Compiling Discrete Probabilistic Programs for Vectorized Exact Inference

Jingwen Pan

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2023

Abstract

Although probabilistic programming has existed for decades, the implementation of probabilistic programming languages (PPLs) and related inference algorithms are limited in scope, and their usage has yet to be widely explored. The current stage of probabilistic programming can only achieve at most two of the three characteristics: (1) User-friendly, (2) fast, and (3) general. A competitive state-of-the-art PPL, Dice [14], specializes in fast exact discrete probabilistic programs but obtains a suboptimal inference performance over Bayesian Networks. In this project, we propose a framework, BayesTensor, that supports fast exact inference in discrete probabilistic programs, particularly optimizing inference over Bayesian Networks. Furthermore, we demonstrate three critical applications supported by BayesTensor, including addressing database tasks such as CardEst and AQP. We conduct experiments for each application supported by BayesTensor. Although BayesTensor does not outperform its competitor DeepDB [13] in the AQP application, BayesTensor shows its potential to be fast in AQP. For other applications, we observe an outstanding performance of BayesTensor compared to corresponding competitors.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jingwen Pan*)

Acknowledgements

I sincerely appreciate my supervisor Professor Amir Shaikhha for his outstanding mentorship and great encouragement. In addition, I would like to thank my friends and parents, who supported me in maintaining both physical and mental health throughout this project.

Table of Contents

Chapter 1

Introduction

This chapter aims to provide readers with an overview of this project and explain the thesis of this project in high-level aspects.

1.1 Motivation

Graphical models benefit scientists with statistical modeling, which help model real-world problems. Bayesian Networks (BNs) are one of the popular graphical models widely used. In the recent decade, probabilistic programming has renewed scientists' interests. The applications of probabilistic programming have been widely explored in various fields, such as computer vision [17, 19], query optimization in database systems [33, 31], medical diagnosis [8], particle physics [3], and astrophysics [18]. By probabilistic programming, graphical models can be transformed into programs. However, inference over graphical models is challenging. As a result, Probabilistic programming languages (PPLs) are introduced, which are natural tools for probabilistic programming. PPLs are programming languages that automate inference over graphical models.

Prior work [5, 4, 21, 12] primarily focuses on PPLs in continuous distribution. PPLs in discrete distribution are waiting to be explored further. Recently, there has been a state-of-the-art PPL [14] specialized in discrete distribution but leading to sub-optimal performance in inference over BNs. This project aims to support fast, exact inference over discrete distribution, including discrete probabilistic programs and BNs. In addition, this project supports database tasks such as CardEst and AQP.

In MInf part 1, we benchmark the performances of four probabilistic programming frameworks (i.e., Dice [14], Pgmpy [2], BayesCard [33], and SPPL [27]) in a database application called Cardinality Estimation (CardEst) in terms of latency and accuracy. In part 1's experiments, We train BNs with datasets and formulate each query of CardEst as an inference over related BNs.

This project extends the MInf part 1 and is inspired by prior work [33, 14]. [33] shows the efficiency, potential powerfulness, and conciseness of probabilistic programming in database fields. In MInf part 1, it is found out that although BayesCard [33] is

the fastest among four probabilistic programming frameworks in CardEst, BayesCard is unable to support exact inference over discrete probabilistic programs and general BNs. BayesCard is limited to CardEst and inference over Chow-Liu tree-structured BNs. Meanwhile, Dice [14] is a relative competitor we have benchmarked within the MInf part 1. Dice specializes in fast, exact inference over discrete distribution by exploiting logical expressions and local structure. Nevertheless, Dice obtains subprime performance of inference over BNs [14].

Compared to MInf part 1, which is limited to the Cardinality Estimation, this project aims to support fast, exact inference over discrete distribution with applications in discrete probabilistic programs, Cardinality Estimation, and Approximate Query Processing (AQP).

1.2 MInf Part 1

BayesCard [33] is a competitive state-of-the-art CardEst framework that utilizes probabilistic programming in CardEst. BayesCard reveals the efficiency and succinctness of PPLs in handling database applications. MInf part 1 is motivated to find efficient and appropriate PPLs for DB researchers. In part 1's experiments, we evaluate the performance of four PPLs in the database component, CardEst. The four PPLs used to benchmark are Dice [14], Pgmpy [2], SPPL [27] and Infer.NET [21]. The evaluation metrics are the latency in milliseconds and the estimation accuracy in Q-error. As an implementation of MInf part 1, we adapt the CardEst algorithm from BayesCard with the four selected PPLs. In a nutshell, it is found that Dice outperforms the other three PPLs in CardEst.

1.3 Problem Statement

Although probabilistic programming has existed for decades, the implementation of probabilistic programming languages (PPLs) and related inference algorithms are still suboptimal, which can be further investigated. The current stage of PPLs can only achieve at most two out of three characteristics: (1) User-friendly, (2) fast, and (3) general. Beyond that, prior work primarily researched the continuous distribution [5, 4, 21, 12] while discrete distribution started to receive more attention in recent years. Discrete distribution estimates the uncertainty of a particular outcome. The importance of discrete distribution is noticeable as it commonly exists in the real world (e.g., Bernoulli distribution, Poisson distribution, and binomial distribution). There are many real-world applications of discrete distribution, such as disaster and market recessions forecasting and financial options pricing. This project aims to support the fast, exact inference of discrete probabilistic programming.

1.4 Solution

In this project, we propose a framework, BayesTensor, to achieve fast, exact inference over discrete probabilistic programs, including BNs. Our framework largely benefited

from tensor libraries to handle high-dimensional data distribution and inference over general BNs. There are three applications of our framework: (1) discrete probabilistic programs, (2) CardEst, and (3) AQP.

Cardinality Estimation (CardEst) is a particular case of Approximate Query Processing (AQP). Although we have explored CardEst using PPLs in MInf part 1, AQP is more complex than CardEst by including different aggregation functions (e.g., AVG(), SUM(), and Group By) in database languages. In this project, we will explain how to support fast, exact inference in discrete probabilistic programs, CardEst and AQP. The main architecture of our framework is demonstrated in Chapter 3, and corresponding applications are explained in Chapter 4. The evaluation process of each application supported by our framework is presented in Chapter 5.

1.5 Contribution

Reminder: The work presented in Chapters 3-4.2, and 5.2-5.3 is accepted by the Compiler Construction (CC) Conference (2023) [24].

Our main contributions are the following:

1. We propose a framework, BayesTensor, to tackle the fast, exact inference over discrete probabilistic programs and general BNs by taking advantage of tensor libraries.
2. We present three applications of our framework: (1) Discrete probabilistic programs, (2) CardEst, and (3) AQP.
3. We show the evaluation processes of each application with corresponding competitor(s) in topics on single table datasets.
4. We discuss potential extensions that can be made to our framework.

1.6 Dissertation structure

This dissertation comprises six main chapters, expounding important information about the implementation of this project.

Chapter 2: This chapter includes the fundamental background knowledge required to understand this project and the literature review of two critical research topics concerned by the database (DB) community: (1) CardEst, and (2) AQP.

Chapter 3: This chapter elucidates the architecture of our framework, BayesTensor, and explains how it targets fast, exact inference over discrete probabilistic programs and general BNs. This work [24] is accepted by the Compiler Construction (CC) Conference (2023).

Chapter 4: This chapter presents three applications supported by our framework: Discrete probabilistic programs and two database tasks, i.e., CardEst and AQP.

Chapter 5: This chapter demonstrates the setup and evaluation details of each experiment in this project.

Chapter 6: This chapter summarizes our contributions and the practical work in this project and reveals possible future work to improve the current work.

Chapter 2

Background Chapter

This chapter explains the fundamental knowledge that helps to understand this project and includes the literature review of CardEst and AQP related to this project.

2.1 Probabilistic Programming Languages

Probabilistic programming is a powerful tool enabling scientists to model real-world problems with graphical models incorporating uncertainty. However, performing inference over graphical models is complex, which has led to the development of probabilistic programming languages.

Probabilistic programming languages (PPLs) are domain-specific languages that automate inference over graphical models. Table 2.1 shows some popular PPLs and their specialized domains. Like machine learning with general-purpose programming languages like Python, PPLs have a similar pattern, presented in Figure 2.1.

The training of graphical models in PPLs involves two main components: (1) Structure learning and (2) parameter learning. Structure learning is a process that learns the dependence relationships among variables and constructs the corresponding graphical model. Parameter learning involves learning the distribution for each variable within the graphical model. PPLs encode the graphical model specified by the user and train the graphical model based on training datasets.

During the evaluation, PPLs parse input queries and perform inference over the trained graphical model using selected inference algorithms, where the predictions generated by PPLs are computed distributions.

2.2 Bayesian Networks

Graphical models are probabilistic models that use graphs to represent the dependencies among random variables. The two main types of graphical models are Markov Hidden Fields and Bayesian Networks (BNs). This project focuses exclusively on BNs.

PPL name	Host language	Specializations
PyMC [26]	Python	Bayesian inference, mixed-type modelling
Infer.NET [21]	.NET Framework	mixed-type modelling
Stan [5]	C++	Mixed-type modelling, various inference algorithms
Pgmpy [2]	Python	Bayesian Networks, exact inference
Pyro [4]	Python	Bayesian inference, mixed-type modelling

Table 2.1: Examples of PPLs.

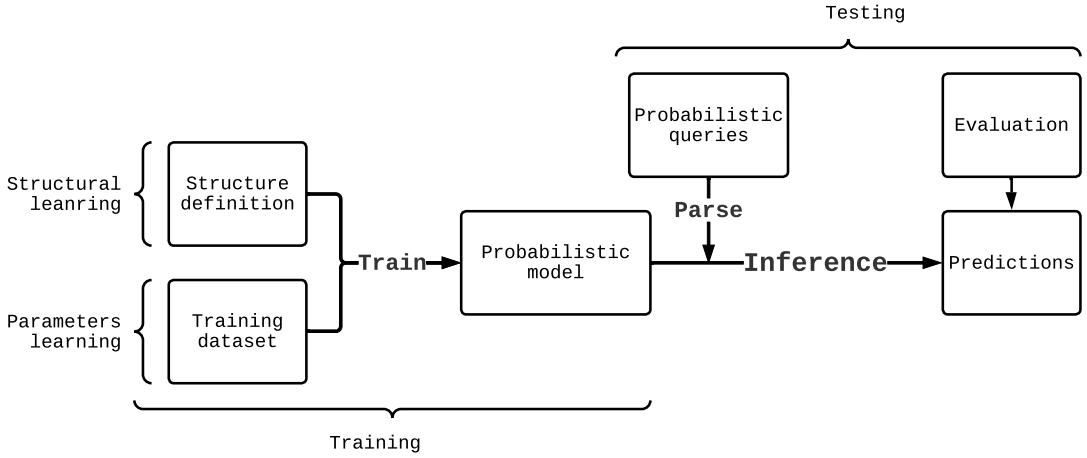


Figure 2.1: The overall workflow of PPLs

BNs are directed acyclic graphs (DAGs) that are widely used in data science. In a BN, each node represents a variable, and each directed edge indicates a dependence between two variables. A node that has an incoming edge is a parent node, while a node that has an outgoing edge is a child node. The joint distribution of a BN can be computed using the equation 2.1 by chain rule:

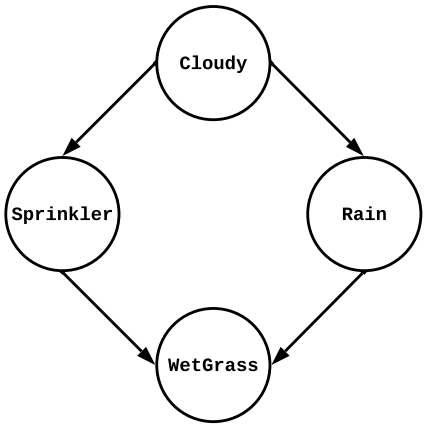
$$\mathbb{P}(X_1, X_2, ..., X_n) = \prod_{i=1}^{n} \mathbb{P}(X_i \mid Par(X_i)) \tag{2.1}$$

where $Par(X_i)$ is the parent node(s) of X_i.

To illustrate the concept of BNs, we will use a well-known example from Kevin Murphy, shown in Figure 2.2. The BN structure can be visualized in Figure 2.2a. This example contains four nodes which are *Cloudy*, *S prinkler*, *Rain*, and *WetGrass*. Since there is no arrow pointing towards *Cloudy*, *Cloudy* is the root node that does not have any parent. *Cloudy* points to two nodes *S prinkler* and *Rain*, which are its child nodes. Table 2.2b summarizes dependence among variables within this example corresponding to Figure 2.2a. The joint distribution of this example is interpreted using the equation 2.1:

$$\mathbb{P}(C, S, R, W) = \mathbb{P}(C) * \mathbb{P}(S \mid C) * \mathbb{P}(R \mid C) * \mathbb{P}(W \mid S, R) \tag{2.2}$$

where C, S, R, W refers to variables *Cloudy*, *S prinkler*, *Rain*, and *WetGrass* in Figure 2.2a respectively.



(a) Wetgrass/sprinkler/rain Bayesian Network.

Node	Parent node(s)	Child node(s)
Cloudy	-	Sprinkler, Rain
Sprinkler	Cloudy	WetGrass
Rain	Cloudy	WetGrass
WetGrass	Sprinkler, Rain	-

(b) Relationships of nodes in 2.2a.

Figure 2.2: Kevin Murphy's wetgrass/sprinkler/rain: An example of BN.

2.2.1 Chow-Liu Tree

Chow-Liu tree [7] is a dependence tree structure proposed by Chow and Liu in 1968. Chow-Liu tree can be seen as a special structure of BNs. Chow-Liu tree efficiently approximates discrete distribution by computing the joint distribution as a product of second-order conditional and marginal distributions. In our implementation related to database components such as CardEst and AQP, we construct BNs in the Chow-Liu structure to speed up the inference by sacrificing some accuracy. The main property of the Chow-Liu tree is that its structure is a first-order dependence tree. In other words, each node in the Chow-Liu tree has only one parent at most. Figure 2.3 below shows an example of the Chow-Liu tree.

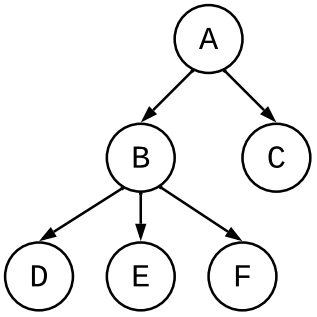


Figure 2.3: An example of Chow-Liu tree

Based on the Chow-Liu tree structure, the joint distribution of the BN shown in Fig-

ure 2.3 can be computed as Equation 2.3:

$$\mathbb{P}(A,B,C,D,E,F) = \mathbb{P}(A) * \mathbb{P}(B \mid A) * \mathbb{P}(C \mid A) * \mathbb{P}(D \mid B) * \mathbb{P}(E \mid B) * \mathbb{P}(F \mid B) \quad (2.3)$$

2.2.2 Variable Elimination

There is a wide range of inference algorithms in probabilistic graphical models. This project explores a general exact inference algorithm called Variable Elimination (VE) [16].

VE performs two operations on factors which are summation and multiplication. A factor of variables, also called the potential, declares the conditional distribution of the related variables. There are three kinds of probability we might be interested in, which are: (1) Marginal probability, (2) conditional probability, and (3) joint probability. By eliminating a variable in VE, we sum out that variable from factors containing that variable and multiply the sum-out result by the remaining factors. To marginalize a variable X, we aim to get $\mathbb{P}(X)$ by eliminating all other variables. To compute the conditional probability, such as $\mathbb{P}(X \mid Y)$ or $\mathbb{P}(X \mid Y = 1)$, instead of eliminating Y by summing all values of Y, we sum over the observation values of Y. The result of a joint probability is a single value that all variables should be eliminated during the VE. When having observation(s), normalization needs to be performed, and the constant used for normalization is also called the partition function Z.

Let us look back to Kevin Murphy's wetgrass/sprinkler/rain example in Figure 2.2a. For this example, according to its joint probability equation 2.2, factors are $\mathbb{P}(C)$, $\mathbb{P}(S \mid C)$, $\mathbb{P}(R \mid C)$, and $\mathbb{P}(W \mid S,R)$. For a query such as $\mathbb{P}(W = w)$ can be computed as below:

$$\begin{aligned} \mathbb{P}(W = w) &\propto \sum_C \sum_R \sum_S \mathbb{P}(C) * \mathbb{P}(S \mid C) * \mathbb{P}(R \mid C) * \mathbb{P}(W = w \mid S,R) \\ Z &= \sum_C \sum_R \sum_S \mathbb{P}(C,S,R,W = w) \\ \mathbb{P}(W = w) &= \frac{\sum_C \sum_R \sum_S \mathbb{P}(C) * \mathbb{P}(S \mid C) * \mathbb{P}(R \mid C) * \mathbb{P}(W = w \mid S,R)}{Z} \end{aligned} \quad (2.4)$$

2.3 Cardinality Estimation

Cardinality Estimation (CardEst) is a critical component of query optimizer in the database management system (DBMS) concerned by the DB community. CardEst approximates the total number of rows to return the final answer for a query to assist the query optimizer in generating the high-quality optimal query plan. In practice, CardEst queries are COUNT(*) queries in database languages (with filter predicates). A summary of the literature review of CardEst is presented in 2.2.

2.3.1 BayesCard

BayesCard [33] is a state-of-the-art framework that utilizes PPL in CardEst and achieves fast CardEst compared to prior work [13, 37, 34, 35, 15]. BayesCard learns a BN in

the Chow-Liu tree structure for every table included in the dataset. If the dataset is a single table, then the inference of CardEst queries of that dataset will be performed on the learned BN based on that dataset. A multi-table dataset comprises multiple tables with data dependent on each other based on the schema. The join condition(s) defined in the database schema declares the dependency among tables in a multi-table dataset. In the case of CardEst on a multi-table dataset, BayesCard constructs a BN ensemble such that each BN in the ensemble is learned based on the join condition. Based on the BN ensemble, the inference of CardEst on the multi-table dataset can be decomposed to multiple single-table queries over each related BN in the ensemble. Up to this stage, the inference of CardEst needs to be normalized because BN in the ensemble can share overlaps of records in the database. As a result, BayesCard applies the fanout method from prior works [37, 34, 13] as a normalization to the inference results for CardEst on multi-table datasets.

2.4 Tensor Libraries

In the recent decade, scientists have widely explored tensor libraries to handle high-dimensional data. For instance, Pytorch, Numpy, Numba, Tensorflow, and Pytensor are well-known tensor libraries. Tensors are high-dimensional arrays with specific types. Tensor libraries support data structures such as arrays and matrices, and provide extensive collections of high-dimensional mathematical operations over arrays and matrices. In this project, we represent data distributions with Numpy arrays and empower Opt_einsum [1] to fasten the inference where Opt_einsum is a tensor library that specializes in fast tensor contraction.

2.5 Approximate Query Processing

Framework	Year	CardEst	AQP	Approach
IDEA [11]	2017	✗	✓	IDEA, Online aggregation, Query rewrite, Tail Indexes
VerdictDB [25]	2018	✗	✓	Online analytical processing engine, Variational subsampling
MSCN [15]	2019	✓	✗	Multi-set convolutional network
Naru [35]		✓	✗	Deep autoregressive models, Progressive sampling
DeepDB [13]		✓	✓	Relational Sum Product Netorks (RSPNs)
FLAT [37]	2020	✓	✗	Factorize-sum-split-product network (FSPN)
BayesCard [33]		✓	✗	Bayesian Networks, JIT compilation
EntropyDB [23]		✗	✓	Multi-linear polynomial Maximum Entropy (MaxEnt) Model
VAE-AQP [31]		✗	✓	Variational Auto-encoders, Variational Inference
ML-AQP [28]		✗	✓	Gradient Boosting Machines (GBM), XGBoost, LightGBM
PGMJoins [30]	2021	✗	✓	Probabilistic graphical models (PGMs), Sum-Product Message Passing Algorithm (SP-MPA)
Factorjoin [32]	2023	✓	✗	Bayesian Networks, Join-histograms

Table 2.2: Literature review of CardEst and AQP

As computing exact answers for queries with large databases is costly, Approximate Query Processing (AQP) is introduced. AQP achieves fast approximate query answers

by sacrificing reasonable accuracy, whereas CardEst is a particular case of AQP. Compared to CardEst, AQP enables answering queries with aggregations such as COUNT(), SUM(), AVG(), and GROUP BY.

2.5.1 DBEst

DBEst [20] is a model-driven AQP engine composed of supervised-machine-learning (SML) models. DBEst supports aggregations including COUNT(), SUM(), AVG(), GROUP BY, VARIANCE, STDDEV, and PERCENTILE. Although DBEst can answer a broad range of aggregation functions, due to DBEst being model-driven (i.e., queries are answered by the model(s) of the data. However, not the data itself), the accuracy of unseen queries can be unreliable due to biased sampling to ensure that categorical columns can meet certain conditions.

2.5.2 DeepDB

DeepDB [13] a competitive framework that supports both CardEst and AQP. DeepDB is a data-driven AQP framework that benefits from Relative Sum-Product Networks (RSPNs). RSPNs is a probabilistic model extended from Sum-Product Networks (SPNs) to optimize the usage in the relational database. RSPNs enable capturing data distribution from the dataset and NULL value handling in the database. The model training of DeepDB is similar to BayesCard, where each table from the dataset is learned as an RSPN. DeepDB also employs the fanout method as BayesCard to handle multi-table datasets and joins in database languages. The main difference between BayesCard and DeepDB is that DeepDB provides AQP, whereas BayesCard is limited to CardEst. Our framework, BayesTensor, is inspired by BayesCard and DeepDB to achieve CardEst and AQP by PPLs.

Chapter 3

Compiling Discrete Probabilistic Programs

This chapter will introduce our framework, BayesTensor, and how it can support fast exact inference over discrete distributions and Bayesian Networks.

3.1 Overview

Our framework, BayesTensor, contains a front-end and a back-end. The front-end compiles discrete probabilistic programs in BayesTensor Language (BTL) and simplifies the program where possible. The back-end transforms the program from BTL to Tensor Intermediate Language (TIL) to generate Python code that empowers tensor libraries and performs the inference with the generated code. The overall workflow of BayesTensor is shown in Figure 3.1.

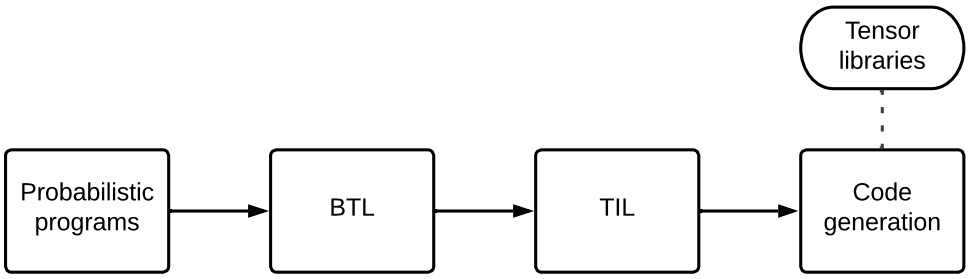


Figure 3.1: The workflow of our framework BayesTensor.

BayesTensor has four main components: A front-end, Normalization, Tensorization, and a back-end. The main architecture of our framework is shown in Figure 3.2. The core grammar of BTL is shown in Figure 3.3.

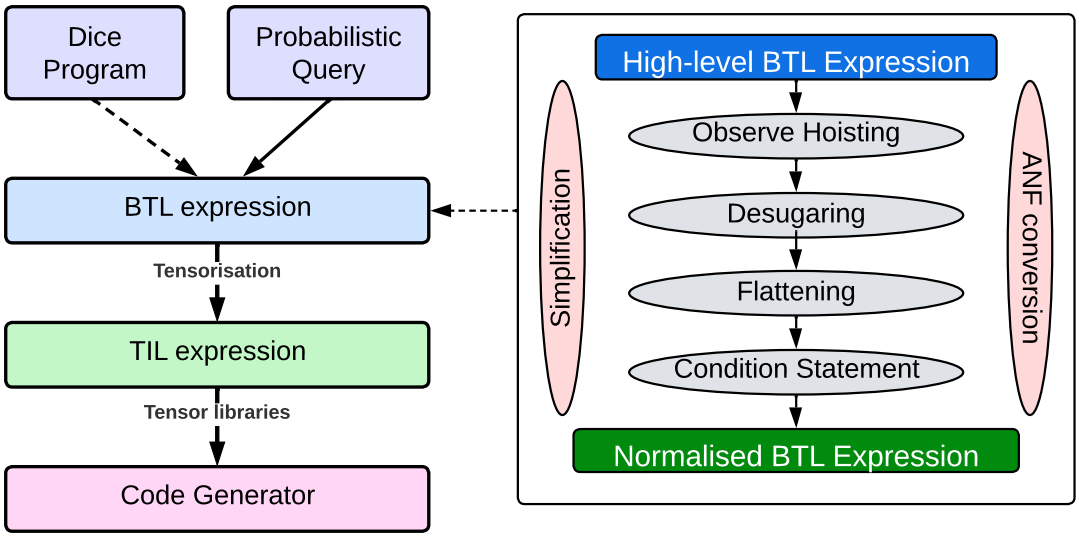


Figure 3.2: The overall architecture of our framework BayesTensor with detailed normalisation steps

```
e   ::=  x | v | fst e | snd e | (e, e) | f(e)
     |   let x=e in e | match e with { v -> e | ... }
     |   discrete [p, ..., p] | observe e [v, ..., v]
     |   query [x, ..., x] [x, ..., x]
P   ::=  e | fun f(x: T): T { e } P
T   ::=  int | (T, T)
v   ::=  n | (v, v)
```

Figure 3.3: Grammar of the core BTL language. The meta-variables p, n, x, and f range over real numbers in the range $[0, 1]$, natural numbers, variable names, and function names.

3.2 Frontend Language

The front-end of BayesTensor accepts inputs in BayesTensor Language (BTL), a tensor-centric PPL. This section demonstrates the main functionalities of the front-end language BTL with its core grammar and extensions in Section 3.2.1 and 3.2.2, respectively.

3.2.1 Core BTL

The core grammar of BTL is shown in Figure 3.3 and explained in the following sections. `observe` is the only effectful construct in BTL, which considers observations in `query` construct.

3.2.1.1 Generic constructs

BTL provides features of variable access, constant value representation, tuple projection, tuple creation, and functional call by the term `e` which can be further expanded by terms $\mathcal{P}$, $\mathcal{T}$ and v. The let-bindings syntax `let` x`=e1` `in` `e2` avoids redundant computations of the same expression for later reuse.

3.2.1.2 Control flow

BTL enables pattern matching by control flow constructs through the syntax **match** `e0` **with**`{ v1 -> e1 | ... }`, where `e0` represents scrutiny and the symbol `|` separates each unique case of the control flow.

3.2.1.3 Distribution

BTL defines a distribution by the syntax **discrete** `[`p_1`, ..., `p_n`]`, where p_1, ..., p_n are numbers in range of 0 to 1 such that $\sum_{i=1}^{n} p_i = 1$. This syntax represents a prior distribution of a discrete random variable, a marginal distribution, or a posterior distribution.

3.2.1.4 Conditional Distribution Tables (CDTs)

BTL provides the representation of conditional distributions by defining a prior distribution with **discrete** `[`p_1`, ..., `p_n`]` and specifies the dependency among variables by pattern matching. An example of the usage is shown in Figure 3.7.

3.2.1.5 Conditioning

To make observations with certain variables, BTL provides the conditioning operation **observe** and supports range conditioning referring to observing a variable `e` with multiple unique states v_1, ..., v_k by the syntax **observe** `e [`v_1`, ..., `v_k`]`.

3.2.1.6 Querying

BTL enables unconditional and conditional probabilistic queries. To query for a marginal probability such as $\mathbb{P}(X)$, specify the variable to be marginalized by its name at the end of the program. If multiple variables are specified for the query, a joint probability of the specified variables will be computed. To query conditional probability, BTL provides the syntax **query** `[`x_1`, ..., `x_k`] [`y_1`, ..., `y_d`]`, where `[`x_1`, ..., `x_k`]` declares variables to be queried and `[`y_1`, ..., `y_d`]` states the list of observations.

3.2.2 High-level Extensions

In order to recognize a wider range of discrete probabilistic programs, such as discrete probabilistic programs in Dice [14], we extend BTL to make representations in probabilistic programs more flexible and concise. The extended grammar of BTL is shown in 3.4, and the full desugar patterns are presented in Figure 3.5.

```
e  ::=  cf. Figure 3.3 | if(e) then e else e
     |  e && e | e || e | e ~> e | ~e | e == v
     |  flip p | observe e | observe e n:n
v  ::=  cf. Figure 3.3 | true | false
```

Figure 3.4: Extended high-level constructs for BTL. These constructs are syntactic sugar extensions that make BTL source-compatible with Dice [14] programs.

⟦**true**⟧	=	`1`
⟦**false**⟧	=	`0`
⟦**flip** p⟧	=	**discrete** `[(1-p), p]`
⟦**if** e1 == 0 **then** e2_0 ... **else if** e1 == n **then** e2_n **else** e3⟧	=	**match** ⟦e1⟧ **with** { 0 => ⟦e2_0⟧ ... | n => ⟦e2_n⟧ | n+1 => ⟦e3⟧ }
⟦**if** e1 **then** e2 **else** e3⟧	=	**match** ⟦e1⟧ **with** { 1 => ⟦e2⟧ | 0 => ⟦e3⟧ }
⟦e1 && e2⟧	=	⟦**if** e1 **then** e2 **else false**⟧
⟦e1 || e2⟧	=	⟦**if** e1 **then true else** e2⟧
⟦e1 ~> e2⟧	=	⟦**if** e1 **then** e2 **else true**⟧
⟦~e1⟧	=	⟦**if** e1 **then false else true**⟧
⟦**observe** e1⟧	=	**observe** ⟦e1⟧ `[1]`
⟦**observe** e1 v1:v2⟧	=	**observe** e1 `[v1, ..., v2]`

Figure 3.5: Syntactic sugars for the BTL.

3.2.2.1 Logical expression

To encode logical expressions, BTL is extended with operations `&&`, `||`, and `~` to represent logic operations `AND`, `OR`, and `NOT` respectively, presented in Figure 3.4. BTL also provides logical implication by the syntax `e ~> e`, equivalent to the syntax `~e || e`. To express an equivalence comparison, BTL offers the syntax `e` $== v$ where v is extended express boolean values **`true`** and **`false`**, presented in Figure 3.4.

3.2.2.2 Control flow

To simplify the decision representations in BTL, we replace **`match`**-statements with **`if`**-statements in high-level BTL.

3.2.2.3 Boolean distribution

Boolean distribution can be expressed using the syntax **`flip`** θ in the extended BTL, where θ represents the probability of a condition is true while `1-`θ refers to the probability that the condition is unsatisfied. The syntax **`flip`** θ can be further desugared to **`discrete`** `[`θ`, (1-`θ`)]` using the core grammar of BTL.

3.2.2.4 Boolean conditioning

Inspired by Dice [14], the extended BTL simplifies the expression of observing a variable `e` with boolean distribution to **`observe`** `e`, presented in Figure 3.4.

3.2.2.5 Range conditioning

In extended BTL, we shorten the range conditioning syntax `observe e [n, n+1, ..., m]` in the core grammar of BTL to `observe e n:m`, where `n<m`, shown in Figure 3.4.

3.2.3 Examples

This section illustrates two discrete probabilistic programs that can be expressed in BTL.

3.2.3.1 Simple probabilistic programs

Figure 3.6 defines a simple probabilistic program called Observe1. This example is taken from Dice [14]. In this example, we are curious about the marginal probability of the evidence, with the condition that when the evidence is true, considering that we will get a head while flipping an unbiased coin.

```
let evidence = flip 0.5 in
let coin = flip 0.5 in
if evidence then
  let obs = observe coin in
  evidence
else
  evidence
```

Figure 3.6: Observe1: A discrete probabilistic program in BTL.

3.2.3.2 Bayesian Networks

BNs can be seen as a special case of discrete probabilistic programs. We show how to express BNs using BTL with Kevin Murphy's wetgrass/sprinkler/rain example introduced in Section 2.2 in Figure 3.7. Figure 3.7 demonstrates the BTL construct of this example. The distribution of this BN is presented in Figure 3.7a, and the corresponding BN definition is shown in Figure 3.7b.

3.3 Normalization

This section explains the transformations used to normalize BTL expressions. The normalization process transformed BTL programs to be prepared for the tensorization (i.e., lowering to tensor representations) and the code generation processes. The right-hand side of Figure 3.2 illustrates the main normalization steps to convert high-level BTL expressions, which may contain side-effect observe statements, into normalized purely functional code.

In the following sections, we will show how BayesTensor performs normalization steps with the example Observe1 presented in Figure 3.6.

3.3.1 Observe Hoisting

Observe hoisting addresses semantics ambiguities during the inference such that outcomes of the same probabilistic query are always consistent. An example of semantics ambiguities can be referred to the program Observe1 originated from Figure 3.6.

In the example Observe1, there is a semantics ambiguity caused by the presence of a non-deterministic choice defined by the syntax **`flip`** θ and an observe statement inside an **`if`**-statement. The main ambiguity arises from the non-deterministic choice **`flip`** `0.5` statement, which can be interpreted as either **`true`** or **`false`** with equal probability. Defining non-deterministic choice statements can result in multiple execution paths and outcomes during the inference.

Moreover, the conditional **`let`** `obs =` **`observe`** `coin` **`in`** is nested in the **`if`**-statement **`if`** `evidence` **`then`** such that the conditional will be considered only when the evidence is true. However, if the evidence is false, the conditional of coin flip does not contribute to the program's outcome since the conditional of the coin flip will never be reached and executed in this case.

Observe hoisting contains three main steps: (1) Analyzing and collecting the context of each **`observe`** statement, (2) analyzing and collecting all statements that each **`observe`** statement is dependent on, and (3) pushing all statements collected in steps (1) and (2) outside the conditionals, as well as the **`observe`** statement itself. The result of performing observe hoisting to the example Observe1 is shown in Figure 3.8.

3.3.2 Desugaring

Desugaring decomposes high-level BTL constructs in terms of BTL core grammar. The desugaring transformation rules are depicted in Figure 3.9. Figure 3.9 shows the desugaring result of Observe1 originated from Figure 3.6.

3.3.2.1 Simplification

Various simplifications can be performed during the desugaring, such as partial evaluation, constant propagation, β-reduction, and dead-code elimination. Possible simplification details can be found in Sections 4.3-4.5 in [24]. An example of dead-code elimination is shown in Figure 3.9: In Figure 3.9, lines 4-7 at the left-hand side repeat querying over the variable `evidence`, which can be simplified to line 10 at the right-hand side of the same Figure.

3.3.3 Flattening

This transformation aims to flatten nested pattern matching. Considering expressions in Figure 3.10a, flattening transformation is composed of two steps: (1) Compensating missing pattern scrutinies if it happens, and (2) performing the permutation and combination of possible outcomes that can be generated within each nested branch and pushing those combinations to form integrated homogeneous pattern matching expression. Step (1) inspects all patterns if any pattern scrutiny is missing and records

the range and symbol of each pattern scrutiny in case value expansion is needed, which happens in Figure 3.10a. The result of step (1) Figure 3.10a is shown in Figure 3.10b where the missing pattern scrutiny when x is `true` is expanded. The result of performing step (2) is shown in Figure 3.10c.

Figure 3.11 shows the result of flattening transformation on Observe1 based on Figure 3.9, where lines 1-2 and 8-9 at the left-hand side remain unchanged before and after the flattening.

3.3.4 Query Inference

This transformation encodes conditionals within inference with the `query` statement in high-level BTL construct (Figure 3.5). An example of a normalized query is presented in Figure 3.12, where query inference transformation has been outlined with an arrow to interpret related expressions before and after the query inference transformation.

3.4 Tensorization

With tensor representations, BayesTensor can support fast inference with high-dimensional data distribution by tensor libraries. In this section, we introduce Tensor Intermediate Language (TIL), used to transform a probabilistic program in normalized BTL into tensor representations. The core grammar of TIL is shown in Figure 3.13. The usage standard of variables and let-bindings within TIL is the same as BTL.

3.4.1 Tensor Construction

A distribution expressed in BTL `discrete` $[r_1, \ldots, r_k]$ can be seemed as a tensor of k real numbers $r_1, \ldots, r_k$. TIL provides tensor representations by the syntax `tensor`$([r_1, \ldots, r_k], [n_1, \ldots, n_d])$, where $[n_1, \ldots, n_d]$ specify the shape and order of a tensor and each n_i is a natural number representing the number of elements over a specific dimension of a tensor and the number of n_is (specified by d) represent the tensor order. When defining conditional probabilities, each r_i in TIL tensor construct is a real number in the range of 0 to 1 and $\sum_{i=1}^{n} r_i = 1$.

3.4.2 Filtering

Incorporating observations of a variable in a query (i.e., a conditional query) can be treated as a slice of the tensor representing the distribution of that variable to consider only related observation values in the inference. TIL provides the syntax `select`$(\texttt{t}, [n_1, \ldots, n_k], d)$, where $[n_1, \ldots, n_k]$ specify the indices of the elements to be sliced at the specified dimension. This syntax can slice a tensor `t` based on dth dimension.

3.4.3 Normalization

The inference result must be normalized with the partition function Z to correspond to a probability distribution while incorporating observations within an inference. TIL provides normalization construct `normalize(t)` to ensure the summation over a normalized tensor should be one.

3.4.4 Contraction

TIL offers tensor contraction by the syntax `contract(`s`, `t_1`, ..., `t_k`)`, where s corresponds to an einsum expression, and t_1, ..., t_k refers to corresponding inputs obtained within the einsum expression separated by commas.

The einsum expression `"i,...,i->i"` indicates the indices of input tensors on the left-hand side of the arrow and the indices of the output tensor on the right-hand side. For instance, the dot product over matrices can be expressed as `"ik,kj->ij"`, which requires the column index of the first matrix and the row index of the second matrix to be the same (denoted by the index "k"). Similarly, the Hadamard product (element-wise multiplication) of two matrices can be represented as `"ij,ij->ij"`. The transpose of a matrix can be expressed as `"ij->ji"` in the einsum notation.

3.4.5 Examples

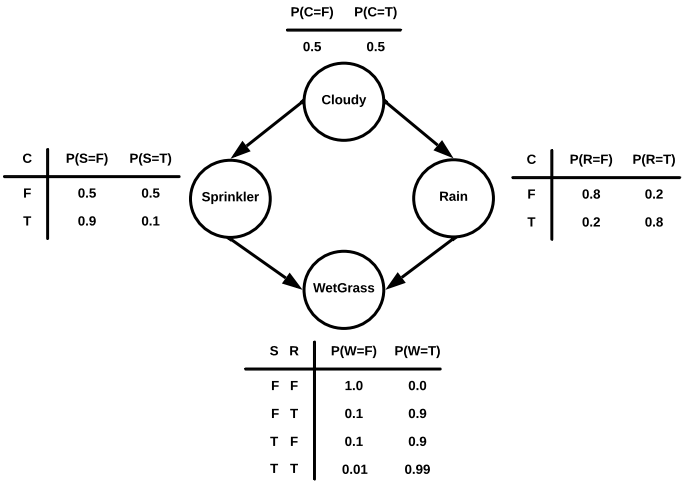
Examples of TIL programs are presented in Figure 3.14 by performing an unconditional query (Figure 3.14a) and a conditional query (Figure 3.14b) over the Kevin Murphy's wetgrass/sprinkler/rain example.

3.5 Code Generation

In practice, we empower code generation to generate tensor programs in Python. The main challenge is computing probability with high-dimensional distributions. Handle shape matching with Numpy arrays during variable elimination (VE) is hard. As a result, we utilize the einsum operation, which automatically handles the shape matching in computing summation and factorization of distributions during the inference.

3.5.1 Examples

Figure 3.15 demonstrates the generated Python code of an unconditional query (Figure 3.15a) and a conditional query (Figure 3.15b) over the Kevin Murphy's wetgrass/sprinkler/rain example in Figure 3.14. Line 6-7 in Figure 3.15b conditions on that grass is wet by extracting the conditional probability of *WetGrass* representing it is wet with the Numpy function `np.take`.



(a) Kevin Murphy's wetgrass/sprinkler/rain example with distribution.

```
let Cloudy = discrete [0.5, 0.5] in
let Rain = match Cloudy with {
0 -> discrete [0.8, 0.2]
| 1 -> discrete [0.2, 0.8]
} in
let Sprinkler = match Cloudy with {
0 -> discrete [0.5, 0.5]
| 1 -> discrete [0.9, 0.1]
} in
let WetGrass = match (Sprinkler, Rain) with {
(0,0) -> discrete [1, 0]
| (0,1) -> discrete [0.1, 0.9]
| (1,0) -> discrete [0.1, 0.9]
| (1,1) -> discrete [0.01, 0.99]
} in
WetGrass
```

(b) BTL program with unconditional query over the corresponding BN in 3.7a.

```
let Cloudy = flip 0.5 in
let Rain =
if(Cloudy) then flip 0.8 else flip 0.2 in
let Sprinkler =
if(Cloudy) then flip 0.1 else flip 0.5 in
let WetGrass =
if(Sprinkler) then
if(Rain) then flip 0.99 else flip 0.9
else
if(Rain) then flip 0.9 else flip 0.0 in
WetGrass
```

(c) Corresponding high-level constructs of BTL in 3.7b.

Figure 3.7: An example of the Bayesian Network and its definition in different level of BTL.

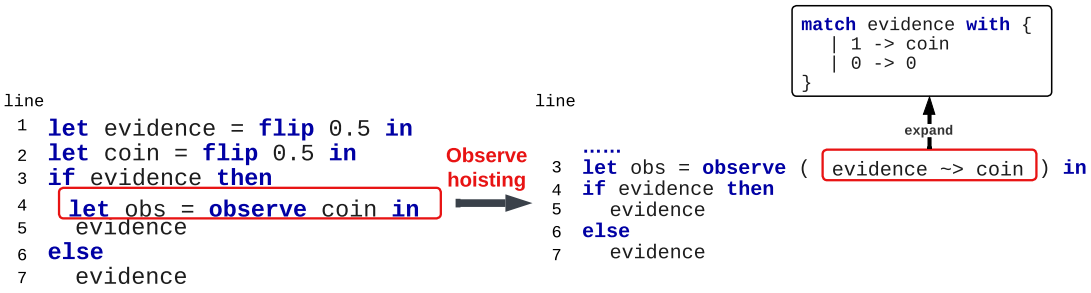


Figure 3.8: Performing observe hoisting on the example Observe1 originated from Figure 3.6.

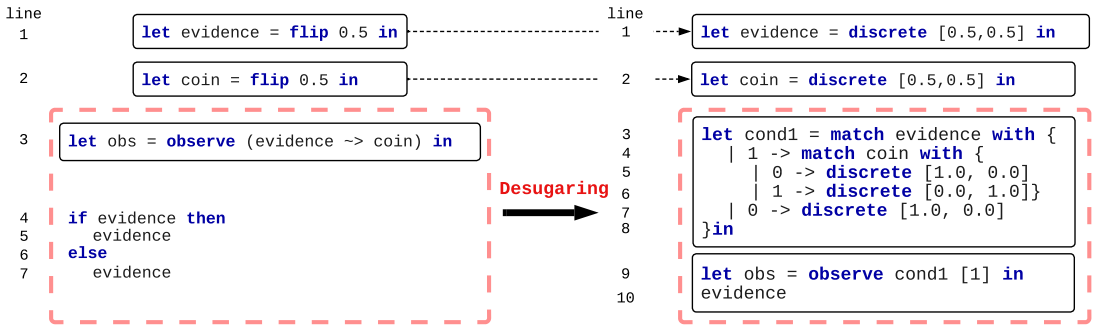


Figure 3.9: Desugaring the example Observe1 based on the result of observe hoisting transformation in Figure 3.8

```
match x with {
| 0 -> match y with {
  | 0 -> dist0
  | 1 -> dist1 }
| 1 -> dist2 }
}
```

(a) An example of nested pattern matching needs to be flatten.

```
match x with {
| 0 -> match y with {
  | 0 -> dist0
  | 1 -> dist1 }
| 1 -> match y with {
  | 0 -> dist2
  | 1 -> dist2 }
}
```

(b) Compensation of mssing pattern scrutities in Figure 3.10a.

```
match (x, y) with {

| (0, 0) -> dist0
| (0, 1) -> dist1

| (1, 0) -> dist2
| (1, 1) -> dist2
}
```

(c) Final flatten result of 3.10a based on 3.10b.

Figure 3.10: An example of BTL expressions before and after the flattening transformation.

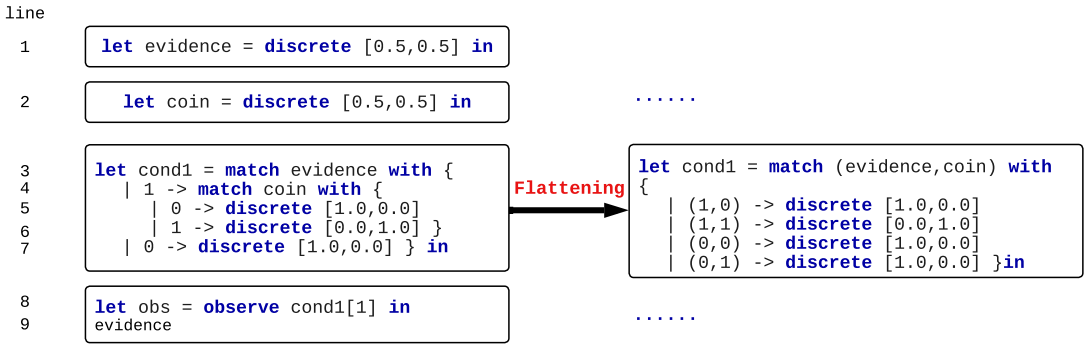


Figure 3.11: Flattening result of the example Observe1 based on its desugaring result in Figure 3.9.

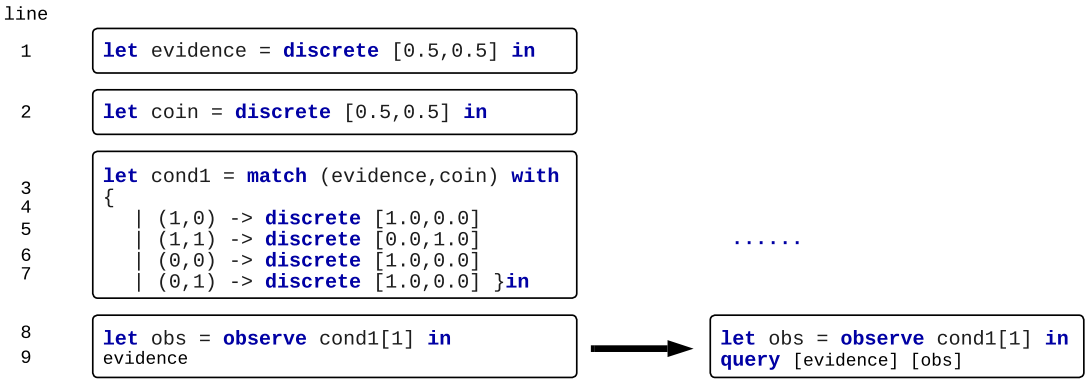


Figure 3.12: The normalization on the query expression of the example Observe1.

```
t  ::=  tensor([r, ..., r], [n, ..., n])
     |  x | let x=t in t | select(t, [n, ..., n], n)
     |  normalize(t) | contract(s, [t, ..., t])
s  ::=  "i,...,i->i"
```

Figure 3.13: Grammar of TIL. The meta-variable r ranges over real numbers, x over variable names, and n over natural numbers. The string literal `i` consists of letters `[a-zA-Z]` corresponding to each dimension of a tensor.

```
1 let Cloudy = tensor([2], [0.5,0.5]) in
2 let Rain = tensor([2,2], [0.8,0.2,0.2,0.8] in
3 let Sprinkler = tensor([2,2], [0.5,0.5,0.9,0.1]) in
4 let WetGrass = tensor([2,2,2],
5     [1,0,0.1,0.9,0.1,0.9,0.01,0.99]) in
6 let prob = contract('a,ab,ac,bcd->d',
7     Cloudy, Rain, Sprinkler, WetGrass) in
8 prob
```

(a) Translated BTL program from Figure 3.7b in TIL. As there is no conditioning, there is no need for normalizing the distribution.

```
1 let Cloudy = tensor([2], [0.5,0.5]) in
2 let Rain = tensor([2,2], [0.8,0.2,0.2,0.8] in
3 let Sprinkler = tensor([2,2], [0.5,0.5,0.9,0.1]) in
4 let WetGrass = select(tensor([2,2,2],
5     [1,0,0.1,0.9,0.1,0.9,0.01,0.99]), [0], 2) in
6 let prob = contract('a,ab,ac,bcd->d',
7     Cloudy, Rain, Sprinkler, WetGrass) in
8 normalize(prob)
```

(b) Query with conditioning in TIL over Kevin Murphy's wetgrass/sprinkler/rain example when grass is not wet

Figure 3.14: Translating the Kevin Murphy's wetgrass/sprinkler/rain example of Figure 3.7 from BTL to TIL.

```
import numpy as np
from opt_einsum import contract
Cloudy=np.array([0.5,0.5])
Rain=np.array([0.8,0.2,0.2,0.8]).reshape((2,2))
Sprinkler=np.array([0.5,0.5,0.9,0.1]).reshape((2,2))
WetGrass=np.array([1.0,0.0,0.1,0.9,
       0.1,0.9,0.01,0.99]).reshape((2,2,2))
prob=contract('a,ab,ac,bcd->d',
       Cloudy, Rain, Sprinkler, WetGrass)
print(prob)
```

(a) Code generation of an unconditional query based on Figure 3.14a.

```
import numpy as np
from opt_einsum import contract
Cloudy=np.array([0.5,0.5])
Rain=np.array([0.8,0.2,0.2,0.8]).reshape((2,2))
Sprinkler=np.array([0.5,0.5,0.9,0.1]).reshape((2,2))
WetGrass=np.take(np.array([1.0,0.0,0.1,0.9,
  0.1,0.9,0.01,0.99]).reshape((2,2,2)), [0], axis=2)
prob=contract('a,ab,ac,bcd->d',
       Cloudy, Rain, Sprinkler, WetGrass)
print(prob / np.sum(prob))
```

(b) Code generation of a conditional query based on Figure 3.14b.

Figure 3.15: The generated Python code for the Kevin Murphy's wetgrass/sprinkler/rain example.

Chapter 4

Applications

In this chapter, we will introduce a variety of applications supported by BayesTensor.

4.1 Simple probabilistic programs

BayesTensor supports the inference over discrete probabilistic programs written in BTL. Chapter 3 demonstrates how BayesTensor decodes and compiles the discrete probabilistic programs. An example of a simple probabilistic program is TwoCoins, shown in Figure 4.1. This example is a benchmark in our experiment of simple probabilistic programs in Section 5.2.1. TwoCoins queries the marginal probability of the first coin when flipping two coins in order and getting heads from both coins. Line 8 and 9 in Figure 4.1b refer to the conditional `bothHeads` and the observation `tmp` in Figure 4.1a at Lines 3 and 4, respectively. The evaluation results of simple probabilistic programs are presented in Chapter 5.2.

4.2 Cardinality Estimation

BayesTensor also supports database tasks such as CardEst and AQP. In this section, we will explain how to apply BayesTensor to do CardEst.

The model training process is similar to BayesCard. In our implementation, since Pgmpy does not support any structure learning algorithm for the Chow-Liu tree, we perform the structure learning from Pomegranate [29], a Python framework for PGMs construction. Then, perform a parameter training with Pgmpy.

Since CardEst queries are COUNT(*) queries in database languages, each CardEst query can be treated as an inference of joint probability with observations specified by the query predicates. In Chapter 3, we have seen how BayesTensor encodes inference of a joint distribution. In order to recognize the queries in database languages, we adapt the query parser from the BayesCard [33], which decomposes queries in database languages to help us generate the corresponding probabilistic programs.

```
let firstCoin = flip 0.5 in
let secondCoin = flip 0.5 in
let bothHeads = (firstCoin && secondCoin) in
let tmp = observe !bothHeads in
firstCoin
```

(a) TwoCoin in BTL.

```
import numpy as np
from time import perf_counter
import opt_einsum
t_start = perf_counter()
firstCoin = np.array([0.5, 0.5])
secondCoin = np.array([0.5, 0.5])
bothHeads = np.array([1.0, 0.0, 1.0, 0.0, 1.0,
    0.0, 0.0, 1.0]).reshape((2,2,2))
x13 = np.take(np.array([0.0, 1.0, 1.0,
    0.0]).reshape((2,2)), [1], axis=-1)
normalized14 = np.einsum('a,b,abc,cd->a',
    firstCoin, secondCoin, bothHeads, x13)
res15 = normalized14 / np.sum(normalized14)
res15
```

(b) Corresponding code generation of TwoCoins originated from 4.1a.

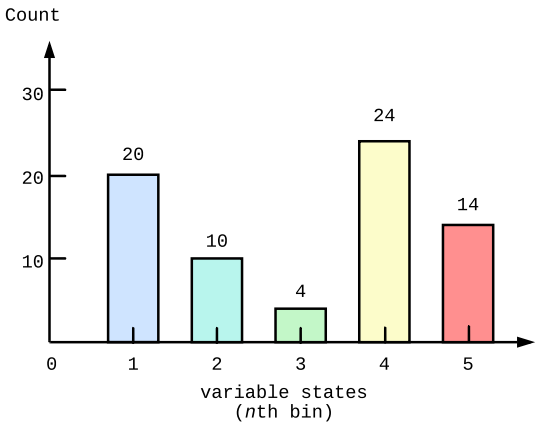
Figure 4.1: TwoCoins: An example of simple probabilistic program from Dice [14].

4.3 Approximate Query Processing

This section shows how BayesTensor supports AQP. BayesTensor provides three aggregates (both scalar and group by aggregates), which are COUNT(*) (i.e., CardEst), AVG(), SUM(), and GROUP BY aggregation in database languages. AVG and SUM aggregates can only apply to numeric variables.

During the training process, since we train BNs over real-world datasets that contain a mixture of the discrete and continuous distribution, before the training, we discretize those datasets into discrete data distribution by histogram for each continuous variable, where each bin represents a unique discrete state of that variable in integers. For each discretized continuous variable, each bin in the histogram records a range based on the related data from the dataset. There is no need to apply the histogram approach for categorical variables because they already have discrete distributions with discrete states. During querying, observations made with values of that variable that belong to the same bin will be parsed to the same discrete state by the query parser.

Suppose we train on a continuous variable x and Figure 4.2 is the corresponding histogram we construct during the training. In Figure 4.2a, the x-axis represents discrete states, also called bins, that are used to bin the values of variable x, and the y-axis shows the number of values that belong to each distinct bin from the dataset. Based on this example, we will explain how we perform SUM() and AVG() on the single-table dataset



(a) The histogram of discretized continuous variable x.

Discrete state (nth bin)	Value range
1	0-15
2	15-30
3	30-45
4	45-60
5	60-75

(b) The range of value for each bin corresponding to 4.2a

Figure 4.2: An example of discretizing a continuous variable x.

in the following two sections.

4.3.1 SUM

This section will show how to perform scalar SUM() (i.e., SUM aggregates without GROUP BY) on a single-table dataset.

Based on the context demonstrated by Figure 4.2, scalar SUM aggregates can be computed by summing up the product of the mean of the range from each bin and the estimated count for each bin by doing a COUNT(*) GROUP BY on the variable x. If each bin represents a single value instead of a range, then sum over the product of the value represented by each bin with the corresponding COUNT value for that bin. For instance, for variable x in Figure 4.2, we take means from each bin which are 7.5 for 1th bin, 22.5 for 2th bin, 37.5 for 3th bin, 52.5 for 4th bin, and 67.5 for 5th bin. Assume results of perform a COUNT(*) GROUP BY on variable x are 19 for 1th bin, 6 for 2th bin, 2 for 3th bin, 20 for 4th bin, and 16 for 5th bin. Then, we will get the SUM(x) as below:

$$SUM(x) = 7.5 * 19 + 22.5 * 6 + 37.5 * 2 + 52.5 * 20 + 67.5 * 16 = 2482.5 \quad (4.1)$$

In our implementation, we perform AQP based on the Flights dataset, which has a Chow-Liu tree BN visualization in Figure 4.3:

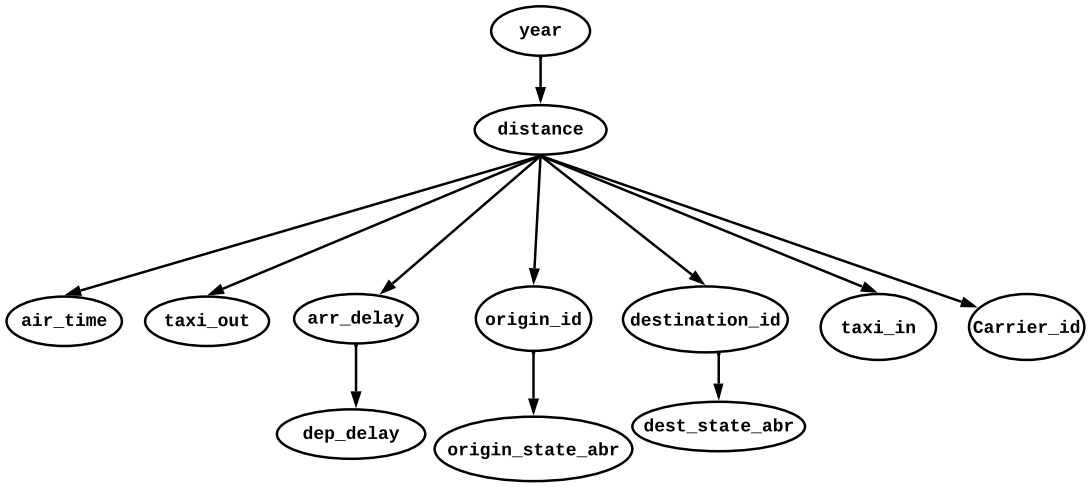


Figure 4.3: The BN visualization of the Flights dataset in our implementation.

An example of SUM() queries in the Flights benchmark is presented in Figure 4.4, where Figure 4.4b is the generated Python program that performs inference according to the query specified in PostgreSQL in Figure 4.4a. Since we use *exec*() command to execute the generated Python code with the local environment, no distribution specification is obtained in the generated code, presented in Figure 4.4b.

In Figure 4.4b, Line 5 incorporates the observation with the variable *unique_carrier* is '9E'. Line 6 performs the scalar SUM aggregate, SUM(*distance*), by pushing computation of computing SUM(*distance*) and COUNT(*) GROUP BY *distance* into one line of einsum operation. At line 6, since we want to sum over distance, we expect the marginal inference of the variable *distance* and treats its total number of discrete states at the output shape in einsum notation with the einsum inputs wf_carrier_id, wf_distance, wf_year which are the variables from BN (Figure 4.3) used for VE. After this, we multiply the cardinality of Flights (i.e., the input "nrows" in einsum operation at line 6 in Figure 4.4b) with the marginal probability of *distance* which is performing COUNT(*) GROUP BY *distance*. At line 6, the other einsum input "bins_avg" is the pre-computed means from the histogram of *distance*, and we again multiply it with results from COUNT(*) GROUP BY *distance* to get the result of SUM(*distance*).

4.3.2 AVG

In our implementation, we compute AVG() by dividing SUM by the COUNT. For instance, for AVG(x), we divide the results from SUM(x) by the results of COUNT(*) GROUP BY on x with Numpy function `np.divide`.

4.3.3 GROUP BY aggregates

BayesTensor supports GROUP BY aggregates on scalar aggregates COUNT(), AVG() and SUM(). We treat GROUP BY aggregates as the variables that should not be eliminated during the VE, such that the inference produces a marginal distribution of variables obtained in scalar aggregates and GROUP BY aggregates.

```
SELECT SUM(distance) FROM flights WHERE unique_carrier='9E';
```

(a) An example of SUM() queries from Flights benchmark.

```
import numpy as np
from time import perf_counter
from opt_einsum import contract
def infer(wf_distance,wf_carrier_id,wf_year,nrows,bins_avg):
        wf_carrier_id = contract("AB->B", wf_carrier_id[[18]])
        res = contract('B,Bw,w,B,->w',
            wf_carrier_id,wf_distance,wf_year,bins_avg,nrows,
            optimize='dp')
        return res
t_start = perf_counter()
infer_res = infer(wf_distance,wf_carrier_id,wf_year,nrows,bins_avg)
```

(b) Corresponding code generation of 4.4a by BayesTensor.

Figure 4.4: An example of SUM() queries supported by BayesTensor.

Chapter 5

Experiments

This chapter demonstrates three experiments to evaluate the performance of different applications introduced in Chapter 4. Experiments in this Chapter are conducted on single-table datasets. In this chapter, we evaluate the performance of BayesTensor by comparing it with corresponding competitors in topics.

5.1 Experimental Setup

In this section, we introduce the setup of our experiments, including the device used for experiments and detailed insight into each dataset we used.

5.1.1 Datasets and query workloads

Three datasets are used for benchmarking: Census, DMV, and Flights. All datasets are single-table datasets. The detail of each dataset is presented below.

5.1.1.1 Census

The Census dataset is generated using Data Extraction System by Microsoft and has 2,458,285 tuples and 68 categorical attributes. The data included is a segment of the U.S. census survey carried out by the United States (U.S.) Department of Commerce Census Bureau in 1990. This dataset obtains relatively data distribution complexity. We use the same attributes and query workload from [33].

5.1.1.2 DMV

The DMV dataset involves vehicle, snowmobile, and boat registration records in the State of New York (NYS). Although the local government of NYS continuously updates this dataset, there is no noticeable shift in data distribution among versions. The DMV snapshot we used has 12,593,240 tuples and 20 attributes. In experiments, we use the same attributes as [36] and the same query workload from [33].

5.1.1.3 Flights

The Flights dataset aims to benchmark Interactive Data Exploration (IDE) systems which have a majority of ad-hoc queries incrementally built by users during the query process. The original Flights dataset is a real-world dataset that contains information about the United States (U.S.) domestic flights managed by the Bureau of Transportation Statistics (BTS). The Flights dataset we used is rescaled by the data generator of the Interactive Data Exploration Benchmark (IDEBench) [10] with a scale factor of 1 billion (SF=1,000,000,000). We use the same sampling rate of 0.01 and query workloads as [13]. The sampled snapshot we experimented with has 12 attributes and 10002716 tuples.

5.1.2 Competitors

This section aims to introduce competitors for each experiment in this project. In experiments, we did not include Pyro [4] and PyTAC [9]. In sections 5.1.2.2 and 5.1.2.3, we will explain the reason for the exclusion of Pyro and PyTAC, respectively.

5.1.2.1 Dice

Dice [14] is a PPL specialized in discrete distribution. Dice supports fast, exact inference by logical expressions and exploiting the local structure with weighted model counting (WMC). Dice compiles a discrete probabilistic program into a binary decision diagram (BDD), a compression of logical expressions, which naturally encodes independence among variables within the program. The WMC approach exploits the BDD's structure to make inferences as a product of weights of the conjunctive normal form (CNFs) from the BDD. The efficiency of Dice is linearly dependent on the size of the BDD. [14] shows that Dice achieves fast inference over discrete probabilistic programs. However, the inference of Dice over Bayesian Networks is suboptimal (compared with ACE in [14] 's Table 3). Dice has limitations led by binary decoding of discrete distribution and point conditioning, while our framework supports range conditioning by integer decoding. In this project, we compare the performance of our framework in simple probabilistic programs and CardEst with Dice.

5.1.2.2 ACE

ACE is a Bayesian Networks (BNs) inference engine that uses similar techniques for inference as Dice. Instead of using BDDs as Dice, ACE compiles BNs into ACs in terms of CNFs with weights. ACE supports exact inference by exploiting the deterministic local structure and amortizes the pre-compiled offline phase for online queries. Nevertheless, ACE has the same problem as Dice: it only supports point conditioning instead of range conditioning, which vastly slows the inference for range queries.

Similarly, PyTAC [9] performs inference based on tensor-implemented Arithmetic circuits (ACs). Since PyTAC is not publicly available, we omitted PyTAC in our experiments.

5.1.2.3 Pgmpy

Pgmpy [2] is a Python framework for probabilistic graphical models (PGMs), particularly Bayesian Networks, targeting three aspects: (1) learning (including structural and parameters learning), (2) inference, and (3) simulation. The latest version of Pgmpy empowers tensor contraction. In CardEst's experiment, Pgmpy outperforms Dice for most point and range queries.

In a similar vein, Pyro is a Python framework that supports a variety of inferences (e.g., exact message passing, stochastic variational inference, etc.) with tensor variable elimination [22]. However, Pyro does not have an official API clarification for BNs construction. Hence, we omitted Pyro in our experiments as well.

5.1.2.4 DeepDB

DeepDB [13] is the state-of-the-art framework that supports CardEst, AQP, and Machine Learning (ML). DeepDB is data-driven and implemented with Relational Sum-Product Networks (RSPNs) to capture the data distribution. During the training, based on the database schema, every single-table dataset will learn an RSPN, while the multi-table dataset will learn an ensemble of RSPNs where each RSPN is learned based on a unique join condition. The training of our framework is similar to that of DeepDB by replacing RSPNs with BNs.

5.1.3 Experimental environment

This project conducts all experiments on Mac OS X, equipped with a 2.6GHz Intel Core i7 CPU with 6-core, 16GB RAM, and 500GB SSD. In all experiments, we use Python 3.9.7, Numpy 1.21.5, and opt_einsum 3.3.0. In the CardEst experiment, we use ACE 3.0. For both simple probabilistic programs and CardEst experiments, we use Dice with the latest version (2023-04-10).

5.2 Simple probabilistic programs

In this experiment, we evaluate two categories of discrete probabilistic programs: (1) seven simple probabilistic programs implemented by Dice [14], and (2) nine well-known discrete BNs from the online Bayesian Network Repository[1]. The evaluation results are shown in Tables 5.1 and 5.2. For end-to-end latency measurement in this section, we use the command line benchmarking tool `hyperfine` [2] to be consistent with Dice, with a warmup of three runs and an evaluation averaged over five runs. In this section, we used `-determinism -eager-eval`. The first flag optimizes deterministic, probabilistic choices, e.g., flip 0 is replaced by false.

[1] https://www.bnlearn.com/bnrepository/
[2] https://github.com/sharkdp/hyperfine

5.2.1 Dice's Discrete Probabilistic Programs

This experiment evaluates the inference performance of seven simple probabilistic programs established by Dice [14].

The evaluation result of this experiment is presented in Table 5.1. Each row in Table 5.1 represents a different discrete probabilistic program as a benchmark. The "Benchmark" column denotes the name of each benchmark. The "Dice" and "BayesTensor " columns indicate the end-to-end performance time measurement in milliseconds for Dice and our framework, respectively. For each simple probabilistic program, we observe an outperformance of BayesTensor compared to Dice, with a modest gap between the performances of the two systems.

Benchmark	Dice (ms)	BayesTensor (ms)
Grass	26.0	**19.3**
Burglar Alarm	26.1	**18.1**
Coin Bias	25.4	**17.1**
Noisy Or	28.7	**19.8**
Evidence1	25.2	**16.9**
Evidence2	25.3	**17.1**
Murder Mystery	26.0	**16.9**

Table 5.1: End-to-End time for simple probabilistic programs.

Size	Number of Nodes	Benchmark	Dice (ms)	BayesTensor (ms)
Small	<20	Cancer	28.1	**17.8**
		Survey	29.5	**18.2**
Medium	20-50	Alarm	480.9	**46.6**
		Insurance	843.6	**60.2**
		Water	655.7	**343.6**
Large	50-100	Hepar2	329.3	**104.1**
		Hailfinder	Time out	**123.9**
Very Large	100-1000	Pigs	621.8	**430.1**
Massive	>1000	Munin	37692.0	**1997.0**

Table 5.2: End-to-End time for single marginal inference on Bayesian networks. We consider a time out of two hours.

5.2.2 Single marginal Inference on Bayesian Networks

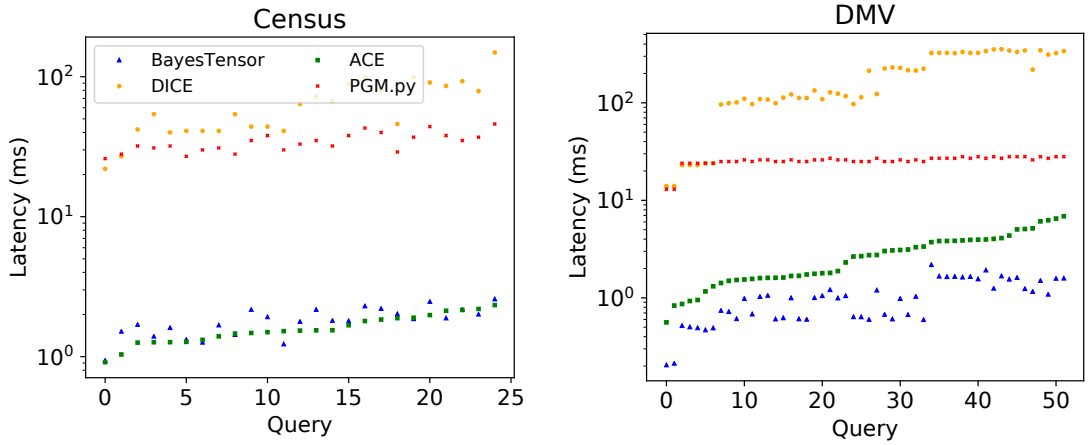
This experiment evaluates the performance of single marginal inference over nine renowned BNs in different sizes from the bnlearn. In this experiment, we perform inference over BNs of various sizes, from small to massive BNs. Table 5.2 shows the number of nodes corresponding to each size. For each evaluation, we perform the marginal inference of a specific node in each BN used by Dice.

The evaluation results are presented in Table 5.2. Unlike Dice's Discrete Probabilistic Programs experiment that performances between systems are similar, we observe a significant outperformance of BayesTensor compared to Dice in single marginal inference over BNs, particularly for BNs Munin and Hailfinder. BayesTensor is 19x faster than Dice in marginal inference over the large BN Munin. For Hailfinder, Dice fails the single marginal inference within two hours, represented as a timeout in Table 5.2, while the inference latency of BayesTensor is within 125 milliseconds.

5.3 Cardinality Estimation

BayesTensor supports CardEst inspired by BayesCard [33] introduced in Chapter 2.3.1. This experiment examines the performance of BayesTensor with three other competitors, which are ACE [6], Pgmpy [2] and Dice [14], in the application of CardEst. In this experiment, since we perform the exact inference supported by each system, the accuracy of all four systems is approximately the same. The evaluation results of CardEst are presented in Figure 5.1 and Figure 5.1.

Figure 5.1 demonstrates the performance of CardEst with point queries. There is a considerable gap in Figure 5.1 where BayesTensor and ACE significantly outperform Dice and Pgmpy for all Census's point queries. In Figure 5.2, the difference among systems is slightly reduced, but apparently, BayesTensor outperforms all other systems for all DMV's point queries. The latency of DMV point queries is observed as: BayesTensor < ACE < Pgmpy < Dice, where < represents that the former system obtains a faster inference than the latter system.



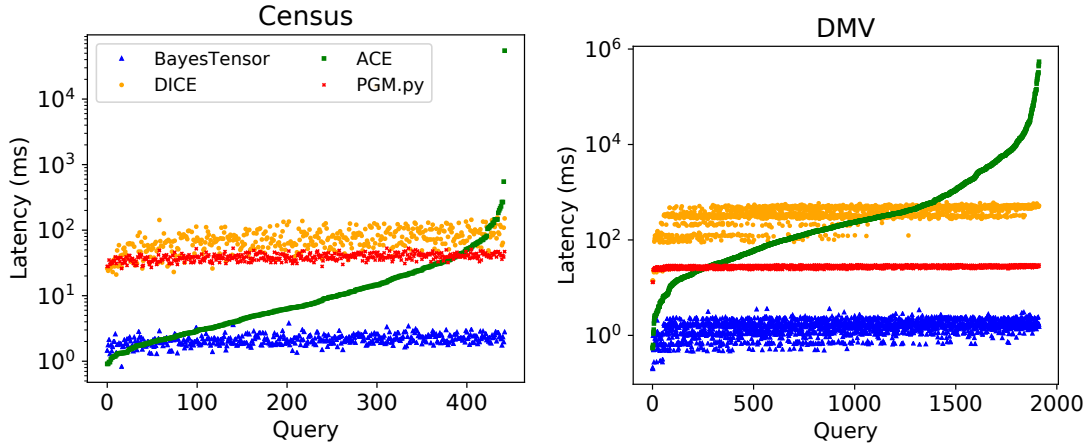
(a) Evaluation of Census with point queries

(b) Evaluation of DMV with point queries

Figure 5.1: Comparison of the latency performance for CardEst among systems on queries with single value for unique selections.

Figure 5.2 compares the system performance among BayesTensor, ACE, Pgmpy, and Dice of range queries on Census and DMV. The high density of results in Figure 5.2 illustrates that there are more range queries than point queries in the query workload for both Census and DMV. Thanks to highly dense result points, it is apparent that

BayesTensor exceeds the other three systems in most cases for both Census and DMV range queries. There is an increase in the latency of ACE performance because ACE does not support range conditioning. In other words, to perform range conditioning, ACE requires the user to manually write a batch of point queries, which is handled by code generation in our case. For a range query, ACE performs a Cartesian product of point queries with observation values of each distinct variable generated from that range query, which is time-consuming during the inference. It is evident that as the range conditions of queries increase, the performance of ACE deteriorates even more compared to Dice and Pgmpy. However, for range queries of both Census and DVM datasets, which have minimal predicates, ACE can infer in 2 milliseconds. While Dice maintains a stable performance for Census and DMV queries, it falls significantly short of BayesTensor. The overall performance of Pgmpy lags behind BayesTensor but exceeds Dice in most cases.



(a) Evaluation of Census with range queries (b) Evaluation of DMV with range queries

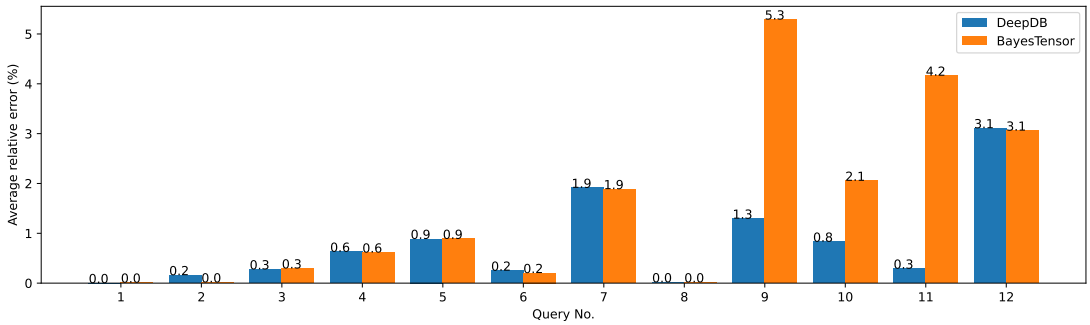
Figure 5.2: Comparison of the latency performance for CardEst among systems on queries with various values for unique selections.

5.4 Approximate Query Processing

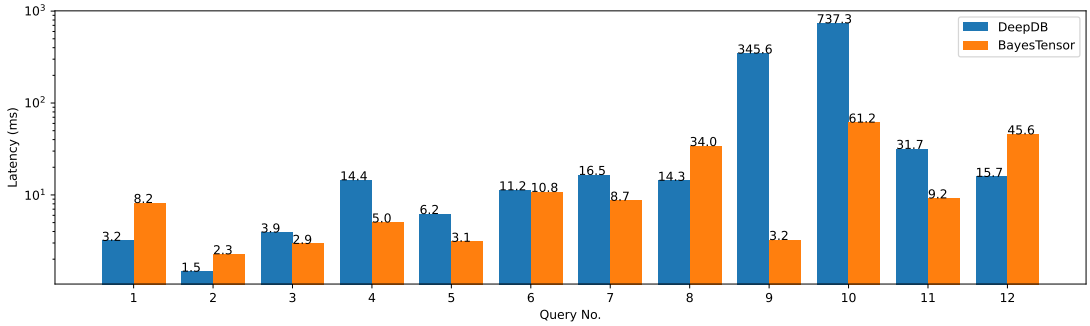
This section describes how to conduct the AQP experiment. In this experiment, we conduct 12 AQP queries based on the Flights dataset and compare the performance in terms of accuracy and latency with the state-of-the-art AQP framework DeepDB. The 12 AQP queries in the Flights benchmark are: Two AVG() queries without GROUP BY aggregates, five GROUP BY COUNT() queries, and four GROUP BY SUM() queries where two GROUP BY SUM() queries includes arithmetic operations within the SUM() function, such as SUM(A-B) and SUM(A*B). The evaluation results are averaged over five runs with a warm-up of five runs.

AQP Results based on the Flights dataset are shown in Figure 5.3. In this experiment, Q1-2 are scalar AVG() queries, Q3-5 and Q9-10 are COUNT() GROUP BY queries, Q6-7 and Q11-12 are SUM() GROUP BY queries where Q11 contains a subtraction within the SUM() and Q12 contains a multiplication within the SUM(). Figure 5.3a shows

the comparison of BayesTensor and DeepDB in the accuracy of each query from its truth value computed by Postgres, and Figure 5.3b compares the inference performance between BayesTensor and DeepDB, corresponding to queries in Figure 5.3a.



(a) Comparison of average relative error (in percentage).



(b) Comparison of latency (in milliseconds).

Figure 5.3: Comparison of AQP experiment performance between DeepDB and our framework.

As shown in Figure 5.3a, BayesTensor is less accurate for queries Q9, Q10, and Q11 than DeepDB. However, in Figure 5.3b, although BayesTensor does not exceed DeepDB in most cases, BayesTensor maintains a stable performance for all benchmark queries, especially for Q9, Q10, and Q11 compared to DeepDB. Considering only the latency performance, it is found that BayesTensor is faster than Dice for all COUNT queries, which are Q3-5 and Q9-10 in Figure 5.3b. However, the latency of BayesTensor for SUM queries is suboptimal and so as affecting the AVG queries.

5.5 Productivity

This section presents our productivity events by timeline. The details of this project management are shown in Table 5.3 below.

We first implement the AQP experiment. We reproduce the results of the Flights dataset with DeepDB. Then, we implement our framework in order of COUNT, AVG, SUM, and Group By aggregations. During the implementation, to maintain accuracy consistency with DeepDB, we debug for parameters tuning with our framework for around one week. Then, we do evaluations and comparisons for the CardEst experiment.

Since AQP includes CardEst, our framework adapts to datasets Census and DMV for evaluation. For Dice and Pgmpy, in MInf part 1, we have already implemented CardEst for those two systems in MInf part 1. As a result, we adapt the code from MInf part 1 with the latest Dice compiler and Pgmpy implementation. Finally, we implement and evaluate the performance of ACE in CardEst, taking around two weeks. For simple probabilistic programs benchmark, we reproduce the results of Dice in two days while the implementation and evaluation with BayesTensor take around three weeks.

Experiment	System	Datasets	Coding effort
AQP	Reproduce DeepDB	Flights	1 week
	Implement our framework		2 months
	Optimise our framework		5 months
CardEst	Evaluate our framework	Census DMV	1 week
	Implement and evaluate ACE	Census DMV	2 weeks
	Evaluate Dice	Census DMV	1 day
	Evaluate Pgmpy	Census DMV	1 day
Simple probabilistic programs	Reproduce Dice	Dice benchmark	2 days
	Evaluate our framework		3 weeks

Table 5.3: Productivity of our project

Chapter 6

Conclusions

In this chapter, we review our contributions to this project and summarize the main achievements of our project and potential improvements to our current work.

6.1 Project Contributions

The main contributions of this project are:

1. We propose a framework, BayesTensor, that specializes in fast, exact inference over discrete probabilistic programs and general BNs, capitalizing on tensors.
2. We demonstrate three applications of BayesTensor: (1) Discrete probabilistic programs, (2) CardEst, and (3) AQP.
3. We evaluate each application with competitors of BayesTensor for that application on single-table datasets (excluding the simple probabilistic programs benchmark from Dice).

6.2 Result overview

In this project, we introduce our framework, BayesTensor, and demonstrate its main applications in discrete probabilistic programs, CardEst and AQP. By evaluating three experiments, which are simple probabilistic programs, CardEst experiment, and AQP experiment, we prove that BayesTensor enables fast, exact inference over discrete distribution. By evaluating the simple discrete probabilistic programs experiment from Dice, we observe that BayesTensor outperforms Dice in inference over discrete probabilistic programs and BNs. In addition, in the CardEst experiment, BayesTensor exceeds the other three systems (i.e., ACE, Pgmpy, and Dice) in most cases with benchmarking on single-table datasets Census and DMV. However, the performance of BayesTensor shown in the AQP experiment is suboptimal, and the implementation of this feature is limited to single-table datasets, not general to support AQP over multi-table datasets, which can be improved as future work.

6.3 Future Work

This section discusses the potential extensions that can be made to BayesTensor. Up to this stage, we have performed AQP over a single-table dataset as DeepDB [13]. As a future work, we intend to extend the AQP application of BayesTensor with multi-table datasets, which contains more complex data distribution and dependencies caused by the join operation in database languages. The idea of performing AQP on multi-table datasets is similar to that of the DeepDB by replacing the RSPN ensemble with the BN ensemble. In addition, the AQP aggregates that BayesTensor can perform are limited to COUNT, AVG, SUM, and corresponding GROUP BY aggregates. In the future, we intend to support as many aggregates with BayesTensor as possible.

Bibliography

[1] Daniel G. a. Smith and Johnnie Gray. opt_einsum - a python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software*, 3(26):753, 2018.

[2] Ankur Ankan and Abinash Panda. pgmpy: Probabilistic graphical models using python. In *Proceedings of the 14th Python in Science Conference (SCIPY 2015)*. Citeseer, 2015.

[3] Atilim Gunes Baydin, Lukas Heinrich, Wahid Bhimji, Bradley Gram-Hansen, Gilles Louppe, Lei Shao, Prabhat, Kyle Cranmer, and Frank D. Wood. Efficient probabilistic inference in the quest for physics beyond the standard model. *CoRR*, abs/1807.07706, 2018.

[4] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20:28:1–28:6, 2019.

[5] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

[6] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, 2008.

[7] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467, 1968.

[8] Daniel-Ioan Curiac, Gabriel Vasile, Ovidiu Banias, Constantin Volosencu, and Adriana Albu. Bayesian network model for diagnosis of psychiatric diseases. In Vesna Luzar-Stiffler, Iva Jarec, and Zoran Bekic, editors, *Proceedings of the ITI 2009 31st International Conference on Information Technology Interfaces, Cavtat/Dubrovnik, Croatia, June 22-25, 2009*, pages 61–66. IEEE, 2009.

[9] Adnan Darwiche. An advance on variable elimination with applications to tensor-based computation. *arXiv preprint arXiv:2002.09320*, 2020.

[10] Philipp Eichmann, Carsten Binnig, Tim Kraska, and Emanuel Zgraggen. Idebench: A benchmark for interactive data exploration, 2018.

[11] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. Revisiting reuse for approximate query processing. *Proc. VLDB Endow.*, 10(10):1142–1153, 2017.

[12] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. PSI: exact symbolic inference for probabilistic programs. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 62–83. Springer, 2016.

[13] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: Learn from data, not from queries *CoRR*, abs/1909.00607, 2019.

[14] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang. (OOPSLA)*, 2020.

[15] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[16] F.R. Kschischang, B.J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

[17] Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash K. Mansinghka. Picture: A probabilistic programming language for scene perception. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 4390–4399. IEEE Computer Society, 2015.

[18] Maggie Lieu, Will M. Farr, Michael Betancourt, Graham P. Smith, Mauro Sereno, and Ian G. McCarthy. Hierarchical inference of the relationship between concentration and mass in galaxy groups and clusters. *Monthly Notices of the Royal Astronomical Society*, 468(4):4872–4886, 03 2017.

[19] Jiebo Luo, Andreas E. Savakis, and Amit Singhal. A bayesian network-based framework for semantic image understanding. *Pattern Recognit.*, 38(6):919–934, 2005.

[20] Qingzhi Ma and Peter Triantafillou. Dbest: Revisiting approximate query processing engines with machine learning models. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1553–1570. ACM, 2019.

[21] T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. /Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

[22] Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Justin Chiu, Neeraj Pradhan, Alexander Rush, and Noah Goodman. Tensor variable elimination for plated factor graphs, 2019.

[23] Laurel J. Orr, Magdalena Balazinska, and Dan Suciu. Entropydb: a probabilistic approach to approximate query processing. *VLDB J.*, 29(1):539–567, 2020.

[24] Jingwen Pan and Amir Shaikhha. Compiling discrete probabilistic programs for vectorized exact inference. In *Proceedings of the 32nd International Conference on Compiler Construction (CC)*, 2023.

[25] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. Verdictdb: Universalizing approximate query processing. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1461–1476. ACM, 2018.

[26] Anand Patil, David Huard, and Christopher J Fonnesbeck. Pymc: Bayesian stochastic modelling in python. *Journal of statistical software*, 35(4):1, 2010.

[27] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. SPPL: probabilistic programming with fast exact symbolic inference. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 804–819. ACM, 2021.

[28] Fotis Savva, Christos Anagnostopoulos, and Peter Triantafillou. ML-AQP: query-driven approximate query processing based on machine learning. *CoRR*, abs/2003.06613, 2020.

[29] Jacob Schreiber. Pomegranate: fast and flexible probabilistic modeling in python. *CoRR*, abs/1711.00137, 2017.

[30] A. M. Shanghooshabad, M. Kurmanji, Q. Ma, Michael Shekelyan, Mehrdad Almasi, and Peter Triantafillou. Pgmjoins : random join sampling with graphical models. In *ACM Sigmod Conference on the Management of Data*, pages 1610–1622. ACM, June 2021.

[31] Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. Approximate query processing for data exploration using deep generative models. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1309–1320. IEEE, 2020.

[32] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. FactorJoin: A New Cardinality Estimation Framework for Join Queries. In *SIGMOD 2023*, 2023. To Appear.

[33] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou.

Bayescard: Revitilizing bayesian frameworks for cardinality estimation. *arXiv preprint arXiv:2012.14743*, 2020.

[34] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: One cardinality estimator for all tables. *Proc. VLDB Endow.*, 14(1):61–73, 2020.

[35] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.*, 13(3):279–292, 2019.

[36] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.*, 13(3):279–292, 2019.

[37] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. FLAT: fast, lightweight and accurate method for cardinality estimation. *Proc. VLDB Endow.*, 14(9):1489–1502, 2021.