap between the algorithms increases. *PGMJoins* is ∼23 times faster than *EW* for 20X data.

3.7.3 Twitter Experiments

Both [Cha et al., 2010] and [Zhao et al., 2018b] replicate the twitter-user file to test scalability. We also do this, not only to show scalability, but also to show that our *PGMJoins* becomes better when there are more repetitions in join attribute values. So, we increase 10x the file twitter-users then sample from it with 1x, 2x, 4x, 6x, 8x and 10x. We do not change the popular-users file.

Figure 3.13 shows the inference and generation times for QT with a sample size of 1 million for different SFs. It shows that by increasing the frequency values, the comparative gains of *PGMJoins* improve. The reason for this improvement is because *PGMJoins* learns the distribution of the join result first, and the repeated

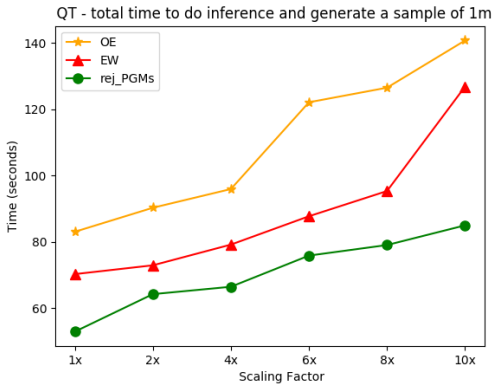


Figure 3.13: Query processing time to generate a sample of 1 million for QT

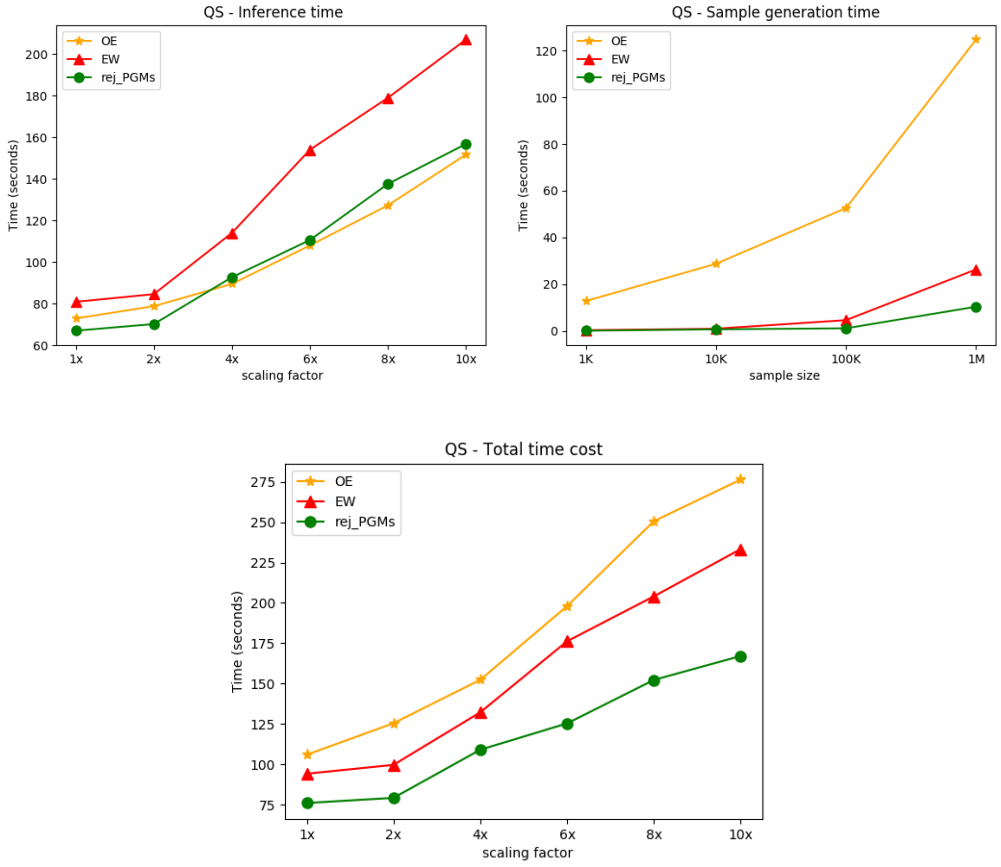


Figure 3.14: Times for QS Inference, Sample generation and total cost to generate 1M-sample

values do not affect its performance (only the frequencies become larger). However, EW scans the large tables with lots of repetitions to calculate the weights of join tuples.

Figure 3.14.a shows that the efficiency of the inference phase follows the same trend as QT by scaling up the data for QS.

Figure 3.14.b shows the generating time with different sample sizes with SF10. The improvement is due to the lower rejection rate, since as mentioned before EW pulls out tables. This makes the DP phase faster with a slower generation phase. Figure 3.14.c shows the overall time for DP (Inference) and generation phases to generate a sample of a million for QS.

3.7.4 TPC-DS Experiments

Calculating the full join of TPC-H queries posed by [Zhao et al., 2018b] is very expensive as shown in their experiments. For instance, the authors say, *"As a result, the full join becomes too expensive to complete in a day"*, as shown in Figure 3b of the paper [Zhao et al., 2018b]. TPC-H and twitter queries are computationally expensive with barely small enough join sizes to store in external storage, but TPC-DS queries are not only computationally expensive, but they are also expensive storage-wise. For example, query QDS1 has more than $5 * 10^{27}$ tuples in the join result, and $PGMJoins$ can generate a uniform sample of 10 million in just a few seconds.

Table 3.2 shows that $PGMJoins$ is dramatically better (up to 28X) than EW in both cyclic and acyclic queries and both scale factors. (OE results are not included for TPCDS queries the same as Q16b on JOB– the authors of [Zhao et al., 2018b] have not provided the code for them. They expected results to be similar to EW).

In inference time for QDS1, $PGMJoins$ is ∼20X and ∼85X times faster than EW, with scaling factors 1 and 10 respectively. For QDS2, it is ∼7X and ∼30X times faster with different scaling factors. It is expected with larger scale factors the difference will increase more. This difference is because $PGMJoins$ focus on the frequencies of the distinct values, and the number of distinct values in our TPC-DS queries are small with lots of repetitions. With respect to sample generation time for a sample size of 10^6, the PGM-Join sampler is slightly better than others on QDS1, but on QDS2 its generation time is ∼16X and ∼28X times faster than EW for scaling factors 1 and 10, respectively. This difference also exists with different sample sizes. One reason for that, as mentioned, is that our method pulls out the edges that make cycles, but not tables.

The take-away message is that across all studied scenarios sometimes EW is

TPCDS	QDS1				QDS2			
	SF=1		SF=10		SF=1		SF=10	
	EW	*PGMJoins*	*EW*	*PGMJoins*	*EW*	rej_PGMs	*EW*	rej_PGMs
Inference	0.720	0.037	9.330	0.109	0.270	0.038	3.640	0.120
1k	0.003	0.002	0.003	0.002	0.038	0.002	0.053	0.002
10k	0.022	0.015	0.025	0.016	0.386	0.020	0.521	0.018
100k	0.207	0.169	0.228	0.180	3.795	0.240	5.220	0.180
1m	2.077	1.846	2.286	1.860	37.887	2.275	52.390	1.917
10m	20.782	17.542	22.866	20.141	378.474	21.891	524.412	19.843

Table 3.2: TPCDS query response time results in seconds

better (worse) than OE. Nonetheless, $PGMJoins$ is never outperformed by OE nor EW and often achieves strong improvements.

3.7.5 Uniformity Test

$PGMJoins$ generates a uniform sample of the join as it uses exact inference on exact frequencies. To show the uniformity of the samples, smaller tables are used so that the exact join result could be computed readily. After the inference phase, the exact join size is computed. If we need, say, a 1%percent sample, this can then be generated.

The KS-test [Massey Jr, 1951] calculates the maximum difference between the CDF of distinct tuples in the exact join results and the CDF of distinct tuples in the $PGMJoins$ results. It reveals whether the generated sample is a uniform sample of the exact join result. The null hypothesis is that the sample is a uniform sample. The two-sample KS-test is used, which is calculated as follows:

$$D_{n,m} = \mathrm{Sup}_x |F_{1,n} - F_{2,n}| \tag{3.11}$$

where $F_{1,n}$ and $F_{2,m}$ are the CDF functions of the first and the second samples. First sample should be the exact join result[2]. Sup_x is the supremum function. With significance level α the null hypothesis is rejected if $D_{n,m} > C(\alpha)\sqrt{\frac{n+m}{n\times m}}$, and $C(\alpha)$ is calculated from $\sqrt{-\frac{1}{2}\ln\frac{\alpha}{2}}$ which gives us the critical value. To use the KS-test we need to order the sample tuples in the same way as in the join result and give an index number for each distinct tuple in both tables. If $D_{n,m}$ is less than the critical value, we conclude that the null hypothesis holds (i.e., the sample is a uniform sample of the exact join).

The full results of the ks-test (ks-statistic and critical values) on different queries are illustrated in Table 3.3. Table 3.3 shows that all of the samples for all

[2]While the exact join result and the join sample are strictly speaking not a sample from a continuous distribution as required by the KS-test, they behave similarly enough that no practically noticeable effects can be observed in this context.

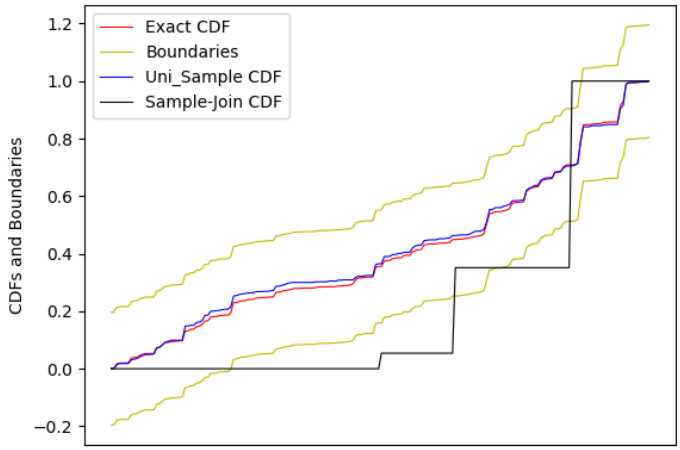


Figure 3.15: CDFs of KS-tests for Q16b

KS-test	*PGMJoins*		sample-then-join	
	CV	KS	CV	KS
Q3	0.430	0.005	0.615	0.242
QX	0.036	0.005	0.050	0.131
QY	0.147	0.002	0.208	0.633
Q16b	0.196	0.024	0.26	0.394
QT	0.396	0.004	0.558	0.396
QS	0.242	0.020	0.343	0.323
QDS1	0.052	0.048	0.072	0.317
QDS2	0.056	0.049	0.079	0.082

Table 3.3: KS-test results for all queries

the queries are not rejected by the test and they are uniform. Figure 3.15 is an example to show how ks-test works. Figure 3.15 compares the CDFs of indexed tuples for the sample generated by *PGMJoins* for Q16b against the sample-then-join method and exact join result. The Y axis are CDFs and X axis are the tuple indices. The red line is for the ground truth exact join result, the blue line is for the sample generated by *PGMJoins*, the black one is for the sample-then-join method and yellow lines are the boundaries for the sample generated by *PGMJoins*. If the blue line crosses the yellow lines, it means that our generated sample is not uniform with significance level of 0.01.

3.8 Summary

This study focused on uniform sampling over joins, contributing a principled solution. The solution, PGM-Join sampler, generates uniform samples efficiently and independently. It leverages PGM principles and shows optimizations for deriving the PGM graph structure and for running inference over the graph. It proposes a modified version of the message passing algorithm, and a new way to handle cyclic joins. Generated samples are proved to be uniform, leveraging PGM fundamentals. The experimental results showed that the PGM-Join sampler is consistently faster than competitors, for both cyclic and acyclic joins.

Chapter 4

A New Physical Join Algorithm with PGMs

Physical n-way join operations (especially when involving many-to-many joins) are known to be very time- and resource-consuming. At large scales, with respect to table and join-result sizes, state of the art PJAs, even fail to produce any answer given reasonable resource and time constraints. In this chapter, we introduce a scalable PJA for equi-join queries, coined the Graphical Join (GJ), as it leverages PGMs. GJ deals with both fundamental problems facing PJAs, namely Unneeded Intermediate Results (UIRs) and redundancy in the join result. The novelties lie in that: (i) GJ leverages PGMs, tweaking known exact inference algorithms in order to derive a statistical summary of the join result, which can be optionally stored and retrieved for reuse very efficiently; (ii) GJ offers algorithms that desummarize the summary, enumerating thus all join result tuples. The proposed enumeration introduces performance gains due to its columnar orientation, producing result values on a per attribute basis, independently of other attribute values. (iii) GJ's summary is a type of Run-Length Encoding (RLE) over join results, but here for the first time it is shown how to generate it without first generating the join result. (iv) It is shown for the first time that a join algorithm, like GJ, which produces the above join-result summary and then desummarizes it, can be highly competitive against binary-join plans and WOJAs in both time and space. Comprehensive experimentation is undertaken with join queries from the JOB, TPCDS, and lastFM data and queries, comparing GJ against PostgreSQL and MonetDB and a state of the art WOJA implemented within the Umbra RDBMS. We also implement a parallel version of the GJ and compare it against the parallel versions of the other competitors. In addition, we compare GJ against FDB in calculating the aggregations. The results

show that GJ can often achieve dramatic performance gains in space and time and also exemplify under which circumstances GJ is more appropriate.

4.1 Motivation

PJAs are concerned with generating (enumerating) all tuples in the join result satisfying the join predicates – other predicates and operations (e.g., aggregations, group-by, etc.) are not of interest.

PJAs are known to be expensive operations, especially for multi-way joins with many-to-many relationships (involving joins on non-key attributes) which are frequent in analytics. In these cases, the state of the art approaches suffer in both time and space. The main source of inefficiency stems from *algorithms exerting redundant effort*. First, some produce intermediate join results (after binary joins) which contain tuples which do not make it in the final n-way join result. Second, even the final join result (tuples) typically has a lot of redundancy which introduces space and time inefficiencies. One can distinguish two use cases for PJAs. In "compute-and-forget", the join result is to be computed once, typically in memory (and then likely forgotten). In "compute-and-reuse", a join result is to be stored on disk (e.g., in order to be re-used in the future). In both cases computing unnecessary intermediate tuples and/or storing, and retrieving redundant result tuples incurs significant overheads.

Despite the large attention PJAs have received, there is still lots to be done in order to minimize both time and space overheads and be applicable in both the compute-and-forget and compute-and-reuse scenarios. Specifically, the most popular PJAs (such as the Nested-loop/Hash/Sort-merge Join algorithms and their derivatives [Mishra and Eich, 1992; DeWitt et al., 1993; Dittrich et al., 2002; Graefe, 1994; Kitsuregawa et al., 1983; DeWitt et al., 1984]) occupy a significant part of the code base in all RDBMS products. These algorithms become inefficient when handling multi-way joins with at least one many-to-many relationship in the joined tables. This inefficiency stems from their binary nature: they join two tables at a time [Avnur and Hellerstein, 2000; Zhang et al., 2012] and this yields intermediate join results that include tuples that are not included in the final result. Avoiding this problem has been the source of many investigations. The Yannakakis algorithm (YA) described in [Yannakakis, 1981; Bagan et al., 2007] can be thought of as a PJA that handles UIR, especially on acyclic queries. YA employs semi-join reduction operations to remove dangling tuples from tables. Later, Ngo et al. [Ngo et al., 2012, 2018] introduced a new "worst case optimal" join algorithm (WOJA) which

avoids generating UIR. A WOJA behaves similarly to a multi-way sort-merge join algorithm in that it does not generate UIR for acyclic queries and minimizes UIR for cyclic queries. WOJAs are the cutting-edge solution for UIR. Nonetheless, even WOJA algorithms do not fully address space concerns, as redundancy exists even in the final result. This is especially important in the compute-and-reuse case where extra space directly translates to extra time (to store and retrieve this redundant data).

A related research thread addressed both UIR and redundancy in the result offering algorithms and analyses for computing analytics (e.g., aggregations and regressions) over the join result. Factorized Data Bases (FDB) [Olteanu and Schleich, 2016] and the solutions introduced in [Abo Khamis et al., 2016] for the Functional Aggregate Query (FAQ) problem are leading exemplars of this thread. We will refer to those solutions in [Abo Khamis et al., 2016] as "FAQ" in this thesis. FAQ works were never intended as a PJA – they skip join result generation and calculate the aggregations directly without computing the join. They also target the compute-and-forget scenario only. Algorithms for enumerating the join result tuples do exist to be used for FDB and FAQ. However, there is much room for improvement as the existing join result-generating algorithms are all based on row-by-row enumeration, this method is inefficient in comparison to columnar generation.

With this work, we show how to leverage PGMs to develop factorized distributions of the join result (as opposed to factorized join-result data) and how to utilize these to develop PJAs that define the new state-of-the-art for compute-and-reuse and compute-and-forget scenarios.

FDB and FAQ research are important, as there are many applications for efficient analytics over joins. However, this does not take away from the importance of PJA efficiency, as producing the actual join result is very important, e.g., for training any kind of models (not just simple regressions), deriving different types of samples, facilitating query pipelines, etc., or for migrating from one database to another (for example, converting structured data to unstructured data in BigQuery) [Melnik et al., 2010; Gubarev et al., 2020]. As a result, two new side problems emerge, which this work targets: for a given factorized distribution, how can one make join result tuple enumeration much faster and what is a better mechanism for efficiently/simply supporting the compute-and-reuse scenario (e.g., storing and retrieving the join result).

In summary, our goal is to introduce a PJA that deals with both UIR and redundancy problems, introduce a statistical join summary that is compatible with RDBMSs, facilitating an efficient mechanism to store/retrieve the summary, and

D_1

...	**A**	**B**	...
...	a_0	b_0	...
...	a_0	b_0	...
...	a_0	b_0	...
...	a_1	b_1	...
...	a_1	b_1	...
...	a_2	b_1	...
...	a_3	b_3	...
...	a_3	b_3	...
...	a_3	b_4	...
...	a_3	b_4	...
...	a_3	b_4	...
...	a_3	b_4	...

D_2

...	**B**	**C**	...
...	b_0	c_0	...
...	b_0	c_0	...
...	b_1	c_0	...
...	b_1	c_0	...
...	b_1	c_0	...
...	b_2	c_1	...
...	b_2	c_1	...
...	b_2	c_1	...
...	b_3	c_2	...
...	b_4	c_3	...
...	b_4	c_3	...
...	b_4	c_4	...

D_3

...	**C**	**D**	...
...	c_1	d_0	...
...	c_1	d_0	...
...	c_1	d_0	...
...	c_1	d_0	...
...	c_2	d_2	...
...	c_2	d_2	...
...	c_2	d_2	...
...	c_2	d_2	...
...	c_3	d_3	...
...	c_3	d_3	...
...	c_4	d_4	...
...	c_4	d_4	...

Figure 4.1: The running example tables (the same tables from Chapter 1)

introduce a new way to enumerate join tuples in a columnar way, leveraging this summary. And, perhaps most importantly, to show that the above approach to PJA can be highly competitive vis-a-vis the current state of the art PJA.

4.2 The Problems and Goals

More formally, the sources of inefficiency with PJAs are as follows. Assume T_1, T_2, T_3 $, ..., T_n$ are the tables to be joined. Given a query execution plan consisting of binary joins, call $S_{tmp} = \{R_{\{1,2\}}, R_{\{1,2,3\}}, ... , R_{\{1,2,...,n-1\}}\}$ the set of all intermediate temporary tables. Any tuple $t \in R_t$ where $R_t \in S_{tmp}$ is called an intermediate tuple and if $t \in R_t$ but $t \notin R_{\{1,2,...,n\}}$, we call t a UIR. Generating UIR tuples is a key source of inefficiency. Especially when the size of any $R_t \in S_{tmp}$ becomes much larger than $R_{\{1,2,...,n\}}$.

Consider our running example from the introduction section (re-shown in Figure 4.1): If D_1 and D_2 are joined first, there will be 15 tuples with b_0 and b_1 which will be ignored when they are joined with D_3 because there is no c_0 in D_3. This makes binary join plans sub-optimal. Note that this sub-optimality still remains if the D_2 and D_3 are joined first because D_1 does not include b_2, so this is not a join-ordering problem.

Another inefficiency is the redundancy in the join result itself. Consider the join result of the running example in In Figure 4.2. Column A has one distinct value (a_3), repeated 32 times, while a_3 is repeated only 6 times in (normalized) D_1. Such redundancy leads to inefficient join algorithms (in time and space), especially in compute-and-reuse cases where results are to be stored and reloaded.

As mentioned, FDB and FAQ research have tried to address both UIR and redundancy problems. FDB and FAQ produce different types of factorized join result representations, but it is unknown how one can store and reuse such factorized

Algorithms	Physical Join	Specific for RDBMs	UIR	Redundancy	Storing method	RLE	Uniform Sampler	Columnar Enum.
Binary JAs	S-I-C	✓	×	×	Tuples	×	×	✓
WOJAs	S-I-C	Some	✓	×	Tuples	×	×	×
FAQ	S	Deductive DBMS	✓	✓	Unknown	Not Studied	Not Studied	×
FDB	S	✓	✓	✓	Unknown	×	×	×
Yannakakis	S-I-C	✓	✓	×	Tuples	×	×	×
PGM-Join	×	✓	✓	✓	Samples	Not Studied	✓	✓
GJ	S-I-C	✓	✓	✓	RLE	✓	✓	✓

Table 4.1: A summary of all related works. Abbreviations: S (studied and discussed), I (implemented), C (compared to the other physical JAs) and RLE (run-length encoding)

representations in RDBMSs efficiently. And, the enumeration algorithm outlined in both FDB and FAQ proceeds row by row, seemingly requiring the actual tables and indexes for efficiency as it jumps from attribute value to attribute value. Although there is no available implementation of this enumeration algorithm, it is apparent that the row-oriented result enumeration and the dependence on indexes are significant performance disadvantages.

4.3 Related Work

In the background section, we have already covered PJAs, WOJAs, analytics over joins such as FDB, FAQ, and join sampling approaches. In Table 4.1, we provide a comprehensive summary and comparison of related work.

All binary join algorithms can be regarded PJAs for RDBMs that have been thoroughly studied, Implemented, and Compared (S-I-C). They all store the results as tuples but are plagued by UIRs and redundancy inefficiencies. To generate RLE or uniform samples over joins, binary PJAs must first produce results. WOJAs and YA are similar to binary PJAs, but they also handle UIRs.

The FDB and FAQ enumeration methods have been discussed, but none of them have been implemented and compared to existing PJAs. FAQ and FDB both address UIRs and redundancy, but generate distinct types of representations. One join result representation by FAQ is the factorized distribution of the result using PGMs (this is the same for GJ), while one join result representation by FDB is the factorized join result (data, not distribution). Since neither FAQ nor FDB has a strategy to store/retrieve their result representations in a manner compatible with RDBMSs, they must first enumerate the tuples before storing them. As FAQ maintains a factorized distribution (with frequencies), it is able to construct an RLE or a uniform sample of the join result, but none of these has been examined or addressed (but FDB cannot because it does not know the frequencies in advance). FAQ and FDB cannot list the join tuples in a columnar manner.

The PGM-Join sampler is a sampler over joins that can provide columnar

Full join result

id	A	B	C	D
0	a_3	b_3	c_2	d_2
...	...	...	...	...
7	a_3	b_3	c_2	d_2
8	a_3	b_4	c_3	d_3
...	...	...	...	...
23	a_3	b_4	c_3	d_3
24	a_3	b_4	c_4	d_4
...	...	...	...	...
31	a_3	b_4	c_4	d_4

Summary of the join result

A	B	C	D
a_3,32	b_3,8	c_2,8	d_2,8
	b_4,24	c_3,16	d_3,16
		c_4,8	d_4,8

Figure 4.2: Join result and GFJS

sample points, however, it is not considered a PJA.

Overall, to the best of our knowledge, this work is the first to derive a fresh approach for n-way join query processing, putting forth its principled, PGM-inspired, summarization-desummarization approach, studying its performance in detail, comparing it against the state-of-the-art physical join processing approaches, and showing that (and when) it is a preferable approach in both compute-and-reuse and compute-and-forget scenarios. GJ is the first method for generating RLE-type summaries over joins without first computing the join results. This RLE is efficient and simple to store and retrieve in RDBMSs. And, although the asymptotic complexity of the join result generation is the same as that of FDB and FAQ's row-oriented enumeration algorithms, GJ can generate the join tuples in a columnar manner, which is more efficient in practice.

4.4 A Brief Overview of GJ Algorithm

For a given join query, GJ learns an exact MRF model from the data. The aim of GJ is to first derive the factorized distribution from the normalized tables. Learning the factorized distribution is accomplished by scanning each table just once and calculating the exact frequencies as explained in Chapter 3. The denormalization (join) in relational databases can be considered as the de-factorization in PGMs, but with the key difference that in PGMs what is defactorized is the distributions, not data. However, please note that storing, retrieving and defactorizing a factorized distribution is a complex task in a RDBMS. GJ's solution for this is to build a kind of RLE over join (called GFJS, to be defined below) from the factorized distribution, and store/retrieve/desummarize GFJS rather than the factorized distribution itself. GFJS, unlike factorized distributions, can be easily stored/retrieved in RDBMS and

desummarized in a columnar way. Below, we formally define our GFJS.

Definition 3 (Grouped Frequentist Join Summary (GFJS))**.** *GFJS is a result of the summarization over the grouped join result per column. It is formed by replacing the repeated values, v, with pairs $(v, freq)$ where v is the value and $freq$ is its frequency.*

One may view GFJS as follows. Calculate the full join result, and then sort it according to all columns in the output. Then, for each column, produce an RLE encoding of its values: starting from the top of each column, replace all repeated values v in the column with the pair of the value and its frequency, like $(v, freq)$. This will yield GFJS.

Figure 4.2 shows the GFJS for the join result of the running example join. The join result in that figure is already sorted, so per column in the join result we can start from the top and replace the repeated values with a pair of the value and its frequency. For column A, there is one distinct value (a_3) with frequency of 32. For column B, there are two distinct values, b_3 and b_4, with the frequencies of 8 and 24, respectively. The sum of all frequencies in different columns should be equal.

The whole process of GJ is shown in Figure 4.3. The user submits a join query. GJ builds a PGM (MRF) for the given query. Note if the graph is cyclic, the junction tree creation is applied on the graph to make it acyclic (unlike the PGM-Join sampler). The aim of junction tree creation is to find a tree of maxcliques where the size of the largest maxclique in that tree is minimum in comparison to all other possible trees of maxcliques. Once the junction tree with all potentials for the query is ready, GJ uses VEA to build the generative model for GFJS based on an elimination order (recall trees have always at least a perfect elimination order). During variable elimination, GJ prunes all MRF paths related to UIR using a dynamic programming based approach with time complexity of the size of the largest potential function in the model. Next, GJ uses Algorithm 5 to generate GFJS by using the GFJS generator. The generation process is done in the reverse order of the elimination order. Here, the RLE generation over joins without doing the actual join is accomplished. Note that GJ does not go through the paths related to UIR tuples, and this is how it achieves WOJA status. After GFJS generation, it can be stored for later use. In the end, GFJS is desummarized columnarly whenever the full join result is needed. Desummarization is straightforward.

4.5 Building MRF Graphs

As explained before, the structure and the factors of the PGM for a join query are already available in the relational databases and we just need to scan the tables

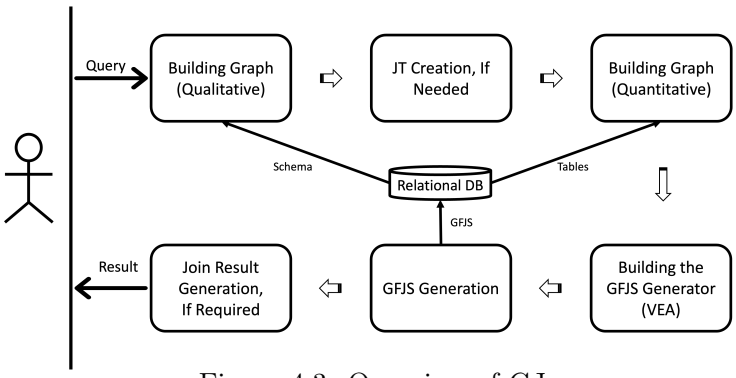


Figure 4.3: Overview of GJ

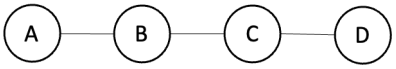


Figure 4.4: Graph for the example join

once and find all the potentials. Here, with a few exceptions, the mapping of the PJA problem to PGMs is similar to that of the PGM-Join sampler: i) the non-join attributes are not separated from the skeleton in this case, and ii) the junction tree creation is used to deal with cyclic joins.

In qualitative learning, for each join attribute and non-join attribute (projection attribute) in the query, GJ adds a node in the graph. If two attributes are in the same table, an edge is added between their corresponding nodes in the graph. All involved attributes from the same table make a clique in the graph.

Examples: Figure 4.4 shows an acyclic MRF for the running example (natural join of the tables) in Figure 4.1. Figure 4.5(a) is for the lastFM_cyc query (refer to Section 4.12). Figure 4.5(b) is for the junction tree derived from the cyclic graph. Please see the background section for more information on how the junction tree is created.

Quantitative learning starts after identifying the cliques and creating any junction trees. Cliques can be considered as hyper edges. In this step, GJ scans

Figure 4.5: a. Triangulated graph for lastFM_cyc b. Junction Tree with three maxcliques of size 3 for the graph

all the tables just once and calculates the potential functions of all variables whose corresponding attributes come from the same table. Each table is scanned once and the complexity of learning the model fully is $O(N)$, where N is the size of the largest table (assuming the number of the tables is small and constant). After creating the junction tree, if there is a macxlique with nodes from multiple tables, we must calculate the product of all the potentials in that maxclique. One of our innovations is that we do not join tables and then find potentials; instead, we present a novel approach for joining distributions (potentials).

Note that a potential can be calculated on a table only once for all queries. In other words, quantitative learning can be offline which can reduce join-query processing time.

In practice, these potentials/factors are implemented with hash maps which give the frequencies of given keys in $O(1)$. The potentials are not necessarily unconditional; they could be conditional in which case, nested hash maps are built for them. Converting a joint distribution to a conditional one and vice versa needs to go through all the entries in the potential, so the complexity is equal to the size of the potential, $O(M)$ where M is the size of the largest potential.

Joining the Distributions (Potentials) for Cyclic Queries

Figure 4.5.a is for the lastFM_cyc query after triangulation. The dotted lines show the fill-in edges after running the min fill-in heuristic. Figure 4.5.b is for the junction tree derived from the cyclic graph. In this junction tree, there are maxcliques whose edges come from different tables, so we need to join (product) the potentials inside the maxcliques. For example the edges (*Ar, U1*) and (*Ar, U4*) come from different tables. With our implementation, our inference algorithm requires a single potential per maxclique. To join the potentials and make the joint distribution (a single potential) for the maxclique, a novel WOJA algorithm is devised. This is shown in Algorithm 4. This algorithm is very similar to generic WOJAs (introduced in the background section), but the novelty is that it joins potentials (distributions) rather than the data.

Algorithm 4 is a recursive function that takes a maxclique $\mathcal{C}$ whose clique potentials come from different tables and generates a single potential that includes all the variables in $\mathcal{C}$. In Algorithm 4, $V_{\mathcal{C}}$ is the set of all variables in $\mathcal{C}$, $O_{\mathcal{C}}$ is the elimination order, $E_{\mathcal{C}}$ contains all hyper edges (which represent the cliques coming from different tables) in $\mathcal{C}$, m is the number of potentials in $\mathcal{C}$, $\Psi_{\mathcal{C}}$ contains all potentials in $\mathcal{C}$ and i is the index of the variables to be considered. The algorithm starts with $i = 0$ and a recursion happens for each next variable. For each variable v_i it finds

the potentials that include v_i as $\Psi'_{\mathcal{C}}$ and those that do not include v_i as $\Psi''_{\mathcal{C}}$ in lines 4 and 5, respectively. Next, for each distinct value k_i, a shared value for v_i among all potentials in $\Psi'_{\mathcal{C}}$, the algorithm filters all the entries from all potentials in $\Psi'_{\mathcal{C}}$ that have the value k_i for their v_i and makes new smaller potentials in Ψ_{next} which contains only the entries with k_i (in line 7). Then a new potential set $\Psi_{\mathcal{C}}$ from the union of Ψ_{next} and $\Psi''_{\mathcal{C}}$ is made to be used to materialize the entries for the next variables recursively (in line 8). If v_i is the final variable, function **Bucket_Product** is called. This function calculates a product of all the entries in all the potentials in the last $\Psi_{\mathcal{C}}$ and makes the final joint potential for $\mathcal{C}$. Each entry has a frequency, so they are multiplied during the product calculation. For example, assume a triangle query having three potential functions coming from $T_1(A, B), T_2(B, C)$ and $T_3(C, A)$ which can be considered as a maxclique (A, B, C). For the entries $(a_1, b_1, 5)$, $(b_1, c_1, 10)$ and $(c_1, a_1, 20)$ in the three potentials, an entry is added to the joint potential for the maxclique (A, B, C) like $(a_1, b_1, c_1, 1000)$ which shows the frequency of a_1, b_1, c_1 in the join result of the three potentials.

Asymptotic Complexity Analysis for Algorithm 4: As explained, Algorithm 4 follows the same paradigm as all WOJAs, but the difference here is that Algorithm 4 joins the distributions (frequency tables or potentials). In other words, the same WOJA method joins the potentials' entries and computes the product of their frequencies. The important note here is that the size of potentials is always equal to or smaller than the size of the tables and thus joining the potentials is faster than joining the data tables. It has already been shown in all WOJA papers (such as [Freitag et al., 2020; Ngo et al., 2012; Veldhuizen, 2012]) that the WOJAs have the complexity of $O(N^{\rho})$ where ρ is the fractional edge cover and N is the largest table size. Thus assuming that the largest potential in the maxclique is M, the complexity of Algorithm 4 is $O(M^{\rho})$. It always holds that $M \leq N$.

4.6 Inference Phase

Similarly to the PGM-Join sampler, GJ employs VEA to perform inference; however, the goal of GJ is to construct a GFJS generator (not a sample generator), and all of the algorithms have been modified accordingly. GJ utilizes the junction tree creation for cyclic joins; consequently, the VEA changes as well. Additionally, it is no longer necessary to calculate cumulative distributions because GJ is a PJA and not a sampler.

Algorithm 4 A New WOJA algorithm to calculate the joint potential of all cliques in a given maxclique

For a given maxclique $\mathcal{C}$:
$V_\mathcal{C} = \{v_1, v_2, ..., v_n\}$ is the set of variables in $\mathcal{C}$.
$O_\mathcal{C}$ is a set of all $v \in V_\mathcal{C}$ but with the elimination order.
$E_\mathcal{C} = \{e_1, e_2, ..., e_m\}$ is the set of hyper edges which makes the cliques in $\mathcal{C}$. Each e contains a subset of $V_\mathcal{C}$
$\Psi_\mathcal{C} = \{\psi_1, \psi_2, ..., \psi_m\}$ is the set of to-be-joined potentials in $\mathcal{C}$ per hyper edge.
i is the index for the current variable.

procedure POTENTIAL-JOIN($i, \Psi_\mathcal{C}$)
  if $i < n$ **then**
    $v_i \leftarrow O_\mathcal{C}[i]$ ▷ v_i is the variable to be eliminated
    $\Psi'_\mathcal{C} \leftarrow \{\psi_j \in \Psi_\mathcal{C} | v_i \in e_j\}$ ▷ All the potentials that include v_i
    $\Psi''_\mathcal{C} \leftarrow \Psi_\mathcal{C} - \Psi'_\mathcal{C}$ ▷ All the potentials that exclude v_i
    for each $k_i \in \bigcap_{\psi_j \in \Psi'_\mathcal{C}} \pi_{v_i}(\psi_j)$ **do** ▷ k_i is the shared value
      $\Psi_{next} \leftarrow \{\sigma_{v_i = k_i}(\psi_j) | \psi_j \in \Psi'_\mathcal{C}\}$ ▷ σ, selection operator
      Potential-Join$(i+1, \Psi_{next} \bigcup \Psi''_\mathcal{C})$ ▷ The recursive part
    end for
  else
    Bucket_Product $(\underset{\psi_j \in \Psi_\mathcal{C}}{\times} \psi_j)$ ▷ Potential product with freqs
  end if
end procedure

4.6.1 Building GFJS Generator: Inference on Trees

As stated, the inference process is quite similar to the inference algorithm for the PGM-Join sampler (see Algorithm 2). Here, we clarify the distinctions using some examples.

Consider the graph in Figure 4.4 (our running example). The potentials for the edges are shown in the top part row of Figure 4.7. These potentials are calculated by scanning the tables. Based on Equation 2.1, the joint distribution of the three tables is:

$$p(A, B, C, D) \propto \psi(A, B)\psi(B, C)\psi(C, D) \tag{4.1}$$

Given an elimination order $O = \{D, C, B, A\}$, GJ eliminates the variables D, C, B and A, respectively. A is the root according to O.

$$\sum_{D,C,B,A} \psi(A, B)\psi(B, C)\psi(C, D) \tag{4.2}$$

Based on the distributive law, this can be represented as:

$$\sum_{C,B,A} \psi(A, B)\psi(B, C) \sum_{D} \psi(C, D) \tag{4.3}$$

At the beginning, we have the full graph with no eliminated nodes and an empty GFJS generator, please refer to the first row of Figure 4.6. The black graph on the left is the main graph, while the orange graph on the right shows the GFJS generator.

After summing out variable D, a new potential (factor) is added for C. This new factor is shown in the middle row of Figure 4.7 (the right most factor).

$$\sum_{B,A}\psi(A,B)\sum_{C}\psi(B,C)\phi_{\not{D},C}(C) \tag{4.4}$$

While eliminating D, a conditional $\psi(D|C)$ is also calculated to be added in our GFJS generator. This conditional potential gives the frequency of D values given C values over the sub-tree starting from C to the leaves (D). Since D is a leaf, the sub-tree contains only the edge from C to D. The conditional potential is shown in the bottom row of Figure 4.7 (the right most one). Each entry of the conditional factors contains the variable(s) v_j on which the condition is on (C in our example), the dependent variable/variables v_i (D in our example), a *bucket* value which shows the local frequency of v_i given v_j in the table containing both v_j and v_i (the table D_3, in our example), and a fac value which shows the frequency of v_i in the factor coming from the children of v_i (there are no children in this step for D). For the conditional factors where the dependent variable is a leaf, fac values are 1s. The graph for this conditional factor is shown in the second row of Figure 4.6 (the right one).

Keeping the *bucket* makes it possible for GJ to produce the full join result. With PGM-Join sampler, we did not have to keep track of *bucket* values because sampling disregards the number of times a value has already been generated. This is one of the distinctions between GJ and the PGM-Join sampler.

Next, C is eliminated. After elimination, we have

$$\sum_{B,A}\psi(A,B)\phi_{\not{D},\not{C},B}(B) \tag{4.5}$$

This operation will add a factor for B (refer to the middle factor in the middle part of Figure 4.7), and a conditional factor $\psi(C|B)$ in the GFJS generator as shown in the bottom part of Figure 4.7. The new conditional factor is shown in the third row of Figure 4.6 (on the right), and C is eliminated from the main graph (on the left). In fact, $\psi(C|B)$ gives the frequency of C given B over the sub-tree from B to the leaves (D).

Next, B is eliminated. This will add a factor for A, as shown in the middle part of Figure 4.7 (the left most one), and a conditional factor $\psi(B|A)$ in the GFJS

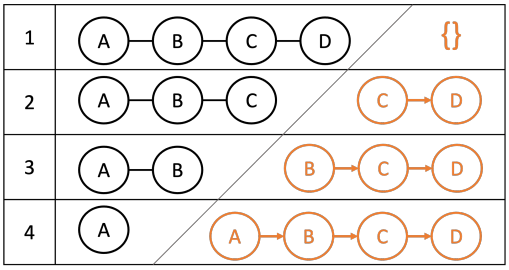


Figure 4.6: Join graph and the GFJS generator for the running example join

as shown in the bottom part of Figure 4.7 (the left most one). $\psi(B|A)$ gives the frequency of B given A over the whole tree. The changes in the main graph and the GFJS generator are shown in the lowest row of Figure 4.6.

$$\phi_{\cancel{D},\cancel{C},\cancel{B},A}(A) \tag{4.6}$$

The unconditional $\phi_{\cancel{D},\cancel{C},\cancel{B},A}(A)$ is the last factor in the GFJS generator. The sum of all the frequencies in $\phi_{\cancel{D},\cancel{C},\cancel{B},A}(A)$ gives the join size.

Asymptotic Complexity of Inference on Trees

Since trees have a perfect elimination order and each variable has a single parent, each elimination involves a single potential with two variables. This potential comes from a single table and there is no repetition in the entries of the potentials. Thus, given that the size of the biggest potential is M, the complexity of eliminating all variables is $O(M)$.

4.6.2 Building GFJS Generator: Inference on Graphs

If the query graph is not a tree, it must be translated to a new structure having the same properties (e.g., R.I.P) as trees. Hence, a junction tree (a tree of maxcliques) is created using standard algorithms as explained in the Background section. Once the junction tree is created, the above inference algorithms run over junction trees.

Concretely, query lastFM_cyc from our experiments is cyclic yielding a cyclic graph. One of the possible junction trees for that query is shown in Figure 4.5. The joint distribution for the junction tree in Figure 4.5 is:

$$p(Ar, U1, U2, U3, U4) \propto \psi(U2, U3, U4)\psi(U1, U2, U4)\psi(Ar, U1, U4) \tag{4.7}$$

Given an elimination order like $O = \{Ar, U1, U2, U3, U4\}$, the variables are summed out from Equation 4.7 one by one. Below all the steps of eliminating the variables are shown. Figure 4.8 depicts the modifications made to the main junction

Potential for D_1		
A	**B**	**freq**
a_0	b_0	3
a_1	b_1	2
a_2	b_1	1
a_3	b_3	2
	b_4	4

Potential for D_2		
B	**C**	**freq**
b0	c0	2
b_1	c0	3
b_2	c_1	3
b_3	c_2	1
b_4	c_3	2
	c_4	1

Potential for D_3		
C	**D**	**freq**
c_1	d_0	4
c_2	d_2	4
c_3	d_3	2
c_4	d_4	2

$\phi_{B\to A}(A)$	
A	**freq**
a_3	32

$\phi_{C\to B}(B)$	
B	**freq**
b_2	12
b_3	4
b_4	6

$\phi_{D\to C}(C)$	
C	**freq**
c_1	4
c_2	4
c_3	2
c_4	2

$\psi(B\|A)$			
A	**B**	**bucket**	**fac**
a_3	b_3	2	4
	b_4	4	6

$\psi(C\|B)$			
B	**C**	**bucket**	**fac**
b_2	c_1	3	4
b_3	c_2	1	4
b_4	c_3	2	2
	c_4	1	2

$\psi(D\|C)$			
C	**D**	**bucket**	**fac**
c_1	d_0	4	1
c_2	d_2	4	1
c_3	d_3	2	1
c_4	d_4	2	1

Figure 4.7: The first row contains the raw potentials from the tables; the second row contains the factors resulting from each sum-product operation; and the third row contains the conditional potentials added to the GFJS generator after each elimination. All of these are for the running example with the graph shown in Figure 4.4

tree and the GFJS generator for each of the elimination steps detailed below. The difference here is that the conditions are on composite keys (the separator sets).

The elimination steps:

1– First step with all potentials:

$$\sum_{U1,U2,U3,U4} \psi(U2,U3,U4)\psi(U1,U2,U4)\sum_{Ar}\psi(Ar,U1,U4) \tag{4.8}$$

2– After eliminating Ar:

$$\sum_{U2,U3,U4} \psi(U2,U3,U4)\sum_{U1}\psi(U1,U2,U4)\phi_{\cancel{Ar},U1,U4}(U1,U4) \tag{4.9}$$

3– After eliminating U1:

$$\sum_{U2,U3,U4} \psi(U2,U3,U4)\phi_{\cancel{Ar,U1},U2,U4}(U2,U4) \tag{4.10}$$

4– After eliminating U2:

$$\sum_{U3,U4} \phi_{\cancel{Ar,U1,U2},U3,U4}(U3,U4) \tag{4.11}$$

5– After eliminating U3

$$\phi_{\cancel{Ar,U1,U2,U3},U4}(U4) \tag{4.12}$$

$\phi_{\cancel{Ar,U1,U2,U3},U4}(U4)$ is the last factor which is added to the GFJS generator in the 6_{th} step. We show $\phi_{\cancel{Ar,U1,U2,U3},U4}(U4)$ as $\psi(U4)$ and consider it as the starting point to generate the GFJS.

Asymptotic Complexity of Inference on Graphs

The asymptotic complexity of the inference on cyclic graphs is different from that on trees. In a junction tree, it is possible to have a maxclique with some potentials (cliques) that come from different tables. In this case, we use the Algorithm 4 to join potentials (frequency tables, not data tables) and make a single potential for the maxclique. Recall that the complexity of the inference on trees was equal to the size of the largest potential ($O(M)$), but here the largest potential can be larger as the maxclique may involve several potentials. In the worst-case (which is very rare),

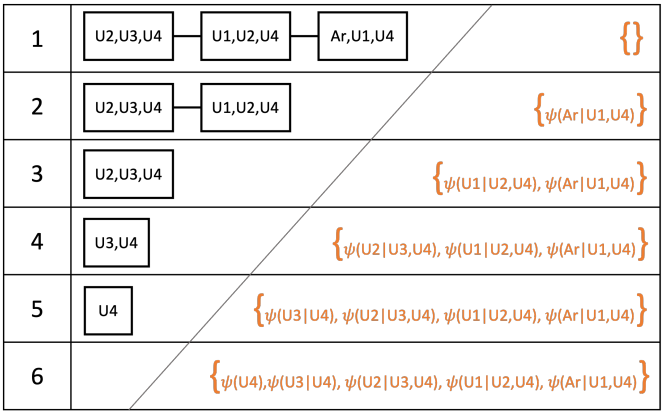


Figure 4.8: Inference on the junction tree of Figure 4.5

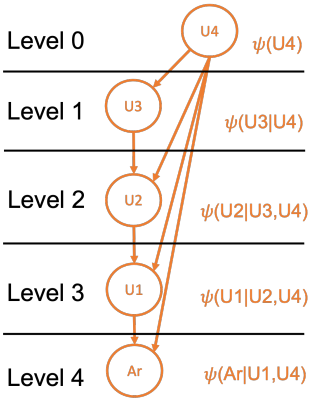


Figure 4.9: The DAG for the GFJS generator in Figure 4.8

all variables are added into a single maxclique during the junction tree creation. In this case, joining all potentials of the graph with Algorithm 4 is $O(M^{\rho})$. This is because Algorithm 4 is a WOJA.

4.7 Summary (GFJS) Generation

We have selected GFJS as the join result representation. GFJS has less redundancy and can be stored/retrieved and de-summarized efficiently and with minimal effort.

Let's show the GFJS generators with ϑ. ϑ for a chain MRF is a chain, for a tree MRF is a tree and for an MRF with a junction tree, it is a directed acyclic graph (DAG).

The orange graph in the lowest row of Figure 4.6 is the GFJS generator for our three-table running example. This generator has 4 levels and the root is A. For our cyclic query, it is shown in Figure 4.9 with 5 levels and the root is $U4$. Note, in our

examples, we have a variable per level, but it is possible to have several variables in a level. For example, if O is $\{D, C, A, B\}$ to do inference in our three-table example, B would be the root and it would have two children A, C. Note that on the same level of trees and DAGs, it is possible to have nodes with different parents.

The summary generation is done in reverse order of the elimination order.

The summary for the root is already ready. Assume ψ_0 is the last (root) potential from the inference and keeps the frequency of the root variable over the join result. For our running example, the potential for the root in the GFJS generator has only a single value a_3 with the frequency of 32, so $(a_3, 32)$ is added to the GFJS of A.

GFJS for other variables is generated recursively, per level of the generator. In other words, all the variables in the same level are dependent to each other when we want to generate the exact full join result even if their parents are different. This is because we want to generate the exact join result, not a sample (instance) of the join result. In sampling, if a node has two children, the sample for each of them is generated independently.

Recall the GFJS generator for our running example shown in the bottom row of the Figure 4.7. After finding an entry for A, then B entries are generated for the GFJS of B. For this, $\psi(B|A)$ of the GFJS generator is used. So, for a_3, there are 2 buckets of b_3 and each bucket's size is 4, hence an entry $(b_3, 8)$ is added to the GFJS of B. The entry for b_4 is added later after backtracking. In fact, our algorithm is a kind of DFS algorithm with backtracking steps on a factorized *distribution*. Using $\psi(C|B)$ in the GFJS generator, for each bucket of b_3, there are 1×4 of c_2 values. Here, the bucket size of b_3 (which is 2 in $\psi(B|A)$) is used to find the right frequencies for c_2. To do so, the bucket size of b_3 from $\psi(B|A)$ and the bucket size of c_2 from $\psi(C|B)$ are multiplied then the result is multiplied with the fac of c_2. The result is $(c_2, 8)$ which means all values related to b_3 is c_2. The bucket values are recursively pushed down until we reach the leaves. Thus, although the bucket of c_2 is 1 in $\psi(C|B)$, the bucket value that is pushed down to level of D is 2 (the product of all the buckets seen in the path). One of the differences between the GJ's inference and the PGM-Join sampler's inference is that the GJ algorithm retains bucket values throughout inference. In the PGM-Join sampler, buckets were not necessary. Once the entry related to c_2 was added to the GFJS of C, an entry $(d_2, 8)$ is added to the GFJS of D by multiplying the pushed-down bucket values and the fac of d_2. Now, the algorithm has reached the leaves, hence the backtracking happens to generate the other values of GFJS for C. This backtracking is continued till all the entries for all variables are added. The final GFJS for the running example is shown in Figure

4.2.

Algorithm 5 is for generating the GFJS for a given GFJS generator ϑ. In Algorithm 5, for each entry in the root potential ψ_0 of the generator, the recursive function of Algorithm 6 is called which generates the summaries per level (all the variables in the same level) of the generator.

In line 2 of Algorithm 6, all potentials in level i of the generator are conditioned on the already observed values in their parents, and are added to Φ. *keys* is a map from the variable names to their values that are already observed in upper levels. The operator [] makes the conditional ψs, unconditional by applying the conditions using the observed values in *keys*. Φ holds the unconditional potentials for all the variables in the same level. If there are more than one variable in the current level, Algorithm 6 calculates the Cartesian product of values. In the Cartesian product, the values for all variables are combined, the *bucket* values are multiplied together and fac values are also multiplied. The result is stored in $\mathcal{R}$. For each row r in $\mathcal{R}$, the recursive function is called again to add the summaries for the next variables. In line 5, the already observed values are combined with the values in r for the current level variables. Also, the *bucket* value from the upper level and the bucket value for r are multiplied. Line 7 adds the summary for the variables in the current level i into s_i by multiplying the new bucket value with the fac values for each r. If it is not the last level, the recursive function is called again in line 9.

Since ψ_0 does not include any unneeded value for the root, and since we do not have any entry in $\psi \in \vartheta$ with zeros, GFJS generation will avoid UIR; hence, GJ is a WOJA.

To handle projection operations, if a node in the graph is not in the projection list (i.e., an output attribute), one can ignore the summary for that node and simply pass to the next level. This happens when some of join attributes are not in the select clause (non-join attributes that are not involved in the join query are simply ignored during MRF building). However, Section 4.9 will explain how one can efficiently apply projections on graphs before starting the summary generation, by deleting those variables from the models. Consequently, the join attributes that are not in the output will be deleted from the GFJS generator.

Asymptotic Complexity Analysis for GFJS Generation

The asymptotic complexity for GFJS generation is $O(M^{\rho})$ since it does not generate any information that is not in the join result and it adds an entry in the GFJS per distinct value (not for the whole data).

Algorithm 5 Generates the GFJS given a generator ϑ

ψ_0 is the root potential for the GFJS generator ϑ.
$S = \{s_0, s_1, ..., s_{m-1}\}$ is the set of summaries per level of the GFJS generator.
m is the number of levels in ϑ.
root is the variable name for the root.

```
1: procedure GEN_GFJS( )
2:     for each entry in ψ0 do                          ▷ each entry is (value,freq)
3:         keys ← {root,entry.value}                    ▷ root value, observed
4:         s0 ← s0∪(entry.value,entry.freq)             ▷ GFJS for the root
5:         REC_GFJS(1, 1, keys)          ▷ which generates {s1, ..., sm−1} for other levels
6:     end for
7:     S ← {s0, s1, ..., sm−1}                          ▷ S is the final GFJS
8:     store S on disk, if needed
9: end procedure
```

Algorithm 6 Recursive algorithm used in Algorithm 5

ϑ is the GFJS generator.
ϑ^i gives all the generative potentials ψs for the level i of ϑ.
$s_1, s_2, ..., s_{m-1}$ are the summaries per level.
m is the number of levels in ϑ.
keys is a map of the variable names to their chosen values.
i is the current level index.
p_bucket is the bucket size from the upper level.

```
1: procedure REC_GFJS(i, p_bucket, keys)
2:     Φ ← ⋃_{ψ∈ϑ^i} ψ[keys]                  ▷ keys contains the observed values
3:     R ← ×_{ψ∈Φ} ψ                          ▷ all ψ ∈ Φ are unconditional
4:     for each row in R do
5:         keys_new ←keys ⋃ row.keys          ▷ new values for next keys
6:         bucket_new ←p_bucket×row.bucket
7:         s_i ← s_i ∪ (row.keys,bucket_new× row.freq)
8:         if i < m − 1 then                  ▷ if any variable remains then recur
9:             REC_GFJS(i + 1,bucket_new, keys_new)
10:        end if
11:    end for
12: end procedure
```

4.8 Join Result Generation (Desummarization)

One of the advantages of RLE (in our case, the columns of GFJS) is that RLE can be stored/retrieved/desummarized straightforwardly. Each column of the RLE can be stored/retrieved and de-summarized separately. De-summarization starts from the top of the column (let us say s_1) in S and replaces any $(v, freq)$ pair with $freq$ of v values, where v is the value and $freq$ is the value's frequency. The cost of de-summarization is exactly the same as the join size $|Q|$. De-summarization is avoided if the join result is to be stored on disk. Please note that if one needs to enumerate the join result tuples from a factorized join representation (as we do in the case for FDB and FAQ) then several index accesses should be done to enumerate each tuple every time the join result is needed. However, with generating or storing/retrieving GFJS, the columns are generated independently, without care about the values of other columns.

The best known upper bound for the join size is N^{ρ} and hence the complexity of the join result generation is $O(N^{\rho})$.

4.9 Early Projections

Not all join attributes are output attributes. In other words, we could have nodes in the graph that are not supposed to be generated in the join result. The idea of early projection is that, before starting the GFJS generation, one can delete these nodes from the join graph and the GFJS generator. When finding the elimination order, one could find two elimination orders O and O', where O is related to the output (projection) attributes and O' is for the join attributes that are not output attributes. O' is always considered before O. First, the unneeded variables based on O' are deleted. Then the others are eliminated, based on O. When "a variable is eliminated", it means that a factor for its parent is calculated (recall the second row of Figure 4.7) and also a conditional factor is calculated (recall the third row of Figure 4.7). But when "a node is deleted", it means that the conditional factors for the node are not generated, and the node is deleted from the graph; note that the factor for its parent is still calculated. Projections do not affect the asymptotic complexity of the inference which is still $O(M^{\rho})$, in the worst case. *GJ is, to our knowledge, the first WOJA able to apply early projections.*

4.10 Overall Asymptotic Complexity of GJ

Overall, the four main operations of GJ are: i) scanning the tables (when potentials are not already available). This is accomplished in $O(N)$ for tree queries and $O(M^\rho)$ for junction trees; ii) running inference on the query graph to build the GFJS generator. This operation on trees is done in $O(M)$. On graphs, it is equal to the size of the largest potential for maxcliques. In the worst (rare) case, when creating junction trees, we add all nodes in a single maxclique. Then the asymptotic complexity is $O(M^\rho)$; iii) generating GFJS, which needs to traverse the graph one time. GFJS generation is carried out in $O(M^\rho)$. Note that summary generation is performed on graphs whose unneeded intermediate paths have already been pruned. Moreover, since repeated values are replaced with frequencies, the summary is typically much smaller than the join result; iv) desummarizing, whose cost is equal to the join size $|Q|$. The join size complexity is $O(N^\rho)$. Thus, if the join tuples are to be generated, all WOJAs, FDB, FAQ, and GJ have the same asymptotic complexity of $O(N^\rho)$. In practice, however, as GJ retains the GFJS, de-summarization is performed columnarly, which is significantly faster than other competitors. This is because GJ has the frequencies of distinct values beforehand. We shall illustrate this through our experiments.

4.11 Parallelism

We employ OpenMP [Chandra et al., 2001] with parallelism at two levels. In two-level parallelism, tasks that can be executed concurrently are formed, and each task can contain several threads. The tasks themselves can be run in parallel. A task consists of executable code and a data environment. Each variable elimination (sum-product operation) leads to the creation of a task, and each task can be handled by many threads. This allows GJ to utilize all CPU cores effectively. Note that all leaves can be eliminated from a tree in parallel with parallel tasks. Once the old leaves have been eliminated, the new leaves can be eliminated in parallel too. This continues until the root is reached by the algorithm.

Each sum-product operation in a task is comprised of four steps: i) compute the product of all potentials ii) sum out the to-be-eliminated variable from the product result; iii) construct a new factor for the remaining variables. iv) construct the conditional potential of the GFJS generator.

For (i), there may be multiple potentials for a sum-product operation. GJ divides the smallest potential into n segments (for each thread), where n represents

the number of available cores. Each thread takes a segment and, for each entry within that segment, checks all other potentials and calculates the products. Consequently, each thread will result in a different product result. Due to the use of hash tables, the complexity of calculating the products over all threads is equivalent to the size of the smallest potential. Since the hash table for each product result (of each thread) does not share distinct values with other product results (from other threads), each thread sums out the to-be-eliminated variable and calculates a sub-factor for the remaining variables, separately. The thread also makes the conditional sub-factor from its product result to be used in the GFJS generator. At the end, all the sub-factors are merged together to build a single factor for the remaining variables with the complexity of the size of the factor. And all the conditional sub-factors are merged together to build the conditional factor to be added into the GFJS generator. The complexity of this is also the same as the size of the conditional factor as there is no shared distinct values among the conditional sub-factors.

The elimination of leaves in node-level parallelism is known as *topological* parallelism, and the elimination of each node using a parallel sum-product operation is known as *in-clique* parallelism. Please refer to the papers [Kozlov and Singh, 1994; Xia et al., 2009] for additional information about these.

As GJ stores the summary of the join result, which is significantly smaller than the join result, GJ is a CPU-bound join method, whereas other algorithms spend a great deal of time storing large join results on disk.

Experiments will demonstrate that the performance of the parallel GJ algorithm is superior to that of its competitors.

4.12 Experimental Evaluation

Setup and Baselines. We study the performance of GJ against PostgreSQL (PSQL), MonetDB, and the WOJA in Umbra, hereafter referred to as "Umbra" which is a state-of-the-art WOJA for RDBMSs. We set Umbra to use its WOJA algorithm ($Umbra_{EAG}$) as the aim is to use it to compare GJ against WOJAs. These competitors are full-fledged systems, whereas GJ is a C++ code library working with CSVs. Systems incur additional overheads when executing a join. Thus, to ensure a fair comparison, we obtained (from the authors of Umbra) a version that works with CSV files (e.g., escaping other system overheads, such as dealing with ACID properties, logging, etc). Thus, two versions of Umbra are used in the experiments, one as an RDBMS and another one which works with CSV files (like GJ). Also, note that the aforementioned overheads in full-fledged RDBMSs represent typically

a small percentage of execution time, whereas the differences between GJ and them are large, as will be shown later.

We test under two different scenarios. The first scenario, compute-and-reuse, involves taking into account the costs of completing joins, storing results on disk, and then reloading results into memory. In this scenario, GJ stores/reloads GFJS on/from disk, and then desummarizes it as a flat join result (whereas PSQL, MonetDB and Umbra store/reload the flat join result). We also compare the storage cost of GJ and other competitors.

In the second (compute-and-forget) scenario, GJ generates GFJS and from it then generates the flat join result. Furthermore, we show the PGM building time for GJ.

Additional experiments are conducted to demonstrate the difference between a factorized join result (by FDB) and the factorized join distribution (by GJ) in regards to GJ's performance against a simulated version of FDB.

We use a server with one Intel Core™ i5-8500 3.00GHz CPU, with 32GB RAM, and 1TB SSD disk. We run each query 5 times and then report averages. Parallelism on MonetDB, PSQL, Umbra and GJ is disabled in an effort to compare solutions without spending more resources to address inefficiencies. We have also developed a parallel version of GJ and compared against the parallel versions of the competitors.

Data: We use the JOB [Leis et al., 2015] and TPCH benchmarks[1] (with scaling factor 1), as well as lastFM[2], a real-world data set [Brusilovsky et al., 2010]. MonetDB builds its own indices. For PSQL, we pre-built B-trees on all join attributes before running queries. Umbra builds its own tries.

Queries: We use queries from JOB, lastFM, and TPCH. When selecting our queries for experiments we wanted to bring out salient features of joins and how they impact GJ and the competitors. JOB queries include some many-to-many relations. Hence they have more redundancy in the join result and UIR. lastFM queries have less redundancy in the result, but suffer from higher UIR. TPCH queries occupy the other extreme, being foreign key (FK) joins, having low redundancy in the join result and there are no UIR. Since we focus on physical joins, all non-join predicates, aggregations and group-by statements are removed. The queries are as below :

JOB–A (JOB):

```
SELECT t.title
FROM keyword AS k, movie_info AS mi, movie_keyword AS mk, title AS t
WHERE t.id = mi.movie_id AND t.id = mk.movie_id AND mk.movie_id = mi.movie_id AND
k.id = mk.keyword_id;
```

[1] http://tpc.org/

[2] https://grouplens.org/datasets/hetrec-2011/

JOB_A	JOB_B	JOB_C	JOB_D	lastFM_A1	lastFM_B	lastFM_cyc	FK_A	FK_B
444,340,632	31,588,599,792	1,037,051,092	146,527,949,388	61,664,382	12,538,289,270	706,351,348	6,001,194	5,760,999

Table 4.2: Join sizes per query

```
JOB-B (JOB):
SELECT mi.info, t.title
FROM aka_title AS at, company_name AS cn, company_type AS ct,info_type AS it1,
keyword AS k,movie_companies AS mc,movie_info AS mi,movie_keyword AS mk, title AS t
WHERE t.id = at.movie_id AND t.id = mi.movie_id AND t.id = mk.movie_id AND
t.id = mc.movie_id AND mk.movie_id = mi.movie_id AND mk.movie_id = mc.movie_id AND
mk.movie_id = at.movie_id  AND mi.movie_id = mc.movie_id AND mi.movie_id = at.movie_id
AND mc.movie_id = at.movie_id AND k.id = mk.keyword_id AND it1.id = mi.info_type_id AND
cn.id = mc.company_id AND ct.id = mc.company_type_id;

JOB-C (JOB):
SELECT mi.info as i,mi_idx.info as ii, t.title
FROM cast_info AS ci, info_type AS it1, info_type AS it2, movie_info AS mi,
movie_info_idx AS mi_idx, name AS n, title AS t
WHERE t.id = mi.movie_id AND t.id = mi_idx.movie_id AND t.id = ci.movie_id AND
ci.movie_id = mi.movie_id AND ci.movie_id = mi_idx.movie_id AND mi.movie_id = mi_idx.movie_id AND
n.id = ci.person_id AND it1.id = mi.info_type_id AND it2.id = mi_idx.info_type_id;

JOB-D (JOB):
SELECT mi.info , mi_idx.info, n.name , t.title
FROM cast_info AS ci, info_type AS it1, info_type AS it2, keyword AS k, movie_info AS mi,
movie_info_idx AS mi_idx, movie_keyword AS mk, name AS n, title AS t
WHERE t.id = mi.movie_id AND t.id = mi_idx.movie_id  AND t.id = ci.movie_id AND
t.id = mk.movie_id AND ci.movie_id = mi.movie_id  AND ci.movie_id =mi_idx.movie_id AND
ci.movie_id = mk.movie_id AND mi.movie_id = mi_idx.movie_id AND mi.movie_id = mk.movie_id AND
mi_idx.movie_id = mk.movie_id AND n.person_id = ci.person_id AND it1.id = mi.info_type_id AND
it2.id = mi_idx.info_type_id AND k.id = mk.keyword_id;

lastFM-A1 (LastFM):
SELECT ua1.userid as u1,ua1.weight as w1, ua2.userid, ua2.weight
FROM user_artists ua1, user_artists ua2, user_friend uf1
WHERE ua1.userid=uf1.userid  AND uf1.friendid=ua2.userid;

lastFM-A2 (LastFM):
SELECT ua1.userid as u1, ua1.weight as w1,ua2.userid, ua2.weight
FROM user_artists ua1, user_artists ua2, user_friend uf1, user_friend uf2
WHERE ua1.userid=uf1.userid AND uf1.friendid=uf2.userid AND uf2.friendid=ua2.userid;

lastFM-B (LastFM):
SELECT ut1.artistID, ut1.userID,ut2.userID,ut2.artistId
FROM usertag ut1, usertag ut2, user_friend uf1, user_friend uf2
WHERE ut1.userID=uf1.userID AND uf1.friendId=uf2.userID AND uf2.friendId=ut2.userID;

lastFM\_cyc (LastFM):
SELECT uf1.userID, uf1.friendId as f, uf2.friendId as f1, uf3.friendId as f2
FROM usertag ut1, usertag ut2, user_friend uf1, user_friend uf2, user_friend uf3
WHERE ut1.userID=uf1.userID AND uf1.friendId=uf2.userID AND uf2.friendId=uf3.userID
AND uf3.friendId=ut2.userID AND ut1.artistID=ut2.artistID;

FK-A (TPCH):
SELECT name,discount
FROM customer, orders,lineitem
WHERE customer.custkey=orders.custkey AND orders.orderkey=lineitem.orderkey;

FK-B (TPCH):
SELECT regionkey,supplier.suppkey
FROM customer, orders,lineitem, nation, supplier
WHERE customer.custkey=orders.custkey AND orders.orderkey=lineitem.orderkey AND
customer.nationkey=nation.nationkey AND lineitem.suppkey=supplier.suppkey;
```

The join sizes per query are listed in Table 4.2 where sizes range from ca. 5 million tuples to ca. 146 billion tuples.

4.12.1 Results for Query Time and Space

The compute-and-reuse scenario. Here, we show the key costs for: i) running the join and storing the result on disk, ii) loading time of the result into memory, and iii) storage. Time costs to run and store the join result on disk are shown in Table 4.3. Results with '>' mean that after a certain time the database crashed (typically, running out of storage - exceeding the available 1TB). The "-" means that after exceeding 1TB, Umbra-CSV neither crashed nor finished the job. The results show that GJ is always better than competitors, except for the FK TPCH joins (as expected). Since GJ does not store the actual join results (but GFJS), the speedup is drastic: up to 820X faster than PSQL; more than 717X faster than MonetDB; up to 165X faster than Umbra; and 94X faster than Umbra-CSV. Table 4.3 shows that the idea of calculating the summary of the join result without computing the join result, can have a significant impact on join performance. MonetDB is always better than PSQL when the join results are supposed to be stored on disk, likely as a result of the columnar nature of MonetDB. FK queries have less redundancy, no UIR and also their join sizes are small; hence, GJ is worse than MonetDB and Umbra. None of the competitors could run JOB_D due to its large join size, exceeding 1TB with PSQL/MonetDB/Umbra/Umbra-CSV. GJ managed JOB_D in 51.8 seconds with a meagre storage cost of 260MB! Queries like JOB_D showcase the scalability afforded by GJ.

Other competitors must construct the join result fully, sort the result, and afterwards calculate the frequencies per column to generate RLE over joins. As a result, in the RLE case, the cost of sorting and calculating frequencies should be added to the competitors' time cost. GJ, on the other hand, generates the RLE directly.

Time costs for loading the join result from disk are shown in Table 4.4. GJ needs to load GFJS into memory and desummarize it. Results in Table 4.4 show that GJ is always faster that PSQL in preparing the join result in memory up to 356X (with query JOB_A). GJ is also always better than MonetDB (up to 11.5X with query FK_A) except for lastFM_A1. GJ is always better than Umbra (up to 118X with query JOB_B) except for lastFM_A1 and lastFM_cyc. GJ is always faster than Umbra-CSV (up to 132X with query FK_A).

The reason that Umbra-CSV is slower than Umbra in loading results into memory is that Umbra-CSV needs to parse strings from CSV files, which takes significant time. Note that GJ also parses the strings in CSV files. This overhead is not incurred in an RDBMS which may make GJ even faster if implemented within an RDBMS.

	JOB_A	JOB_B	JOB_C	JOB_D	lastFM_A1	lastFM_B	lastFM_cyc	FK_A	FK_B
GJ	**8.8**	**27.6**	**45.2**	**51.8**	**5.8**	**80.8**	**9.8**	2.49	5.12
Umbra	36.2	4,560	140.8	>2,664	7.1	1,792	97.6	1.68	**2.71**
Umbra-CSV	23.7	2,610	157.6	-	11.9	423	154.7	1.82	2.74
MonetDB	42.3	>19,794	480.9	>4,537	8.4	6,810	1,639	**0.91**	2.85
PSQL	324.5	22,633	808	>26,533	61	11,508	862	7.3	8.67

Table 4.3: Time cost in seconds for generating and storing the join result in disk (GJ stores the GFJS)

	JOB_A	JOB_B	JOB_C	JOB_D	lastFM_A1	lastFM_B	lastFM_cyc	FK_A	FK_B
GJ	**0.17**	**26.7**	**3.7**	**264**	3.4	**83.4**	**1.9**	**0.004**	**0.007**
Umbra	0.58	3,156	139.8	-	**0.72**	2,016	118.8	0.056	0.055
Umbra-CSV	15.96	2,731	113.2	-	10.2	2,418	136.7	0.53	0.57
MonetDB	1.34	-	9.4	-	0.99	763	10.1	0.046	0.044
PSQL	60.6	7,063	232	-	6.4	3,458	203	0.91	0.39

Table 4.4: Time cost in seconds for loading the results into memory

In general, GJ shines when there are UIR, high redundancy in the results, and join sizes are big.

Storage costs per query are shown in Table 4.5. For all queries GJ is dramatically more efficient, up to 21,488X (with query JOB_B) better than PSQL, up to 38,333X (with query FK_A) better than MonetDB, and up to 78,750X (with query FK_A) better than Umbra and up to 54,666X (with query FK_A) better than Umbra-CSV. MonetDB could not store the result for JOB_B and JOB_D, and all competitors could not finish JOB_D.

The compute-and-forget scenario. The times to compute the join result in memory are shown in Table 4.6. GJ is always better than PSQL and MonetDB, especially on lastFM queries, as expected. The impact of GJ for FK joins is less, again as expected. GJ is better than Umbra for all many-to-many queries except for JOB_A and JOB_C. GJ is faster than PSQL by up to 64X (with query lastFM_B); faster than MonetDB by more than 388X (with query JOB_B); and faster than Umbra by more than 6X (with JOB_B and lastFM_B). The results for Umbra-CSV are almost the same as Umbra as we do not generate, store and reload the CSV files for in-memory runs. Generally, wherever the join size is larger, the efficiency gain from GJ's summarization/desummarization is higher.

Table 4.7 shows the percentage of GJ's in-memory runtime which is spent on building the PGM - actually, this refers to the cost to compute the potentials

	JOB_A	JOB_B	JOB_C	JOB_D	lastFM_A1	lastFM_B	lastFM_cyc	FK_A	FK_B
GJ	**4.4**	**50.8**	**351**	**260**	**312**	**5,529**	**89**	**0.0024**	**0.4**
Umbra	7,639	994,099	49,624	-	3,993	777,206	11,195	189	181
Umbra-CSV	3,174	461,824	23,040	-	2,867	592,998	32,870	131.2	120.3
MonetDB	1,976	-	11,980	-	1,845	384,102	21,640	92	148
PSQL	15,360	1,091,584	44,032	-	3,068	623,616	35,148	247	236

Table 4.5: Storage cost in MBs

	JOB_A	JOB_B	JOB_C	JOB_D	lastFM_A1	lastFM_B	lastFM_cyc	FK_A	FK_B
GJ	8.9	**51**	43.9	**302**	**0.62**	**33.14**	**9.96**	2.51	5.12
Umbra	**4.7**	305	**39.2**	1620	1.02	221.2	29.61	1.17	**2.15**
MonetDB	38.5	>19,776	411	>4,155	3.13	3,466	1,713	**0.66**	2.69
PSQL	40.6	2,219	109	10,660	7.51	2,101	253	4.1	5.28

Table 4.6: Time cost in seconds for running the joins in memory

JOB_A	JOB_B	JOB_C	JOB_D	lastFM_A1	lastFM_B	lastFM_cyc	FK_A	FK_B
62%	31%	49%	9%	9%	0.3%	1%	99%	52%

Table 4.7: The percentage of in-memory running times spent on building PGMs

(frequency tables), as creating the graph structure is trivial. For example, 99% of the time for query FK_A, 62% for query JOB_A, and 0.52% for FK_B are for PGM building. We refer to these queries in particular as GJ performs worse for these queries, cf. Table 4.6. Noting this, one could pre-build the frequency tables and keep them in memory for frequently used tables for joins. The pre-built frequency tables can also be used for other purposes (e.g. for cardinality estimation). Prebuilding PGMs for frequent joins can actually make GJ competitive, even for FK-joins.

4.12.2 Sensitivity Analysis: UIR and Redundancy

Query lastFM_A1 is a good candidate to examine the sensitivity of the algorithms on UIR and the redundancy. This is because most of the algorithms have almost the same performance on this query and also, we can easily control the UIR and the redundancy. So we use this query as our main query. The query is about users and their friendship and it joins tables *user_artists*, *user_friends* and *user_artists*. We now evaluate a variation, lastFM_A2, to see how larger UIR can affect performance. lastFM_A2 entails users and their relation with their friends of friends. lastFM_A2 joins tables *user_artists*, *user_friends*, *user_friends* and *user_artists*. The additional join operation increases UIR. Furthermore, we introduce lastFM_A1_dup, which is the same as lastFM_A1, except the size of each table is doubled by replicating each tuple once. This can show the impact of a higher result redundancy. Figures 4.10(a) (and 4.10(b)) show the times for generating the join result and storing it on a disk (and the in-memory running time). Figure 4.10(c) and Figure 4.10(d) show the storage cost and the loading time for the aforementioned queries.

The results show that the two sources of inefficiency do not affect GJ, but they affect competitors significantly in both in-memory and in-disk runs. Note, the effect of UIR on Umbra is also not significant (as it is a WOJA), but higher result redundancy affects Umbra's performance. In terms of space, GJ is as expected drastically better. Interestingly, the loading time of GJ on lastFM_A1 is slower

than MonetDB and Umbra, but this is not the case for lastFM_A1_dup (with larger redundancy) or for lastFM_A2 (with higher UIR). Finally, GJ performs better as join sizes increase. The join sizes for lastFM_A1, lastFM_A1_dup and lastFM_A2 are ∼61 million, ∼493 million, and ∼2 billion.

4.12.3 Parallel GJ

We use OpenMP for parallelism with both topological and in-clique parallelism ideas for implementing the parallel version of GJ.

We implement the parallel code for the lastFM_A1 query and also run the experiments for other versions of that query lastFM_A2 and lastFM_A1_dup. Our experiments are for both the scenarios compute-and-forget and compute-and-reuse.

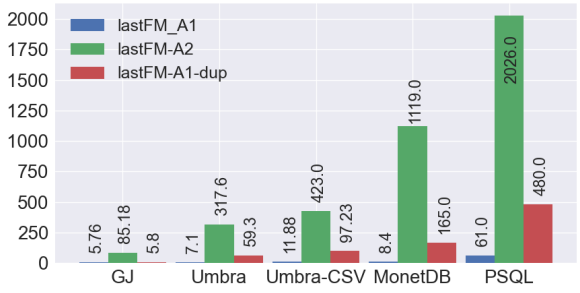
The results in Figure 4.11(a),(b), and (c) show the time cost with 1 to 6 cores for the queries lastFM_A1, lastFM_A2 and lastFM_A1_dup, respectively. The results show that the improvement with GJ and the other WOJA (Umbra) follows the same trend. For all three queries, GJ is faster than others.

The results for on disk runs in Figure 4.11(d),(e), and (f) show the great improvement with GJ when the number of the cores increases. This is because GJ is a CPU-bound algorithm where other algorithms need to store a huge join result. The improvement for other competitors is almost nothing, and sometimes the higher number of cores results in a worse performance because they are I/O-bound.

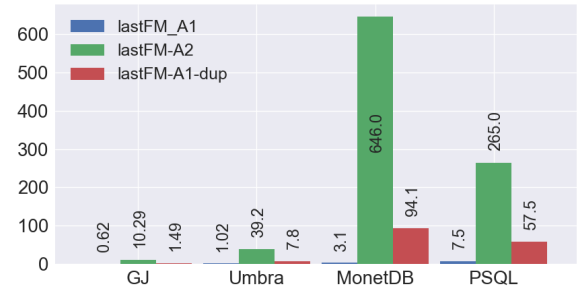
4.12.4 GJ vs. FDB

FDB is not a PJA, making comparisons between FDB and GJ difficult. As described in the background section, FDB's primary goal is to identify the most succinct form of the join result, not to generate the join tuples. The factorization of the data with FDB is a higher level of normalization where any single value is assumed as a table. In contrast, GJ has been designed as a PJA for scenarios where the join result is required. For example, when the result of one query serves as the input for another. Or for migration between databases (for example, converting structured data to unstructured data in BigQuery [Melnik et al., 2010; Gubarev et al., 2020]).

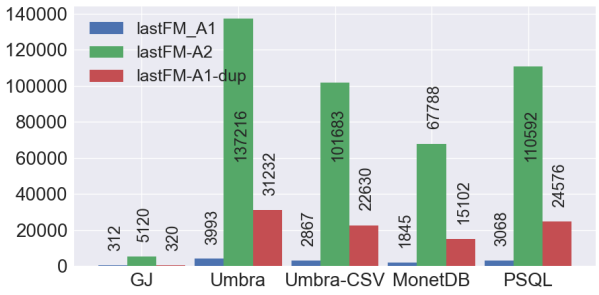
The only conceivable comparison is to run GJ and FDB to calculate aggregations such as *Count* and compare the amount of time required to build their join result representations while calculating the count aggregation. We execute GJ and FDB to determine the count aggregations of the lastFM_A1 and lastFM_A2 queries with varying data scaling factors. With a greater scaling factor, redundancy in the result rises, and this can demonstrate how the factorized distribution of GJ and



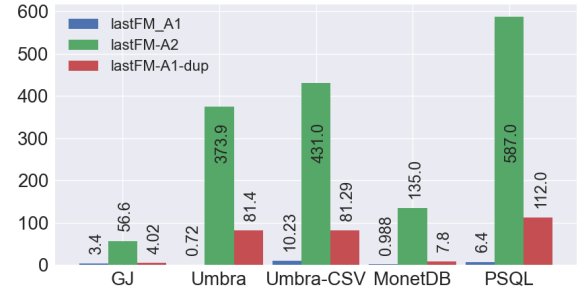
(a)



(b)



(c)



(d)

Figure 4.10: a, b, c and d are for the in-disk time cost (in seconds), in-memory time cost (in seconds), storage cost (in MB) and loading time cost (in seconds), respectively

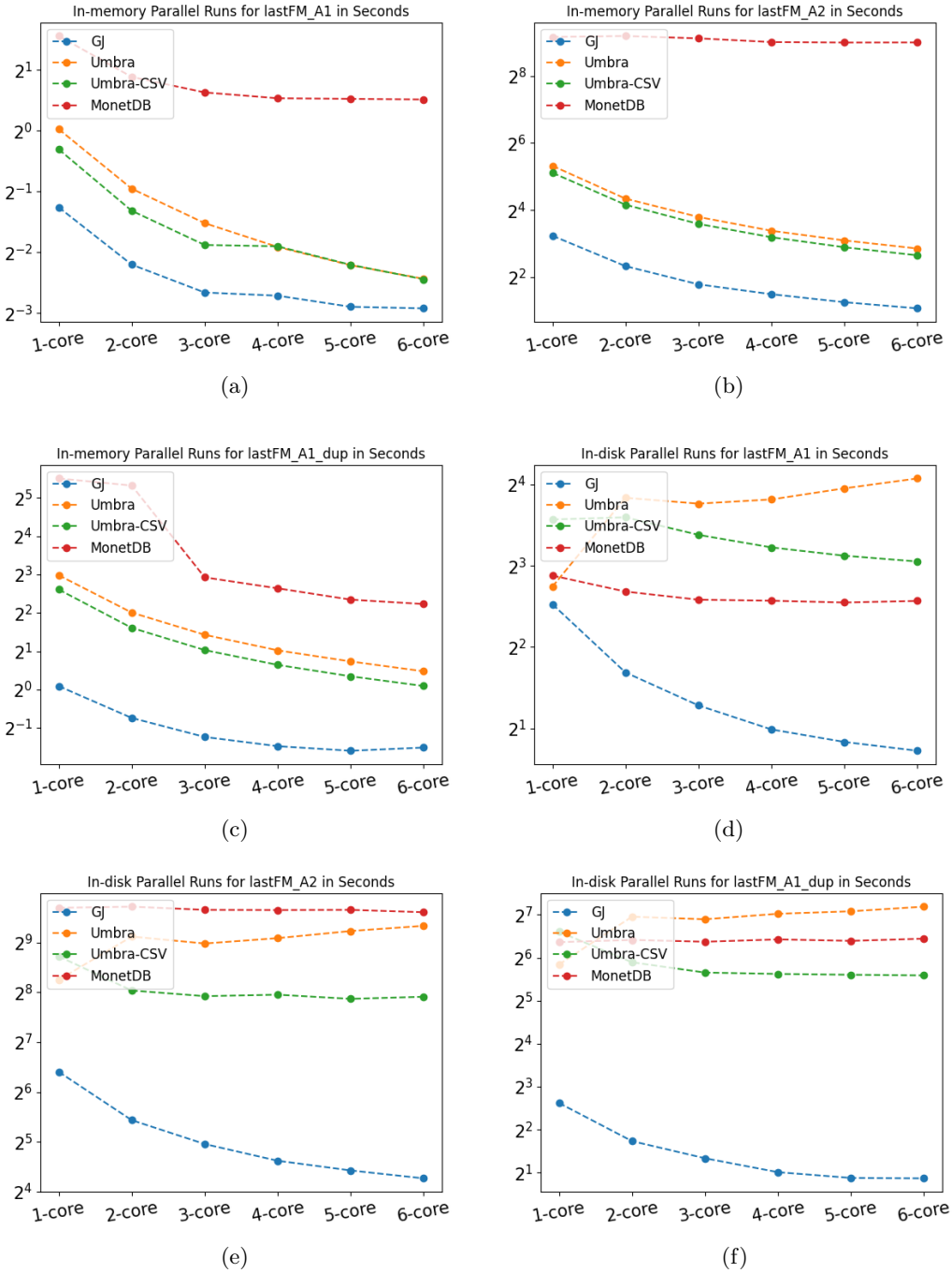
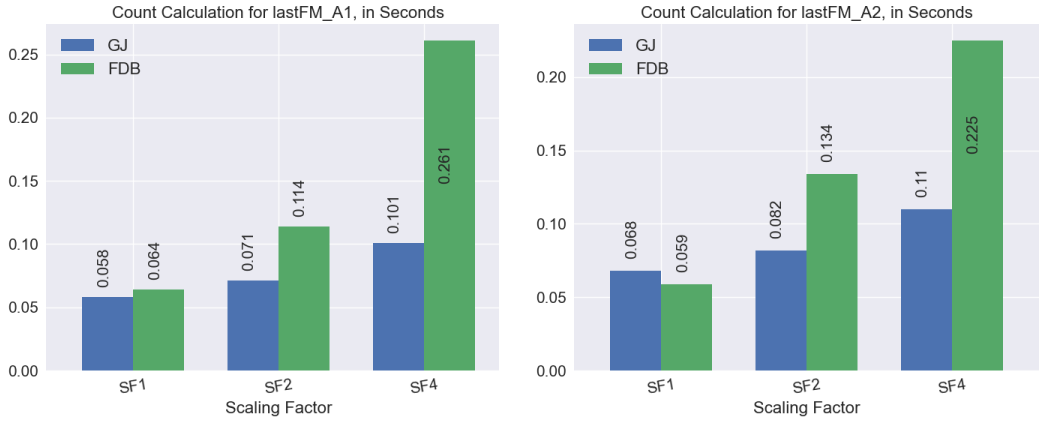


Figure 4.11: a, b and c are the in-memory run time costs in seconds for the three queries, and d,e and f are the in-disk run time costs in seconds for the three queries.



(a) (b)

Figure 4.12: a and b are the time costs in seconds for the queries lastFM_A1 and lastFM_A2 with diferent scaling factors of the data.

factorized join result of FDB are different.

Figures 4.12 (a) and (b) depict the amount of time required for GJ and FDB to calculate the count values for the queries lastFM_A1 and lastFM_A2, respectively. The results indicate that the performance of GJ is not greatly influenced by an increase in the scaling factor, however the performance of FDB is significantly affected.

Remember that the lastFM_A1 query has a lower UIR than the lastFM_A2 query. As both GJ and FDB can effectively manage UIRs, the performance patterns are same for both FDB and GJ.

In [Olteanu and Závodnỳ, 2015], the FDB team has also introduced a constant-delay enumeration algorithm that enumerates the join tuples row by row. The algorithm has never been implemented, tested or compared to other PJAs. They have also not well clarified what the constant-delay cost per tuple exactly is. They claim that using DFS with backtracking to enumerate a tuple on an un-materialized factorized result, the cost is $O(|S|)$, where $|S|$ is the number of attributes in the join result. However, there are some hidden costs associated with the constant-delay term: **(1)** If some of the join attributes are not included in the output (if they are ignored in the final join result), FDB will still pay the enumeration cost for them. Recall, GJ has the early projection solution (described in the section 4.9) to delete join attributes from the factorized joint distribution that are not in the join result. GJ has the frequencies of distinct values in advance, whereas FDB does not; therefore, FDB cannot do early projections. **(2)** another hidden cost with FDB's enumeration

is that FDB needs the tables along with their un-materialized factorized join result (recall FDB uses views; see the background section). Therefore, FDB not only has to go over the factorized representation, but also the tables containing the related information. To enumerate the join result, GJ does not require the actual tables because it already has the materialized factorized distribution. **(3)** one other hidden cost for FDB's enumeration is that it must pay the same cost each time it needs to enumerate the join results. FDB, in other words, is an in-memory engine that does not support the compute-and-reuse scenario. It maintains the factorized result in memory and re-enumerates the tuples whenever required. If FDB wants to store the result on a disk, the alternative is to store the enumerated tuples on the disk. But, if FDB stores/retrieves the factorized result to/from disk, it must pay the serialization/de-serialization cost as well as the enumeration cost each time. While GJ generates the GFJS once and stores it simply and efficiently, and whenever we need the join result, GJ fetches the GFJS and de-summarizes it columnarly without incurring the cost of DFS and backtracking.

In a separate series of experiments, we attempted to simulate the enumeration algorithm of FDB in order to gain a general understanding of how row-by-row enumeration differs from columnar join result generation. In our simulation of FDB, the DFS algorithm is used to enumerate the tuples from our materialized factorized distribution row by row rather than using the un-materialized factorized result. This simulation is less expensive than FDB's actual enumeration algorithm because the simulated algorithm lacks (1) and (2) of FDB's hidden expenses. In other words, the time cost for FDB reported in our experiment is a minimum cost, and the actual costs for FDB may be greater.

For given factorized distributions of queries in the memory, Figure 4.13 shows the speed-up of the columnar join tuple generation (by GJ) vs. row-by-row enumeration of join result (by simulated FDB's enumeration algorithm) for all the queries. While GJ produces the GFJS, it also de-summarizes it columnarly. This is efficient because GJ knows the frequencies of each distinct value in advance and can generate multiples of a distinct value at once without incurring the cost of traversing graphs and accessing indices. When the scaling factor is increased, the frequencies increase as well, hence FDB must pay more for index access and reaching the actual data in the tables. As it shown in Figure 4.13, columnar join result generation is up to 73X faster than row-by-row enumeration.

NB1: Although the asymptotic complexity of both row-by-row and columnar tuple generation is the same (restricted by the join size), in practice, GJ can generate the flat join results much faster.

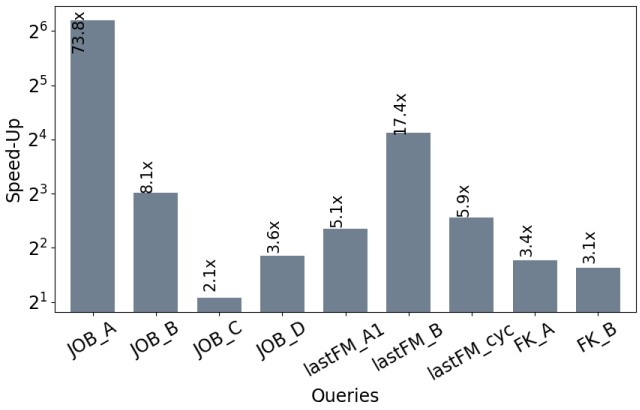


Figure 4.13: Columnar vs. row-by-row result generation

NB2: GJ only creates the GFJS once; thereafter, GJ does not incur the expense of graph traversal and backtracking when reusing a join result. GFJS helps GJ to generate each column of the join result independently from other columns, without accessing several indices.

NB3: As FDB is an in-memory engine, GJ and FDB cannot be compared in a compute-and-reuse scenario. It is unknown how FDB stores and retrieves the un-materialized factorized join result. Consequently, the storage costs for FDB and GJ cannot be compared.

NB4: FAQ employs the same enumeration process as FDB and is also for deductive databases; hence, we do not compare GJ to FAQ. FAQ's primary technical contribution is efficient heuristic techniques for hyper-tree decomposition. These heuristics can also be utilized by our GJ when it builds the junction trees.

4.13 Summary

This work proposes and studies a new approach, GJ, for n-way physical equi-joins in relational databases. GJ effectively tackles the fundamental inefficiencies of join algorithms and it is shown to be worst-case optimal. GJ advocates the generation of an RLE-style, frequency-based summary of the join result. It leverages PGMs and offers the inference algorithms needed to achieve this without computing the join first. GJ then proposes a new approach that consists of first computing the above summary, optionally storing it to disk (to be retrieved later when needed), and followed by desummarization to materialize the flat join result. Detailed experiments reveal that such an approach can yield large performance improvements compared

against both traditional RDBMS (binary) join plans, WOJAs and FDB.

Chapter 5

Sampling Over Joins of Models with PGMs

5.1 Motivation

In many settings, data sizes are growing at an alarming pace. At the same time, machine learning (ML) models, trained over raw tables, are increasingly being used (in lieu of data tables) for many learning and knowledge discovery (LKD) tasks and for replacing traditional DB system components, e.g., for AQP. The existence of such high-quality models can, thus, in principle afford us the ability to "forget" tables and work only with models. This would bear significant benefits. For example, avoiding the ever-increasing maintenance costs with ever-increasing large data by forgetting (deleting) data [Kersten and Sidirourgos, 2017; Milo, 2019], or respecting privacy concerns by learning high-quality models with added noise [Dwork et al., 2006], etc. Given the necessity to perform LKD tasks over joins of tables, we initiate a study of the problems associated with joining models, not tables, putting forth a solution framework, Model-Join, that brings to the surface the key challenges within a principled solution. The aim is to join models in a way that enables LKD over joins as if it were performed on the join of the actual raw tables. The key insight is that the framework integrates the per-table models of the absent tables as a factorized distribution, from which it generates a uniform and independent sample efficiently by utilizing tweaked algorithms of PGMs. The sample obtained by the Model-Join can be used for LKD downstream tasks over joins, such as AQP, or learning models for classification, clustering, regression, cardinality estimation [Yang and al, 2019, 2020; Kipf et al., 2018], or for visualization tasks etc. To our knowledge, this is the first work with this agenda and solutions.

There are many reasons to consider raw data tables as being "absent". For example, big data operators are nowadays faced with formidable challenges managing massive tables (e.g., historical data, IoT data, monitoring/telemetry data, etc.). Hence, they wish to be able to forget data, but without losing the ability to re-obtain the key information of the data [Kersten and Sidirourgos, 2017; Milo, 2019], especially for LKD tasks where approximate answers are acceptable. In fact, the major funder of this research is a leading communications infrastructure company, inundated with telemetry/monitoring data spread across datasets reporting alarms, faults, tickets, and KPI data, such as CPU and network bandwidth utilization, faults, customer SLAs and achieved performance, etc., wishing to be able to perform LKD tasks without maintaining the massive datasets. In response to such requirements, several solutions have been introduced recently to make the learning on tabular data more effective, e.g. Tabnet [Arik and Pfister, 2021] and NODE [Popov et al., 2019] (available in Pytorch Tabular [Joseph, 2021]) and other libraries like Fastai [Howard and Gugger, 2020]). For AQP, methods like [Ma and Triantafillou, 2019; Hilprecht et al., 2020; Ma et al., 2021] are some successful models over relational data tables. Another reason for forgetting tables is that access to raw data tables may be restricted for privacy reasons. Thus, models are trained with added noise to minimize data leaks [Dwork et al., 2014] and they are used in lieu of raw data [Dwork et al., 2006]. Federated learning is another setting where one may have access to models (coming from clients), but not to the underlying local data [Konečný et al., 2016] enabling the resolution of crucial concerns like data privacy, data security, and data access rights. Similarly, in a distributed environment, sharing huge data tables is costly in terms of time/resources/money and it would be highly desirable to transfer compact models rather than the data itself. We will refer to a table as *absent* whenever it undesirable to access it for whatever reason.

However, it is often necessary to execute LKD tasks on the table joins. So how can this be accomplished if the tables themselves are absent? How is a model join query carried out in ML-driven DBs?

Definition 4 (Model Join Query)**.** *A model join query is a join query in which at least one of the tables to be joined is absent and is replaced by a model.*

Definition 5 (Model Join)**.** *The outcome of a join operation on the models of (absent/deleted) tables should enable LKD tasks as if it were conducted on the join of the actual raw tables.*

Note that due to the absence of the underlying tables, none of the existing join sampling methods, such as [Shanghooshabad et al., 2021; Zhao et al., 2018b], are

applicable to a model join query. Also one cannot simply 'chain' models in a row to generate a uniform sample of the join result. Such sample will be statistically poor. Consider the tables and attributes: $D_0(A, B)$, $D_1(B, C)$, and $D_2(C, D)$. Suppose models $\mathcal{M}_0$, $\mathcal{M}_1$, and $\mathcal{M}_2$ and $\mathcal{M}_3$ learn $p(A), p(B|A), p(C|B)$, and $p(D|C)$ on the tables $\{\mathcal{D}_i\}_0^2$. If we use those models in a row, they will generate a uniform sample of $p(A) \cdot p(B|A) \cdot p(C|B) \cdot p(D|C)$, which is not equal to the desired joint distribution $p(A, B, C, D)$ (as per the chain rule). An alternative solution would be to construct a uniform sample of each table $S(D_i)$ using the learnt per-table models and then join the resulting samples. However, $S(D_0) \bowtie S(D_1) \bowtie ... \bowtie S(D_{n-1})$ is not equal to $S(D_0 \bowtie D_1 \bowtie ... \bowtie D_n)$: It is well-known [Shanghooshabad et al., 2021; Zhao et al., 2018b] that this would produce a sample of very poor quality (i.e., not uniform).

Therefore, we wish to start the investigation of joining ML-based per-table models. We will introduce a novel framework Model-Join to generate high quality samples over the join of models. Model-Join treats each relational model as a factor in a factorized distribution represented by a PGM, from which it generates the samples by tweaking the PGMs' inference algorithms. And we will highlight the key challenges and open research problems moving forward. The proposed solution will be, by nature, approximate in the sense that the generated uniform sample will be a uniform sample of the approximate joint distribution. The approximation stems from the given models to the framework. We will prove that, if the models are exact, the generated join samples will also be exact. In other words, the framework's model joining does not add any extra error – it will use exact inference algorithms.

Conversely, the Model-Join framework may act as a facilitator for forgetting raw data tables. In other words, if joining models is feasible, then organizations will not have trouble doing so, and therefore they may be more driven to forget data.

Occasionally, we must join models with existing tables. Therefore, the Model-Join framework must construct its own models based on the existing tables and then join them. Section 5.5 thus provides a discussion of the issues and initial solutions for deriving per-table models in the event that some models are not currently available. However, any other type of model can be integrated in the framework, and custom methods can be utilized. Each per-table model entails a novel blending of embeddings, clusterings, and feed forward Neural Networks.

5.2 Related Work

To the best of our knowledge there is no prior research for enabling LKD tasks over join results of absent tables. This work formulates and solves this problem.

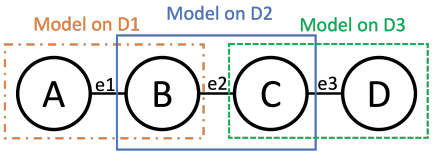


Figure 5.1: The model join MRF for our running example

However, there is some research on join sampling (like the PGM-Join sampler) when the tables are readily available. Regarding the work that is related with the join sampling methods, please refer to Section 3.2.

5.3 Building MRFs for Model Join Queries

The construction of an MRF for a model join query is similar to that of a PGM-Join sampler, with the exception that the potentials in the edges are derived from to-be-joined models. Because of this, we do not generate frequency tables locally because the data is unavailable. The key insight is that gathering all the per-table models in a graph as factors can yield the full factorized joint distribution. Hence, the graph can be considered as a PGM.

As previously explained, there are two components to MRF creation: a qualitative component that includes a graph with nodes and edges, and a quantitative component that contains the real dependencies between the nodes. For the qualitative component, a node is added to the graph per join attribute/variable in tables/-models. The attributes that are not involved in the query, are simply ignored. All involved non-join attributes from the same table are considered as a single node. If any corresponding attributes/variables of two nodes are in the same tables/models, an edge between the nodes is added in the graph, indicating their dependency. Similar to the PGM-Join sampler, we isolate the skeleton (the join attributes) from the non-skeleton of the MRF to simplify the graph and inference process. The cycles are handled in the same way as the PGM-Join sampler.

The Model-Join framework also first generates the samples for the skeleton then it adds the non-skeleton attributes into the samples. Notably, if the existence of a non-join attribute in the MRF graph does not result in a cycle, we can also consider it to be a part of the skeleton, since this will neither increase nor decrease the complexity of MRF construction, inference, or sample production. Therefore, we may consider the attributes A and B to be part of the skeleton in our running example because they do not make any cycles in the MRF graph.

Assume that models $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$ have been learned on our running

example tables $D1$,$D2$ and $D3$, respectively and the tables have been made unavailable (or deleted) for any reason. Figure 5.1 is the MRF tree for that join query. The dependencies in the edges e_1, e_2 and e_3 are obtained from the models $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$ respectively. The factorized joint distribution of the MRF in Figure 5.1 can be represented as below:

$$p(A, B, C, D) \propto \mathcal{M}_1(A, B) \times \mathcal{M}_2(B, C) \times \mathcal{M}_1(C, D) \tag{5.1}$$

The goal is to join the models by using that MRF tree, and generate a (uniform and independent) sample of the join result, "similar" to a uniform sample of the join of the actual tables. This sample will enable LKD tasks over joins of absent tables. The similarity between (a sample of) the actual join result and the uniform sample generated by Model-Join strongly depends on the quality of the per-table models which replace the tables. (Model-Join does not add any error).

Since we use models in the edges of model join MRFs, the inference and sample generation for Model-Join framework is slightly different than for the PGM-Join sampler (to be explained in the next section).

5.4 Inference and Sample Generation Phases

With an example, the differences between the inference and sample generation phases of the Model-Join framework and the PGM-Join sampler are explained.

The first difference is that the edges (dependencies) in the Model-Join framework are represented by pre-learned models. If a model for an edge is not available, a model is trained by the framework. And this leads to the next distinction: the sample generator becomes different than what we saw with the PGM-Join sampler. All that has changed is that the models themselves, along with some supplemental statistics, have been included in the sample generator, as opposed to the PGM-Join sampler that simply a single conditional frequency table for each variable is added to the sample generator.

Consider the query MRF shown in Figure 5.1. Given an elimination order $O = \{D, C, B, A\}$, Model-Join will eliminate (sum out) the variables D, C, B and A, respectively from $p(A, B, C, D)$. A is the root according to O, and D is the first variable to eliminate.

$$\sum_{D,C,B,A} \mathcal{M}_1(A, B) \times \mathcal{M}_2(B, C) \times \mathcal{M}_1(C, D) \tag{5.2}$$

Based on the distributive law, this can be represented as below.

$$\sum_{C,B,A} \mathcal{M}_1(A,B) \times \mathcal{M}_2(B,C) \times \sum_{D} \mathcal{M}_1(C,D) \tag{5.3}$$

After summing out D, a new factor $\phi_{\not{D},C}(C)$ is added:

$$\sum_{B,A} \mathcal{M}_1(A,B) \times \sum_{C} \mathcal{M}_2(B,C) \times \phi_{\not{D},C}(C) \tag{5.4}$$

While eliminating D, a factor $(\mathcal{M}_1(C,D) \times 1)$ is added in the sample generator. Accordingly, $\mathcal{M}_1(C,D)$ can be used to find the frequency of D given C, which can then be multiplied by 1. Note that the model itself is added in the sample generator. Let's call this $(\mathcal{M}_1(C,D) \times 1)$ as $Factor\_D$. $Factor\_D$ gives the frequency of D values given C values over the sub-tree starting from C to the leaves (D), and will be used to generate D in sample generation. As D does not have a child, we multiplied $\mathcal{M}_1(C,D)$ with 1.

Next, C is eliminated. This is done by calculating the product of all the factors containing C then summing out C from the result. After eliminating C, $\phi_{\not{C},B}(B)$ is replaced as below.

$$\sum_{A} \sum_{B} \mathcal{M}_1(A,B) \times \phi_{\not{C},B}(B) \tag{5.5}$$

In addition, a new factor $Factor\_C$ with $(\mathcal{M}_2(B,C) \times \phi_{\not{D},C}(C))$ is added to the sample generator. $Factor\_C$ gives the frequency of C given B over the sub-tree rooted at B.

$\phi_{\not{B},A}(A)$ shows the frequencies of A values over the join result. After elimination of B, a new factor $Factor\_B$ with $(\mathcal{M}_1(A,B) \times \phi_{\not{C},B}(B))$ is added into the sample generator. $Factor\_B$ gives the frequency of B values given A values over the whole tree. What remains is

$$\sum_{A} \phi_{\not{B},A}(A) \tag{5.6}$$

And after eliminating A, we will obtain the join size. The join size is exact if all the models are exact. Since $\phi_{\not{B},A}(A)$ (let's call it $Factor\_A$) is the factor for the root, Model-Join adds this into the sample generator too as the starting point for ancestral sampling.

The final sample generator is shown in Figure 5.2. The sampler will use $Factor\_A$, $Factor\_B$, $Factor\_C$ and $Factor\_D$ to generate the samples for A, B, C and D respectively in the reverse order of O. The sample generation phase is similar to that of the PGM-Join sampler, which employs ancestral sampling. However, in this case, the sample generators comprise the models as well as the factors that

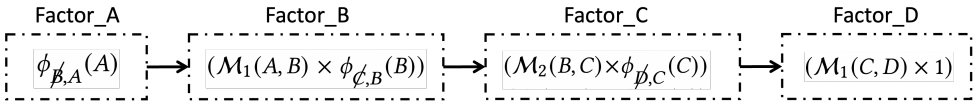


Figure 5.2: The uniform-sample generator for the MRF in Fifure 5.1

should be multiplied while sampling. Moreover, this framework does not build the cumulative distributions in the inference time because of using the models in the sample generator.

Remarks:

- If a node x_i in the MRF has more than one child, to eliminate x_i, there will be more than one factor coming from the children. The product of all factors should be calculated first then x_i is summed out (eliminated) as we explained with the sum-product operation.
- Model-Join framework, similar to the PGM-Join sampler, executes VEA only once to obtain a sample of the joint distribution, as opposed to the conventional method of executing VEA twice to obtain the marginals for all variable pairs.
- If the models to be joined are exact, the Model-Join framework generates a uniform sample of the exact joint distribution. The proof is identical to that of the PGM-Join sampler.

5.5 Learning Per-Table models

The Model-Join framework can build its own models based on the raw data tables. Model-Join does not need to learn the whole table, but just the dependencies among the attributes which are involved in the model join query. Alternatively, one could use a variety of models over relational data – for example, [Ma and Triantafillou, 2019; Ma et al., 2021; Hilprecht et al., 2020] if these are available. However all such models may suffer from poor accuracy (in estimating conditional probabilities) when dealing with attributes having a high number of distinct values (NDVs), as shown in [Ma et al., 2021] or when dealing with non-ordinal categorical attributes. On the other hand, building accurate models is very important in the model join problem because the error (from each per-table model) accumulates when we are to join several models. Thus, we need highly accurate models. Here we present a proposal that addresses these problems. Assume an available table to be joined with other models that has two attributes A and B; we need to learn $p(A|B)$ or $p(B|A)$ depending on the elimination order. Our goal here is to address the two

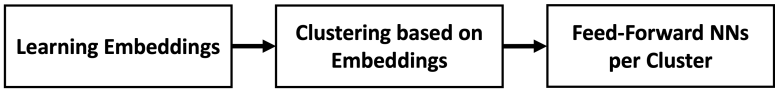


Figure 5.3: Learning models on existing tables

mentioned main challenges: high NDVs and categorical attributes. The key problem with categorical attributes is that even if we use one-hot or binary encoding, we cannot learn a high-quality model over categorical attributes. This is due to the fact that the different values for categorical attributes have no meaningful relationship. The challenge with greater NDVs is that for a given input value, the model may have to predict a huge number of probabilities per distinct value, and learning then becomes impractical when there are large number of distinct values.

To address the first challenge, we learn embeddings per distinct value. In this way, distinct values of an attribute acquire similar embeddings, (positions in the embedded space) if they are related to similar distinct values for the other attributes. This tends to simplify the learning task and improve accuracy as relations between distinct values in the dependent and independent attributes can now be detected based on how close they are in the embedded space. To address the challenge of great NDVs, after finding embeddings for distinct values, clustering is used on the embeddings of the dependent attribute. After clustering the values for the dependent attribute (based on their embeddings), sub-models are learned per cluster, then all sub-models are used as a single model. We use Feed-Forward Neural Networks (FNNs) per cluster. Clustering makes the model more accurate because it deals with smaller learning spaces. Furthermore, it helps to achieve higher efficiency in predicting the conditional probabilities because for a given distinct value of the independent attribute, only a small number of the sub-models are used to predict the probabilities of the distinct values in the dependent attribute. Moreover, this clustering can be considered as a mechanism to guarantee higher accuracy. If the accuracy of the model is low then we can increase it by employing more clusters. It is not difficult to show that clustering can improve accuracy. We may easily employ d clusters, where d is equal to the number of distinct values, to obtain a model with 100% accuracy. This might take the form of a nested hash table, which gives the exact probabilities for each distinct value in the dependent attribute that is associated to the given distinct value in the independent attribute. The exact probabilities are calculated by scanning tables once.

Figure 5.3 shows the main steps of learning models on the existing tables.

5.5.1 The Three Steps of Learning Models

Embedding Learning: Our proposed models are conditional and also discrete in nature (as join attributes are categorical). This calls to mind models used in NLP tasks, operating on individual words (which are discrete values). So, the embeddings we seek will be adapted from NLP models. Embeddings in NLP tasks yield a learned representation of words, so that the words with the same meaning have the same real-valued vector representation. These vector representations are critical in deep learning because if the input of a Neural Network (NN) model does not have meaningful distances between values, it cannot work well. Since finding good models on join attributes is not simple, specially, when the join attributes are keys (i.e., high NDVs), the embedding approach helps us to find more accurate models. There are many embedding approaches; embeddings like Continuous Bag-of-Words (CBOW) and Skip_Gram [Mikolov et al., 2013] have been highly successful for NLP tasks because of the deep linguistic theory behind them (coined distributional hypothesis). In this work we use Skip_Gram with negative sampling [Mikolov et al., 2013] which is faster than the naive Skip_Gram. However, one can employ other kinds of embedding learning methods. The comparison between different embedding learning methods is left for future work.

Notably, the way we adapt these embeddings for our task is based on the values contained in all attributes of interest. Each tuple is viewed as a word sentence and the distinct values for each join attribute as the individual words. The difference between this Skip-Gram and the vanilla Skip_Gram NN is that here the place of words is important (much like in [Song et al., 2018] and [Ling et al., 2015]). Assume a tuple with attributes (A, B, C, D) and the aim is to learn embeddings for A values. The training data will include the pairs of values for (A, B), (A, C) and (A, D). The window size is equal to the size of the tuple. After training the embeddings, the embedding values (real-valued vector representations) are used instead of the distinct values of A and help the final model to distinguish easily among different conditions in the input of the models. This technique significantly increases overall accuracy.

For more information about Skip_Gram refer to the background chapter.

Clustering: After the embedding learning is complete, a clustering method is used to cluster the distinct values according to their embedding vectors. To cluster the distinct values, any clustering technique may be employed; however, we employ the well-known clustering algorithm K-means [MacQueen, 1967; Xu and Wunsch, 2005].

Clustering is only used on the dependent variable, and a sub-model is trained

for each cluster. A dictionary is also created so that for any distinct value v in the independent variable, it returns the clusters in which v exists. We do not need to call all sub-models to predict the probability of the dependent variable if we use this dictionary. Thus, for a given distinct value of the independent variable, the probability of just a restricted group of distinct values is determined from the associated sub-models. The probabilities are considered to be 0 for the remainder of the distinct values in the dependent variable. This is how we treat all sub-models as a single model. The model then substitutes the potential functions in the PGMs' edges.

The clustering of embeddings can enhance accuracy. As our experiments demonstrate, if the accuracy of the sub-models is low, the number of clusters can be raised to improve accuracy. Clustering also guarantees a high level of accuracy. For instance, suppose we use a separate sub-model for every possible combination of two values, so that we always get perfect accuracy.

Feed-Forward NNs as Generative Models: The final component is the NN models. For each cluster a feed-forward NN model is used to find the probabilities for the distinct values of the dependent attribute given the distinct values of the first independent attribute. In the recent literature, one can find several such models, ranging from conditional generative adversarial models (GANs) or auto-encoders (AEs) such as [Mao et al., 2016; Michelsanti and Tan, 2017; Kingma and Welling, 2013]. However, as the NDVs increases, these models underperform. Furthermore, the tuning and training tasks of GANs and AEs are time- and resource-consuming. For these reasons, we turned our attention to simpler (to train, tune, and evaluate) models like probabilistic classifiers, which we adapted to use as conditional-discrete generative models. Here, we re-purpose simple feed-forward NNs, which have been used for classification with high success. Figure 5.4 shows the architecture of our feed-forward conditional-discrete generative NN models. Since join attributes are categorical, the output of the models are also categorical. Then the problem resembles multi-class classification. The input layer of the model are the embeddings of the categorical values in the independent attribute. After hyper-parameter tuning, we chose 5 hidden layers with the *tanh* activation function and a softmax at the end. This softmax layer provides the probabilities per distinct value in the dependent attribute. The loss function is Negative Log Likelihood (NLL). NLL estimates the dissimilarity between the empirical distribution defined by training data and the model distribution. NLL is defined as follows:

$$\mathcal{L} = \mathbb{E}_{x \sim \hat{P}_{data}}[log\ \ p_{model}(x)] \tag{5.7}$$

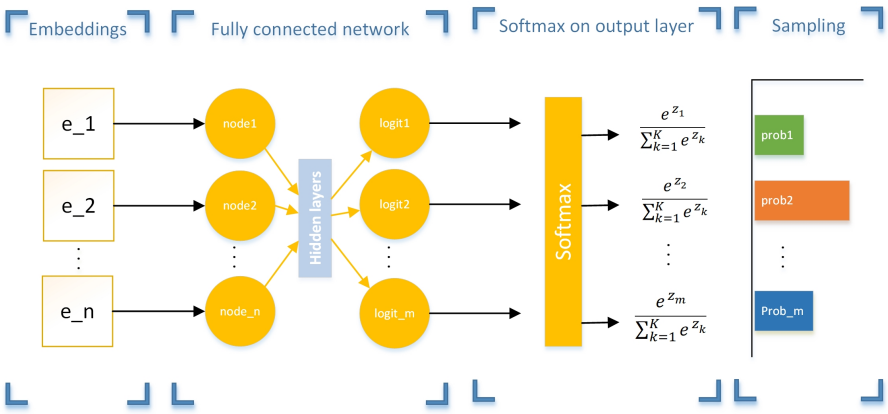


Figure 5.4: Per Cluster Feed-Forward NN Model

When training, the task is like classification, but the usage of the models is slightly different. In a classification task, given an input, the softmax at the last layer yields the probabilities per output class. Then the argmax function is applied on probabilities to find the winning class. Here, instead of using argmax, *we sample using those probabilities per class.* Thus, for each given distinct value in the independent attribute, this finds the probability for each distinct value in the dependent attribute. Then one can generate (as many as needed) distinct values of the dependent attribute.

5.6 Experimental Evaluation

Our experimentation addresses five key issues: (i) quantification of key overheads; (ii) accuracy of the proposed per-table models; (iii) accuracy of the overall framework; (iv) showing that the framework's output sample is indeed a uniform sample of the *true join result*; and (v) exemplify high-quality downstream LKD tasks being performed on the framework's output.

Two sets of experiments are presented. One with very large tables to show that table sizes do not affect the framework's efficiency and only NDVs do. However, when tables are too large (expensive to join), the ground truth to measure actual errors cannot be calculated. So, in the 2nd set of experiments, smaller tables are used so the join can be computed and errors be measured.

Metrics. All reported times (space) are in seconds (MBs). For the quality of generated samples we use (i) the KS-test for sample uniformity and (ii) the F-score for quantifying distances from uniform-random samples of the exact join.

Note that other join sampling methods cannot be used as the to-be-joined

DB name	table name	NVDs in 1st Att.	NVDs in 2nd Att.	1st_2nd Distincts	Num. of records
SynthDB1	tbl0	100,000	100,000	100,000	149,895,600
	tbl1	100,000	100000	500,000	749,967,400
	tbl2	100,000	100000	1,000,000	1,499,818,300
	tbl3	100,000	100000	2,500,000	3,747,965,800
	tbl4	100,000	100000	5,000,000	7,499,960,400
	tbl5	100,000	100000	7,500,000	11,244,111,000
SynthDB2	tbl0	10,000	5000	1,000,000	991,047,100
	tbl1	50,000	5000	1,000,000	998,403,000
	tbl2	100,000	5000	1,000,000	1,284,291,300
	tbl3	5,000	10000	1,000,000	988,931,000
	tbl4	5,000	50000	1,000,000	998,533,100
	tbl5	5,000	100000	1,000,000	1,379,785,400
	tbl6	10,000	10000	1,000,000	995,563,500
	tbl7	50,000	50000	1,000,000	1,000,400,900
TPC_DS	Store_sales	customers=273,443	items=54,000	13,745,062	1,375,167,200
	Store_returns	stores=27	reasons=39	1053	136,278,000
	Store_sales	items=54,000	stores=183,284	1,452,252	1,375,193,700
	Catalog_sales	ship_mode=20	items=54,000	1,064,990	716,351,500

Table 5.1: Data characteristics

tables are absent.

5.6.1 Experimental Setup

System configuration. Model training uses a GPU GeForce RTX 2080 Ti, with 11 GB memory. Model joining uses a system with 64GB main memory and the E5-2660 CPU with 20, 2.60GHz cores.

Data and queries. In Table 5.1 we show statistics for the synthetic and benchmark data. The first synthetic data $SynthDB1$ has a fixed NDVs, but variant NDVs pairs (independent attribute, dependent attribute). The second synthetic data $SynthDB2$ has variant NDVs in the attributes, and the number of distinct pairs (independent attribute, dependent attribute) is variant. All of our synthetic tables have 2 columns, and the values are generated randomly. We also generated data from the TPC-DS benchmark [Nambiar and Poess, 2006] with a scaling factor 10 and then replicated the data 100 times.

In Table 5.1, the first and second attributes are independent and dependent variables of the models. The table displays the number of distinct values in the independent and dependent variables, as well as the number of distinct pair values in both variables and the number of records in the tables.

Note that table sizes run in the billions of rows, which is a good reason for replacing tables with learned models.

We analyze the performance of the following four join queries, which are sufficient to reveal the key features:

Q1(SynthDB2): $tbl2 \bowtie tbl3 \bowtie tbl0$
Q2(SynthDB2): $tbl0 \bowtie tbl1 \bowtie tbl2 \bowtie tbl3 \bowtie tbl4$

DB	Table	Clusters	Hidden_nodes	Max_epochs
SynthDB1	tbl0 (att0 → att1)	100	200	3
	tbl1 (att0 → att1)	100	200	3
	tbl2 (att0 → att1)	100	200	3
	tbl3 (att0 → att1)	100	200	3
	tbl4 (att0 → att1)	100	200	5
	tbl5 (att0 → att1)	100	200	5
SynthDB2	tbl0 (att0 → att1)	50	200	3
	tbl1 (att0 → att1)	50	200	3
	tbl2 (att0 → att1)	50	200	3
	tbl3 (att0 → att1)	50	200	3
	tbl4 (att0 → att1)	50	200	3
	tbl5 (att0 → att1)	50	200	3
	tbl6 (att0 → att1)	50	200	3
	tbl7 (att0 → att1)	50	200	3
TPC_DS	Store_sales(customers→ items)	50	300	20
	Store_sales(items→ store_id)	5	10	10
	Catalog_sales(SM_id→ items)	20	30	10
	Store_returns(store_id→ reason_id)	5	10	10

Table 5.2: Hyper-parameters for training the models

Q3(TPC-DS): $Web\_sales \bowtie_{cus} Store\_sales \bowtie_{items} Store\_returns$
Q4(TPC-DS): $Ship\_mode \bowtie_{SM\_id} Web\_sales \bowtie_{SM\_id} Catalog\_sales \bowtie_{items}$
$Store\_sales \bowtie_{store\_id} Store\_returns \bowtie_{reason\_id} Reasons$

With synthetic data, we test the worst case when there are no meaningful relationships between the independent and dependent attributes' values, and we examine our learning method when the distinct values of variables are selected randomly (uniformly).

Reproducibility. Table 5.2 shows the hyperparameters for models per table involved in the join queries. The third column shows the number of clusters, the fourth shows the number of nodes per layer (in all cases we use 5 layers), and the last shows the maximum epochs. AdamOptimizer is used for all models with learning rate 0.0005. Implementation used Python and Tensorflow. The code and documentation can be found at: `https://github.com/shanghoosh1/ModelJoin`

5.6.2 Efficiency and Overheads

We first illustrate the results of learning per-table models. Then, given the models, we evaluate Model-Join.

Learning the models consists of embeddings, clustering and training NNs. Table 5.3 shows the learning time costs per table. Table 5.4 shows total storage costs for everything needed to treat all sub-models as a single model. These run from a few MBs to a few hundred MBs even for tables of size of 100s of GBs which is one of the good reasons to forget the data.

Table 5.5 shows sample-generation times. The time cost in Table 5.5 varies

DB name	table name	1st Embedding	2nd Embedding	Clustering	Training	Total
SynthDB1	tbl0 (att0 → att1)	385	336	74	1009	1804
	tbl1 (att0 → att1)	801	763	186	1939	3689
	tbl2 (att0 → att1)	1389	1185	177	3655	6406
	tbl3 (att0 → att1)	1562	1211	257	8595	11625
	tbl4 (att0 → att1)	1046	1069	345	27344	29804
	tbl5 (att0 → att1)	1190	763	434	41462	43849
SynthDB2	tbl0 (att0 → att1)	415	614	20	1295	2344
	tbl1 (att0 → att1)	621	606	22	1341	2590
	tbl2 (att0 → att1)	809	826	30	1691	3356
	tbl3 (att0 → att1)	500	512	21	1398	2431
	tbl4 (att0 → att1)	608	607	61	2186	3462
	tbl5 (att0 → att1)	812	853	193	4001	5859
	tbl6 (att0 → att1)	596	437	22	1402	2457
	tbl7 (att0 → att1)	616	622	57	2203	3498
TPC_DS	Store_sales(customers → items)	834	793	354	24537	26518
	Store_sales(items → store_id)	116	167	26	575	884
	Catalog_sales(SM_id → items)	97	68	28	5754	5947
	Store_returns(store_id → reason_id)	33	40	3	33	109

Table 5.3: Time cost in seconds

DB name	table name	Embedding	Clustering	Freq	NN	Total
SynthDB1	tbl0 (att0 → att1)	70	2	2	379	453
	tbl1 (att0 → att1)	110	7	3	448	568
	tbl2 (att0 → att1)	111	11	3	473	598
	tbl3 (att0 → att1)	114	23	3	473	613
	tbl4 (att0 → att1)	116	40	3	473	632
	tbl5 (att0 → att1)	117	53	3	473	646
SynthDB2	tbl0 (att0 → att1)	9	4	0.2	120	133.2
	tbl1 (att0 → att1)	31	9	0.7	120	160.7
	tbl2 (att0 → att1)	59	12	1	120	192
	tbl3 (att0 → att1)	9	2	0.2	133	144.2
	tbl4 (att0 → att1)	31	2	0.7	234	267.7
	tbl5 (att0 → att1)	59	2	1	361	423
	tbl6 (att0 → att1)	11	4	0.2	135	150.2
	tbl7 (att0 → att1)	56	9	1	234	300
TPC_DS	Store_sales(customers → items)	186	78	2	430	696
	Store_sales(items → store_id)	30	3.6	0.7	0.5	34.8
	Catalog_sales(SM_id → items)	6.3	0.005	0.7	24	31
	Store_returns(store_id → reason_id)	0.008	0.002	0.85	0.5	1.36

Table 5.4: Storage cost in MBs

Query	Inference	Sampling	Total
Q1	45	799	844
Q2	157	1032	1189
Q3	4060	2646	6706
Q4	35	3492	3527

Table 5.5: Time-cost for generating a uniform 100k sample

from ca. 14 minutes to ca. 112 minutes to generate a sample with 100k rows. As each sample data point is generated independently from others, sample generation is embarrassingly parallelizable, so times will be decreased by N with N cores.

5.6.3 Quality of Models and Join Sample

The major purpose is to demonstrate that, despite the fact that the per-table models have some errors, the generated samples are nonetheless uniform. Hence, we do not compare our learning approach to other options (which will be considered as future work.) First, we demonstrate the accuracy of the models, and then, utilizing those models in the Model-Join framework, we demonstrate the uniformity of the generated samples using the KS-test. The KS-test is given the actual join result and the sample generated by the framework, just like our uniformity evaluation in Chapter 3.

Per-table Model Accuracy.

Table 5.6 shows the accuracy of per-table models. Note that F-score is already very high. This is one of the reasons why tables can be "absent" safely, and we will not lose much by forgetting the data. Note: in these experiments we purposely do not allow models to be 'too' accurate (keeping the number of the clusters low) because we wish to see exactly the effect of errors in the samples. Recall, we proved if the models are 100 percent accurate, the samples will be an exact uniform sample of the join result since the Model-Join does not introduce any additional error. This has been shown in with the PGM-Join sampler where all the potentials are exact.

KS-test of Generated Samples

The null hypothesis is that the sample is a uniform sample of the join result.

Figure 5.5 shows the actual CDFs, the CDFs of our generated samples and the boundaries of KS-test that come from the critical values on TPC-DS queries. If our approximate CDF line goes outside the boundaries, it means the null hypothesis is rejected. The pattern for all queries is the same.

For Q1 and Q2, we made small tables from $SynthDB2$ with 20k samples, then built models from the independent attribute to the dependent attribute in the tables. The exact join result sizes are 162,271 and 11,840 tuples respectively for Q1 and Q2. We generate samples with sizes 20k and 2k for Q1 and Q2, respectively. The critical values become 0.012 and 0.039. The KS-statistics for Q1, Q2 are 0.0029, 0.011. Thus the null hypothesis (easily) holds and the sample is declared uniform.

DB name	table name	F-score	Intervals
SynthDB1	tbl0 (att0 → att1)	0.984	0.009
	tbl1 (att0 → att1)	0.9926	0.005
	tbl2 (att0 → att1)	0.9929	0.005
	tbl3 (att0 → att1)	0.9823	0.008
	tbl4 (att0 → att1)	0.948	0.013
	tbl5 (att0 → att1)	0.93	0.02
SynthDB2	tbl0 (att0 → att1)	0.944	0.014
	tbl1 (att0 → att1)	0.968	0.01
	tbl2 (att0 → att1)	0.973	0.009
	tbl3 (att0 → att1)	0.922	0.16
	tbl4 (att0 → att1)	0.918	0.017
	tbl5 (att0 → att1)	0.93	0.015
	tbl6 (att0 → att1)	0.932	0.015
	tbl7 (att0 → att1)	0.955	0.13
TPC_DS	Store_sales(customers→ tems)s	0.941	0.014
	Store_sales(items→ store_id)	0.91	0.01
	Catalog_sales(SM_id→ items)	0.986	0.007
	Store_returns(store_id → reason_id)	0.99	0.005

Table 5.6: F-score and Confidence intervals with $\alpha = 95\%$

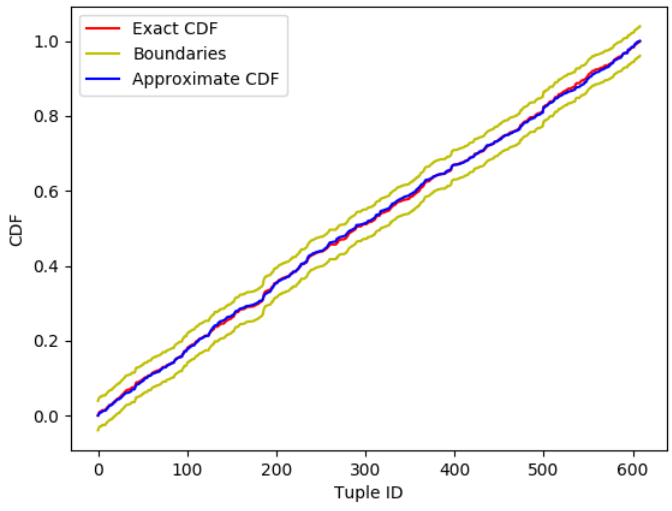


Figure 5.5: CDFs comparison, KS-test

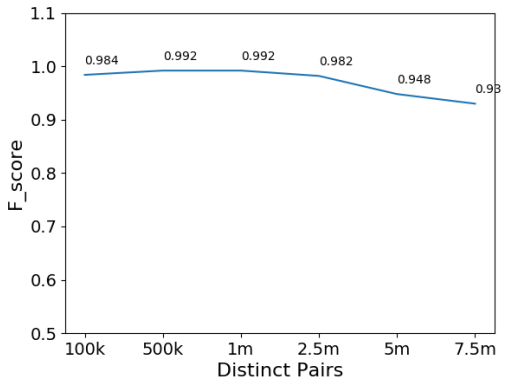


Figure 5.6: F-score vs. Number of Distinct Pairs

For Q3 and Q4, again, we create smaller tables. For Q3 (Q4) we take 100k (5k) of the data points in the involved tables. Next, we build our models over them. The sizes of the exact join results of the small tables for Q3 and Q4 in the skeleton are 84,279 and 28,407,118. For Q3 and Q4, Model-Join generates 50k and 100k, respectively. The KS critical values for these sample sizes are 0.0091 and 0.0051 respectively for Q3 and Q4 with $\alpha = 0.01$. The KS-statistic values are 0.0032 and 0.0048, respectively. These results imply that the null hypothesis (of the generated sample being uniform) is not rejected for both of them.

5.6.4 Impact of Numbers of Clusters

Here, we use $SynthDB1$ tables. The relation between the pair of attribute values here is random to stress-test the models and highlight the clustering benefits. Figure 5.6 shows F-scores on the 6 $SynthDB2$ tables (with 50 clusters). Each x-axis point refers to the number of distinct pairs (NDPs) in one of the 6 tables. The F-score worsens when increasing NDPs.

Figure 5.7 shows the positive effect on F-score when using more clusters with model of $tbl2$ in $synthDB1$. Thus, when NDPs and NDVs in the independent and dependent attributes increase and accuracy worsens, more clusters can improve the F-score. Increasing the number of clusters increases the number of models per table, but the size of each model per cluster becomes smaller. Nonetheless, clustering helps us have accurate models. If more clusters are used, the smaller the cluster sizes will be, the higher the accuracy will be. Thus, we can use this knob to achieve high accuracy. In theory, the number of clusters could be equal to the number of distinct independent-dependent attributes' pairs. Then, accuracy always will be perfect, but

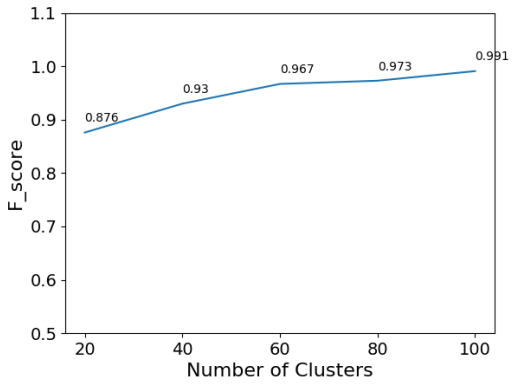


Figure 5.7: Effect of increasing the number of clusters

with a much higher cost.

5.6.5 Downstream LKD Over the Model Join Samples

We exemplify the usefulness of Model-Join for downstream LKD tasks. Works in data analytics and management have already shown that uniform samples can be used to train accurate LKD models [Hilprecht et al., 2020; Ma and Triantafillou, 2019; Ma et al., 2021; Yang and al, 2020]. Here we add to this by exemplifying that the uniform sample generated by Model-Join can be used for classification tasks. We show results using the join result of Q4 for a binary classification task. The labels are the store-id (1 or 2) and the independent variables are all the other attributes in the result. The popular XGBClassifier from XGBoost library is used to learn the classification models with the same hyperparameter tuning.

First, we take the exact join result of Q4 (which is 7m rows) and split it into training (80%) and testing (20%) parts, randomly. Note, this testing set is used to test all three scenarios below: (1) We train XGBClassifier with the training set (producing $\mathcal{M}_{main}$). (2) We create a 10k-sample from the training set (i.e., the 80% of the true join result) and train XGBClassifier on it (producing $\mathcal{M}_{sample}$). (3) we train XGBClassifier over the approximate 10k-sample produced by Model-Join (producing $\mathcal{M}_{ModelJoin}$).

The F-scores for $\mathcal{M}_{main}$, $\mathcal{M}_{sample}$ and $\mathcal{M}_{ModelJoin}$ are 64.33, 62.89 and 62.71, respectively. This shows two facts: i) The F-scores for $\mathcal{M}_{sample}$ and $\mathcal{M}_{ModelJoin}$ are close to $\mathcal{M}_{main}$ which shows that using uniform samples from large tables instead of the raw tables for training is a good choice. In other words, the error due to sampling is small. ii) $\mathcal{M}_{sample}$ and $\mathcal{M}_{ModelJoin}$ have very similar F-scores. This

shows that Model-Join can indeed enable high-quality downstream LKD tasks over absent tables, despite creating a sample of an approximated join result.

5.7 Open Challenges

Model joining presents a new suit of key open challenges. Below is a list of open challenges, and in the next section, we will elaborate on them and propose preliminary solutions for some of them.

- How can Model-Join generate other kinds of samples such as weighted samples?
- How can one predict the error of the generated samples for given to-be-joined models as the per-table models' errors propagate down the chain and accumulate?
- How does the error of the samples impact the downstream analytics tasks?
- When Model-Join operates with different types of per-table models, how will different kinds of models behave when integrated with the Model-Join framework? And how does our model learning method compares to other methods?
- How might the inference time and sampling time and quality be improved?
- How can Model-Join be utilized to improve performance for (analytics over) distributed joins in which the models are shared as opposed to the data?
- Applying the Model-Join framework as a federated learning approach for scenarios in which the data in several clients are conditionally independent.
- To handle cyclic queries, one can utilize rejection sampling or hyper-tree decomposition like junction tree creation (to convert the cyclic graph to an acyclic one). Which one is superior?
- What other applications exist for the PGM of a join query?
- The KS-test reveals if the sample is uniform or not, but does not quantify the sample's deviation from uniformity. Are other measures for gauging uniformity better suited?

5.8 Summary

We studied an emerging problem in modern ML-driven databases in which ML models substitute the tables for analytics. We discussed and evaluated new solutions to

the model join problem and pointed out the key issues to be addressed at the levels of integrating the per-table ML models (using the principles of PGMs) and addressing the key challenges for deriving efficient per-table models (namely, high NDVs and categorical attributes, using embeddings, clustering, and feed-forward NNs). To our knowledge, Model-Join is the only existing solution for the model join problem. The experimental results showed the high quality of the generated samples. We hope this work will motivate a new line of research in ML for databases.

Chapter 6

Conclusion and Future Work

The main goal of this thesis is to study how ML and statistical models might assist data management systems in more successfully carrying out their tasks, specifically in dealing with the often costly process of joining normalized tables.

Joins are considered as ever-present and expensive operations in relational databases. When the tables and the join size are large, especially when they involve several tables with many-to-many relationships, a number of difficulties arise: the processing time for the join increases, the space becomes difficult to manage and support, and the costs in the cloud skyrocket. Apart from these, this thesis also defines and investigates a new challenge associated with the join operation. When the tables are too huge or include sensitive information (private data), models are typically constructed over the massive tables, and only the models are made accessible to others. Therefore, it is impossible to perform analytics on joins because there is no solution for joining models.

To address these challenges, we presented multiple solutions in various contexts based on a single core idea: Learn the distribution of the join result. The key insight is that everything becomes simple if we know what is the distribution of the join result. Because of the joint distribution, we may quickly and simply generate uniform samples, enumerate join tuples, and solve the model join problem.

This thesis demonstrated a method for learning the joint distribution of a join query without having to generate the join result itself. It was clarified how factorization in PGMs is related to normalization in relational databases. We showed that a PGM can be easily derived for join queries in relational databases, as the non-join attributes from the different tables are independent of each other when the join attributes are observed. This brings to mind the conditional independence that can be found in PGMs. Thus, we attempted to train PGMs for join queries and modified

PGM's efficient, principled, and easy-to-understand algorithms to address a variety of problems.

We identified three categories of join problems where PGMs could be useful in cutting costs: (1) join sampling problem, (2) physical join algorithm problem, and (3) model join problem.

6.1 Sampling Without Generating the Join Result

Uniform sampling over joins without generating the join result is a promising solution to deal with both the time and space costs of the joins. In this thesis, the join sampling problem was adopted to PGMs for the first time and the algorithms were tweaked to efficiently generate uniform and independent samples of join results. The learned PGMs are exact, so the drawn samples come from the exact joint distributions. We also proved that the generated samples are uniform, and by using the KS-test we showed experimentally that our proof is indeed true. Moreover, a way of dealing with cyclic join sampling was introduced and tested.

Lots of experiments with benchmarks and real databases (TPCH, Twitter, and JOB) were carried out, and the results showed that the table sizes do not affect our sampling method because of the repetitions in the data, but they do affect other competitors. The performance of our method improves as the size of tables and the number of repetitions in tables increase. This is an important achievement regarding our core idea of learning the joint distribution of a join query. The improvement seen in the experiments was up to 28X faster than other competitors.

6.2 A new PJA with Summarization/Desummarization Technique

Sampling is not always sufficient; even when the join size is too large, it is occasionally necessary to generate and store/retrieve the full join result. This is especially true when we require the exact answers to the queries. Or when the results of one query are entered into another.

In this thesis, we presented a mapping from the physical n-way equi-join problem to PGMs, allowing us to create the join result efficiently and columnarly. Instead of constructing indexes and utilizing binary-join algorithms or WOJAs over raw data tables, our new PJA (named GJ) first produces a (PGM-based) join summary, then optionally stores it to disk and retrieves it from disk when requested to (re)produce the join result, and lastly desummarizes it. The summary is a kind of RLE over

join, and for the first time, we showed how to generate it without generating the join result.

We also provided a complexity analysis of GJ, showing that GJ is a worst-case optimal join algorithm.

Experiments with JOB, TPCH and lastFM data and queries showed GJ to be faster than PSQL up to 64X, more than 388X faster than MonetDB, and up to 6X faster than Umbra. GJ shines when large join results are to be stored on disk and reused (up to 820X faster than PSQL, up to 717X faster than MonetDB, up to 165X faster than Umbra and 94X faster than Umbra-CSV). GJ uses less storage than all competitors, up to 21,488X better than PSQL, up to 38,333X better than MonetDB, up to 78,750X better than Umbra and 54,666X better than Umbra-CSV. Because of this, GJ can actually produce the join result even in cases of very large join results where currently RDBMS join plans and WOJAs fail. We also compared GJ and FDB and showed that GJ's factorized joint distribution is better than FDB's factorized join result as the scaling factor of the data goes up. Additionally, we demonstrated that enumeration with FDB can be less efficient than columnar tuple generation with GJ.

6.3 A New Framework for Joining Models Rather Than Tables

Throughout our investigation, we uncovered a novel problem facing today's ML-driven, relational databases. While we may not always have access to the underlying tables, we may be able to employ models built on top of the tables. In this thesis, we defined a novel problem that we refer to as the *model join* problem; in this scenario, ML models are joined rather than tables, and our PGM-based method is suggested to do this.

We also looked into two fundamental problems with learning models over tabular data (the large number of different values and categorical attributes) and came up with solutions to solve them.

At the end, a list of open challenges to be addressed en route of *model join* problem was provided (please see Section 5.7).

6.4 Future Work

Throughout the research for this thesis, we have uncovered and recognized a number of interesting opportunities for future work to improve and promote the utilization of PGMs in relational databases.

For the inference phase of the PGM-Join sampler, Model-Join sampler, and the GJ algorithm, approximation ideas could be applied. This may aid in improving the performance of the inference phase, as well as the sample and summary generation phases. If the inference is approximate, the resulting joint distribution is approximate, and consequently, the generated samples and summaries become approximate as well. For example, potential functions could be generated by scanning samples of tables rather than whole tables. Or, approximate inference algorithms from the PGMs' world could be used.

As another opportunity, the distributed version of the three proposed methods also appears interesting for future investigation. This is particularly promising for the GJ algorithm.

As previously stated, various open challenges remain to be investigated in relation to the model join problem (please see Section 5.7). Here, we discuss a few of them and suggest potential solutions.

Beyond Uniform Sampling: How can Model-Join generate other kinds of samples over joins such as weighted sampling [Shekelyan et al., 2022] or stratified sampling [Kandula et al., 2016]? Below, we sketch an alternative for these sample types. Stratified sampling involves taking samples for each stratum independently. Within each stratum, simple random sampling is used. Typically, the stratum in a database is defined by group-by attributes. Thus, to construct a stratified sample based on the strata of the group-by attributes, all that is required is to generate uniform samples per group. Our adaptation approach is to keep the group by attributes as the MRF's root. The VEA is the same, and by using the same VEA, we eliminate all nodes except the root attributes. At sampling time, samples are generated for each entry in the root. Take note that in uniform sampling, root entries are selected at random based on their frequencies and the join size, but in stratified sampling, samples are generated independently for each group. Uniform samples should be generated for all group entries. For weighted sampling, the procedure is the same. The attributes that specify the weights should be retained as the MRF's root. All of the remaining nodes in the MRF are eliminated using the same VEA. The weights per entry in the root are calculated by combining the factors received by the root from its children and the weights obtained from users. Then, during sampling time,

the entries for the root are chosen based on their new weights, and the values for all other nodes are added in the same way as the uniform sampling.

Aggregation Calculation without Generating Samples: How can aggregations be calculated without generating the samples? Aggregations such as Sum, Average, Count, can be calculated over joins while performing the variable elimination algorithm. In doing so, there is no need to build the sample generator anymore. One solution could be to keep the attributes that the aggregations are on as the root of the MRF and then eliminate all the other nodes with VEA. Methods like FDB [Olteanu and Schleich, 2016] and [Abo Khamis et al., 2016] are some good examples of calculating the aggregations over factorized data. In our case, those methods can be used on our factorized models. Furthermore, if we want to calculate many aggregations over the join of the models with different attributes, we can construct samples and calculate all aggregations over the sample.

Efficiency Improvements: Future improvements in performance are warranted. One option is to adapt approximate inference algorithms at the PGM level [Murphy et al., 2013]. Another option is to exclude some of the distinct values when building the per-table models or when using the models in inference. In this case, the inference and sample generation will be faster because the number of times the per-table models will be called will be decreased. Nonetheless, both of the above techniques raise key issues, with respect to trade-offs between the quality of samples for downstream tasks and efficiency, which deserve attention.

Federated Learning: Federated learning has received increased attention in recent years as a new collaborative learning paradigm [McMahan et al., 2017; Yang et al., 2019], and the limitations of federated learning from non-i.i.d data have been identified, particularly for supervised learning tasks [Li et al., 2022; Zhao et al., 2018a; Li and et al., 2020; Xie et al., 2019; Yu et al., 2020]. Using Model-Join as a federated learning strategy for scenarios where data across several clients is (fully) conditionally independent appears to be an appealing idea. In fact, one appealing goal will be to learn a general AQP model from the AQP models provided by clients.

We hope this Model-Join work will motivate a new line of research in ML for databases.

Bibliography

Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. Faq: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 13–28, 2016.

Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 275–286, 1999.

Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42, 2013.

Christos Anagnostopoulos and Peter Triantafillou. Learning set cardinality in distance nearest neighbours. In *2015 IEEE international conference on data mining*, pages 691–696. IEEE, 2015.

Christos Anagnostopoulos and Peter Triantafillou. Efficient scalable accurate regression queries in in-dbms analytics. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 559–570. IEEE, 2017a.

Christos Anagnostopoulos and Peter Triantafillou. Query-driven learning for predictive analytics of data subspace cardinality. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11(4):1–46, 2017b.

Sercan Ö Arik and Tomas Pfister. Tabnet: Attentive interpretable tabular learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6679–6687, 2021.

Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L Reutter, Javiel Rojas-Ledesma, and Adrián Soto. Worst-case optimal graph joins in almost no space. In

Proceedings of the 2021 International Conference on Management of Data, pages 102–114, 2021.

Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 739–748. IEEE, 2008.

Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 261–272, 2000.

Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*, pages 208–222. Springer, 2007.

Christopher M Bishop. Mixture density networks. 1994.

Andrew Blake, Pushmeet Kohli, and Carsten Rother. *Markov random fields for vision and image processing*. MIT press, 2011.

Peter Brusilovsky, Iván Cantador, Yehuda Koren, Tsvi Kuflik, and Markus Weimer. Workshop on information heterogeneity and fusion in recommender systems (hetrec 2010). In *Proceedings of the fourth ACM conference on Recommender systems*, pages 375–376, 2010.

Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, P Krishna Gummadi, et al. Measuring user influence in twitter: The million follower fallacy. *Icwsm*, 10(10-17): 30, 2010.

Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. *ACM SIGMOD Record*, 28(2):263–274, 1999.

Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

Robert G Cowell, Philip Dawid, Steffen L Lauritzen, and David J Spiegelhalter. *Probabilistic networks and expert systems: Exact computational methods for Bayesian networks*. Springer Science & Business Media, 2007.

Amol Deshpande and Sunita Sarawagi. Probabilistic graphical models and their role in databases. In *Proceedings of the 33rd international conference on very large data bases*, pages 1435–1436, 2007.

David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data*, pages 1–8, 1984.

David J DeWitt, Jeffrey F Naughton, and Joseph Burger. Nested loops revisited. In *[1993] Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 230–242. IEEE, 1993.

Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, and Peter Widmayer. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 299–310. Elsevier, 2002.

Joshua D Drake and John C Worsley. *Practical PostgreSQL*. " O'Reilly Media, Inc.", 2002.

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

Alan Fekete, Brody Franks, Herbert Jordan, and Bernhard Scholz. Worst-case optimal radix triejoin. *arXiv preprint arXiv:1912.12747*, 2019.

Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment*, 13(12):1891–1904, 2020.

Kamel Garrouch and Mohamed Nazih Omri. Bayesian network based information retrieval model. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 193–200. IEEE, 2017.

Walter R Gilks and Pascal Wild. Adaptive rejection sampling for gibbs sampling. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 41(2):337–348, 1992.

Goetz Graefe. Sort-merge-join: An idea whose time has (h) passed? In *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*, pages 406–417. IEEE, 1994.

Andrey Gubarev, Dan Delorey, Geoffrey Michael Romer, Hossein Ahmadi, Jeff Shute, Jing Jing Long, Matt Tolton, Mosha Pasumansky, Narayanan Shivakumar, Sergey Melnik, et al. Dremel: A decade of interactive sql analysis at web scale. 2020.

Peter J Haas. Large-sample and deterministic confidence intervals for online aggregation. In *Proceedings. Ninth International Conference on Scientific and Statistical Database Management (Cat. No. 97TB100150)*, pages 51–62. IEEE, 1997.

Peter J Haas and Joseph M Hellerstein. Ripple joins for online aggregation. *ACM SIGMOD Record*, 28(2):287–298, 1999.

Marios Hadjieleftheriou, Xiaohui Yu, Nikos Koudas, and Divesh Srivastava. Hashed samples: selectivity estimators for set similarity selection queries. volume 1, 2008.

W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. 1970.

Joseph M Hellerstein, Peter J Haas, and Helen J Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 171–182, 1997.

Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: learn from data, not from queries! *Proceedings of the VLDB Endowment*, 13(7):992–1005, 2020.

Jeremy Howard and Sylvain Gugger. Fastai: a layered api for deep learning. *Information*, 11(2):108, 2020.

S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin*, 2012.

Finn V Jensen and Thomas Dyhre Nielsen. *Bayesian networks and decision graphs*, volume 2. Springer, 2007.

Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable approximate query processing with the dbo engine. *ACM Transactions on Database Systems (TODS)*, 33(4):1–54, 2008.

Manu Joseph. Pytorch tabular: A framework for deep learning with tabular data. *arXiv preprint arXiv:2104.13638*, 2021.

Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 international conference on management of data*, pages 631–646, 2016.

Ahmet Kara and Dan Olteanu. Covers of query results. *arXiv preprint arXiv:1709.01600*, 2017.

Robert E Kass, Bradley P Carlin, Andrew Gelman, and Radford M Neal. Markov chain monte carlo in practice: a roundtable discussion. *The American Statistician*, 52(2):93–100, 1998.

Martin L. Kersten and Lefteris Sidirourgos. A database system with amnesia. In *CIDR*, 2017.

Ross Kindermann and Laurie Snell. *Markov random fields and their applications*, volume 1. American Mathematical Society, 1980. ISBN 978-0-8218-5001-5. doi: http://dx.doi.org/10.1090/conm/001.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtarik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016. URL `https://arxiv.org/abs/1610.05492`.

Alexander V Kozlov and Jaswinder Pal Singh. A parallel lauritzen-spiegelhalter algorithm for probabilistic inference. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pages 320–329. IEEE, 1994.

Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. 2019.

Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1969–1984, 2015.

Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, pages 615–629, 2016.

Qinbin Li, Yiqun Diao, Quan Chen, and Bingsheng He. Federated learning on non-iid data silos: An experimental study. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 965–978. IEEE, 2022.

Tian Li and et al. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020.

Wang Ling, Chris Dyer, Alan W Black, and Isabel Trancoso. Two/too simple adaptations of word2vec for syntax problems. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1299–1304, 2015.

Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 631–645, 2018.

Qingzhi Ma and Peter Triantafillou. Dbest: Revisiting approximate query processing engines with machine learning models. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1553–1570, 2019.

Qingzhi Ma, Ali Mohammadi Shanghooshabad, Mehrdad Almasi, Meghdad Kurmanji, and Peter Triantafillou. Learned approximate query processing: Make it light, accurate and fast. In *CIDR*, 2021.

J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

Xudong Mao, Qing Li, Haoran Xie, Raymond YK Lau, and Zhen Wang. Multi-class generative adversarial networks with the l2 loss function. *arXiv preprint arXiv:1611.04076*, 5, 2016.

Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

Daniel Michelsanti and Zheng-Hua Tan. Conditional generative adversarial networks for speech enhancement and noise-robust speaker verification. *arXiv preprint arXiv:1709.01703*, 2017.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

Tova Milo. Getting rid of data. In *JDIQ*, 2019.

Priti Mishra and Margaret H Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.

Kevin Murphy, Yair Weiss, and Michael I Jordan. Loopy belief propagation for approximate inference: An empirical study. *arXiv preprint arXiv:1301.6725*, 2013.

Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *VLDB*, volume 6, pages 1049–1058, 2006.

Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, page 37–48, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312486. doi: 10.1145/2213556.2213565. URL `https://doi.org/10.1145/2213556.2213565`.

Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

Frank Olken. *Random sampling from databases*. PhD thesis, University of California, Berkeley, 1993.

Dan Olteanu and Maximilian Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, 2016.

Dan Olteanu and Jakub Závodnỳ. Factorised representations of query results: size bounds and readability. In *Proceedings of the 15th International Conference on Database Theory*, pages 285–298, 2012.

Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):1–44, 2015.

Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, pages 1–4, 2018.

Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 587–602, 2017.

Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1461–1476, 2018a.

Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. Verdictdb: universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1461–1476. ACM, 2018b.

Judea Pearl. Bayesian netwcrks: A model cf self-activated memory for evidential reasoning. In *Proceedings of the 7th conference of the Cognitive Science Society, University of California, Irvine, CA, USA*, pages 15–17, 1985.

Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 689–690. IEEE, 2011.

Sergei Popov, Stanislav Morozov, and Artem Babenko. Neural oblivious decision ensembles for deep learning on tabular data. *arXiv preprint arXiv:1909.06312*, 2019.

Christian P Robert and George Casella. The metropolis—hastings algorithm. In *Monte Carlo Statistical Methods*, pages 231–283. Springer, 1999.

Ali Mohammadi Shanghooshabad, Meghdad Kurmanji, Qingzhi Ma, Michael Shekelyan, Mehrdad Almasi, and Peter Triantafillou. Pgmjoins: Random join sampling with graphical models. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1610–1622, 2021.

Michael Shekelyan, Graham Cormode, Peter Triantafillou, Ali Shanghooshabad, and Qingzhi Ma. Weighted random sampling over joins. *arXiv preprint arXiv:2201.02670*, 2022.

Sameer Singh and Thore Graepel. Compiling relational database schemata into probabilistic graphical models. *arXiv preprint arXiv:1212.0967*, 2012.

Yan Song, Shuming Shi, Jing Li, and Haisong Zhang. Directional skip-gram: Explicitly distinguishing left and right context for word embeddings. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 175–180, 2018.

Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. Approximate query processing using deep generative models. *arXiv preprint arXiv:1903.10000*, 2019.

Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. Approximate query processing for data exploration using deep generative models. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1309–1320. IEEE, 2020.

Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. Lightweight graphical models for selectivity estimation without independence assumptions. *Proceedings of the VLDB Endowment*, 4(11):852–863, 2011.

Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. Efficiently adapting graphical models for selectivity estimation. *The VLDB Journal*, 22(1):3–27, 2013.

Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.

Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 363–378, 2016.

Martin J Wainwright and Michael Irwin Jordan. *Graphical models, exponential families, and variational inference*. Now Publishers Inc, 2008.

Daisy Zhe Wang, Eirinaios Michelakis, Minos Garofalakis, and Joseph M Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. *Proceedings of the VLDB Endowment*, 1(1):340–351, 2008.

Yinglong Xia, Xiaojun Feng, and Viktor K Prasanna. Parallel evidence propagation on multicore processors. In *International Conference on Parallel Computing Technologies*, pages 377–391. Springer, 2009.

Cong Xie, Sanmi Koyejo, and Indranil Gupta. Asynchronous federated optimization. *arXiv preprint arXiv:1903.03934*, 2019.

Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on neural networks*, 16(3):645–678, 2005.

Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(2):1–19, 2019.

Zongheng Yang and et al. Deep unsupervised cardinality estimation. In *Proceedings of the VLDB Endowment*, 2019.

Zongheng Yang and et al. Neurocard: One cardinality estimator for all tables. In *Proceedings of the VLDB Endowment*, 2020.

Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: one cardinality estimator for all tables. *arXiv preprint arXiv:2006.08109*, 2020.

Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

Felix Yu, Ankit Singh Rawat, Aditya Menon, and Sanjiv Kumar. Federated learning with only positive labels. In *International Conference on Machine Learning*, pages 10946–10956. PMLR, 2020.

Xiaofei Zhang, Lei Chen, and Min Wang. Efficient multi-way theta-join processing using mapreduce. *arXiv preprint arXiv:1208.0081*, 2012.

Yue Zhao, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. Federated learning with non-iid data. *arXiv preprint arXiv:1806.00582*, 2018a.

Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1525–1539, 2018b.