Benchmark of Probabilistic Programming Languages

Jingwen Pan

MInf Project (Part 1) Report
Master of Informatics
School of Informatics
University of Edinburgh

2022

Abstract

Probabilistic programming is a tool for statistical modelling that automatically infers statistical information to extract the data pattern and supports experts in analysing the surrounding world by observing the current world of knowledge without hypotheses. Probabilistic programming languages (PPLs) are the intuitional programming languages for probabilistic programming. Compared with other general-purpose programming languages like C, Java and Python, PPLs naturally infer based on the data distribution and the prior possibilities that experts define and grasp the data pattern for research and applications purposes. Like machine learning models (e.g., deep neural networks (DNNs)), model training with PPLs has an analogous process by replacing the machine learning model with the graphical model. However, although PPLs seem to be essential tools in modern machine learning, there has been no sufficient research into PPLs in the last decades. Recently, PPLs were anew raising researchers' interests in exploring them. In this case, helping researchers find the most efficient PPL with an adequate trade-off of accuracy becomes crucial. This project examined four PPLs based on three Bayesian Networks (BNs), mainly based on a Cardinality Estimation (CardEst) use case called BayesCard. We focus on making exact inferences on BNs. By experiments, we found that the PPL named dice has outperformed our baseline, BayesCard, in terms of latencies on the Census dataset while achieving almost the same accuracy as BayesCard.

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 kind support and guidance. I would also like to thank my friends and parents for their support and understanding throughout the project year.

Table of Contents

Chapter 1

Introduction

In this chapter, an overview of this project is presented to explain the thesis of this project and inform readers main sections related and the experimentation process taken to support our results in high-level aspects.

1.1 Motivation

Using probabilistic models to capture data patterns, Model-based machine learning has continuously raised interest in the research community for years. Probabilistic programming languages (PPLs) are essential frameworks used to declare such probabilistic models and perform automatic probabilistic inference. As there is an increasing number of probabilistic programming languages developed and a boost in research interests in probabilistic models, it is crucial to help researchers find the most efficient and appropriate programming framework for their research. This project focuses on comparing probabilistic programming languages in criteria of accuracy and inference time based on a database application called Cardinality Estimation (CardEst).

CardEst is a fundamental and essential component of modern database usage. Figure 1.1 describes the modern DBMS usage workflow in aspects of querying. In this project, we mainly focus on CardEst, and in our plan for next year, we pursue exploring the approximate query processing (AQP) based on the implementation of this year. The overview of our CardEst implementation is presented in figure 1.2. Our main contributions are the PPLs programs generator and the query evaluation with PPLs inference based on the benchmarks of the corresponding training dataset. We do not always include the training PPLs models due to different PPLs syntaxes. A detailed version of the demonstration in 1.2 refers to figure 3.1.

By reviewing state-of-art, we decide to choose BayesCard [36] as our baseline because it outperforms almost all existing CardEst methods. Besides, BayesCard has achieved all Algorithm-Data-System (ADS) criteria while no state-of-art has accomplished, shown in table 1.1. However, BayesCard has limitations about not much implementation with different PPLs. As a result, we are curious about the performance

of BayesCard CardEst method in our selected PPLs: Infer.NET [22], Dice [15] and SPPL [28].

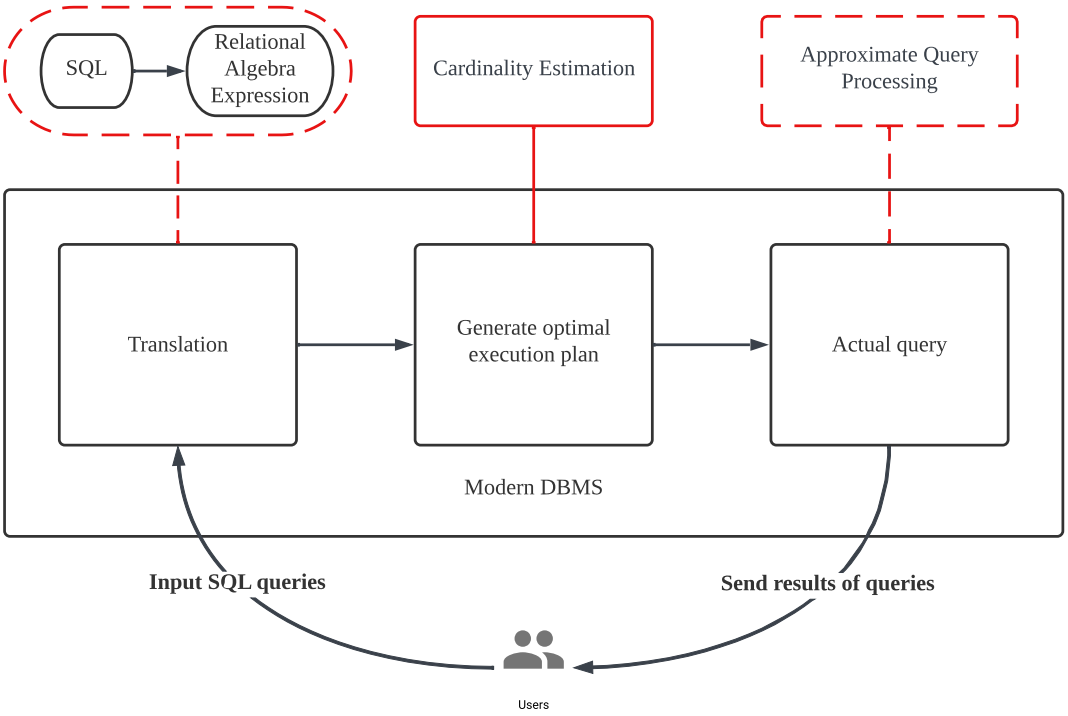


Figure 1.1: Overview of DBMS usage

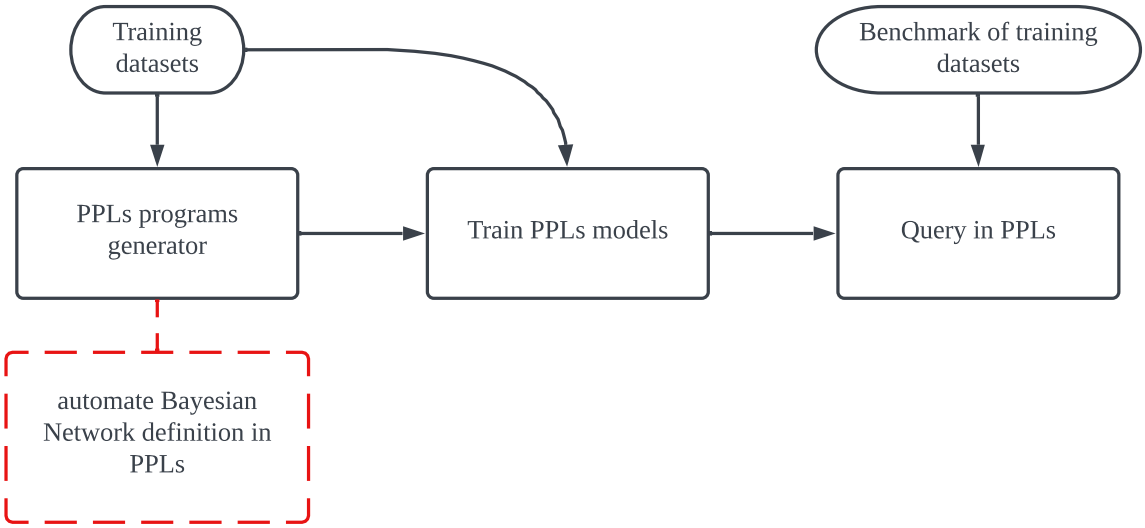


Figure 1.2: Overview of our implementation with CardEst

Table 1.1: Simplified version of status of CardEst methods [36].

CardEst methods	Accuracy	Latency	Updating	Interpret	Predict	Reproduce
Historgram	-	✓	✓	✓	✓	✓
Sampling	-	-	✓	-	✓	-
Naru	✓	-	-	-	✓	-
DeepDB	✓	✓	✓	-	✓	✓
...	...	...	...	...	...	...
BayesCard	✓	✓	✓	✓	✓	✓

1.2 Problem statement

As graphical models are naturally designed for statistic modelling and probabilistic programming, probabilistic programming languages (PPLs) are tools that automatically make inferences from graphical models and learn the data patterns in an intuitional way. Although PPLs have existed for a decade, there is not much benchmarking about PPLs. Instead, most papers and projects related are targeted to solve real-world problems and emphasise applications of PPLs. Under such a situation, we aim to focus on evaluating the efficiency of PPLs based on three datasets which include a variety of data distributions.

1.3 Solution

Our main objective is to evaluate four PPLs with three Bayesian Networks(BNs) in terms of accuracy and latencies: Pgmpy [1], Infer.NET [22], Dice [15] and SPPL [28].

The PPLs evaluated in this project are composed of two PPLs (Infer.NET and Pgmpy) presented from 2011 to 2015 and two PPLs (Dice and SPPL) from recent research in 2020 and 2021, respectively. In experiments, we use the same datasets (Census, DMV and IMDB) as BayesCard. For each dataset, we build a BN with each of our PPLs. To examine PPLs performance, we translate CardEst methods for single and joined tables based on BayesCard paper into our PPLs. We construct the evaluation process by adapting the decoding queries from BayesCard implementation for each dataset by combining the four PPLs selected and Python.

1.4 Contribution

Our contributions are shown as following:

1. We implement PPLs program generator in Python based on each PPLs syntax. We automate the BNs construction for each PPLs.
2. We construct BNs inlcuding adapting structure learning results from BayesCard, parameter learning and making inference.

3. We translate CardEst approaches of BayesCard into other three PPLs: Infer.NET, Dice and SPPL.
4. We adapt benchmarking resources from BayesCard to support implementations of the other three PPLs.
5. We evaluate the performance of four PPLs in terms of accuracy and latencies.

1.5 Dissertation structure

This dissertation comprises five main chapters, expounding important information about the project implementation.

Chapter 2: This chapter explains the literature review of the CardEst and the fundamental background knowledge needed to understand this project implementation.

Chapter 3: This chapter presents the design of this project and the detailed coding decision made during the implementation.

Chapter 4: This chapter aims to show how to set up experiments and the associated workflow of our evaluation process. This chapter includes our evaluation of four PPLs and detailed information about their performance.

Chapter 5: This chapter concludes the dissertation in the above aspects (Chapters 2, 3 and 4) via analysing the results given in chapter 4.

Chapter 6: This chapter evaluates our work and describes the plan we will take for next-year work and decisions made to complement and integrate our existing work.

Chapter 2

Background Chapter

In this chapter, we will review state-of-art approaches about CardEst with related use cases and existing PPLs.

2.1 Probabilistic Programming Languages

Probabilistic programming languages (PPLs) are programming languages used to do probabilistic programming. PPLs encode the probabilistic models and perform the inference of these models automatically. Because of this characteristic, implementing Bayesian networks with PPLs are more efficient and less expensive to infer than traditional methods. Examples of PPLs are below (table 2.1):

PPL name	Extends from	Host language
Picture [18]	Julia	Julia
PyMC [26]	Python	Python
Infer.NET [22]	.NET Framework	.NET Framework
Stan [2]	-	C++

Table 2.1: Examples of Probabilistic programming languages (PPLs)

As table 2.1 shows, there are a wide range of PPLs that can be used. As the machine learning process with general programming languages like python, the workflow of machine learning using PPLs is similar. Instead of using a traditional machine learning model (e.g., neural networks), PPLs first specify the structure of a probabilistic model based on the specific task that the model is designed for. Then, this probabilistic model will be trained with training datasets to learn parameters in order to make predictions. With the trained model, probabilistic programming inference methods will be selectively applied to the test datasets and perform evaluation for that specific task. The general workflow of probabilistic programming is shown in figure 2.1 below:

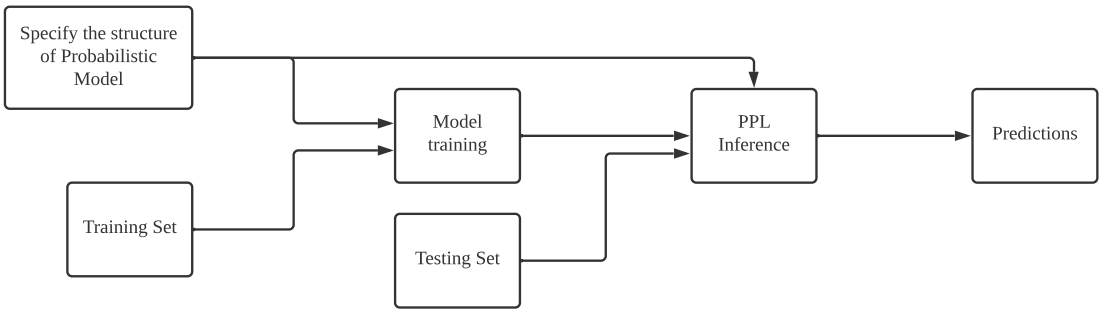


Figure 2.1: Simplified workflow of probabilistic programming

2.2 Bayesian Networks

Graphical models are a kind of probabilistic models which represent the dependencies among random variables in a structure of graphs. There are two main kinds of graphical models: Markov Hidden Fields and Bayesian Networks (BNs). In this project, we only focus on BNs.

Bayesian Network is a type of graphical model that demonstrates the joint probability of all random variables. It is a directed acyclic graph (DAG) where each node in the graph declares a variable and each directed edge represents the relationship between two connected nodes as parent and children. A parent node is a node which has an arrow line pointing out to another node, and the node that is pointed towards is the corresponding children node. Each parent node will have at least one children node and each children node can have at least one parent node.

In the Bayesian Network, besides the relationship, each directed edge is allocated a probability to indicate how likely will children node happen dependent on its parent node. The formula of the joint probability of a Bayesian Network is:

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

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

For example, the figure 2.2 at below is an example of the Bayesian Networks. In figure 2.2, there are five nodes A, B, C, D and E. For node A, since there are three directed edges out from node A pointing to nodes B, C and D, A is the parent node of nodes B, C and D. In other words, nodes B, C and D are children nodes of node A. Also for node D, because there are three directed edges pointing to node D from nodes A, B and C respectively, node D has 3 parent nodes which are nodes A, B and C. Node D has one children node E as it has a directed edge connected towards the node E. The overall relationships in figure 2.2 are summarised in table 2.2.

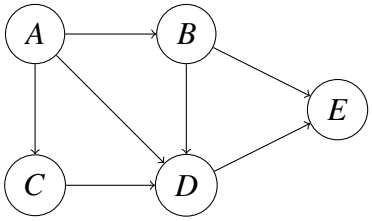


Figure 2.2: An example of Bayesian Networks

Node	Parent node(s)	Children node(s)
A	-	B, C, D
B	A	D, E
C	A	D
D	A, B, C	E
E	B, D	-

Table 2.2: Relationships of nodes in figure 1

BNs and Bayesian techniques are proved to have remarkable achievements on some data-analysis problems such as image processing [21], database query optimization [36] [33], medical diagnosis [7] and so on. BNs have following properties [12] [36]:

1. BNs can handle incomplete observations on data by encoding variables based on dependencies among variables.
2. BNs explore casual relationships among random variables. This allows to answer questions like "To increase A, what should be done to B?" or "To increase A, is it worth increasing B?", which are useful in tackling real-world problems.
3. BNs can be easily updated using the Bayes theorem and have stable performances because they capture the intrinsic data pattern at a general level.

All these shreds of evidence prove that BNs are practical, predictable, easy to be updated and maintained. Because of above advantages, BNs can be applied to applications in lots of fields and thus have a great development potential.

In this project, our benchmarking is developed based on the baseline model **BayesCard** [36], a BN model implemented in probabilistic programming languages **Pomegranate** [31] and **Pgmpy** [1] to do CardEst.

2.3 Database

As mentioned in section 2.2, there is a wide range of fields that Bayesian Networks can be used. This project will explore the use in database field in the following sub-sections, including the Cardinality Estimation (CardEst). An introduction to the database and Database Management Systems (DBMS) is described below.

A database is a collection of data or information that is managed in structure and stored inside a computer system. A database usually has two components: Entities and relationships among entities. For example, a database of a Google branch office can have entities such as staffs, offices, products and research programmes. The relationships among entities in this database can be the participation of staff members in a research programme, products' design attendance of staff members and so on.

Database Management Systems (DBMS) are software systems that are used to manage the databases. DBMS enable the users to access the databases to store information and

enable information to be retrieved, deleted or updated for users' purposes. DBMS can be used by multiple users simultaneously. The examples of DBMS are MySQL, Google PostgreSQL, Microsoft Azure and Oracle. In DBMS, the process of running queries will be:

1. The user codes the queries and submits to run these queries with the user interface of DBMS software.
2. The DBMS software sends an API request with all details of the queries to the DBMS server. API request managements such as authenticating and authorising the API request, and build the metadata such as SQL statements and query parameters during this stage.
3. Lexing and parsing the SQL statements. In other words, scanning raw SQL statements which are the arrays of bytes and converting those arrays into a series of tokens. And then, build up an syntactical tree representation of arrays with the tokens, which can be understood by the DBMS software.
4. The parsed queries will be passed to a query optimiser. Query optimisers will generate efficient execution plans for running the query based on how data are stored in DBMS. An execution plan is a tree of relational operators that will be used in running queries, including information about how will the data be accessed for satisfying queries.
5. By comparing and evaluating all query plans, query optimisers generate the most optimal query execution plan to run.
6. Results of running queries will be returned to users based on that optimal query execution plan.

The illustration of the above description is shown in figures 2.3 (the red circle denotes where CardEst should happen in usage) and 2.4, and 2.5 is an example of an execution plan that can be viewed in DBMS softwares (e.g., Microsoft Azure Data Studio).

2.3.1 Cardinality Estimation

This sub-section is about the work done related to the Cardinality Estimation (CardEst), which is an essential component of the query optimiser. In practical use, it is always expected to obtain the most efficient execution of SQL statements at the lowest cost. A query optimiser is designed to reduce the loss and cost. The query optimiser determines the optimal execution strategy in two aspects:

1. The number of rows that will be accessed to satisfy what query requests. This seems as the cardinality of the execution plan.
2. The cost model that estimates the run time of CardEst of the performing execution query plan with inference algorithms.

To ensure the optimal execution strategy, the query optimiser compares all query execution plans. Each query execution plan has its cardinality estimator which estimate the cardinality of that query plan with inference algorithms. These inference algorithms

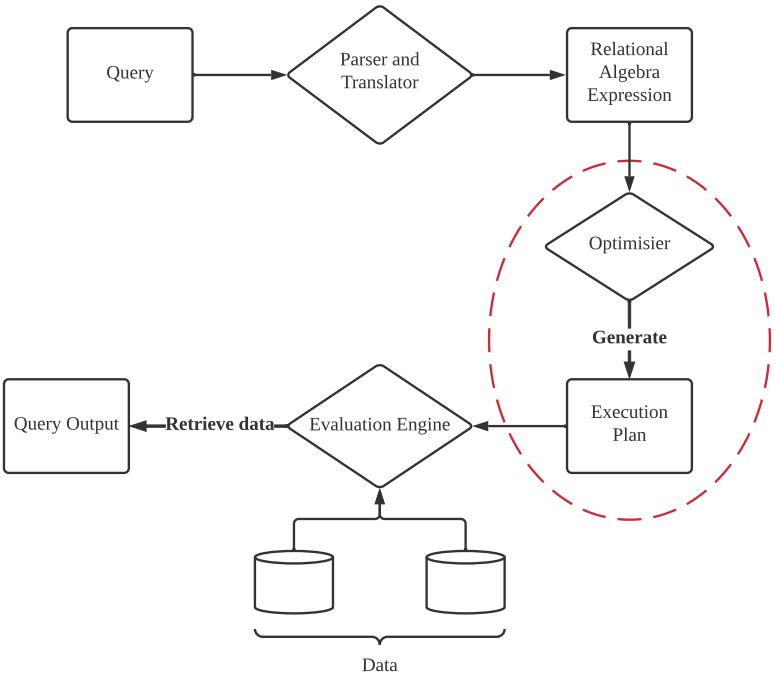


Figure 2.3: DBMS Query Process [27]

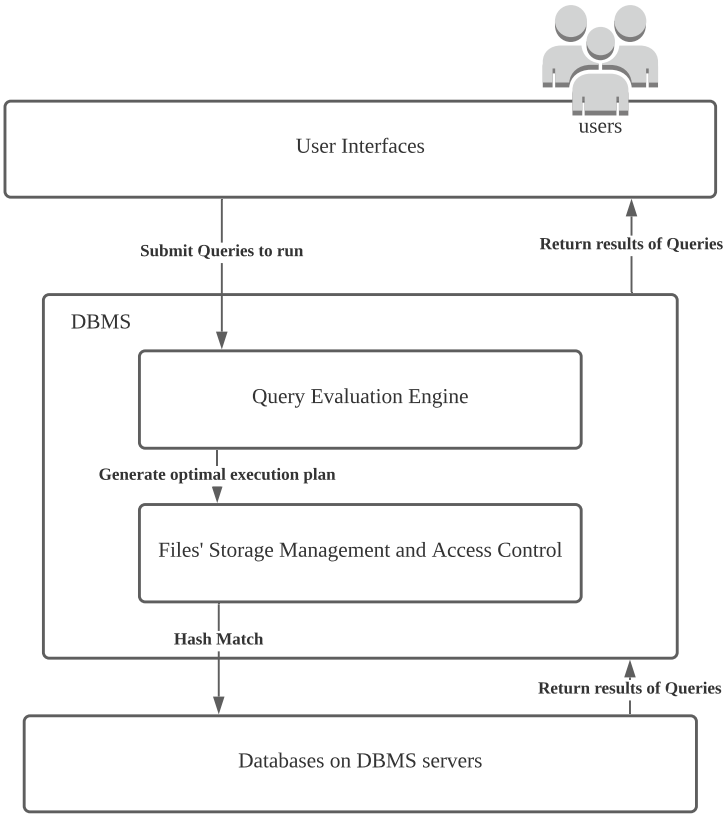


Figure 2.4: DBMS Architecture [27]

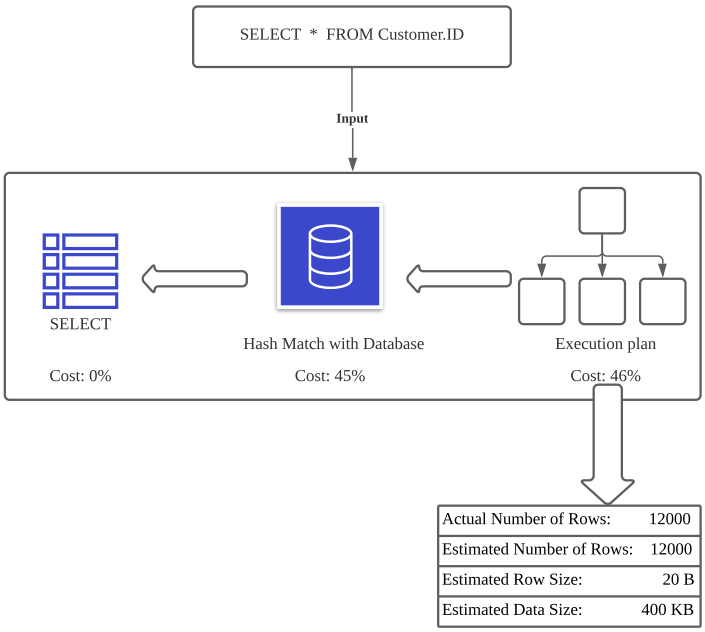


Figure 2.5: The possible execution plan review in real-world DBMS softwares

estimate the number of rows that will be considered to contain relative information requested by a query and seems them as the cardinality of that execution plan. The optimal execution plan will be generated based on the previous estimation.

In recent years, Cardinality Estimation is extensively explored by researchers. Table 2.3 below lists the Cardinality Estimation methods designed recently:

Model Name	Year	Real-world Application
Histogram [32]	-	PostgreSQL, SQL Server
MSCN [17]	2019	-
Naru [38]	2019	-
DeepDB [13]	2019	-
FLAT [40]	2020	-
Sampling [6]	2020	MySQL, MariaDB
BayesCard [36]	2020	-

Table 2.3: Cardinality estimation methods in recent years

In this project, only the BayesCard method will be benchmarked and BayesCard is the only CardEst method in construction of Bayesian Networks.

2.3.2 BayesCard

BayesCard [36] is a CardEst method that estimates the joint probability of each query with Bayesian Networks. It is designed with a data-driven probabilistic model called

BN-Ensemble, which organises the DBMS in terms of Bayesian Networks.

To construct a BN-Ensemble, BayesCard method takes a database **DB** as the input which contains **n** tables and a join scheme **J** of the *DB*. The join schema **J** in BayesCard is a tree of nodes where each node represents as a table and each edge represents the relationship between two tables. By this join scheme, the full outer join table of all tables will be generated from the join scheme **J** and based on this full outer join table, some unbiased samples will be generated to build up a Bayesian Network [37]. In BayesCard, the BayesCard model might not only contain one Bayesian Network but an ensemble of Bayesian Networks (see sections 3.3 and 3.4).

To understand this, the process of how can the full outer join table be produced is important. For example, assume that there are two tables *C* (2.4) and *E* (2.5):

C.key	C1	C2	C3
1	100	50	90
2	40	50	70

Table 2.4: Table C

E.key	E1	E2	E3
1	110	40	80
1	0	10	0
2	20	10	20
3	40	50	70

Table 2.5: Table E

Based on table *C* and *E*, a full outer join table Ω can be produced as below (table 2.6), which includes both the matched and unmatched tuples in two tables. In table 2.6, the third row in full outer join table Ω is produced by matching *C.key* and *E.key* when they are both equal to two and combining the corresponding rows in table *C* and *E* into one row. When a table has multiple records for the same key, like in the table *E*, there are two records when *E.key* equals to one. At this case, each row that has *E.key* equal to one will be matched to each row in table *C* that has *C.key* equal to one.

In the full outer join table Ω, if table *E* has n records when *E.key* equals to a certain value v and table *C* has m records when *C.key* equals to the same value v, then the number of full outer join records when $C.key = E.key = v$ will be mn, which is m multiplied by n.

C.key	C1	C2	C3	E.key	E1	E2	E3
1	100	50	90	1	110	40	80
1	100	50	90	1	0	10	0
2	40	50	70	2	20	10	20
∅	∅	∅	∅	3	40	50	70

Table 2.6: Full outer join table Ω of tables C and E

Let us have a look at how to contruct the BN-ensemble in BayesCard paper and 1

describes the construction algorithm from the paper [36].

Algorithm 1 BN Ensemble Construction Algorithm

Input: a DB schema with n tables $T_1, ..., T_n$ and a budget k

1: Create the join tree $T = (V, E)$ for the schema
2: Generate unbiased samples S for full outer join of the entire schema
3: Initialize a dependence matrix $M \in R^{n \times n}$
4: **for** Each pair of tables $e = (T_i, T_j)$ **do**
5: Calculate the RDC dependence level scores between all attributes in T_i and attributes in T_j
6: $w_e \leftarrow$ average RDC scores
7: **end for**
8: **if** $k = 1$ then **then**
9: **return** T and learn a single PRM for each table
10: **end if**
11: **for** $k^{'} \leftarrow 2, ..., k$ **do**
12: Sort E in decreasing order based on w_e
13: **for** $e = (u, v) \in E$ **do**
14: **if** u and v contain exactly $k^{'}$ tables in total **then**
15: Update T by contracting nodes u, v to a single node u, v
16: **end if**
17: **end for**
18: **end forreturn** T and learn a single PRM for each node in T

Looking at 1, from line 3 to line 7, BayesCard calculates the dependency level between each two of the tables in terms of Randomised dependence coefficient (RDC) values [20]. RDC values measure the level of dependence between two attributes, which reflects the likelihood when attribute A happens while the attribute B co-occurs. RDC values help to optimise CardEst by variable elimination (VE). In 1, BayesCard takes in a budget k to limit the number of tables can be taken into produce one full outer join table. If the budget equals to one, then a Bayesian Network will be constructed at each node, for each table. Otherwise, with a for loop iterating from two to the value of the budget, BayesCard applies the VE based on the highest dependency level among the connected tables. This is done to join the tables. Finally, BayesCard produces several Bayesian Networks, which is an ensemble of Bayesian Networks.

Having the BayesCard probabilistic model, inference algorithms can then be run with the test datasets. To understand how can the BayesCard estimate the cardinality of a query for joined tables, the process of applying fanout method is needed to know first.

BayesCard uses the fanout method for calculating the joint probability of a query, based on the example of full outer join table Ω shown above (table 2.6). In each full outer join table, the number of tuples in the full outer join table that matches the tuples in the original tables will be counted at each row, and this number is called a fanout attribute. For example, for full outer join table Ω of tables C and E mentioned above,

we have the table 2.7 below:

C.key	C1	C2	C3	E.key	E1	E2	E3	$F_{C\to\Omega}$	$F_{E\to\Omega}$
1	100	50	90	1	110	40	80	2	1
1	100	50	90	1	0	10	0	2	1
2	40	50	70	2	20	10	20	1	1
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	3	40	50	70	0	1

Table 2.7: Adding fanout attributes to full outer join table Ω

Fanout attributes in full outer join table Ω are the $F_{C\to\Omega}$ and $F_{E\to\Omega}$. In table 2.7, fanout attributes are marked in blue. Looking into table 2.7 (figures in magenta), the value of $F_{C\to\Omega}$ is two at first row. This is because the corresponding tuple from table C has appeared twice in this join table Ω: On the first row and the second row. By using fanout method, the dependencies between tables can therefore be represented in a simpler way in terms of fanout attributes. Fanout attributes will be used to estimate the joint probability of the query which captures the intrinsic pattern of the relations between tables in a database.

As a result, referring to theorem 2 in [36], we can calculate the joint probability for each attribute in the joined table query as following:

$$p_i = \frac{|v_i|}{|v|} * \sum_{f,v} \left(P_{V_i}(Q_i \wedge F = f \wedge F_{V_i}, V = v) * \frac{max\{v,1\}}{dlm} \right) \tag{2.2}$$

and the cardinality of query Q is

$$Cardinality = |v| * \prod_{i=1}^{d} p_i \tag{2.3}$$

where

1. i represents which attribute is queried.
2. $V = \{V_1, V_2, V_3, \ldots, V_d\}$ is the BN graph (join tree) vertices touched during query.
3. v is the full outer join of all tables in V.
4. For each node, (A_j, B_j) is a distinct join in the $F = \{F_{A_1,B_1}, F_{A_2,B_2}, \ldots, F_{A_n,B_n}\}$ where B_j is not in Q.
5. $f = (f_1, f_2, \ldots, f_n)$ represents as assignments to F as $F_{A_j,B_j} = f_j$ when $1 \le i \le n$, and $dlm(f) = \prod_{j=1}^{n} max\{v,1\}$

The BN-ensemble model also conducts graph reduction to ensure faster and more efficient estimation. When doing CardEst, BayesCard preprocesses each query and collects its predicates as a domain and does estimation on full outer join tables with fanout attributes. BayesCard's inference algorithms are based on inference methods defined

in a PPL package called Pgmpy [1]. BayesCard re-implement the original variable elimination (VE) inference method with just-in-time compilation (JIT) to maintain efficient estimation and progressive sampling [38] to compute the joint probability of query Q over tables $T_1, T_2, ..., T_n$. BayesCard first uses progressive sampling to compute the probability if a table (children node) is related to query Q if its dependent table (parent node) is relevant to Q. Then by this probability, it selects the fanout attributes that will be looked at and calculates the probabilities of how likely tables relevant to fanout attributes are related to query Q by compiled variable elimination (VE+JIT). The overall joint probability of the query Q is the multiplication of factorised results from progressive sampling and results from compiled variable elimination 2.3.2.

Chapter 3

Implementation

As all our implementations are based on BayesCard, we translate the approach BayesCard to evaluate CardEst on single tables and the BayesCard fanout method into Infer.NET, Dice and SPPL, where the fanout method estimates the cardinality of joined tables.

Specifically, although we have supported three PPLs, we mainly focus on Dice implementation in the following sections. The descriptions of other PPLs implementations refer to section 3.5.

3.1 Overview

Referring to figure 2.1 in section 2.1, to construct, train and evaluate graphical models, we need to define our graphical models, which are BNs. We train BNs to learn from data and evaluate with test datasets, in our case, which are queries, by making inferences for CardEst. Mainly, we use structural learning to define our BNs, and by parameters learning, we train our BNs to learn the data distribution.

The illustration of our implementation is in figure 3.1.

3.1.1 Structural Learning

Stucture learning is the process that helps to learn the structure of a BN, including learning the way how nodes connect.

Treated as the baseline, Pgmpy does not provide any structural learning method. As a result, the baseline uses the probabilistic modelling package Pomegranate to learn its BN structures. Pomegranate has a method called "BayesianNetwork.from_samples", which automatically learn the links of a specific BN from the given data. For example, assuming we have the following BN 3.2a and corresponding data to represent such relationships.

By Pomegranate, the relationship between nodes will be learnt as 3.2b at below.

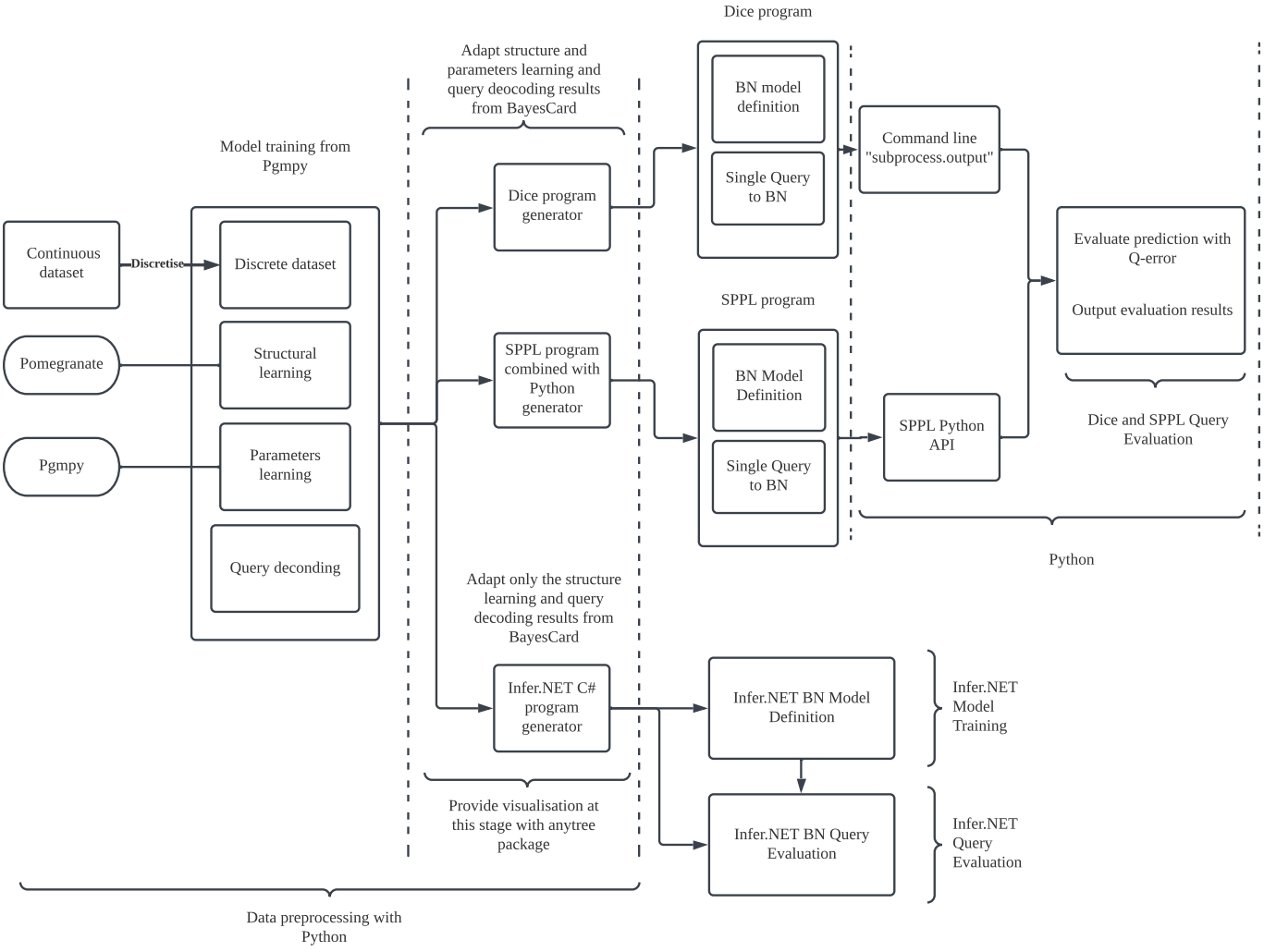
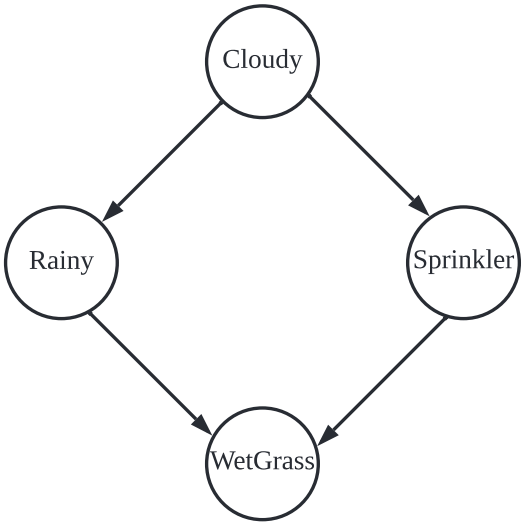


Figure 3.1: Overview of this project implementation



(a) Kevin Murphy's wet grass/sprinkler/rain example

```
1   [(Cloudy,Rainy),(Rainy,WetGrass),(Cloudy,Sprinkler),(Sprinkler
        ↪ , WetGrass)]
```

(b) The corresponding BN structural learning result

Figure 3.2: An example of BN structural learning

By sturctural learning, the links we learn enables the Pgmpy to do the parameters learning.

3.1.1.1 Chow-liu tree

Chow-Liu tree is a tree structure first proposed by Chow and Liu [5] in 1968. This tree structure can efficiently estimate the joint distribution as a product of second-order conditional and marginal distributions by sacrificing accuracy.

An example of Chow-Liu tree is presented in figure 3.3. The joint distribution of this Chow-Liu tree can be seemed as

$$\mathbb{P}(A, B, ..., F, G) = \mathbb{P}(E|C) * \mathbb{P}(F|C) * \mathbb{P}(g|C) * \mathbb{P}(C|A) * \mathbb{P}(B|A) * \mathbb{P}(D|A) \quad (3.1)$$

Pomegranate package has the Chow-Liu tree algorithm for constructing BN links by learning the structure from data, and the results of learnt BN links are in a similar way described in 3.2b. By adapting the BN structure from Pomegranate to Pgmpy or other PPLs, BNs with Chow-Liu tree structure can then be constructed.

In order to assist project implementation and provide a better presentation to help readers understand the project, we provide the visualisation of each BNs with the Python package Anytree. Anytree package can help build up almost any kind of tree structure and provide decent visualisations with the Python package networkx. Figure

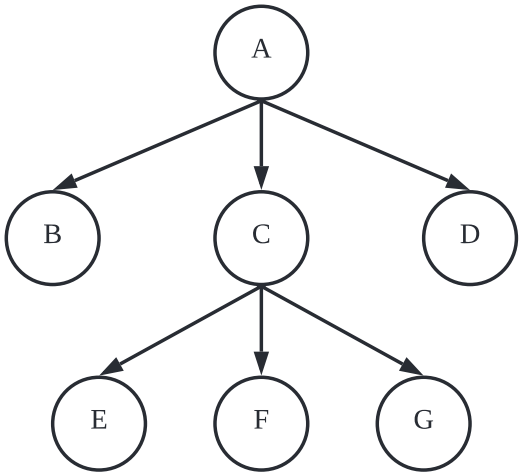


Figure 3.3: An example of Chow-Liu tree

3.4 demonstrates an example of the visualisation functionality in our implementation, which presents one of the BNs constructed based on the joined table dataset IMDB.

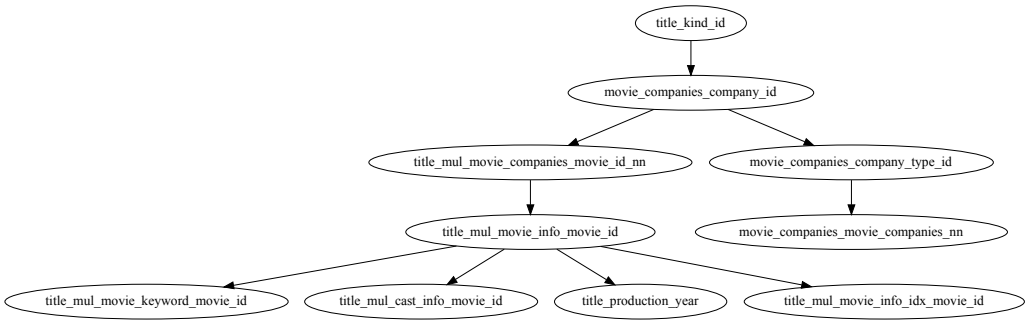


Figure 3.4: Visualisation of Chow-Liu Bayesian Network provided in implementation

To optimise the efficiency of the BN inference, we support the graph reduction in making inferences for the evaluation process. Chris Wadsworth first developed this technique in 1971. The main idea of graph reduction is that when making inference, we ignore the partial structure of the BN, which will not be used by probability inference. Graph reduction reduces the time for BN inferring the joint distribution for query evaluation [36].

3.1.2 Parameters Learning

Parameters learning is the process that helps to learn the distribution of each attribute/node from the data given. In our case, the parameters learning enables our BNs to learn probabilities from the training data. Package Pgmpy has a "BayesianModel.fit" method for Bayesian Networks parameters learning.

However, there is no parameters learning process for Dice and SPPL. As a result,

we adapt the BayesCard parameters learning results to these two PPLs. Compared to the baseline BayesCard, the implementation of Dice and SPPL does not include any structural learning and parameters learning process. Although Infer.NET does not have structural learning implemented, Infer.NET syntax enables us to implement parameters learning from the given training data.

3.1.3 Inference

Making Inference with graphical models is that by conditioning on specific attributes/nodes, graphical models return the joint distribution of the query results based on the parameters learnt from training data.

Each PPL has a different programming syntax for its inference method. For instance, Pgmpy has an Inference module that provides Variable Elimination and Maximum Likelihood to query BNs about probabilities. Compared to Pgmpy, it is more straightforward for Dice since Dice is a PPL focusing on the exact inference of discrete BN only. Details about how Dice does inference are explained in the following section.

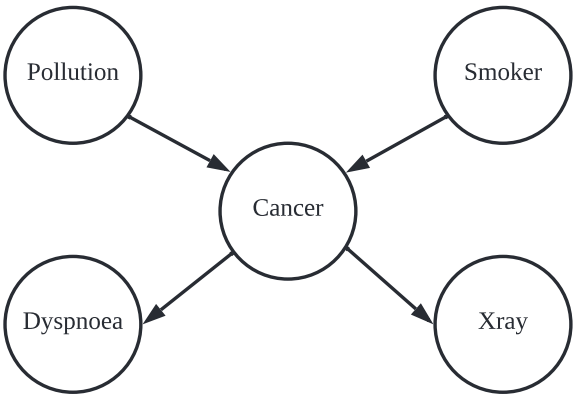
3.2 Dice

Dice is a PPL proposed by Holtzen et al. in 2020 [15]. Even though Dice paper itself includes benchmarking with different PPLs like Psi [9] and the Bayesian Network solver ACE [4], its evaluation does not obtain benchmarks based on complex real-world datasets and specific use cases. Since Dice is recently proposed, we believe that exploring the usage of Dice is potential. Although Dice does not provide any Python API, it has Dice compilers that can be invoked from the command line for use. Dice provides two compilers: Dice compiler from the Github *master* branch and native Dice compiler from Github *oopsla-artifact* branch. We find that the native Dice compiler sacrifices inference accuracy for efficiency by observation and testing while the Dice compiler does the opposite. Looking at the Dice syntax (3.5a), Dice provides neither structural learning nor parameters learning. Instead, to query Dice the joint distribution, we need to define the data distribution learnt from the parameters learning process and specify the attributes we would like to query in the same .dice file and run it with the Dice compiler.

In 3.5a, this BN has five attributes: Pollution, Smoker, Cancer, Dyspnoea and Xray. The Dice program 3.5a actually defines the BN relationships in 3.5b. In Dice syntax, "if ((attribute_name == int(N, X))) then ... else ...", N is the binary bitwidth of the attribute value range for Dice compiler (while for native Dice compiler, N is the attribute value range in decimal) and X is the attribute value to condition. The syntax "discrete(..., ..., ...)" defines the data distribution of the attribute following behind the syntax word "let". In the example 3.5a, we make a query for asking the joint distribution when all attributes occur at the same time by "(Xray,(Dyspnoea,(Cancer,(Smoker,Pollution))))" and the result of this query is shown in 3.5c.

```
1  let Pollution = discrete(0.5,0.4,0.1) in
2  let Smoker = discrete(0.3,0.7) in
3  let Cancer = if ((Pollution == int(2, 0))) then (if ((Smoker
       ↪ == int(1, 0))) then (discrete(0.03,0.97)) else (
       ↪ discrete(0.001,0.999))) else (if ((Pollution == int(2,
       ↪ 1))) then (if ((Smoker == int(1, 0))) then (discrete
       ↪ (0.03,0.97)) else (discrete(0.001,0.999))) else (if ((
       ↪ Smoker == int(1, 0))) then (discrete(0.05,0.95)) else (
       ↪ discrete(0.02,0.98)))) in
4  let Dyspnoea = if ((Cancer == int(1, 0))) then (discrete
       ↪ (0.65,0.35)) else (discrete(0.3,0.7)) in
5  let Xray = if ((Cancer == int(1, 0))) then (discrete(0.9,0.1))
       ↪  else (discrete(0.2,0.8)) in
6
7  (Xray,(Dyspnoea,(Cancer,(Smoker,Pollution))))
```

(a) The BN definition in Dice



(b) Corresponding visualisation of the Bayesian Network

```
1  ================[ Joint Distribution ]================
2  Value Probability
3  (0, (0, (0, (0, 0)))) 0.0026325
4  (0, (0, (0, (0, 1)))) 0.002106
5  (0, (0, (0, (0, 2)))) 0.0008775
6  (0, (0, (0, (0, 3)))) 0
7  (0, (0, (0, (1, 0)))) 0.00020475
8  (0, (0, (0, (1, 1)))) 0.0001638
9  .......
```

(c) The corresponding inference result of BN "Cancer" 3.5a in Dice

Figure 3.5: The BN example "Cancer" in Dice

However, when querying for joint distribution, both Dice compilers will compute all combinations of the joints and thus leading to inefficiency when querying probabilities for a list of attributes based on their specific values. As a consequence, we adapt the Dice syntax to do queries described in 3.6a.

```
1       let q = if (((dHours == int(6,3))||(dHours == int(6,0))||(
           ↪ dHours == int(6,5)))&&(dIncome4 == int(2,0))&&(
           ↪ dIncome5 == int(2,0))&&((iMobility == int(3,1))||(
           ↪ iMobility == int(3,0)))&&((dPwgt1 == int(4,0))||(
           ↪ dPwgt1 == int(4,1)))&&(iSex == int(2,1))&&(iSubfam1
           ↪ == int(4,0))) then (discrete(1.0, 0.0)) else (
           ↪ discrete(0.0, 1.0)) in
2
3       q
```

(a) Second example query in Dice

```
1   Value Probability
2   0 0.160781
3   1 0.839219
4   Final compiled size: 3490
5   Live: 37468
```

(b) The corresponding query result

Figure 3.6: An example of Dice query optimisation

The query in 3.6a is one of the queries we have with the single dataset Census. This query syntax describes how to query the Dice about the joint distribution when attributes dHours values zero, five and six, dIncome4 values zero, dIncome5 values zero, iMobility values zero and one, dPwgt1 values zero or one, iSex values one and iSubfam1 values zero. The probability syntax "then discrete(1.0, 0.0) else discrete(0.0, 1.0)" means that we only query for the situation q is true or not. The query result of 3.6a is shown in 3.6b.

The result in 3.6b describes that the probability that query q is true is 0.160781, and there is a probability of 0.839219 that query situation q might not occur. Obviously, instead of having a long list of joint distributions that requires us to do more computations for our evaluation process, the joint distribution of specific situations can be directly given by the Dice compilers. This kind of query syntax optimises Dice compiler processing time and the convenience for processing data for further operations when needed.

When dealing with joined tables, due to BayesCard implementation with fanout methods, we need to know the probability of each combination that made up the joint distribution for the evaluation data processing, similar to the result in 3.5c. At this time, although the query syntax is similar to 3.5a, this syntax does not satisfy us when we ask for the joint distribution conditioning on other attributes. Consequently, we combine

the Dice syntax "observe" to indicate the conditions when querying. The example is below (3.1):

Listing 3.1: Dice query third example

```
1 let _ = observe (((title_production_year == int(7,15))||(
    ↪ title_production_year == int(7,16))||(title_production_year
    ↪  == int(7,17))|| ..... ||(title_production_year == int
    ↪ (7,75)))&&(movie_companies_movie_companies_nn == int(1,0))
    ↪ &&(movie_companies_company_type_id == int(2,1))) in
2
3 (title_mul_movie_keyword_movie_id,(title_mul_movie_info_movie_id)
    ↪ )
```

In 3.1, we not only query for the joint distribution of attributes title_mul_movie_keyword _movie_id and title_mul_movie_info_movie_id but also query this joint distribution based on the occurrence of other attributes. The syntax "let _ = observe ..." indicates that the following conditions will be counted in computing the final joint distribution, where syntax "_" represents ignoring the name of this situation as we do not use this name in the query sentence "(title_mul_movie_keyword_movie_id,(title_mul_movie_info_movie _id))".

The above syntaxes are the foremost syntaxes we use for our implementation. We then think about how to combine Python with Dice to do the same evaluation as BayesCard. Since there are limitations about Dice, like no structure and parameters learning implemented, we adapt the results of those learning processes from Pomegranate (Structural learning) and the Pgmpy (Parameters learning) to define syntaxes like 3.5a in Dice. Meanwhile, we also reuse the query decoding from BayesCard and use that information to generate the Dice program by using Python to write .dice files. Then we invoke these Dice programs from the command line by "subprocess.getoutput" and processed the results with Python. In our case, we implement the CardEst. As a result, we need to multiply the probability results from Dice with the number of data we have to predict the number of rows that might be visited by queries. We reuse the same testing datasets for querying on tables Census, DMV and IMDB from BayesCard, where each query uses the symbol "||" to differentiate the query content and true cardinality for this query. Example of the query in Structured Query Language (SQL) is in 4.1c and its true cardinality is 148552.

After getting the predicted CardEst results, in order to maintain consistency with BayesCard, we evaluated our predictions by q-error, i.e.,

$$\textbf{Q-error} = max(\frac{\text{Estimated Cardinality}}{\text{True Cardinality}}, \frac{\text{True Cardinality}}{\text{Estimated Cardinality}}) \tag{3.2}$$

BayesCard and recent research about CardEst have widely used this evaluation metric for evaluation. Although there are other evaluation metrics for CardEst, we believe that q-error is reliable since it has been examined over the years. The more that the value of q-error is closed to 1, the better accuracy results we have.

3.3 Single tables

To bring into correspondence with BayesCard, we reuse the BayesCard training datasets: Census, DMV and IMDB. This section focuses on describing the single tables: Census and DMV. A single table is where all data and distribution relevant to the query are included in one table only. In database querying, sometimes we need complicated operations like the "join" operation to query combined with multiple tables simultaneously as data we would like to query can be stored in different tables, and we have more conditions for querying. The single table does not obtain any "join" operations because all data are stored together as a single table. For querying a single table, we can have all the information we want from that table unless attributes in queries do not exist in the database storage. As a result, we only construct one BN for each single table during model construction and evaluation implementation.

3.4 Joined tables

By starting from doing CardEst on single tables, the CardEst method for joined tables can be developed upon that. Joined tables are tables that have multiple outgoing connections. Referring to Chapter 2, section 2.3.2, we have gone through how BayesCard implements its fanout methods for multi-tables, which are also called joined tables. Referring to 1, by looking at the BN-ensemble construction algorithm, we need to generate different BNs for different joined tables. BayesCard already provides the implementation for discretising the IMDB dataset and categorises possible full outer joined tables. We adapt the generated structures of full outer joined tables from BayesCard, and for each full outer joined table, we construct a single BN for querying. During the evaluation, BayesCard has the query decoding method implemented. The query decoding method analyses the joined table queries into multiple single BN queries and is treated as a dictionary regarding each BN used for the query. By adapting this dictionary to our PPLs evaluation process, we enable to accomplish querying to multi-tables with the "join" operation.

3.5 Other Probabilistic Programming Languages

Despite the Dice, we have evaluated the Infer.NET and SPPL as well. Since both PPLs do not significantly improve performance compared to baseline Pgmpy, in the following section, we will briefly explain how we implement BayesCard CardEst methods in these two PPLs.

3.5.1 Infer.NET

Infer.NET is the probabilistic programming framework proposed by Microsoft in 2008, aiming for machine learning. Infer.NET is developed based on the .NET framework and the C# to run Bayesian inference in graphical models. Since Infer.NET is a .NET framework, we try to find a Python API for convenience. This refers to the IronPython

developed by Microsoft. However, in late 2010, the official development and maintenance of the IronPython were terminated. During implementation, several kinds of compatibility problems occurred among Python, Mono (a platform that helps .NET framework run on Python interface) and IronPython. In order to maintain reliable results, we directly use the Csharp (C#) to develop the BayesCard CardEst methods.

Infer.NET has three inference algorithms: Expectation Propagation [23], Variational Message Passing [35] and Gibbs sampling [16]. To construct BNs in Infer.NET, for each attribute, the syntax of Infer.NET requires us to define global primary random variables for the observed occurrence record of this attribute, prior probability, posterior probability, and the learnt BN parameters as an N-D array of variable length. The dimensions of the array N depend on the number of values that an attribute can have, and the type of value stored in the array is determined by what the variable stores. An example of the Infer.NET random variable definition is below:

Listing 3.2: An example of Infer.NET random variable definition

```
1 ......
2 public Variable<int> TotalNumberOfExamples;
3 public InferenceEngine Engine = new InferenceEngine();
4
5 public VariableArray<int> dAge;
6 public Variable<Vector> ProbdAge;
7 public Variable<Dirichlet> ProbdAgePrior;
8 public Dirichlet ProbdAgePosterior;
9 ......
```

Other than that, we need to specify our BN model. The Infer.NET syntax requires us to define our inference engine with an inference algorithm, and the BN links similarly to the primary variables. For example, to construct BN for the Census dataset, we need to define 4 * 68 = 272 variables to specify the BN parameters, excluding defining the BN links, which leads to duplication of defining the syntax and the consumption of time. As a result, we decide to automatically generate the C# code for constructing BNs dependent on the dataset with Python. Because Infer.NET does not provide structural learning, we adapt the BN structure learnt by Pomegranate from BayesCard to automate C# code with Python. Since Infer.NET requires defining BN attributes in order, we solve this with the anytree package in Python and generate the BN model definition in C# based on the Chow-Liu tree depth. We implement a method for parameters learning by reading the training dataset and using the inference engine to infer the data distribution of attributes in BN.

Since the automated generation of BN construction obtains thousands of lines of C# code, for better visualisation and debugging, we separate the BN model definition and the query evaluation into two C# files (figure 3.1). As BN construction, we also automate generating the Infer.NET query process with Python. After reading the training dataset and training our BN model, we adapt the query decoding results from BayesCard stored as a JSON file and read them into C#. We support the BayesCard

CardEst method [36] for point queries of the single table benchmark, in which each attribute is queried with one value only. However, we do not successfully support the CardEst method with ranged queries in an efficient way. Ranged queries are queries in which each attribute can be conditioned on a list of values. We have tried to implement the variable elimination and the cartesian product methods to realise this solution (The result of this solution is explained in section 4.3.1.1).

3.5.2 SPPL

Sum-Product Probabilistic Language (SPPL) [29] is a probabilistic programming language infer with sum-product expressions, proposed in 2021. Compared to Dice which only provide exact inference with discrete BNs, SPPL supports exact solutions to graphical models with discrete, continuous and mix-typed data distributions. SPPL provides multi-stage workflow (Figure 7(a) in [29]). In our implementation, similar to Infer.NET and Dice, we automate to generate SPPL program of BN definition with Python with anytree package. For query evaluation, we invoke the SPPL compiler provided from SPPL Python API to compile and query our BN model. As mentioned before, SPPL does not provide structure and parameters learning. We again adapt the BN structure learnt by Pomegranate and the data distributions learnt by Pgmpy from BayesCard to define our BN models in SPPL. An instance of SPPL BN definition is in 3.7a.

For each attribute, we defines its data distribution as 3.7a. In the example 3.7a, the attribute dOccup has two children: dIncome3 and dIndustry. The syntax "~= choice("0": ..., "1": ..., ...)" defines the data distribution of a specific attribute in terms of attribute value as a string following semicolon and the probability in float. For example, dIncome3 has the probability of around 1.0 being zero, when dOccup values zero.

We used the SPPL compiler to translate and construct our BN for later usage and Python API to evaluate queries. Different from Dice (Dice has both BN definition and query in .dice file), for each query, we generate a python file including defining the above SPPL program 3.7a and compiled the corresponding BN to query our SPPL BN model with the SPPL Python API. For instance, a query evaluation is executed as 3.7b.

In the 3.7b, with SPPL python API, we define our SPPL compiler to compile our BN construction called source and refer our variables namespace and our BN to variables "namespace" and "model" for querying. The SPPL syntax "X.Y.prob(X.attribute « "1","2",.....)" helps us query our BN model, where X is the name of variables namespace and Y is our BN name. The symbol "«" can be seemed as equivalence symbol. In the 3.7b, we query the joint probability when attribute dOccup values zero to two and four to six, and so on.

```
1  ......
2  if (dOccup == "0"):
3      dIncome3 ~= choice({"0":0.9999620006459891,"1"
           ↪ :3.799935401098181e-05})
4      dIndustry ~= choice({"0":1.0,"1":0.0,"2":0.0,"3":0.0,"4"
           ↪ :0.0,"5":0.0,"6":0.0,"7":0.0,"8":0.0,"9":0.0,"10"
           ↪ :0.0,"11":0.0,"12":0.0})
5  elif (dOccup == "1"):
6      dIncome3 ~= choice({"0":0.993047758585807,"1"
           ↪ :0.006952241414193067})
7  ......
```

(a) BN definition in SPPL

```
1  ......
2  from sppl.compilers.sppl_to_python import SPPL_Compiler
3  compiler = SPPL_Compiler(source)
4  namespace = compiler.execute_module()
5  model = namespace.model
6  prob = namespace.model.prob(namespace.dOccup << {"4","1","6","
       ↪ 0","2","5"} & namespace.dPwgt1 << {"0"} & namespace.
       ↪ iRelat1 << {"0","2","1"} & namespace.iYearwrk << {"0","
       ↪ 1","6","3","7"})
7  predict = round(p * 2458285) # Census table has 2,458,285 rows
```

(b) The corresponding query evaluation of the example 3.7a

Figure 3.7: An example of SPPL BN definition and evaluation of Census dataset

Chapter 4

Experiment

In this chapter, we will go through experiments taken to support our findings.

4.1 Experimental setups

In this section, we mainly focus on how to set up experiments about Dice, and briefly explain how to build Infer.NET and SPPL. The installation about how to establish BayesCard with Pgmpy refers to [36].

4.1.1 Datasets and query workloads

Our experiments are performed with the following three datasets: single table experiments on Census and DMV and multi-table experiments on IMDB.

4.1.1.1 Census

The Census dataset is part of US census survey results collected in 1990 and donated by Microsoft members. This dataset has 2,458,285 instances and 68 attributes, including multivariate data distribution. By experiments, BayesCard finds that the data in Census dataset are highly correlated. As this dataset is large in scale and obtains considerable data distribution complexity, it can better examine the inference performance of PPLs.

4.1.1.2 DMV

The DMV dataset contains the real-world registration information of the vehicle, snowmobile and boat in New York State (NYS). We reuse the same attributes as BayesCard [36]. However, we use the latest DMV instead of the same snapshot as BayesCard. Compared to the DMV snapshot that BayesCard uses, the data distribution of our DMV dataset has some shift, and there are out-of-order 794,873 instances added to the original dataset. Considering we reuse the BayesCard DMV benchmark, we use the first 11,575,483 tuples of our DMV dataset to ensure we will get similar CardEst q-error results as the BayesCard for successful evaluation, maintaining the consistency with the number of instances that BayesCard DMV has used.

4.1.1.3 IMDB

The Internet Movie Database (IMDB) dataset is a multi-table dataset. Based on IMDB, JOB (Join Order Benchmark) is published for research purposes as a standard testing dataset for researching the database field, query optimizer and is first used by the original paper [19]. Prior work [19] states that the IMDB has considerably complicated data distribution. We evaluate our Dice CardEst programs with the first 65 queries in the Job-light benchmark, which obtains six tables: title, cast_info, movie_info, movie_companies, movie_keyword and movie_info_idx, where only the primary table "title" can be joined by other five tables.

4.1.2 Experimental environment

For Dice and SPPL, all models are evaluated with 2.6 GHz 6-Core Intel Core i7 CPU, 16 GB 2667 MHz DDR4 main memory. We have also built Dice and SPPL successfully on Ubuntu 20.04 and Ubuntu 18.04 on Virtual Machine (VM) Software UTM, respectively. However, considering the evaluation efficiency with VM, we choose to build Dice on Mac. The installation information of Dice and SPPL refer to the official Github repositories. However, for the official version of Dice (Ubuntu 20.04), the installation on Mac OS cannot be successfully built from the source. This is because the installation guide generates file "../rsdd/target/release/librsdd.dylib" on Mac and the same file with suffix ".so" on Ubuntu, which makes the installation process cannot find "librsdd.so" needed. To solve this problem and successfully build Dice on Mac, when installing on Mac, add the instruction at line 3 between lines 2-4 as follows:

Listing 4.1: Dice installation setups

```
1 ......
2 (chdir ../rsdd (run cargo build --release))
3 (copy ../rsdd/target/release/librsdd.dylib ../rsdd/target/release
    ↪ /librsdd.so)
4 (copy ../rsdd/target/release/librsdd.so dllrsdd.so)
5 ......
```

For Infer.NET, experiments are built with Visual Studio 2019 for Mac, which is the natural platform for running C# and .NET framework officially developed by Microsoft. All .NET packages using in our implementation can be installed and referenced by Visual Studio directly.

4.2 Baseline model

Our baseline BayesCard was recent research in 2021. There are several reasons why we consider BayesCard as our baseline:

1. By reviewing state-of-art, BayesCard almost has the overall optimal performance on CardEst, while other methods could be better in specific tasks.

2. When considering our graphical model to be BNs, in the existing research about CardEst, BayesCard is the only open-sourced CardEst method building with the Bayesian Network structure.
3. BayesCard implementation is naturally built with Python, and packages that BayesCard uses have developed for years, which are stable.

4.2.1 Pgmpy

Our baseline model, BayesCard, is implemented with Pgmpy [1], by learning parameters with the maximum likelihood estimator from Pgmpy, which maximums the likelihood of the parameters of the graphical model based on the observation [1] [10]. For the inference in BNs, Pgmpy supports several inference algorithms such as Variable Elimination (VE) [3], Belief Propagation [39] and Casual Inference [11]. BayesCard mainly focuses on using VE with the elimination ordering method from Pgmpy, which helps find the order of attributes for efficient VE.

In addition, one of the main achievements that BayesCard holds is that it supports the JIT version of VE. JIT compilation translates the python program into linear algebra for faster read speed in binary, and JIT improves the efficiency of the compilation by remembering the VE orders of the previous query and reusing them for the next query during the evaluation [36].

4.3 Evaluation

```
1  SELECT COUNT(*) FROM climate WHERE iKorean = 0 AND dPoverty =
       ↪ 2||2043794
```

(a) Benchmarking query from Census

```
1  SELECT COUNT(*) FROM DMV WHERE Registration_Class IN [PAS, COM
       ↪ , LTR, BOT, MOT, TRL] AND Fuel_Type IN [GAS, NONE
       ↪ ]||10033557
```

(b) Benchmarking query from DMV

```
1  SELECT COUNT(*) FROM movie_companies mc,title t,movie_keyword
       ↪ mk WHERE t.id=mc.movie_id AND t.id=mk.movie_id AND mk.
       ↪ keyword_id=117||148552
```

(c) Benchmarking query from IMDB

Figure 4.1: Examples of queries from our benchmarks

In this section, we will explore our observations and analyse the results of our experiments. In the previous section 4.1.1, we describe the datasets used for model training.

For all three datasets benchmarking, we reuse the benchmarks from BayesCard. For Census, we evaluate our CardEst implementation in PPLs with 468 queries in total, combining 55 point queries and 413 ranged queries. DMV benchmark supports 1965 queries which cannot be directly defined as the point or ranged queries for evaluation. As previously mentioned in 4.1.1.3, we reuse the IMDB benchmark job-light from [19] as BayesCard, which includes 70 queries in total. During the evaluation, there are five queries in the job-light that Dice cannot recognise. As a result, we exclude these queries and finally maintain to evaluate with 65 queries. Example queries from Census, DMV and IMDB can refer to 4.1.

4.3.1 Experiments Results

This section analyses our experiment results presented in tables and discusses why we observe such phenomenons. Each table in this section mainly focuses on q-error and the averaged latency each query takes during the evaluation. All results presented below are averaged over five runs.

4.3.1.1 Census

Tables 4.1 and 4.2 present the experiment results averaged over five runs on the Mac OS X and Ubuntu 20.04, respectively. For each experiment of Dice and SPPL, we first build our experiments on Ubuntu, and if the results are ideal, we move our build to Mac OS X to further improve the performance of our programs.

Table 4.1: Ablation study of different inference algorithms of BayesCard and Dice on CENSUS (run on Mac OS X).

	BayesCard		Dice (*master*)		Dice (*oopsla-artifact*)	
	VE+GR	VE+GR+JIT	No GR	GR	No GR	GR
95% q-error	2.04	2.05	2.04	2.05	2.04	2.04
Latency (ms)	390	**2.6**	1624.6	705.8	88.1	46.1

Table 4.1 compares the results between BayesCard, the Dice compiler from the Github *master* branch, and the native Dice compiler from the Github *oopsla-artifact* branch. In table 4.1, there is not much difference in accuracy among BayesCard and two Dice compilers, while the performance in terms of latency varies. Chapter 3 mentions that we implement graph reduction (GR). For the Census dataset, the averaged compilation latency of both Dice compilers with GR is around half less than the implementation without GR. Especially, the native Dice compiler (*oopsla-artifact*) with GR can achieve approximately the same accuracy as BayesCard results, while the averaged compilation latency (46.1ms) is about eight times less than that of the BayesCard VE+GR (390ms). Although Dice still cannot outperform the BayesCard VE+GR+JIT (2.6ms), the result given by Dice *oopsla-artifact* with GR is impressive.

Table 4.2: Ablation study of different inference algorithms of BayesCard and Dice and SPPL on Census (run on Ubuntu 20.04).

	BayesCard		Dice (*oopsla-artifact*)		SPPL
	VE+GR	VE+GR+JIT	No GR	GR	No GR
95% q-error	2.06	2.05	2.05	2.05	2.06
Latency (ms)	461	**3.0**	106.5	51.4	11275.1 ≈ 11.3 s

In Table 4.2, we examine the results of the experiments among BayesCard VE+GR and VE+GR+JIT, Dice *oopsla-artifact* and the SPPL on Ubuntu 20.04. The is not much difference in Dice results unless the performance latencies are higher than the results in table 4.1. To run on Ubuntu, we use the virtual machine (VM) software UTM, which leads to losses in our machine and results in higher averaged compilation latency. Table 4.2 includes the performance of SPPL. However, although SPPL has almost the same accuracy as BayesCard and Dice, the average compilation latency of SPPL is terrible, at around 11.3 seconds (shown in 4.2). By previous observation, the GR implementation will reduce the averaged compilation time by half less, but even for each query of SPPL takes around 5 seconds, this result is still bad. Therefore, we did not continue further experiments with SPPL, including moving to Mac OS X and supporting GR and datasets DMV and IMDB.

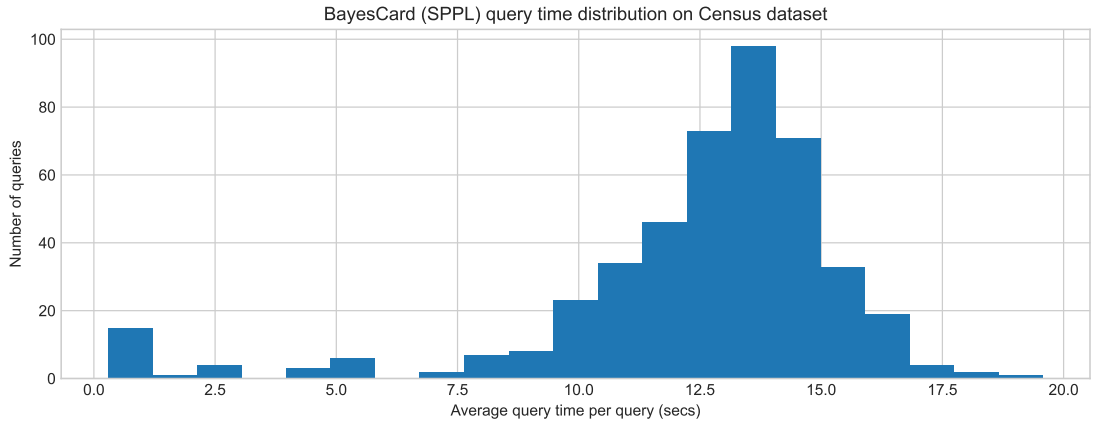


Figure 4.2: BayesCard(SPPL) query time distribution on Census

As mentioned before in Chapter 3, section 3.5.1, we try to support the evaluation implementation to ranged queries of Infer.NET with the Census dataset. Nevertheless, at the same time, during the implementation, the efficiency of the current manipulation is terrible. The evaluation result to point queries of each inference algorithm is between 86 and 117 seconds, while BayesCard VE results in 134ms. Considering that the performance of Infer.NET with point queries of the single table is not ideal, we stop our implementation of Infer.NET and reschedule our plan to move to the subsequent PPL implementation, Dice.

4.3.1.2 DMV

We present our experimental results among BayesCard and Dice of the DMV dataset in table 4.3.

Table 4.3: Ablation study of different inference algorithms of BayesCard and Dice on DMV (run on Mac OS X).

	BayesCard		Dice (*master*)		Dice (*oopsla-artifact*)	
	VE+GR	VE+GR+JIT	No GR	GR	No GR	GR
95% q-error	1.38	1.39	1.47	1.47	1.47	1.47
Latency (ms)	223.7	**1.77**	3140	1731	726.1	342.0

Compared to Census dataset, the performance of both versions of Dice is not as good as the BayesCard. In terms of accuracy, there is no difference between the two versions of Dice. However, both Dice compilers result in a q-error of 1.47 while BayesCard results are around 1.38. We have not found a reasonable explanation for this observation and we need to experiment with more datasets like DMV to support our guess. We think that this can be caused by DMV BN structure which can have affect the inference in Dice.

Being focus on the average compilation latency only, the influence of GR in performance is consistent with the results in tables 4.1 and 4.2. However, for DMV evaluation, none of our results outperform any BayesCard results. In future, we seek solutions to improve the performance of Dice.

4.3.1.3 IMDB

Table 4.4 shows the experiment results of the IMDB dataset running on Mac OS X.

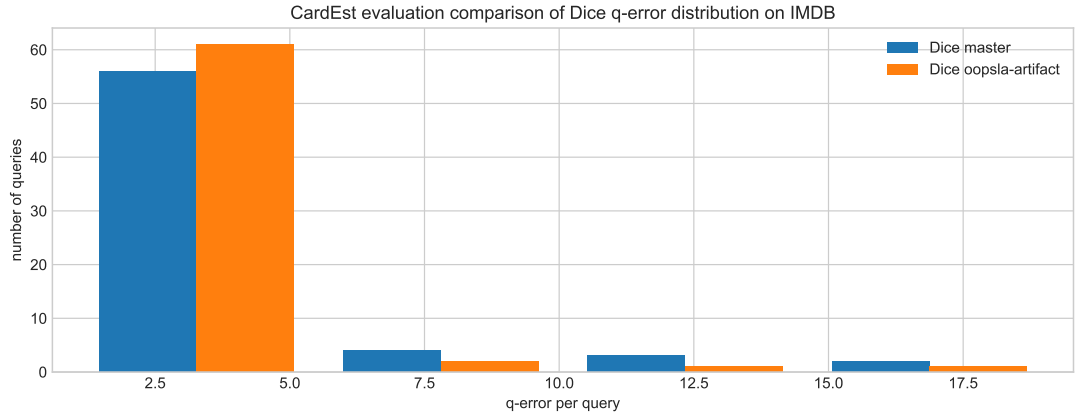
Table 4.4: Ablation study of different inference algorithms of BayesCard and Dice on IMDB (run on Mac OS X).

	BayesCard		Dice (*master*)		Dice (*oopsla-artifact*)	
	VE+GR	VE+GR+JIT	No GR	GR	No GR	GR
95% q-error	4.98	4.90	13.5	13.5	6.60	6.60
Latency (ms)	225.4	**6.30**	1752268	1625360	64562	71334

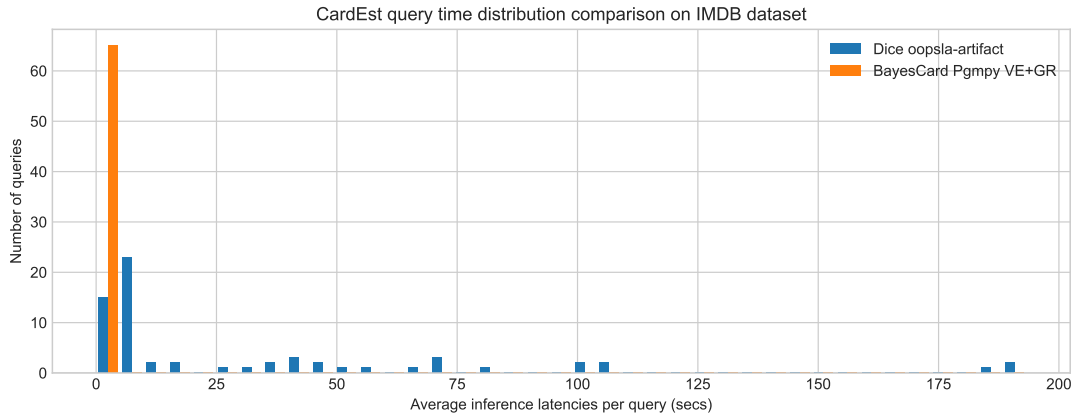
It is evident that Dice results are all worse than BayesCard results. We show the comparison of average query time distribution between BayesCard Pgmpy VE+GR and Dice *oopsla-artifact* in 4.3b. Especially for Dice *master*, the q-error is apparently higher than both BayesCard and Dice *oopsla-artifact* results. We have broken up the query evaluation experiment and analysed this observation stepwise. We find that, for some queries, Dice *master* has better q-error results, while for some queries, *oopsla-artifact* has better q-error results. Compared to Dice *master*, Dice *oopsla-artifact* obtains more queries with q-errors lower than 2.0, shown in 4.3a.

For some queries inference by Dice *oopsla-artifact*, the probability of some attributes

could be tiny, approximating zero. Dice *oopsla-artifact* then generalise this kind of results to zero. Until now, we cannot adequately explain why Dice *master* results are so terrible. More multi-table evaluation experiments should be undertaken to explore the trade-off between accuracy and inference latency for both Dice compilers. Also, for average inference latency, the results of Dice *master* are underestimated. We guess that this could be caused by the heat up of the system due to running several experiments continuously over time, thus resulting in such unacceptable results. In table 4.4, it seems that the GR does not fasten the inference compilation process. This is because of the IMDB BNs structures and the corresponding benchmark. It looks like that queries visit every attribute of IMDB BNs During the evaluation. As a result, GR does not have influence in table 4.4.



(a) CardEst evaluation comparison of Dice q-error distribution on IMDB dataset.



(b) CardEst query time distribution comparison between Dice *oopsla-artifact* and BayesCard Pgmpy on IMDB.

Figure 4.3: CardEst evaluation results distribution on IMDB

4.4 Productivity

This section describes our productivity during the project. Table 4.5 demonstrates our productivity events by timeline. We first reproduce the BayesCard results, and based on those results, we try to support the Infer.NET with Census dataset. However, since Infer.NET seems to have no proper support for Bayesian Networks (details in section 4.3.1.1), we move to support Dice and SPPL with Census dataset. Nevertheless, SPPL performance is not ideal yet. Consequently, we decide to integrate our support with Dice by implementing with single dataset DMV and multi-table dataset IMDB in Dice.

System	Datasets	Coding effort
Reproduce BayesCard	all datasets	1 week
Infer.NET	Census	2 months
Dice	Census	1 week
SPPL	Census	1 week
Dice	DMV	1 week
Dice	IMDB	2 weeks

Table 4.5: Productivity of our project

As database (DB) components like CardEst, the results of those DB components are dependent on the data distribution. Using PPLs and generating PPLs programs, DB developers do not need to implement probability inference from scratch since PPLs handle such problems automatically. DB developers can rely on the infrastructure provided by PPLs and directly use the inference results from PPLs.

Chapter 5

Conclusions

5.1 Project Contributions

This project contributes to four main parts:

1. We reproduce the BayesCard results reported in the [36].
2. We implement the PPLs program generator to automate our graphical models definition and query in PPLs
3. We construct BNs based on three different datset to support the Cardinality Estimation, obtaining three different BNs in total.
4. We evaluate our BNs performances implemented in four PPLs built with diversified data distributions in terms of accuracy and latencies.
5. Our finding suggests that the native Dice compiler of *oopsla-artifact* version with graph reduction (GR) outperforms the BayesCard VE+GR, evaluating with Census dataset.

5.2 Results overview

In this project, we have examined the performance of four PPLs (Pgmpy, Infer.NET, Dice and SPPL) in the database component, Cardinality Estimation, with the current best state-of-art CardEst approach, BayesCard. Our project idea is to explore the inference performance of PPLs on different types of Bayesian networks in terms of inference time in milliseconds and the accuracy measured in Q-error. We evaluate whether other PPLs (e.g., Dice, SPPL and Infer.NET) will be more adaptive or not to the BayesCard inference process in terms of inference time and accuracy compared to baseline implementation Pgmpy.

Overall we find that neither Infer.NET nor the SPPL results outperform the BayesCard results. Our best CardEst performance is inference by Dice *oopsla-artifact* with GR on Census dataset, which achieves around eight times less than BayesCard VE+GR

in terms of average inference latency. Looking at all performances of Dice with three datasets (Census, DMV and IMDB), although the inference latencies of Dice with DMV and IMDB are higher than that of BayesCard, Dice achieves to obtain similar accuracy as BayesCard always. Since Dice is recently proposed, the development of Dice is still ongoing. We keep in touch with Dice authors, and they plan to support a python API to Dice, which could vastly improve the efficiency of calling Dice with Python.

Chapter 6

Future Plan

During this project, we have reviewed the literature for both CardEst approach and the Approximate Query Processing (AQP) (but AQP is not presented in this dissertation because it is not the main implementation at present). Due to time limitations, we plan to support the AQP functionality with Dice as an extension of the existing CardEst implementations as the next-year project.

We have reviewed several AQP state-of-art approaches as the following:

Publication Year	Paper Title	Approach
2017	Revisiting Reuse for Approximate Query Processing [8]	Bayes theorem
2018	VerdictDB [25]	Sampling
2019	**DeepDB** [13]	RSPN
2020	EntropyDB [24] Deep Generative Models [34] ML-AQP [30]	Multi-linear polynomial Variational Auto-encoders Sampling

Table 6.1: Literature review of AQP

By literature review, we intend to implement our next-year work based on [14] and this project. Despite that, we are seeking solutions to improve the current Dice performance, especially on IMDB dataset, which is much worse than expected compared to results with Census and DMV dataset.

Bibliography

[1] Ankur Ankan and Abinash Panda. pgmpy: Probabilistic graphical models using python. In *Proceedings of the 14th python in science conference (scipy 2015)*, pages 6–11. Citeseer, 2015.

[2] 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.

[3] Mark Chavira and Adnan Darwiche. Compiling bayesian networks using variable elimination. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2443–2449, 2007.

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

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

[6] Corporation Oracle. Mysql 8.0 reference manual. `https://dev.mysql.com/doc/refman/8.0/en/performance-schema-statement-digests.html`, 2020.

[7] 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.

[8] 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.

[9] 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.

[10] Daniel Grossman and Pedro M. Domingos. Learning bayesian network classifiers by maximizing conditional likelihood. In Carla E. Brodley, editor, *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*, volume 69 of *ACM International Conference Proceeding Series*. ACM, 2004.

[11] David Heckerman. A bayesian approach to learning causal networks. In Philippe Besnard and Steve Hanks, editors, *UAI '95: Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence, Montreal, Quebec, Canada, August 18-20, 1995*, pages 285–295. Morgan Kaufmann, 1995.

[12] David Heckerman. A tutorial on learning with bayesian networks. *CoRR*, abs/2002.00269, 2020.

[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] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: Learn from data, not from queries! *Proc. VLDB Endow.*, 13(7):992–1005, 2020.

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

[16] Tomas Hrycej. Gibbs sampling in bayesian networks. *Artif. Intell.*, 46(3):351–363, 1990.

[17] 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.

[18] 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.

[19] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[20] David Lopez-Paz, Philipp Hennig, and Bernhard Schölkopf. The randomized dependence coefficient. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

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

[22] 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.

[23] Thomas P. Minka. Expectation propagation for approximate bayesian inference. In Jack S. Breese and Daphne Koller, editors, *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, University of Washington, Seattle, Washington, USA, August 2-5, 2001*, pages 362–369. Morgan Kaufmann, 2001.

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

[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] Johannes Gehrke Raghu Ramakrishnan. *Database Management Systems*. McGraw-hill Companies, Inc, third edition, 2003.

[28] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. SPPL: probabilistic programming with fast exact symbolic inference. In *PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Design and Implementation*, pages 804–819, New York, NY, USA, 2021. ACM.

[29] 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.

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

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

[32] The PostgreSQL Global Development Group. Postgresql 10.3 documentation. `https://www.postgresql.fastware.com/hubfs/_Global/Manuals/V10-PostgreSQL10-3Documentation.pdf?hsLang=en-us`, 2018.

[33] 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.

[34] 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.

[35] John M. Winn and Christopher M. Bishop. Variational message passing. *J. Mach. Learn. Res.*, 6:661–694, 2005.

[36] 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.

[37] 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.

[38] 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.

[39] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Generalized belief propagation. In Todd K. Leen, Thomas G. Dietterich, and Volker Tresp, editors, *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA*, pages 689–695. MIT Press, 2000.

[40] 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. *CoRR*, abs/2011.09022, 2020.