THÈSE

En vue de l'obtention du
DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Université Toulouse 3 - Paul Sabatier

Présentée et soutenue par

Max HALFORD

Le 28 septembre 2020

Apprentissage statistique pour l'estimation de sélectivité en bases de données relationnelles

---

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse**

Spécialité : **Mathématiques et Applications**

Unité de recherche :
IMT : Institut de Mathématiques de Toulouse

Thèse dirigée par
Franck MORVAN et Philippe SAINT PIERRE

Jury

Mme Chirine GHEDIRA-GUEGAN, Rapporteure
M. Aurélien GARIVIER, Rapporteur
M. Nicolas BOUSQUET, Examinateur
Mme Lynda TAMINE-LECHANI, Examinatrice
M. Franck MORVAN, Directeur de thèse
M. Philippe SAINT-PIERRE, Co-directeur de thèse

Summary

Databases, and in particular relational databases, are a common paradigm for storing and querying data. Data is stored in relations which are linked with each other. Users may interrogate the database by issuing queries via a declarative query language. One of the challenges for a relational database management system (RDBMS) is to evaluate each query as quickly possible. However, as it turns out, there are many ways in which a query can be executed. The RDBMS delegates the choice of the query execution plan to a query optimiser. In turn, the query optimiser tasks a cost model with the responsibility of estimating the cost of each execution plan considered by the optimiser. The cost estimation therefore has a large influence on the quality of the execution plans, which in turns affects the running time that the user perceives.

The most important input in the cost model is the selectivity, which is the amount of data that is expected to flow in and out of each operator in a plan. From a statistical point of view, this is a case of density estimation, where the goal is to determine the amount of data that verifies certain conditions. It is common practice to use density estimation models that make strong simplifying assumptions, in large part for efficiency reasons. For instance, it is usually assumed that attributes are independent with each other. Alas, in practice, these assumptions are more often that not unverified, which leads to large estimation errors. The goal of this PhD is thus to propose models that offer a better compromise between selectivity estimation accuracy and computational cost. First of all, we propose a methodology based on Bayesian networks, where we constrain each network to have a tree topology, therefore allowing a linear computational complexity. Secondly, we train online machine learning models to correct an existing selectivity estimation model, whilst adapting to concept drift. Our experimental results, which are based on the TPC-DS and JOB benchmarks, are very encouraging with respect to both of our lines of work.

Résumé

Les bases de données relationnelles sont couramment utilisées pour stocker et interroger des données. Les données y sont stockées dans des relations qui sont liées entre elles. Un utilisateur interroge ces données via des requêtes exprimées dans un langage déclaratif. Un des défis pour le Système de Gestion de Bases de Données Relationnelles (SGBDR) est d'évaluer les requêtes le plus rapidement possible. Pour cette évaluation, il existe un large choix de plans d'exécution associé à une requête donnée. Le SGBDR délègue le choix du plan d'exécution le plus efficace à un optimiseur. Dans ce processus, le modèle de coûts, quand à lui, a la responsabilité d'estimer le temps de calcul de chaque plan d'exécution considéré par l'optimiseur. Par conséquent, le modèle de coûts influe sur la qualité des plans d'exécution produit par l'optimiseur, et donc sur le temps de réponse perçu par l'utilisateur.

Le paramètre le plus influant dans le modèle de coûts est la sélectivité, qui permet d'estimer la quantité de données qui transite entre chaque opérateur d'un plan d'exécution. D'un point de vue statistique, ceci correspond à de l'estimation de densité, qui consiste à déterminer la quantité de données qui vérifie un ensemble de conditions. Il est d'usage courant d'utiliser des modèles avec des hypothèses simplificatrices pour des raisons de performances. Par exemple, il est souvent supposés qu'il n'existe pas de dépendances de valeurs entre différents attributs de relations. Dans la majeure partie des applications cette hypothèse se révèle non vérifiée et conduit à de grandes erreurs d'estimation. Le but de cette thèse est donc de proposer des modèles plus réalistes qui offrent un meilleur compromis entre la précision des estimations et la complexité calculatoire requise. Premièrement, nous proposons une méthode s'appuyant sur les réseaux Bayésiens dont la représentation sous forme d'arbre nous permet d'améliorer la précision des estimations avec une complexité linéaire. Par la suite, nous explorons l'usage de modèles d'apprentissage en ligne pour corriger un modèle d'estimation existant tout en s'adaptant à l'évolution des données. Nos

résultats expérimentaux basés sur les bechmaks JOB et TPC-DS montrent des résultats très encourageants.

Contents

List of Figures

List of Tables

Acknowledgements

I wish to thank each living being, whom over the past three years, made my life pleasant and interesting. They know who they are.

Sometimes it's good to do
what you're supposed to do
when you're supposed to do it.

FRANCES HA

Chapter 1

Introduction

1.1 Problem statement

In a relational database management system (RDBMS) – which we equivalently refer to as "database" in this manuscript – data is scattered across relations that are themselves linked with each other. From a user's point of view, querying a RDBMS is easy. Indeed, queries to the latter can be expressed via interrogation languages, such as *structured query language* (SQL), or via *object-relational mapping* (ORM) frameworks. Users express *what* they want, whilst the RDBMS is in charge of determining *how* to retrieve what the users want. In this sense, SQL is a declarative language rather than a procedural language. This abstraction allows users and applications to delegate the gritty details to the RDBMS. However, as is often the case in computer science, abstractions come at a cost. In this case, by only telling the RDBMS what she wants, a user is expecting the RDBMS to figure out the details of how to perform the query. In fact, relational databases usually have a dedicated module to deal with this: the *query optimiser*.

The query optimiser is in charge of translating a *logical execution plan* into an optimised *physical execution plan*. Typically, the logical execution plan is produced by a query parser, which is an earlier step of the query execution pipeline. The query optimiser's output is commonly referred to as a *query execution plan* (QEP). A QEP is usually represented as an inverted tree. The leaves of the tree represent the relations that are part of the query, whilst the root is the final output. Each node of the tree represents an algebraic operator, such as a filter or a join. For any given user query, there exists many possible QEPs. Once executed, each of these QEPs will produce the same result. This stems from the relational

model and the mathematical properties of the operators used in a QEP. For instance, join operators are commutative. That is, joining a relation A with a relation B will produce the same result as joining B with A. Likewise, filtering a relation is transitive with respect to the join operator. In other words, the same result will be obtained by filtering a relation before a join or filtering the output of the join.

Although there are many QEPs that may answer a user query, not all of them have the same cost. In fact, they can vary wildly. The goal of the query optimiser is thus to pick the QEP with the least *expected* cost. However, the only way to know for certain the cost of a QEP is to execute it and measure it's running time. Alas, doing so over a set of candidates execution plans defeats the purpose of selecting one in the first place. Therefore, the query optimiser has to *estimate* the cost of a QEP without executing it. This is a difficult task in itself, and is the responsibility of the *cost model*. The cost model estimates the running time of a QEP through an equation which attempts to model the steps taken by said QEP. This equation may vary from one RDBMS implementation to another, but in general it always captures the same idea. It is essentially a sum of the costs of each of it's operator. The cost of an operator is in turn a function of the number of tuples that it needs to process. In the case of a distributed RDBMS, the equation also incorporates some from of communication cost. When multiple processors are available, the cost of moving objects back and forth has to be considered as well. There are also some parameters that depend on the specific hardware that the RDBMS is being run on – for instance, the CPU speed. Finally, the query optimiser has to consider system specifications, such as memory page sizes and the amount of available rapid access memory.

The cost of each operator in a QEP is directly tied to the number of tuples that will transit through it. For example, the cost of a `WHERE` statement on a relation scales linearly with the length of said relation – which is also referred to as it's *cardinality*. It is thus crucial to accurately estimate the amount of tuples that will flow through a particular operator in a given QEP. In fact, these amounts are mostly sufficient in order to perform a successful query optimisation. However, the issue is again that we do not have the luxury to execute parts of the QEP in order to determine these quantities. Instead, we can use the statistical properties of the attribute values to estimate these quantities. For instance, consider a join between a `cars` relation and a `passages` relation. The cost of the join between both relations is a function of the cardinality of both relations – the details of said

function depend on the join algorithm that is being used, here we are instead interested in the general idea. The cardinality of each relation can assumed to be known as it is simply the number of tuples that each relation contains. However, consider the case where we only want to consider electric cars. Without explicitly filtering and counting the number of electric cars, it is extremely difficult to determine the cost of the join. A potential solution is to store statistics which summarise a relation's attributes. For instance, the percentage of electric cars can be precomputed and stored aside prior to the query optimisation phase. This leads to a flurry of questions, such as how to update statistics when new data is inserted, or how to deal with multiple statements. In general, the fraction of tuples that flow through part of a QEP is called the *selectivity*. The selectivity can be seen as the percentage of input tuples that satisfy a query statement, the latter being implemented with a physical operator.

There is little doubt that selectivity is by far the most important input to the cost model [Gu et al., 2012, Lohman, 2014, Leis et al., 2018]. *Selectivity estimation*, which is the task of estimating the selectivity of a sequence of operators, is thus the most important task a cost model has to perform. It is also a difficult task. It is in fact an entire research topic by itself, even though it is a sub-task of query optimisation, which is itself a sub-field of database research. Fundamentally, the goal of selectivity estimation is to determine the amount of tuples that satisfy a query statement. In other words, it boils down to determining the amount of tuples that flow out from each operator in the QEP. From a statistician's perspective, this is nothing more than a case of *density estimation*. In this regard, the goal is to estimate the underlying *probability density function* of the database's attribute values. This is a well-known statistical problem, which may be attested by the many honed solutions that are available. However, there are a few specific points that make selectivity estimation a more difficult beast to tame. First of all, density estimation algorithms are traditionally tailored to work with a single entity, for which the equivalent in our case is a relation. More often than not, we are mostly interested in estimating selectivities over multiple relations, a problem towards which much less work has been poured into. Additionally, a relation's attributes can be both discrete and continuous, which goes against the settings of many density estimation algorithms. Meanwhile, attributes are regularly updated with new values, as a result of new information flowing towards the database. Therefore, it is important to develop methods that are able to efficiently detect changes and adapt when

new data is inserted into the database. Finally, and probably most importantly, selectivity estimation is a critical task that needs to be performed as quickly as possible. Indeed, the total query time, as perceived by the user, is the sum of the query optimisation and the query execution time. Therefore, finding a good QEP is useless if it takes longer than running a less efficient QEP that takes less time to find. The selectivity estimation module, which is part of cost model, is called many times for each candidate QEP considered by the query optimiser, and consequently has to be extremely efficient.

Selectivity estimation has traditionally been performed with simplistic methods. The reason why is because keeping track of the distribution of each attribute's values can quickly become an unwieldy task. Indeed, the naive way to go about selectivity estimation is to store the frequency of the values of each attribute, as well as for each combination of said values. For a single relation, with 10 attributes each with an average cardinality of 50, this would require storing 119,042,423,827,613,000 values, which is typically more than the total number of values in the relation. Therefore, simplifying assumptions have been proposed to offer alternative methods. For instance, the System R database used in the seminal Selinger paper [Selinger et al., 1979] assumed a default selectivity of 10% for each attribute condition. In other words it assumed that every `WHERE` statement reduces the relation's cardinality by 90%, regardless of the particular attribute and value that are used. This is based on the assumption that all the values of an attribute occur at the same frequency, which is obviously wrong. Thankfully, this assumption is easy to relax. Indeed, the distribution of a numerical attribute can be approximated with a histogram. In case of a categorical attribute, a common solution is to store the exact frequencies of the most common values, and assume that the rest of the values appear at the same frequency. A lot of methods have since been proposed to improve the accuracy of estimating selectivities for predicates that involve a single attribute. However, the crux of the selectivity estimation problem lies in estimating selectivities where multiple attributes are involved, often across relations.

Estimating selectivities when multiple attributes are involved is difficult because there are many possible combinations of attributes. This is a prime example of the well-known *curse of dimensionality* [Bellman, 2015]. Nonetheless, many approaches have been proposed. A first approach has been to extend histograms to multiple dimensions. One of the earliest methods of such a kind was proposed in [Poosala and Ioannidis, 1997]. Another

seminal proposal was made in [Bruno et al., 2001]. The issue at this point, however, is to determine on *which* attributes to build the histogram. Indeed, relations usually possess hundreds of attributes, and therefore building a histogram for every combination of attributes is unfeasible. The problem becomes increasingly difficult considering the fact that the most crucial attribute dependencies that need to be captured are those that span multiple relations. Indeed, such a paradigm has not received a lot from statistical communities, who mostly work in the context of single entities. Therefore, many methods have been proposed to capture dependencies between attributes, within and between relations, at as little a cost as possible by making compromises and simplifying assumptions. For instance, methods have been proposed to identify small groups of dependent attributes and build histograms on said groups [Deshpande et al., 2001, Ilyas et al., 2004]. A related method is to use the dependencies to build a graphical model which encodes attribute dependencies in a network structure. These are called *probabilistic relational models*, which are extension of the well-known *probabilistic graphical models*. The most prominent methods in this category are Bayesian networks, which have been proposed in [Getoor et al., 2001] as well as [Tzoumas et al., 2011]. The two latter proposals are important because they paved the way towards using Bayesian networks for the purpose of selectivity estimation. However, the methods they proposed were not particularly efficient, as they are yet to see widespread adoption in practice.

1.2 Contributions and outline

The contributions of this work can be separated into two parts. The first is concerned with using Bayesian networks for the purpose of selectivity estimation. We begin by reviewing fundamental concepts and discuss that are to be made. We then present our methodology, which aims at capturing attributes dependencies within separate relations. Importantly, we carefully design our method so that it remains efficient. In particular, we restrict our networks to have a tree structure, therefore allowing to compute selectivity estimates in linear time. This is a much a simpler paradigm than what can be found in existing Bayesian network proposals [Getoor et al., 2001, Tzoumas et al., 2011]. We justify this step back because it offers a different compromise between selectivity estimation accuracy and efficiency. Indeed, even though we know our method is less accurate from the get-go, it is also

ensured to be much faster. We believe that it is important to offer such a compromise, as clearly efficiency is of primordial importance in database cost models. Secondly, we extend our methodology to capture dependencies across multiple relations. To this end, we introduce the *dependency preservation assumption*, which states that attributes preserve their dependencies across joins. This allows us to reuse the dependencies found within each relation when estimating selectivities for queries that contain joins involving said relations. By exploiting the tree topology of our Bayesian networks, we are able to estimate the distribution of attributes across joins by only having to compute the new distribution of the root attribute. We validate our methodology by benchmarking it against the method described in [Tzoumas et al., 2011], as well as other recent proposals, including methods based on deep learning. Our benchmarks validate the fact that our method is well-behaved and is a robust selectivity estimation. Although it is not as accurate as more sophisticated, it is immensely more efficient, and is guaranteed to perform better than the simplistic yet efficient methods used in current query optimisers.

In the second part, we look downstream and attempt to learn to correct an existing selectivity estimation model. Instead of building a model from scratch, we learn to predict the mistakes the cost model will make for a given query execution plan. In other words, we frame selectivity estimation as a supervised learning problem, whereby the goal is to improve the accuracy of a base model that is known to be weak, thereby using the latter as a bootstrap. Although this has already been proposed in the past, we propose an *online* methodology. Therefore, instead of training a model on a historical dataset in a batch manner, we train our model in an incremental fashion. This is well suited to selectivity estimation because new queries are constantly flowing in, which offers the chance to continuously update our model. Our method is therefore able to compete with batch methods, including deep learning models, and in fact outperforms them when the querying patterns drift through time. From a larger perspective, we believe that training models this way is particularly well suited to computer systems tasks in general. Indeed, in many computer systems, information is endlessly flowing in, thereby justifying the use of models that can keep learning without having to start from scratch. This has been observed by others, and has been named as a field *machine learning for systems*.

The first chapter of this manuscript enumerates existing related work. We discuss the different approaches and the methods that pertain to each method. We also attempt to

sketch the difficulties that selectivity estimation involves, and explain why various methods are not usable in practice. The second chapter encompasses our work on Bayesian networks. The first part describes how to use Bayesian networks to capture dependencies within a relation. We describe a methodology for building Bayesian networks in an efficient manner, and also show how this leads to an efficient inference procedure. The second part extends the work of the first part. We discuss a newly introduced simplifying assumptions, which allows us to reuse existing knowledge within each relation and apply across joins. We provide some benchmarks that exemplify the trade-offs that our method involves. The third chapter concerns our work on online supervised learning. We discuss in further depth some related work, and justify the merits of online learning. We then describe an implementation of our proposal and shows how it fairs against other methods. Finally, we conclude by taking a step back and reviewing our work, before hinting at some further directions.

Chapter 2

Related work

2.1 Concepts

2.1.1 Preliminaries

Before delving into existing methods, it is important to understand where we stand currently. From a statistician's perspective, selectivity estimation is nothing more than a density estimation problem. Therefore, a lot of the work in that field can be applied. However, this completely ignores the point that query optimisation is a critical application, and therefore any method that is employed has to be blazing fast. In fact, as of writing this manuscript, selectivity estimation is still considered to be an unsolved problem [Barbar'a et al., 1997, Hellerstein and Stonebraker, 2005, Wu et al., 2013, Lohman, 2014]. First of all, it is a difficult task that requires making very hard compromises between accuracy, speed, and memory consumption. Many methods have been proposed, but very few of them have made the cut. Instead, the simple methods that were proposed decades ago [Selinger et al., 1979, Chaudhuri, 1998] are still being used. Secondly, query optimisers are still extremely fast overall, even though they use simplistic cost models. Indeed, querying times are long-tailed, and on average are considered fast enough to justify using said simplistic models.

To justify their choices, existing cost models all make simplifying assumptions which we discuss in the next subsection. This is true for established battle-tested query optimisers such as that of PostgreSQL [Stonebraker and Rowe, 1986], as well as new ones such Apache Calcite [Armbrust et al., 2015, Begoli et al., 2018] and Presto [Traverso, 2013, Sethi et al., 2019]. A lot of query optimisers also make use of "tricks", which are complementary to

other methods. Some of these are documented in the literature, but many or not. For instance, the *exponential backoff* trick, which is used by some commercial query optimisers, is only briefly mentioned in research papers such as [Müller et al., 2018], and is discussed on various database forums [1]. We mention this to highlight the fact that there is a gap between research and is what actually used in database systems. Most of the methods discussed in this chapter have in fact never seen the light of day.

2.1.2 Simplifying assumptions

Selectivity estimation is fundamentally a density estimation task. The attributes of each relation can be seen as statistical variables with an unknown underlying probability distribution which we want to estimate. Formally, given a query $\mathcal{Q}(\mathcal{R}, \mathcal{J}, \mathcal{A})$ over a set of relations $\mathcal{R}$, a set of join predicates $\mathcal{J}$ and a set of attribute predicates $\mathcal{A}$, the cardinality of the query $\mathcal{Q}$ is determined as follows:

$$|\mathcal{Q}(\mathcal{R}, \mathcal{J}, \mathcal{A})| = P(\mathcal{J}, \mathcal{A}) \times \prod_{R \in \mathcal{R}} |R| \tag{2.1}$$

where $P(\mathcal{J}, \mathcal{A})$ is the selectivity of the query and $\prod_{R \in \mathcal{R}} |R|$ is the number of tuples in the Cartesian product of the involved relations. The problem is that $P(\mathcal{J}, \mathcal{A})$ is not available. Moreover estimating it quickly leads to a combinatorial explosion. The difficulty is that there are many attributes which are scattered across relations. Therefore, simplifying assumptions have to be made in order to make the computations tractable. Essentially, the goal is to break down the previous formula into simpler parts that are easier to compute. It is important to discuss these assumptions, as the goal of selectivity estimation research is to relax them at as little a cost as possible.

Join uniformity

The *join uniformity* assumption states that attributes preserve their distributions after joins. This allows the following simplification:

$$P(\mathcal{J}, \mathcal{A}) \simeq P(\mathcal{J}) \times P(\mathcal{A}) \tag{2.2}$$

[1] `https://sqlperformance.com/2014/01/sql-plan/cardinality-estimation-for-multiple-predicates`

$P(J)$ is the number of tuples resulting from a join, divided by the size of the Cartesian product of the concerned relations. $P(A)$ is then the percentage of remaining tuples that satisfy the query statement. In other words, making this assumption means that the overall statistical distribution can be estimated locally within each relation. Indeed, if we have an estimate of the distribution of a particular attribute, then with this assumption we can use the same distribution when the relation to which said attribute belongs is joined with another relation. As we will discuss throughout this manuscript, this is an extremely high source of error. Indeed, the distribution of attribute value tend to change by a significant amount whenever relations are joined with each other.

Attribute value independence

Another assumption that is commonly made is that attributes are independent within and between each relation. This is the so-called *attribute value independence* (AVI) assumption. It allows to simplify the computation as follows:

$$P(\mathcal{A}) \simeq \prod_{A_R \in \mathcal{A}} P(A_R) \simeq \prod_{A_R \in \mathcal{A}} \prod_{a_i \in A_R} P(a_i) \tag{2.3}$$

where $P(A_R)$ refers to the selectivity concerning relation R whilst $P(a_i)$ stands for the selectivity of a predicate on attribute a_i. In practice the AVI assumption is very error-prone because attributes often exhibit correlations. However it is extremely practical because each distribution $P(a_i)$ can be condensed into a one-dimensional distribution $\widetilde{P}(a_i)$. The AVI assumption is very strong, and cases that violate it are regularly encountered in practice. First of all, many attributes within a relation have strong dependencies. For instances, a relation of customers might contain attributes that describe nationalities, living locations, incomes, names that are all highly susceptible to exhibit correlations. Therefore, by making the AVI assumption, the selectivity of queries that involve two or more of these attributes would systematically be underestimated. This even more so a problem for normalised databases where the attributes are scattered across relations. Indeed, in this case the AVI assumption compounds with the join uniformity assumption, which leads to huge estimation errors.

Join predicate independence assumption

Next, the *join predicate independence* assumption implies that join selectivities can be computed independently, which leads to the following approximation:

$$P(\mathcal{J}) \simeq \prod_{J_i \in \mathcal{J}} P(J_i) \tag{2.4}$$

Assume we are given two relations R and S. We want to join both relations on their respective attributes $R.K$ and $S.F$. In this case the selectivity of the join (denoted J) can be computed exactly [Selinger et al., 1979]:

$$P(J) = min(\frac{1}{|J.R.K|}, \frac{1}{|J.S.F|}) \tag{2.5}$$

This assumption doesn't usually hold if multiple foreign keys are included in a join [Ioannidis and Christodoulakis, 1991].

Most relational databases – even modern ones [Traverso, 2013, Armbrust et al., 2015] – make all the previous assumptions in conjunction. This leads to the following estimation formula:

$$P(\mathcal{J}, \mathcal{A}) \simeq \prod_{J_i \in \mathcal{J}} min(\frac{1}{|J_i.R.K|}, \frac{1}{|J_i.S.F|}) \times \prod_{A_R \in \mathcal{A}} \prod_{a_i \in \mathcal{A}_\mathcal{R}} \widetilde{P}(a_i) \tag{2.6}$$

In practice this approximation is much too coarse and is frequently wrong by several orders of magnitude. However, it only requires storage space that grows linearly with the number of attributes and doesn't involve any prohibitive computation. In other words, accurate selectivity estimation is traded in exchange for a low computational complexity. The natural question is if a better trade-off is possible. That is, one that relaxes any of the previous assumptions. In this regard, the existing work we will now discuss can be grouped into three categories. First of all, there are methods that attempt to provide a better estimate of the distribution of a single attribute's values, which are called *attribute-level synopsis*. Histograms are a prime example. Secondly, *relation-level synopsis* aim at capturing the distribution of attribute values within a relation. Finally, *schema-level synopsis* are designed to approximate the entire distribution across all the relations. However, there are different ways to skin a cat, and in the rest of this chapter we categorise each method

according to the approach to which it belongs. In fact, we go through each method from the perspective of a statistician rather than that of a database researcher.

2.2 Sampling

Sampling is a natural way to perform selectivity estimation, especially from the point of view of a statistician. The underlying principle is straightforward: execute a logical predicate on a sample of the database and extrapolate. The main advantage is that the underlying data does not have to be summarised because the sample exhibits the same properties as the whole data [Piatetsky-Shapiro and Connell, 1984] – provided the sample is representative and large enough. Given a relation R of size N and a sample r of size n, the cardinality of the output of a predicate $P(R)$ can be estimated as $P(R) = \frac{N}{n} \times P(r) + \delta$. δ is the error that is related to ratio $\frac{N}{n}$. Indeed as n grows larger we expect the sample r to become more representative of R.

2.2.1 Online vs. offline sampling

From a practical point of view, there a few challenging points that need to be addressed in order to make sampling feasible. First of all, sampling the database every time a selectivity estimate has to be produced is extremely wasteful and too burdensome. Sampling can be done with both *online* and *offline*. The obvious advantage of offline sampling is that it doesn't require sampling in real-time, which is extremely helpful in a distributed setting. However offline sampling implies storing a sample of the database. Moreover online sampling can be tailored to each query, whereas an offline sample has to be constructed in such a way that it will be efficient for unknown queries. For offline sampling to work well, it is thus necessary to have an efficient sampling method. Therefore, a sample of the database has to be constructed offline and used at runtime. This brings it's own set of challenges. Indeed, the question is now how much data needs to be sampled in order to answer future selectivity estimates within a desired error interval. The offline sample also has to be refreshed every so often to account for distribution changes and modifications in the database schema [El-Helw et al., 2007]. Finally, the biggest challenging concerns joins. Indeed, if one samples relations independently, then the join of these samples has a high probability of being. This is often coined the *empty join problem* and is presented in detail in [Chaudhuri et al., 1999].

Therefore, specialised sampling methods have been designed for relational data. A thorough review of sampling over joins and joining over samples is given in [Huang et al., 2019].

2.2.2 Sampling sizes

Sampling methods can be either be *fixed-stepped* or *fixed-precision*. In a fixed-step procedure, the sampling size is determined in advance. Whereas in a fixed-precision procedure, the desired precision of the sampling procedure is fixed in advance and the size of the sampling is adjusted to meet the desired precision. [Lipton and Naughton, 1990] were the first to propose a systematic approach of sampling for estimating selectivities for both single and multiple relations. They derived loose bounds for the number of required samples in order to reach a given accuracy. An interesting analysis is also given in [Riondato et al., 2011]. It is based on the *Vapnik-Chervonenkis dimension* and provides theoretical guarantees to correctly estimate the selectivity of a given workload of queries. However, in both cases the required number of samples are quite pessimistic because they are based on random Bernoulli sampling, which samples relations independently.

2.2.3 Sampling relations independently

The simplest method is to sample each relation R_i with a probability p_i. Each resulting sample r_i will then be of size $|R_i| \times p_i$. To estimate the size of the output of a filter condition on R_i one may apply the filter to the sample r_i and multiple the cardinality of the output by $\frac{1}{p}$. As for join predicates, the cardinality of the output of $J(R_1, R_2)$ can be estimated as $\frac{J(r_1,r_2)}{p_1 p_2}$. These estimates can be proven to be without bias and to have a variance with depends on each p_i. The key issue with sampling relations independently is that it ignores the join relationships between relations. In other words, the Cartesian product of uniform samples of a number of tables is different from a uniform sample of the Cartesian product of those tables [Chaudhuri et al., 1999]. A potential solution is to use *cross-sampling* [Olken, 1993], whereby relations are joined before being sampled. This is also discussed in detail in [Haas et al., 1996] and has been applied to the AQUA system [Acharya et al., 1999]. [Ganguly et al., 1996] introduced *bifocal sampling*, which is a method for sampling two relations that is resistant to skew in the key attributes of both relations.

2.2.4 Sampling across relations

The issue with methods that join relations before sampling, such as cross-sampling, is that joins are very expensive. An altogether different approach is to apply *hash functions* to the foreign keys and primary keys that pertain to the joins. Indeed, by constructing deterministic hash functions, it is possible to sample relations independently whilst avoiding the empty-join problem. [Estan and Naughton, 2006] were the first to take this approach in the context of selectivity estimation; they named their proposal *end-biased* sampling. They use a hash function which is biased towards records who's primary key appears frequently in the foreign key attribute of a related attribute, thus reducing the impact of the empty join problem. [Vengerov et al., 2015] pushed into this direction and introduced *correlated sampling*, which improves end-biased sampling by eliminating the need for knowing the frequencies of values in key attributes. Their method uses a random hash function $h(a) \rightarrow [0, 1]$. Say R_1 has an attribute A_1 which is a foreign key to an attribute A_2 of a relation R_2. By applying the same hash function $h(a)$ to both attributes, one may obtain samples which preserve join relationships by keeping all the tuples where $h(a) < p$. This way each both tuples that satisfy $h(a) < p$ will be included in the sample. The unbiased estimator for the size of the result of the join is $J(R_1, R_2)$ is $\frac{J(r_1,r_2)}{p}$. [Leis et al., 2017] proposed a complementary method for speeding up the scan of the key attributes by using existing indexes when available, but their proposal is limited to in-memory databases, which is a niche topic. [Chen and Yi, 2017] introduced *two-level sampling*, which improves on correlated sampling by identifying the frequent items – called *heavy-hitters* – via a initial offline pass through the relations. These ideas are somewhat revisited and discussed in [Zhao et al., 2018]. Correlated sampling has also been implemented within the map-reduce paradigm in [Wilson et al., 2019].

2.3 Supervised learning

2.3.1 Motivation

Another way to frame selectivity estimation is to formulate it as a *supervised learning* problem. Hereby, the goal is to *predict* the selectivity of a query given properties that can be derived from it's raw representation. More precisely, the goal of supervised is to learn a function which maps from a set of inputs – called *features* – to some output(s) –

called *target(s)*. This function can be linear as well as non-linear, and can be defined for either continuous or discrete inputs, or both. In any case, the function typically contains parameters that need to be tuned in order to function adequately. The goal is to learn these parameters by scanning through a training set of examples, whilst keeping in mind that the true goal is to generalise to future examples that are yet unseen. Therefore, the goal is not to *overfit* by finding parameters that perfectly map the inputs and outputs in the training set but break down for unseen data. This can be controlled by so-called *regularisation*, which boils down to limiting the capacity of the function to learn patterns that are too specific and cannot be generalised.

The supervised learning framework is immensely popular in other fields. It has received a considerable amount of attention from the query optimisation community in recent times. In particular, this attention is in large part to the due to the recent trend of revisiting old problems by applying *deep learning* – or simply *neural networks*. [Wang et al., 2016] gives a wide-ranging overview of the opportunities and challenges of applying deep learning to databases. Meanwhile, some criticism of this trend has been formulated in [Hilprecht et al., 2020]. An empirical analysis of deep learning for cardinality estimation is given in [Ortiz et al., 2019]. One of the main downsides of deep learning, and supervised learning in general, is that it requires a costly training phase.

2.3.2 Early days

Before going forward, it is interesting to mention LEO [Stillger et al., 2001], which is query optimiser used by the DB2 database system. The novelty of LEO was to remember the selectivities of query execution plans in order not to have to estimate their selectivities if ever they showed up again. This can be seen as the most basic form of learning: *memorising*. Indeed, in theory it is possible to never make any mistake twice by simply storing the selectivity of every observed query plan. However, such an approach does not allow to generalise to unseen queries, and therefore isn't in fact learning. One of the earliest attempts to perform actual supervised learning for the purpose of selectivity estimation can be retraced back to [Chen and Roussopoulos, 1994]. They proposed a method based on *recursive least squares* for learning the distribution of a single attribute on a continuous domain. In particular, they are able to refine their estimate of the distribution every time a new record is inserted into a relation. However, this method is quite limited because histograms already

do a very good job at approximating the distribution of single attributes, therefore it never picked up in practice. A similar approach for tuning a one-dimensional histogram was also proposed in [Aboulnaga and Chaudhuri, 1999]. Meanwhile, an early attempt at applying off-the-shelf supervised learning can be found in [Malik et al., 2007]. In their so-called "black-box approach", the authors present a methodology for grouping queries by syntax similarity into buckets. Then, they project the queries into a Cartesian coordinate system by extracting features derived from the queries. They fitted a linear model in order to map the features to the query selectivities. Although novel, their methodology was most destined to helping in scheduling workloads, and was not designed to be part of database's cost model. [Akdere et al., 2012] explored a similar method, but provided a more in-depth discussion of practical details, such as offline training and materialisation. Indeed, their work was one of the first to provide a concrete implementation in a query optimiser and expose the ensuing challenges that this incurs, even though they used a simple linear model. More recently, [Ivanov and Bartunov, 2017] proposed a similar methodology, but benchmarked a wider range of common supervised learning algorithms, such as random forests and gradient boosted decision trees. They also used simple features such as how many joins are present, and how many `WHERE` clauses there are. They also used boolean value indicating whether or not a particular attribute, relation, condition is present in a given QEP. This is often called a *one-hot encoding scheme* and is at the core of many of the deep learning methods which we will discuss further on. Indeed, many of the ideas proposed in recent papers based on deep learning have already been discussed in the above papers. They are simply being rediscovered and better understood. Even the idea of using neural networks for selectivity estimation was first introduced in [Boulos et al., 2014] as well as in [Liu et al., 2015], slightly before they were mainstream.

2.3.3 The rise of deep learning

One of the most prominent papers that advocates the use of deep learning for selectivity estimation can be found in [Kipf et al., 2018], as well as in a subsequent work [Kipf et al., 2019b]. They proposed a neural network architecture, which they dubbed MSCN for *multi-set convolutional network*. Their work is in turn inspire from *deep sets* [Zaheer et al., 2017], which is a deep learning framework for handling item sets. For the purpose of selectivity estimation, MSCN featurises a plan with a one-hot encoding scheme. They then group the

different boolean values into sets. Therefore one group pertains to the relations, another to the joins, another to the attributes, etc. A fully-connected neural network is then stacked on top of each set. These sub-networks are then concatenated and a final layer is appended to the overall network. The neural network is then trained by looking at pairs of plans and associated selectivities. The weights of the neural network are obtained via stochastic gradient descent. From the point of deep learning, the neural network architecture from [Kipf et al., 2018] is very standard. The innovation comes from their attention to detail and the detailed description of every step that is taken. Moreover, they describe a way to generate random queries from a database schema, which can be used to warm-up the model before actual queries are seen. Their model is therefore trained on a set of queries which is workload-agnostic, which is not necessarily optimal. Like many selectivity estimation methods, the original MSCN implementation is limited to simple `SELECT-FROM-WHERE` queries that do not involve complex clauses. However, the same authors also proposed an extension to support `GROUP BY` statements, which can be found in [Kipf et al., 2019a]. A plethora of other deep learning based methods have been proposed, some of them being very similar to MSCN, such as [Marcus and Papaemmanouil, 2019], whilst others are complementary. For example [Dutt et al., 2019] proposed a technique for handling `RANGE` queries, while [Hayek and Shmueli, 2019] explored the use of neural networks for predicting the selectivity of containment queries. There also have been methods that attempt to estimate selectivities for similarity queries, such as with `LIKE` statements [Wang et al., 2020]. Recently, [Hayek and Shmueli, 2020] have provided a method for extending MSCN to union queries that contain `OR` statements. Many of these proposals are not necessarily novel, and are rather applications of existing work on boolean algebra to neural networks. Meanwhile, neural networks have also been used in different manners. For instance, in [Wu et al., 2018], a neural network is used to adjust the selectivities estimated by a classical cost model with similar results to MSCN. In [Sun and Li, 2019], the whole cost model is modeled with a deep learning model, not just the selectivity estimation module. Meanwhile, the authors of [Woltmann et al., 2019] have proposed an alternative method which consists in building smaller neural networks that cover specific parts of the schema. According to them, these networks are better able to capture local dependencies and patterns than a single large network. In their experimental results, they are able to improve their accuracy by two orders of magnitude compared to global approaches.

2.3.4 Learning a query optimiser from scratch

It is also worth mentioning that some proposals have been made to design neural networks that can replace a whole query optimiser, not just the selectivity estimation module. For instance, the authors of [Marcus and Papaemmanouil, 2018] describe a hypothetical "hand-free query optimiser" which can learn and perfect all the steps required to optimise a query. They argue that deep learning, and in particular *deep reinforcement learning* [François-Lavet et al., 2018] can be used to perform such a feat. Essentially, they make the case that relational query optimisation boils down to constructing a query plan by making choices. These choices are based on the current state of the plan, a context which pertains to the query at hand, and past experience. In such a scenario, a reinforcement learning algorithm is able to leverage this information in order to make informed decisions and pick the correct relational operators at each stage of the plan. This proposal is in part inspired by a technical report [Tzoumas et al., 2008] that covers the links between adaptive query processing and reinforcement learning. The authors of [Marcus and Papaemmanouil, 2018] then applied their ideas and introduced "Neo" [Marcus et al., 2019]. The query optimiser is able to learn from the mistakes of an existing optimiser, such as the one from PostgreSQL, with promising results. Similar methods, also based on deep reinforcement learning, have been proposed and can be found in [Ortiz et al., 2018] and [Krishnan et al., 2018]. We mention these contributions because, even though they are not pure supervised learning methods, they are based on the assumption that the network is somewhat able to predict the selectivity/cost of a query execution plan. However, historically this has not been the prevalent approach to selectivity estimation. Instead, most methods attempt to understand the distribution of the database's values, without explicitly learning a mapping between the queries and the selectivities. In other words, many methods instead fall under the umbrella of density estimation, which is a form of *unsupervised learning*.

2.4 Unsupervised learning

One of the downsides of supervised learning is that the training data has to resemble the data that will be seen in the future. This is fundamental assumption that drives the workings of many supervised learning algorithms, but alas isn't always verified in practice. A consequent issue is that a sufficient amount of pairs of queries and selectivities has to be

gathered in order for the learning algorithm to get a clear picture. Things become difficult when the data distribution and the query typology are constantly evolving. Indeed, in this case a learning algorithm will constantly be playing a cat and mouse game by catching up with the current state of things.

An alternative approach is to perform *unsupervised learning*. In this paradigm, there is no learning task per se. Instead, the goal is to learn to understand the underlying data generating process. In particular, the sub-field of unsupervised learning which is especially applicable to selectivity estimation is *density estimation*. The goal of density estimation is to build a function which determines the amount of data that satisfies a predicate. These frequencies can then act as direct proxy for selectivities. This is a natural consequence of the fact that selectivity estimation is essentially a density estimation task, albeit with some twists. In this regard, it has a lot of resemblance with the supervised learning approach to selectivity estimation. This is because methods that pertain to the latter approach are essentially trying to do density estimation. The difference is that, with supervised learning, the goal is to explicitly learn to estimate selectivities. With unsupervised learning, this is more of an afterthought, and accurate selectivity estimation is a consequence of being able to correctly summarise the data via a density function.

It is interesting to dwell on the differences between supervised and unsupervised learning approaches. On the one hand, with supervised learning, the goal is to learn to predict the selectivities of queries. The issue is that we don't know in advance what kind of queries are going to be formulated. Therefore, we need to gather a dataset as large as possible to cover all possible cases. On the other hand, with unsupervised learning, the goal is to understand the distribution of the data, and use the acquired knowledge to estimate selectivities. A thorough discussion of the differences between both approaches is given in [Hilprecht et al., 2019]. The advantage of taking a data-driven approach the distribution of the data can be assumed to be stable. Indeed, there is no uncertainty as to what kind of attributes and values will be present in query execution plans. In other words, the database is a *closed world*. In philosophical logic, this is also referred to as *domain closure*. It is therefore possible, ignoring memory constraints, to perfectly estimate selectivities by simply memorising the frequency of every combination of attributes values across all relations. Obviously this is unfeasible, as already discussed. However, this does shed some light on the problem and justifies the use of unsupervised learning. In short, if we cannot predict the queries that

will be thrown at the query optimiser, we can rely on the fact there is no mystery regarding the underlying attribute values. A related consequence of the closed world assumption is that it is impossible to overfit by memorising too much. This is particularly interesting, because it means we can build density estimation models without having to worry about regularising them. Again, this is only possible in the case of using a data-centric approach, not one based on issued queries. In general, this is not the case, as we have to build models that generalise well.

2.4.1 Histograms

One of simplest forms of density estimation, and therefore unsupervised learning, is a histogram. A histogram separates a distribution into n *buckets* – also called *bins* – denoted b_i. A bucket is defined by it's lower bound b_i^- and upper bound b_i^+. For convenience we can define d_i as the distance $b_i^+ - b_i^-$. Each bucket contains values that are comprised in the range $[b_i^-, b_i^+]$. We can denote c_i as the number of values that are contained in bucket b_i. Only b_i^-, b_i^+, and c_i are known. Thus for practical reasons optimisers usually assume that values are uniformly distributed inside each bucket. To obtain the selectivity of a single-attribute predicate one simply has to linearly interpolate within the appropriate bucket b_i – found through binary search – and sum up the previous buckets.

$$s(A < v) = (\sum_{j=1}^{i-1} c_j) + (v - A) \times c_i \tag{2.7}$$

Of course, the next buckets should be summed up in case the range predicate is the other way round.

$$s(A > v) = 1 - s(A < v) = ((A - v) \times c_i) + \sum_{j=i+1}^{n} c_j \tag{2.8}$$

If the number of distinct values u_i per bucket b_i is known, then selectivities for equality predicates can be computed as so:

$$s(A = v) = \frac{c_i}{u_i} \tag{2.9}$$

Histograms are usually built with buckets of the same width (*equi-width*) or of the same height (*equi-height*/*equi-depth*). Using equi-width histograms for selectivity estimation was

first introduced in [Kooi, 1980]. The number of buckets n determines the width of each bucket which is $\frac{max(A)-min(A)}{n}$. The height of each bucket depends on A's distribution. [Piatetsky-Shapiro and Connell, 1984] formalised the use of equi-height histograms. They also showed that equi-height histograms have a much lower worst-case and average error for a variety of predicates compared to equi-width histograms. Opposite to equi-width histograms, the number of buckets n determines the height of each bucket, which is $\frac{1}{n}$. With equi-height histograms, the error bound can be lowered by controlling the bucket height – and *a fortiori* the number of buckets. The number of buckets n exemplifies the compromise that has to made in practice. On the one hand, if n is equal to the number of distinct values in a distribution then the frequency for each particular value will be stored, which implies a large storage cost. On the other hand, if $n = 1$ then a uniform distribution is assumed whilst the storage cost is close to nil – in this case the histogram is called *trivial*. Note that a bucket that contains a single value perfectly captures the frequency of the value, in this case the bucket is said to be *uni-valued*. A bucket that contains more than one value is called *multi-valued*.

Histograms are popular because they are simple and deliver reliable computation guarantees. For instance, insertion and querying operations can both be done in logarithmic time. Histograms also allow a great amount of leeway and can be adapted as necessary. For example, a popular trick is to use uni-valued buckets for the k most frequent values, and store the rest of the values in multi-valued buckets wherein uniformity is assumed. This offers a good compromise between accuracy and storage cost. Many other of such tricks have been proposed throughout the literature. These tricks involve optimising both the width and the height of each bucket in order to minimise the overall selectivity error. For instance, [Ioannidis and Christodoulakis, 1993] showed that histograms that minimise the average selectivity estimation error are ones that minimise the weighted variance:

$$V = \sum_{i=1}^{n} n_i v_i \tag{2.10}$$

where n_i is the number of values in a bucket v_i is the variance of the frequencies inside a bucket. Such histograms are called *V-optimal histograms*. Intuitively, V-optimal histograms have buckets where the distribution of frequencies is somewhat uniform. This leads to reduced errors when performing linear interpolation within a bucket. While V-

optimal histograms are more accurate than equi-width and equi-height histograms, they have some drawbacks. Building a V-optimal histogram is nothing less than an optimisation problem where the parameters are the number of buckets and the size of each bucket. Moreover, V-optimal histogram are difficult to maintain because they can require a complete reconstruction when new tuples are appended to a relation.

[Ioannidis and Christodoulakis, 1993] introduced *biased histograms* in order to build histograms that avoid the worst possible errors. A *biased* histogram is a histograms that contains $n-1$ uni-valued buckets and one multi-valued bucket. On the one hand if these $n-1$ buckets represent the lowest frequencies then the histogram is said to be *low-biased*. On the other hand a *high-biased* histogram is a biased histogram where the $n-1$ uni-valued buckets represent the highest frequencies. Finally the histogram is said to be *end-biased* if the $n-1$ uni-valued buckets capture both the most frequent and the lest frequent values.

End-biased histograms are often used in practice because they offer a good compromise between storing all possible frequencies and storing an approximate distribution. Indeed if the goal is to avoid the worst possible error whilst not storing too much information, then [Ioannidis and Christodoulakis, 1993] advocates that end-biased histograms are optimal – at least if one is comparing different kinds of histograms. Indeed the intuition is store the exact values of the lowest and highest frequencies by means of uni-valued buckets and to store the rest of the frequencies through a single bucket. The intuition is that the rarer frequencies should be stored more precisely because they offer more information about the underlying distribution. For example consider the distribution $[1, 8, 9, 10]$. The frequency of the first element is much smaller than the three other values which relatively similar. In this case a histogram h_1 with buckets $\{1\}$ and $\{8, 9, 10\}$ is preferable over a histogram h_2 with buckets $\{1, 8, 9\}$ and $\{10\}$. Indeed the distribution approximation of h_1 will be $[1, 9, 9, 9]$ whereas it will be $[4.5, 4.5, 4.5, 10]$ for h_2. Moreover, having precise estimates of high frequencies helps a lot in obtaining better join size estimates. Indeed the larger the number of joins the larger the share of frequently occurring values will be [Ioannidis and Christodoulakis, 1993], thus getting precise estimates as early as possible is very beneficial. Histograms have also been tailored for specific types of queries. For instance, [Poosala et al., 1996] did a systematic review of various types of histograms, and benchmarked them on a range selectivity estimation task. They also gave some cues for selecting the number of buckets depending on the shape of the underlying data distribution.

In most cases, histograms are "good enough" for capturing the distribution of an single attribute's values. Most of the error made by the cost model is instead due to missed dependencies between attributes. In spite of this, one-dimensional histograms are ubiquitous in existing relational database cost models. Furthermore, histograms are used as building blocks in more sophisticated methods, such as Bayesian networks. Meanwhile, *multi-dimensional* histograms have been proposed to capture dependencies between attributes. Even though single-column histograms capture the skew in a distribution, they do not take into account correlations between attributes. To compute the selectivity of a conjunctive predicate $s(p_1 \cap p_2)$ with only single-column histograms, one still has to make AVI assumption and compute $s(p_1) \times s(p_2)$. Even if $s(p_1)$ and $s(p_2)$ are perfectly known, an overlap between p_1 and p_2 means that the estimated selectivity will be lower than $s(p_1) \times s(p_2)$. As an example consider a relation *employee* with two attributes *age* and *salary* and that we wish to obtain the selectivity of the predicate $(age > 42) \cap (salary > 24000)$. Because single-column histograms are available we know that $s(age > 42) \simeq 0.4$ and $s(salary > 24000) \simeq 0.7$. If we assume the AVI assumption holds then we can calculate $s((age > 42) \cap (salary > 24000))$ as $0.7 \times 0.4 = 0.28$. However, it might be that *age* and *salary* are correlated – which is quite often the case in reality. In other words it might be that older employees usually have higher salaries. In that case the proportion of employees older than 42 and earning more than 24000 might be equal to, say, 0.36 and not 0.28. Multi-column statistics are necessary to capture this kind of correlation. [Muralikrishna and DeWitt, 1988] introduced multi-dimensional histograms for estimating selectivities for multi-attribute predicates. [Poosala and Ioannidis, 1997] gave a thorough treatment of multi-dimensional histograms, and discussed practical aspects and common operations to be performed.

While multidimensional histograms are certainly of merit, they have enjoyed limited success, albeit more than other methods. A big issue with multi-column histograms is that too many values have to be stored. Let b be the number of bins per dimension and k the number of attributes in a database. Then there are $\sum_{i=1}^{k} \binom{k}{i} \times b^i$ values that have to be stored to capture all the possible interactions. This number grows extremely quick. For example for b equal to 100 and k equal to 30 – quite conservative numbers – the number of possible histograms is 1.347×10^{60}. Therefore, the attributes on which to build multi-dimensional histograms has to be chosen, either manually or automatically. In a similar vein, [Muthukrishnan et al., 1999] proved that finding an optimal partitioning for multiple

attributes is an NP-hard problem. Therefore, approximate algorithms have been proposed. For instance, [Lee et al., 1999] proposed a method based on cosine transforms, which reduces the amount of required storage space. To address the issue of not being able to build histograms for each combination of attributes, [Bruno et al., 2001] proposed a workload aware methodology. Instead of looking at the data, they look at the issued queries and build multidimensional histograms for attributes that are often queried together. A similar proposal was made in [Srivastava et al., 2006]. The main difference is that the latter uses entropy-based histograms, which seek to build histograms wherein the entropy of each bucket is minimised. A simpler description for unidimensional histograms can be found in [To et al., 2013]. In the meantime, [Gunopulos et al., 2005] made some improvements to deal with skewed attributes and provided better experimental results. In the statistical community, histograms have for long been cast aside in favour of more sophisticated methods, such as kernel density estimation.

2.4.2 Kernel density estimation

Kernel density estimation (KDE) have also been studied extensively for the purpose of selectivity estimation. They can be seen as a smooth version of histograms [Blohsfeld et al., 1999]. The general idea behind KDEs is to build a distribution by averaging individual points x_i. Each point contributes to the global distribution through a *kernel function* K. Together with a *bandwidth parameter*, K determines how each point x_i "spreads" in it's neighbourhood:

$$f(x) = \frac{1}{nh} \sum_{i=1}^{n} K(\frac{x - x_i}{h}) \tag{2.11}$$

A KDE can be viewed as a set of averaged "bumps" which are positioned at the samples x_i. The bandwidth h determines the spread of these bumps, whilst the kernel K determines their shape. The global distribution is constructed by averaging up these bumps. The bandwidth h is also called *smoothing parameter* and plays a crucial role in producing good estimates. On the one hand, if h is too large over-smoothing will occur and sharp details of the distribution will be obfuscated. On the other hand if k is too small then the global distribution will seem very rough and spiky. Tuning h is important in order to obtain an accurate estimator, just like determining the number of bins in a histogram is crucial to

it's performance. In order to choose h, [Blohsfeld et al., 1999] chooses to minimise the relative error, turning the problem into an optimisation process with regard to h. As for the kernel function, the general consensus is that it doesn't matter as much [Silverman, 1986]. [Blohsfeld et al., 1999] chooses to use the *Epanechnikov* kernel because it is fast to compute.

The main advantage KDEs have over histograms is that they are smoother, and can thus better approximate real data. Whereas histograms require linearly interpolation within buckets, KDEs can directly be queried at any point. In statistics, KDEs are considered to be extremely accurate estimators assuming the underlying distribution is sufficiently smooth. However, this assumption does not often hold in practice [Blohsfeld et al., 1999]. Indeed, it is often the case that attributes have jagged distributions that oscillate a lot, which is the main reason why KDEs are not used in practice. To counter this issue, [Blohsfeld et al., 1999] propose to combine histograms and KDEs and by building one KDE per bin. Although theoretically sound and accurate, this hybrid method is prohibitively expensive to use in practice. Indeed, KDEs by themselves are already memory intensive, due to the fact that their complexity scales linearly with the amount of data points. To alleviate this computational cost, [Heimel et al., 2015] explored algorithmic tricks and the usage of GPU cards. They also provided a method to evolve the KDE by continuously updating the KDE with new kernels. Meanwhile, [Kiefer et al., 2017] applied KDEs for estimating the selectivity of queries which involve joins. Their main contribution was to show how to merge KDEs from different tables, which is reminiscent of histogram alignment techniques. Nonetheless, KDEs have been used as the main alternative histograms. Indeed, they seem to be more popular than other methods, such as wavelets.

2.4.3 Wavelets

Wavelet decomposition is a method for compressing a signal. For the purpose of selectivity estimation wavelets can be used as a replacement to histograms [Matias et al., 1998] – wavelets are nothing more than more granular histograms. The idea of wavelet decomposition is to "decompose" a signal into a more coarse representation whilst preserving information so as to be able to "recompose" it afterwards. For instance *Haar* wavelets are used in the following manner (taken from [Matias et al., 1998]).

Take the following list:

$$S_0 = [2, 2, 7, 9]$$

Average all contiguous values in S_0 to obtain a coarser signal S_1 and store the differences with each average in another list D_1.

$$S_1 = [\tfrac{2+2}{2}, \tfrac{7+9}{2}] = [2, 8];\ D_1 = [2-2, 9-8] = [0, 1]$$

S_0 can be reconstructed by doing $S_0 = [2-0, 2+0, 8-1, 8+1] = [2, 2, 7, 9]$. Although S_1 is shorter than S_0, we still have to store the list of differences to the average D_1 in order not to lose any information – in other words we still have to store 4 values. The same decomposition process that was applied to S_0 can be reapplied to S_1 to obtain S_2.

$$S_2 = [\tfrac{2+8}{2}] = [5];\ D_2 = [8-5] = [3]$$

S_2 can't be decomposed further so the process has to stop. In the same way that S_0 can be reconstructed from S_1 and D_1, S_1 can be reconstructed thanks to S_2 and D_2. This implies, by recursion, that S_0 can be reconstructed with S_2 and D_2. At this point, all the information needed to reconstruct the original signal can be stored by concatenating S_2 and each A_i as so:

$$\hat{S} = [5, 3, 0, 1]$$

[Matias et al., 1998] proposes to apply wavelet decomposition to the cumulative distribution function (CDF) of a continuous attribute.

As for compression, the idea in [Matias et al., 1998] is to remove the smallest coefficients in $\hat{S}$. For example the third value is 0 and doesn't bring any new information at all when reconstructing the original signal, hence it can be removed without any loss of information. Recall that the 0 value comes from computing the differences of two contiguous and identical values with their average – 2 and 2 in our example. In other words the wavelet decomposition "captured" the fact that the signal didn't vary on it's first two values. In this sense wavelet decomposition is very similar to information compression methods where one makes the most of identifying repetitions. [Matias et al., 1998] proposes different methods for choosing which coefficients to remove from the compressed signal. On the one hand one can keep the m largest coefficients if one wishes to have control over the amount of information stored. On the other hand one may want to minimise the compression error and an optimisation

process can be used to determine the necessary number of coefficients that have to be stored. Effectively the trade-offs that have to be same are not so dissimilar to the ones pertaining to histograms. Once again a compromise has to made between the amount of information and the storage size.

Wavelet decomposition naturally extends to multi-dimensional distributions. Indeed the same process described above can also be applied to CDFs with more than one dimension. A thorough overview of multi-dimensional wavelets is done in [Chakrabarti et al., 2001]. One of the downsides of wavelet decomposition is that the original signal has to be reconstructed at run-time, which is significantly more costly that querying a histogram. Whats more, wavelet decomposition can only be applied to attributes with continuous values.

2.4.4 Probabilistic graphical models

A *probabilistic graphical models* (PGM) is a way of taking into account the joint distribution over multiple attributes. From a mathematical point of view, probabilistic graphical models can compactly represent complex joint distributions over high-dimensional spaces. There are two main families of PGMs, namely *Bayesian networks* and *Markov models*. *Probabilistic relational models* [Friedman et al., 1999] are extension of PGM that are tailored towards relational data. The major contribution in this direction has been to translate the usage of Bayesian networks to encode attribute dependencies across relations [Getoor et al., 2001].

A graphical model relies on the *conditional independence* assumption. This assumption which states that two correlated attributes A and B can be assumed to be independent given the knowledge of a third attribute C. That is, if one knows the joint distributions $P(A, C)$ and $P(B, C)$, then one indirectly knows the joint distribution $P(A, B)$. [Getoor et al., 2001] gives the example of a boolean `home-owner` attribute which is conditionally independent from attribute `education` given attribute `income`. That is, if one knows the link between `income` and `education` and the one between `education` and `home-owner`, then something can be said about the relationship between `income` and `home-owner`.

A *Bayesian network* (BN) is a type of PGM that can be used to make the most of indirect influences between attributes. BNs consist in building a graph where each node represents an attribute and each vertex stands for a direct link. Attributes that do not share a vertex are indirectly connected through a sequence of intermediary connections. Much like histograms, the construction of a BN can be done offline whilst selectivity estimates

can be obtained online. As an example, consider a relation with 3 attributes A_i. Attribute A_1 has 3 distinct values, A_2 has 3, and A_3 has 2. To know the discrete joint probability distribution $P(A_1, A_2, A_3)$ one would have to store $3 \times 3 \times 2 = 18$ values – 17 by using the fact that the probabilities have to sum up to 1. With these values the selectivity estimate for any kind of equality predicate can be obtained. Although feasible for a toy example, the number of values that has to be stored grows exponentially with the number of relations and attributes. To counter this, one may represent the joint distribution in a factored form by leveraging conditional independence. By storing the marginal distribution $P(A_1)$ and the conditional distributions $P(A_2 \mid A_1)$ and $P(A_3 \mid A_2)$, one may retrieve the full joint distribution like so:

$$P(A_1, A_2, A_3) = P(A_1) \times P(A_2 \mid A_1) \times P(A_3 \mid A_2) \tag{2.12}$$

The number of values to store for the factored form is $3 + 9 + 6 = 18$, which is the same as for the non-factored form. However, by taking into account the fact that the probabilities have to add up to 1, only $2 + 4 + 3 = 9$ values have to stored. These savings grow exponentially with the number of relations and attributes. In fact, [Getoor et al., 2001] gives an example where they store 951 values instead of 7 billion. Selectivity estimates for range predicates can also be obtained by summing up the probabilities for each discrete value in the desired range. Although feasible, this quickly becomes a slow process for attributes with many distinct values. To handle this, [Getoor et al., 2001] proposes to discretise the values with a range of techniques, all somewhat similar to the histograms methods described previously.

The difficulty with Bayesian networks is that a network structure has to chosen. There exist many valid structures, as one has to determine between which attributes conditional independence holds. Inevitably, building a BN requires going through a length optimisation process [Heckerman et al., 1998]. The goal of this optimisation process is to find the structure which explains the best a given dataset. However no method that runs in polynomial time has yet been discovered. Therefore, in order to simplify the computation, it is necessary to make simplifying assumptions by constraining the network structure. This was addressed in [Tzoumas et al., 2011], whereby dependencies are limited to be between two attributes. This work was also discussed in [Tzoumas et al., 2013].

2.5 Conclusion

As we have hopefully conveyed, selectivity estimation is a difficult problem in that it requires making difficult compromises. The sampling approach has the benefit of being simple, but it comes at the high computational cost of having the execute nonetheless, albeit on a relatively small excerpt of the database. Meanwhile, the supervised learning approach has received at of attention in recent times. Although it has its merits, we believe it is flawed in that it attempts to perform density estimation indirectly. Indeed, selectivity estimation is nothing more than a case of density estimation, and is usually performed by taking measures of the data, not by learning a mapping function. Learning to predict selectivities by looking at queries seems akin to deliberately tying a hand behind our back. Furthermore, supervised learning methods are limited by the fact that the amount of labeled execution plans depends on the database usage. Meanwhile, all the data from each relation is at hand, which is a low-hanging fruit that can be exploited if one frames the problem as an unsupervised learning problem.

The main advantage of unsupervised learning, and in particular density estimation, is that it essentially boils down to performing selectivity estimation. Once a density estimation model has been decided upon and trained, it can be used to estimate of any query whatsoever. It's performance depends on the amount of capacity that we are ready to allocate to it. As such, Bayesian networks are an elegant method, as it possible to design them ways that each offer a different computational compromise. During this PhD, we explored some mechanisms that allow Bayesian networks to act as a strong selectivity estimation model. Existing proposals are flawed in that their computational cost is too high, essentially because they are trying to do too much. Computational performance is of paramount importance, and is in fact a constraint to which we must abide. At the core of our work is the idea to remain simple, and produce methods that are efficient and have a practical aspect to them.

Chapter 3

Selectivity estimation with Bayesian networks

3.1 Motivation

As we have attempted to explain in the previous chapter, selectivity estimation can be viewed as a case of density estimation. In such a setting, the goal is to determine of data that satisfies a particular predicate. For instance, how many users are blond, are born in Sweden, and have taken a flight between Stockholm and Ankara. Bayesian networks represent a general and powerful framework for reasoning and going about such a task. In particular, they allow to model dependencies between attributes and organise them into a graph-like structure. Therefore, Bayesian networks are a promising candidate for performing selectivity estimation. However, they need to be tutored in such a way that they can be used in our particular context, which in particular imposes strict performance constraints.

3.2 What is a Bayesian network?

Before delving into our method, we will give a short overview of the principles that underpin Bayesian networks. This will allow us to briefly highlight and discuss the benefits and challenges brought by Bayesian networks, in particular in the context of relational data. A Bayesian network (BN) is a probabilistic model. As such, it is used for approximating the probability distribution of a dataset. In theory, the underlying dataset is assumed to be generated from an unknown probability distribution. If we had an infinite amount of time

and resources, then we could faithfully estimate this generative process by computing the *full joint distribution*. The latter is often denoted as $P(X_1, \ldots, X_i, \ldots, X_n)$, whereby X_i are variables. We will also denote it by the more terse $P(\mathcal{X})$. The full joint distribution is perfectly accurate because it takes into account all the interactions between variables. However, it is inordinately large and therefore isn't practical. In order to make things simpler, one idea is to identify couples of variables that are independent and extract them from the joint distribution. The extreme way to proceed is to assume that all variables are independent, and therefore estimate the full joint distribution as so:

$$P(X_1, \ldots, X_i, \ldots, X_n) = \prod_{i=1}^{n} P(X_i) \tag{3.1}$$

This is essentially what is implied by the attribute value independence (AVI) and join uniformity assumptions, which we discussed in the previous chapter. The only benefit of this approach is that it only requires estimating one-dimensional distributions, which is as tractable as it gets. A more sophisticated notion is to identify *conditional independencies* between groups of variables. For instance, take the fact the people who have a headache usually tend to cough – i.e. the values of both variables are correlated. Well, these variables are conditionally independent of each other if we know that a person has the flu or not. In other words, symptoms are independent of each other if we have the knowledge of their common cause. This allows us to perform the following factorisation:

$$P(headache, cough, flu) = P(headache \mid flu) \times P(cough \mid flu) \times P(flu) \tag{3.2}$$

Now we only have to store a couple of two-dimensional distributions along with a one-dimensional one. That is, assuming that the variable relationships are correct. At this point, we have made a compromise between the complete independence assumption and the full joint distribution. The benefit is that we have captured the dependencies that matter and eliminated the redundant one between *headache* and *couch*. The goal of a Bayesian network is to identify these redundant relationships and essentially factor them out of the equation. The particularity of a Bayesian network is that it uses a directed acyclic graph (DAG) to organise relationships between variables. The graph contains one node per variable, whilst each directed edge represents a conditional dependency between two variables. For instance,

if nodes A and B are connected with an edge that points from A to B, then this stands for the conditional distribution $P(B \mid A)$. A Bayesian network is thus a product of many such conditional dependencies, which formally is:

$$P(X_1, \ldots, X_n) \simeq \prod_{X_i \in \mathcal{X}} P(X_i \mid Parents(X_i)) \tag{3.3}$$

Once we have a Bayesian network at our disposal, we can use to answer probabilistic queries, such as "What is the probability of someone having the flu?" or "How likely is a person of having the flu, knowing that they are coughing but don't have a headache?". With a Bayesian network, we are thus able to answer any selectivity estimation problem by converting a logical query into a mathematical formula following standard rules of probability [Jensen et al., 1996], in particular Bayes' rule. In Bayesian network jargon, this process is referred to as *inference*. The important thing to notice is that selectivity estimation is essentially equivalent to answering probabilistic queries. Therefore, if we have a Bayesian network which accurately models the underlying data, then we can do selectivity estimation. Note that in such a case we are necessarily more accurate than when using the AVI assumption. This is an attractive approach because it is a well grounded method and is very versatile. Indeed, it allows to answer *any* selectivity estimation query, not just a subset on which a supervised learning problem would have been trained on. On a side-note, it is interesting to notice that if we had enough resources, then we could answer the probabilistic queries we gave by simply interrogating the data. Indeed, although a Bayesian network summarises the distribution of a dataset, it doesn't bring any added. In fact, it's only benefit is that it provides a much more efficient way to answer a query when compared to scanning the whole dataset from scratch.

There are a few considerations to take into account before using Bayesian networks off-the-shelf. First of all, we need to find the structure of the Bayesian network. Clearly, we cannot expect users to provide a schema of the Bayesian network by themselves. Indeed, we need an automatic manner of determining the optimal structure of the Bayesian network. It is easy to see that this is difficult problem, because it implies some notion of causality. Indeed, human experience tells us that the flu is a cause of headaches, and not the other way round. However, a computer has no such prior knowledge. Additionally, we have to consider that we have strong performance constraints. Even though we *can* answer any

selectivity estimation query, we also want to be able to do so in a reasonable amount of time. As we will see, this is largely determined by a Bayesian network's structure. Finally, we need to deal with the fact that selectivity estimation has essentially to be performed over multiple relations, which isn't a paradigm for which Bayesian networks were designed for initially. Before delving into the latter, we will first discuss our method for applying Bayesian networks within a single relation.

3.3 Applying Bayesian networks to single relations

3.3.1 Structure learning

A Bayesian network (BN) factorises a probability distribution $P(X_1, \ldots, X_n)$ into a product of conditional distributions. For any given probability distribution $P(\mathcal{X})$ there exist many possible BNs. For example $P(hair, nationality, gender)$ can be factorised as:

$$P(hair \mid nationality) \times P(gender \mid nationality) \times P(nationality),$$

as well as:

$$P(hair \mid nationality, gender) \times P(nationality) \times P(gender).$$

Both factorisations are represented graphically in figure 3-1. Both possibilities are *valid* in that they both describe a possible manner in which the data *could* have generated. Only the data can tell if one is better suited than the other.

Figure 3-1: Possible factorisations of $P(hair, nationality, gender)$

Although both factorisations shown in figure 3-1 are legal, they have different storage complexities. On the one hand, the first factorisation requires a couple of two-dimensional distributions and a single one-dimensional distribution. On the other hand, the second factorisation requires a couple of one dimensional distribution, along with a three-dimensional distribution. The problem is that the full joint distribution can be stored with a three-

dimensional distribution by itself. Therefore, the second factorisation requires more information than necessary. As we will discuss in the following section, both factorisations incur different querying costs. In addition to finding a network structure that adequately fits the data stored in a relation, we also would like to be able to constrain the network to have a particular shape. The goal of *structure learning* is to find a BN that closely matches $P(\mathcal{X})$ whilst preserving a low computational complexity. Indeed, for any given BN, the cost of storing it and of answering queries both depend on it's structure.

The classic approach to structure learning is to define a scoring function that determines how good a BN is – both in terms of accuracy and complexity – and to run a mathematical optimisation procedure over the possible structures [Heckerman et al., 1995]. The problem is that such kind of procedures are too costly and don't fit inside the tight computational budget a database typically imposes. Recently, linear programming approaches that require an upper bound on the number of parents have also been proposed [Jaakkola et al., 2010]. In practice these can handle problems with up to 100 variables which is far from ideal. Finally, one can also resort to using greedy algorithms that run in polynomial time but don't necessarily find a global optimum.

Chow-Liu trees [Chow and Liu, 1968] is one such method that finds a BN where dependencies between two attributes are the only ones considered. Building a Chow-Liu tree only involves three steps. Initially the mutual information (MI) between each pair of attributes is computed. Mutual information is defined as so:

$$MI(X_i, X_j) = \sum_{x_i \in X_i} \sum_{x_j \in X_j} P(x_i, x_j) \times \log(\frac{P(x_i, x_j)}{P(x_i)P(x_j)}) \tag{3.4}$$

The mutual information can be seen as the strength of the relation between two variables, whether it be linear or not. The distribution $P(X_i, X_j)$ contains the occurrence counts of each pair (x_i, x_j) in a relation R. It can be obtained with a `SELECT COUNT(*) FROM R GROUP BY` X_i`,` X_j statement in `SQL`. The distributions $P(X_i)$ and $P(X_j)$ can be obtained by marginalising over $P(X_i, X_j)$ with respect to the other attribute. By marginalising, we refer to the process of choosing one attribute and summing over the values of the other attributes. Essentially, this collapses a multivariate distribution into a unidimensional one. The resulting distribution is the distribution of the chosen attribute *ceteris paribus*. Once the mutual information for each pair of attributes is computed, they are organised into a

fully connected weighted graph G. This graph is therefore composed of edges which are each associated with an MI value. For the sake of completeness, we will give an example of our procedure on a toy example. We will reuse this example in further parts of this chapter. Let us start with a single relation called `customers`. The latter contains five attributes, which are `nationality`, `country`, `city`, `eye_colour`, and `hair_colour`. Computing the mutual information values between each pair of attributes and organising them into a fully connected graph leads to the representation displayed in figure 3-2. Note that we have chosen the particular MI values somewhat arbitrarily for the sake of example.

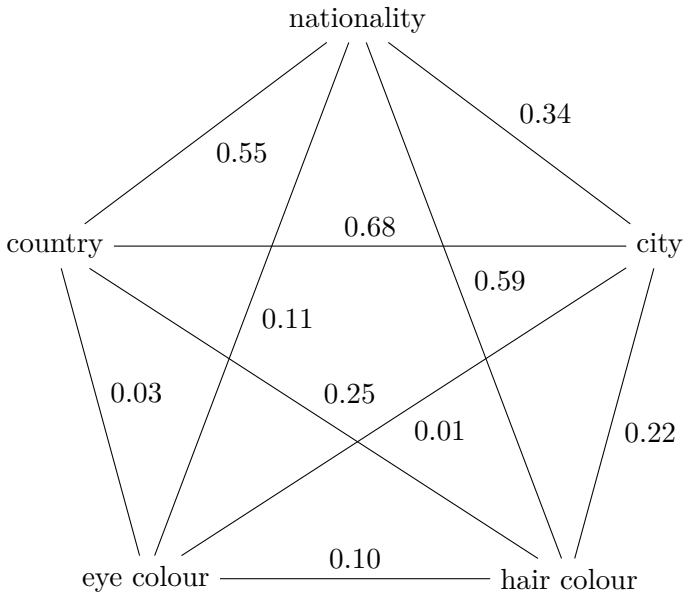

Figure 3-2: Mutual information amounts for five attributes

The next step is to find the *maximum spanning tree* (MST) of the graph, which is the spanning tree whose sum of edge weights is maximal. A spanning tree is a subset of $p - 1$ edges that forms a tree. Finding the maximum spanning tree can be done in $\mathcal{O}(p \log(p))$ time, for example, by using Kruskal's algorithm [Kruskal, 1956]. Other methods, such as Borůvka's algorithm [Nešetřil et al., 2001] and Prim's algorithm [Prim, 1957], are also valid candidates and have the same runtime complexity. The result of applying this procedure to the graph from figure 3-2 is shown in figure 3-3.

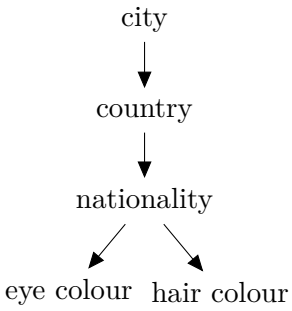

Figure 3-3: Maximum spanning tree of figure 3-2

Finally the MST has to be directed by choosing a node at random and defining it as the root. The choice of the root attribute does not matter because mutual information is symmetric. From a human perspective, one choice make more sense than another. This is because we might an inherent sense of causality that would justify the choice of one particular root node. However, this doesn't have any influence whatsoever on the amount of information that the BN is able to capture. Note however that Bayesian networks can be used to unravel causality, but that is a whole other story [Murphy, 2001]. In our case we are just doing density estimation, so the edge directions do not matter to us. Therefore, we have not paid much attention to the choice of the root node.

We chose to use Chow-Liu trees for two practical reasons. First of all they are simple to construct. The only part that doesn't scale well is computing the MI values. However, this can be accelerated by using a coarser representation of the data such as histograms, effectively discretising the data. Moreover, the process can be run over a sample of a relation. In our experience, these two methods greatly reduce computation time without hindering the accuracy of the resulting trees. Secondly the output network is a tree – hence there is only one parent per node. This is practical because retrieving a marginal distribution – in other words *inferring* – from a tree can be done in linear time [Robertson and Seymour, 1986]. Moreover, storing the network only requires saving $n-1$ two-dimensional distributions and one uni-dimensional distribution. On top of this, [Chow and Liu, 1968] proves that Chow-Liu trees minimise the KL divergence, meaning that they are the best possible trees from an information theory perspective. The downside is that they can't capture events that result from multiple causes. For instance, it only snows if it is both cold and rainy. This is too much of a worry, as we should be content enough of having a cheap method for capturing some dependencies. It's all about making compromises.

3.3.2 Parameter estimation

Once a satisfying structure has been found, the necessary probability distributions have to be computed. Indeed, recall that a Bayesian network is essentially a product of Conditional Probability Distributions (CPD). A CPD gives the distribution of a variable given the value of one or more so-called parent variables. For discrete variables, things are relatively straightforward because CPD are essentially tables. For instance, the two-way tables 3.2 and 3.3 are two CPDs that are both conditioned on the `nationality` attribute. Likewise, the 3.1 is a one-dimensional table because the `nationality` attribute is not conditioned on any other attribute.

American	**Swedish**
0.5	0.5

Table 3.1: $P(nationality)$

	Blond	**Brown**	**Dark**
American	0.2	0.6	0.2
Swedish	0.8	0.2	0

Table 3.2: $P(hair \mid nationality)$

	Male	**Female**
American	0.5	0.5
Swedish	0.45	0.55

Table 3.3: $P(gender \mid nationality)$

Meanwhile, things are much murkier in the case of numeric attributes. Indeed, it is easy to see that tables are an ill-fitting solution for recording probabilities for non-discrete values. This is due to the fact that numeric attributes have an infinite number of possible by definition. In our case, because of the closed-world assumption, this number is not in fact infinite. Nonetheless, it is potentially very large, which can be a problem. In fact, a discrete attribute with a very high cardinality is also a problem. For continuous

variables, a potential solution is to assign a parametric representation to said variables. For instance, we could say that an attribute follows a Gaussian distribution for which the parameters could be estimated from the data [Weiss and Freeman, 2000]. However, this requires choosing a distribution for each given attribute, which isn't very practical. The other way to proceed is to use a non-parametric representation, such as a histogram. Indeed, while coarse, a histogram is able to approximate any attribute value distribution, regardless of it's statistical properties. This works out nicely, because cost models have a rich history of using histograms, as explained in the previous chapter. Therefore, all of the different flavors of histograms can be used for our purpose. As a side-note, note that Bayesian networks that handle both discrete and continuous attributes are called *hybrid Bayesian networks*. Due to the fact that we will essentially be treating them in the same manner via histograms, we will not be making any distinction between discrete an continuous attributes in further parts of this chapter.

The number of values needed to define a CPD is c^{p+1} where c is the cardinality of each variable – for simplicity we assume it is constant – and p is the number of parent variables. This stems from the fact that each CPD is related to $p + 1$ variables and that each and every combination of values has to be accounted for. The fact that Chow-Liu trees limits the number of parents each node has to 1 means that we only have to store c^2 values per distribution. Moreover, a sparse representation can be used to leverage the fact that 0s are frequent. Indeed, a value might only appear for particular conditioning values. This is broadly encompassed by the notion of *deterministic distributions*. Although they are not ubiquitous, they can arise in particular cases, an example being functional dependencies. Indeed, some attributes may have a hierarchical structure, such as cities that belong to countries, which themselves belong to continents. In such cases, some space can be saved by using an adequate data structure other than a histogram. As we mentioned above, if the cardinality of a variable is high, then a lot of values still have to be stored. This can be rather problematic in a constrained environment.

To preserve a low spatial complexity, we found that end-biased histograms were a good choice overall. The idea is to preserve the exact probabilities for the k most common values of a variable and put the rest of the probabilities inside j equi-height intervals. Using equi-height intervals means that we don't have to store the frequency of each interval. Indeed it is simply $1 - \sum_{i=1} P(MCV_i)$, where $P(MCV_i)$ denotes the frequency of the i^{th} most common

value. Instead, by assuming that the values inside an interval are uniformly distributed, we only have to store the number of distinct values the interval contains. Table 3.4 shows what a CPD with intervals looks like. In the example, given that a person is American, there is probability of $1-(0.2+0.5) = 0.3$ that his hair colour is in the [Dark, Red] interval. Because there are 3 distinct values in the [Dark, Red] interval, the probability that an American has, say, hazel hair is $\frac{1-(0.2+0.5)}{3} = 0.1$.

	Blond	**Brown**	**[Dark, Red]**
American	0.2	0.5	3
[British, French]	0.4	0.3	3
Swedish	0.8	0.2	0

Table 3.4: $P(hair|nationality)$ with $k = 2$ and $j = 1$

Compressing a CPD in such a manner means that we only have to store $(k+j)^2$ values per distribution. If we assume that there are n attributes inside a relation, then storing a Bayesian networks requires $(k+j)+(n-1)(k+j)^2$ values in total. Indeed, the first $(k+j)$ corresponds to the network's root node which is not conditioned on any other variable. This has the added advantage that we can handle continuous variables that usually have extremely high cardinalities.

In practice, retrieving CPDs inside a relational database can easily be done with `GROUP BY` and `COUNT` statements. Moreover, each CPD can be obtained from a sample of the relations to reduce computation time. Indeed, having an exact value for each case of each CDP isn't of paramount importance. Many RBDMS implementation in fact resort to sampling in order to compile their sufficient statistics in a timely manner. What's more, if data is appended to or removed from a relation, then only the CPDs have to recomputed. Indeed, if one assumes that the attributes dependency structure remains constant, then there is no point in running a new structure search. However, if new attributes are added to a relation, then the structure of the Bayesian network it should belong to has to be rebuilt from scratch. One may imagine that structure searches can be plumed with CPDs updates on a fixed schedule.

3.3.3 Producing selectivity estimates

As previously mentioned, inference is the task of obtaining a marginal distribution from a Bayesian network. For example we may want to know the probability of Swedish people having blond hair (i.e. $P(hair = \text{“}Blond\text{”} \wedge nationality = \text{“}Swedish\text{”})$). The idea is to treat the obtained probability as the selectivity of the associated relational query. For each relation involved in a query, we identify the part of the query that applies to the relation and determine it's selectivity. Then, by assuming that attributes from different relations are independent, we simply multiply the selectivities together. Although this is a strong assumption, we argue that capturing table-level dependencies can still have a significant impact on the overall selectivity estimation. Of course we would be even more precise if we had determined dependencies between different relations as in [Getoor et al., 2001, Tzoumas et al., 2011], but it would necessarily involve joins. In other words our method offers a different trade-off between accuracy and computational feasibility.

Performing exact inference over a BN is an NP-hard problem [Cooper, 1990] in general. Approximate methods exist based on Monte Carlo estimation exist, such as *belief propagation* [Pearl, 1982] and Gibbs sampling [Gelfand and Smith, 1991], but they are not efficient and don't suit our purpose. Due to the fact that we have restricted our BNs to trees, we can make full use of purpose-built algorithms that only apply to trees. The *variable elimination* (VE) algorithm [Cowell et al., 2006] is a simple exact inference algorithm that can be applied to any kind of network topology. Specifically, the complexity of VE is $\mathcal{O}(n \exp(w))$ where n is the number of nodes and w is the width of the network [Robertson and Seymour, 1986]. However, the width of a tree is necessarily 1, meaning VE can run in $\mathcal{O}(n)$ time. The formula for applying VE is given in (3.5), wherein k attributes are being queried out of a total of n.

$$P(A_1 = a_1, \ldots, A_k = a_k) = \sum_{i=k+1}^{n} \prod_{j=1}^{k} P(A_j = a_j \mid Parents(A_j)) \tag{3.5}$$

For example, if *hair* depends on *nationality* and *nationality* depends on *gender*, then by applying equation (3.5), we obtain:

$$\begin{aligned}
P(hair = \text{“}Blond\text{”}) &= P(hair = \text{“}Blond\text{”} \mid nationality = \text{“}Swedish\text{”}) + \\
&\quad P(hair = \text{“}Blond\text{”} \mid nationality = \text{“}American\text{”}) \\
&= P(hair = \text{“}Blond\text{”} \mid nationality = \text{“}Swedish\text{”}) \times \\
&\quad (P(nationality = \text{“}Swedish\text{”} \mid gender = \text{“}Male\text{”}) + \\
&\quad P(nationality = \text{“}Swedish\text{”} \mid gender = \text{“}Female\text{”})) + \\
&\quad P(hair = \text{“}Blond\text{”} \mid nationality = \text{“}American\text{”}) \times \\
&\quad (P(nationality = \text{“}American\text{”} \mid gender = \text{“}Male\text{”}) + \\
&\quad P(nationality = \text{“}American\text{”} \mid gender = \text{“}Female\text{”}))
\end{aligned} \tag{3.6}$$

The idea of VE is to walk over the tree in a post-order fashion – i.e. start from the leaves – and sum up each CPD row-wise. This avoids unnecessarily computing sums more than needed and ensures the inference process runs in linear time. The computation can be further increased by noticing that not all nodes in a BN are needed to obtain a given marginal distribution [Jensen et al., 1996]. Indeed the VE algorithm only has be run on a necessary subset of the tree's nodes which is commonly referred to as the *Steiner tree* [Hwang et al., 1992]. Extracting a Steiner tree from a tree can be done in linear time (see algorithm 1).

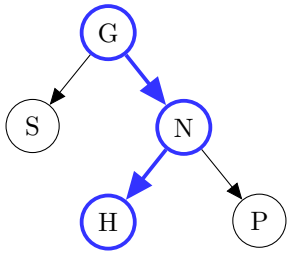

Figure 3-4: Steiner tree in blue containing nodes G, N, and H needed to compute H's marginal distribution

In our case we are using CPDs with intervals, meaning that we have to tailor the VE algorithm around them. Fortunately this is quite simple as we only have to check if a given value is an interval or not. Range queries can be handled by interpolating inside the interval.

Meanwhile, for equality queries we can assume that all distinct values in the interval are equally frequent.

Algorithm 1 Steiner tree extraction

```
1: function WALK(node, required, path, relevant)
2:     if required is empty then
3:         return {}
4:     else if node in nodes then
5:         required ← required \ {node}
6:         relevant ← relevant ∪ path
7:     end if
8:     path ← path ∪ {node}
9:     for child ∈ node.children() do
10:        relevant ← relevant ∪ WALK(child, required, path, relevant)
11:    end for
12:    return relevant
13: end function

14: function EXTRACTSTEINERTREE(tree, nodes)
15:    nodes ← nodes ∪ tree.root()
16:    relevant ← WALK(tree, nodes, {}, {})
17:    return tree.subset(relevant))
18: end function
```

3.3.4 Toy example

To illustrate our model, we will show how it works on a synthetic purpose-built dataset. The dataset contains three relations and is intendedly small. By design, it possesses attribute dependencies inside and between relation. The first relation contains passenger information whilst the second relates to air travel routes. The third relation is a fact table that indicates the routes taken by each passenger – a passenger can take more than one route. We will therefore detail the intermediary results to get a feel of what steps are involves. Before proceding, let us enumerate some records for each relation.

ID	Origin	Destination	Minutes
1	Stockholm	Boston	515
2	Stockholm	San Francisco	830
3	Stockholm	New-York	515
4	Fresno	Seattle	130
5	Fresno	San Francisco	60
6	Fresno	Seattle	110

Table 3.5: Routes

ID	Nationality	Gender	Hair
1	Swedish	Male	Blond
2	Swedish	Female	Blond
3	Swedish	Male	Blond
4	Swedish	Female	Brown
5	Swedish	Female	Blond
6	American	Male	Brown
7	American	Male	Dark
8	American	Female	Brown
9	American	Male	Brown
10	American	Female	Blond

Table 3.6: Passengers

Passenger ID	Route ID
0	0
0	1
0	2
1	0
1	1
1	2
2	2
3	0
4	1
5	3
5	5
6	3
7	3
7	5
8	4
9	4

Table 3.7: Flights

The first step of our model is to determine the structure of a Bayesian network per relation. To do so we first compute the mutual information between each pair of attributes. The following tables contains the mutual information values for the passengers and routes relations. The flights relation only contains key attributes and so we ignore it.

Attribute 1	Attribute 2	MI
Hair	Nationality	0.21801
Gender	Hair	0.07938
Gender	Nationality	0.02013

Table 3.8: Passengers MI values

Attribute 1	Attribute 2	MI
Minutes	Destination	1.32966
Minutes	Origin	0.69314
Destination	Origin	0.46209

Table 3.9: Routes MI values

The highest dependency in the first relation is between the hair and nationality attributes. This stems from the fact that 4 out of the 5 Swedes in the relation have blond hair. However there doesn't seem to be any particular link between American passengers and their hair colour. Dependencies seem to be more pronounced in the routes relation. Indeed there appears to be a strong dependency between the number of minutes a flight can take and the destination. This seems rather counter-intuitive because we know that the duration actually depends on the origin as well as the destination. This oddity is simply due to the small size of the relation. Indeed, apart from San Francisco each destination is only present once, meaning that there is almost a 1-to-1 relationship between the destination and the duration in minutes. There is also a dependency between the origin and the duration. This is more natural as it originates from the fact that Stockholm, being a capital city, has more long-haul flights than it's Fresno counterpart. Finally there is a dependency between the origins and the destinations simply because each airport has a different of available destinations. In other words, given a list of destinations, one should be able to deduct to which airport the list belongs.

The next step is to treat the mutual information values as weighted edges in a fully connected graph. For convenience we call this the *dependency graph*.

Figure 3-5: Dependency graphs for the passengers relation (left) and the routes relation (right) annotated with the according mutual information values

There isn't a noticeable difference between using any of the three most used MST construction algorithms – Kruskall, Prim, or Borůvka. For this example and the rest of the experiments we stuck with Kruskall's algorithm. Once the MST is obtained, we have to determine it's root in order to convert it into a direct graph.

Figure 3-6: Bayesian networks obtained for the passengers relation (left) and the routes relation (right)

Once the structure is determined, the conditional probability distributions have to be computed. Although there aren't any spatial concerns whatsoever, for the sake of this example we only keep the two most common values and put the rest of the values inside two equi-height bins.

515	**830**	**[60, 110]**	**(110, 130]**
$\frac{1}{3}$	$\frac{1}{6}$	2	1

Table 3.10: $P(minutes)$

	Stockholm	**Fresno**
515	1	0
830	1	0
[60, 110]	0	1
(110, 130]	0	1

Table 3.11: $P(origin \mid minutes)$

	New-York	**San Francisco**	**[Boston, Portland]**	**[Seattle, Seattle]**
515	0.5	0	1	0
830	0	1	0	0
[60, 110]	0	0.5	1	0
(110, 130]	0	0	0	1

Table 3.12: $P(destination \mid minutes)$

American	**Swedish**
0.5	0.5

Table 3.13: $P(nationality)$

	Blond	**Brown**	**[Dark, Dark]**
American	0.2	0.6	0.2
Swedish	0.8	0.2	0

Table 3.14: $P(hair \mid nationality)$

	Male	**Female**
Blond	0.4	0.6
Brown	0.5	0.5
[Dark, Dark]	1	0

Table 3.15: $P(gender \mid hair)$

As mentioned previously, the values under each interval column – e.g. [Boston, Portland] – are not probabilities, instead they indicate the number of distinct values that are present given a particular conditional value. For example there is one unique value in the interval [Boston, Portland] given that the number of minutes is equal to 515; there is also a probability of $\frac{1-(0+0)}{2} = 0.5$ that a destination is in the interval [Boston, Portland] given that the number of minutes is 515.

We now have all the elements required to conduct selectivity estimation. Let us start with a simple query over a single relation.

```
SELECT *
FROM passengers p
WHERE p.hair = 'Blond'
```

We simply want to know how many passengers have blond hair. The true selectivity of this query is 0.5. Because the query only targets the passengers relation we only have to look at the corresponding Bayesian network. The selectivity is therefore computed as so:

$$\begin{aligned} P(\text{“}Blond\text{”}) &= P(\text{“}Blond\text{”} \mid \text{“}American\text{”}) \times P(\text{“}American\text{”}) + \\ &\quad P(\text{“}Blond\text{”} \mid \text{“}Swedish\text{”}) \times P(\text{“}Swedish\text{”}) \\ &= 0.2 \times 0.5 + 0.8 \times 0.5 \\ &= 0.5 \end{aligned} \tag{3.7}$$

Paradoxically, it can be that computing the selectivity of a more complex query involving two predicates is simpler. For example consider the following query:

```
SELECT *
FROM passengers p
WHERE p.hair = 'Blond'
AND p.nationality = 'Swedish'
```

In this case we want to retrieve the passengers who are blond *and* Swedish. The textbook approach to compute this selectivity is to consider that hair and nationality are independent attributes: $P(\text{“}Blond\text{”}, \text{“}Swedish\text{”}) = P(\text{“}Blond\text{”}) \times P(\text{“}Swedish\text{”}) = 0.5 \times 0.5 = 0.25$. However, in our dataset, 4 out of the 10 passengers are blond and Swedish. Meanwhile, our Bayesian network knows this dependency and computes the selectivity as so:

$$\begin{aligned} P(\text{“}Blond\text{”}, \text{“}Swedish\text{”}) &= P(\text{“}Blond\text{”} \mid \text{“}Swedish\text{”}) \times P(\text{“}Swedish\text{”}) \\ &= 0.8 \times 0.5 \\ &= 0.4 \end{aligned} \tag{3.8}$$

Now let us turn to a query involving a join. For example let's see what happens if we join the passengers relation with the flights relation.

```
SELECT *
FROM passengers p, flights f
WHERE p.hair = 'Blond'
AND p.nationality = 'Swedish'
AND p.id = f.passenger_id
```

This query returns 8 rows. The selectivity procedure goes as follows:

$$\begin{aligned} P(passenger.id = flights.passenger\_id) &= min(\frac{1}{|passenger.id|}, \frac{1}{|flights.passenger\_id|}) \\ &= min(\frac{1}{10}, \frac{1}{10}) \\ &= 0.1 \end{aligned} \tag{3.9}$$

The selectivity estimate is thus $0.4 \times 0.1 \times 160 = 6.4$ where 160 is the size of the Cartesian product of the relations involved in the query. Alas we have underestimated the selectivity of the query ($6.4 < 8$). The reason behind this error is the fact that the distribution of the attributes of the passengers relation changes after the join. Indeed, as mentioned earlier 4 out of the 10 passengers are blond and Swedish. However, after the join it seems that 8 out the 16 flights were taken by blond Swedes. This can be due to the fact that blond people take more flights, or that Swedes take more flights, or both. In any case, this is a good example of the join uniformity assumption. In our case, the problem is that $P(nationality)$ refers to the distribution of the nationality in the passengers relation, not in the join between the passengers and the flights relations. Interestingly, if we knew $P(nationality)$ in the latter case, then our estimate would be perfect. In other words we wouldn't *also* have to know $P(hair \mid nationality)$ in both cases. This is due to the fact that our probabilities propagate through the network, meaning that the distribution of the root node determines all the rest of the distribution. Obtaining this probability would require a single `GROUP BY` over a single join between the relations involved. This is much more efficient than [Tzoumas et al., 2011]

proposal of looking for the highest dependency between pairs of attributes involving two relations.

Finally let us examine a query involving all three relations. The following query returns all the flights taken by blond Swedes and originating from Stockholm. The query returns 8 rows.

```
SELECT *
FROM passengers p, flights f, routes r
WHERE p.hair = 'Blond'
AND p.nationality = 'Swedish'
AND p.id = f.passenger_id
AND f.route_id = r.id
AND r.origin = 'Stockholm'
```

The attribute predicates belong to two different tables. We thus handle each of these predicates separately. We already obtained the selectivity for the part of the query involving the passengers relation – it was 0.4. As for the selectivity for the routes relation, it is determined as so:

$$\begin{aligned} P(\text{“}Stockholm\text{”}) &= P(\text{“}Stockholm\text{”} \mid 515) \times P(515) + P(\text{“}Stockholm\text{”} \mid 830) \times P(830) \\ &= 1 \times \frac{1}{3} + 1 \times \frac{1}{6} \\ &= 0.5 \end{aligned} \tag{3.10}$$

The selectivity of the join between the flights and the routes is:

$$\begin{aligned} P(flights.passenger\_id = routes.id) &= min(\frac{1}{|flights.passenger\_id|}, \frac{1}{|routes.id|}) \\ &= min(\frac{1}{6}, \frac{1}{6}) \\ &= \frac{1}{6} \end{aligned} \tag{3.11}$$

Our final selectivity estimate is thus $(0.4 \times 0.5) \times (0.1 \times \frac{1}{6}) \times (10 \times 16 \times 6) = 3.2$. Sadly this is well under the expected value of 8. The problem is that Swedes tend to take their flights from Stockholm. In other words the "nationality" and "origin" attributes are dependent. However these attributes are located in different relations and a 3-way join is needed in order to determine this dependency. Even if we used a method that captured dependencies in 2-way joins, we wouldn't have detected this dependency.

3.4 Linked Bayesian networks

3.4.1 Discussion

The Bayesian networks we have discussed in the previous sections are built on individual relations. Therefore, they don't allow capturing dependencies between attributes that belong to one or more different relations. Alas, these are the highest source of error, and are thus the most important ones to estimate correctly. This is also a much difficult task, because the number of attribute combinations is increasingly large. Additionally, attribute value distributions may change once the relation they pertain to is joined with one or more other relations. Many existing methods get around this issue by making the join uniformity assumption, which states that there is an equal chance for each tuple in a relation to be part of a join. In other words, it assumes that attribute values are simply repeated a constant amount of times, which essentially "stretches" the attribute values but doesn't modify the distribution whatsoever. Naturally, this isn't correct in practice.

It is worth taking a short epistemological detour to get a better understanding. From the philosophical point of view of logic, Bayesian networks are in essence *propositional*. *Propositional logic* is a subset of logic that deals with logical operators and the calculus rules that concern them. In a Bayesian network, the variables are part of a single entity. Propositional logic, which is also called zeroth-order logic, has an older sister called *first-order logic*. The latter supersedes zeroth-order logic, and adds a formalism for dealing with relationships and quantities. Our goal is to somewhat extend Bayesian networks through the principles of first-order-logic. A much more in-depth treatment is given in section 14.6 of [Russell and Norvig, 2002]. In fact, the latter discusses *relational probability models* [Friedman et al., 1999], which are an application of first-order logic to Bayesian networks. These were applied in a text-book manner in [Getoor et al., 2001]. Although sound in and

well-grounded in theory – one could even that they are elegant – they are just not amenable to be used in a cost models, where the computational is extremely tight.

Alas, as we are constrained by strict performance requirements, there is seemingly little we can do. Recall that the the AVI assumption implies that the statistical distribution of each attribute is estimated by itself, which leads to the following factorisation:

$$P(X_1, \ldots, X_n) \simeq \prod_{R_j} \Big(\prod_{X_i \in R_j} P(X_i) \Big) \tag{3.12}$$

Methods that work at the resolution of a single relation, which includes the Bayesian networks we have discusses until now, induce independence between relations:

$$P(X_1, \ldots, X_n) \simeq \prod_{R_j} \Big(\prod_{X_i \in R_j} P(X_i \,|\, Parent(X_i)) \Big) \tag{3.13}$$

Our goal is to model the full probability distribution by taking into account dependencies between attributes of different relations, which can summarised as so:

$$P(X_1, \ldots, X_n) \simeq \prod_{X_i} P(X_i \,|\, Parent(X_i)) \tag{3.14}$$

Note that equation 3.14 captures more information than equation 3.12. Modeling the data by taking into account conditional dependencies thus guarantees that the resulting selectivity estimates are at least as accurate as when assuming independence between attributes. In the previous section, we made the assumption that attribute values of different relations are independent. Additionally, we assumed that each attribute value distribution remains the same when the relation it belongs to is joined with another relation. This is a result of the join uniformity assumption and is a huge source of error. Indeed, the distributions of an attribute's values before and after a join are not necessarily the same.

As an example, imagine an e-commerce website where registered customers are stored in a database alongside with purchases. Each customer can make zero or more purchases whilst each purchase is made by exactly one customer. Some customers might be registered on the website, but might not have made a purchase. If the customers and purchases relations are joined together, then the customers who have not made any purchases will not be included in the join. Therefore, the attributes from the customers relation will have different value distributions when joined with the purchases relation. Note, however, that the attribute

value distributions from the purchases relation will not be modified. This stems from the fact that the join between the customers and purchases relations is a one-to-many join. We will now attempt to explain how we can exploit this property to capture dependencies across relations.

3.4.2 The attribute dependency preservation assumption

Let us assume we have two relations R and S that share a primary/foreign key relationship. That is, S contains a foreign key which references the primary key of R. This means that each tuple from S is associated with one tuple from R. Inversely, each tuple from R is associated with zero more tuples from S. A direct consequence is that the size of the join $R \bowtie S$ is equal to $|S|$. The join uniformity assumptions implies that the probability for a tuple r from relation R to be present in $R \bowtie S$ follows a uniform distribution. In statistical terms, that is:

$$P(r \in R \bowtie S) \sim \mathcal{U}(\frac{1}{|R|}) \tag{3.15}$$

Consequently, the expected number of times each tuple from R will be part of $R \bowtie S$ is $\frac{|S|}{|R|}$. Let us now denote by $P_R(A)$ the value distribution of attribute A in relation R. We will also define $P_{R \bowtie S}(A)$ as the value distribution of attribute A in join $R \bowtie S$. The join uniformity assumption thus implies that the distribution of A's values before and after R is joined with S are equal:

$$P_R(A) = P_{R \bowtie S}(A) \tag{3.16}$$

Furthermore, assume we have found and built the following factorised distribution over attributes A, B, and C from relation R:

$$P_R(A, B, C) \simeq P_R(A \,|\, B) \times P_R(B \,|\, C) \times P_R(C) \tag{3.17}$$

If we hold the join uniformity assumption to be true, then we can use the factorised distribution to estimate selectivities for queries involving A, B and C when R is joined with S without any further modification. The issue is that this is an idealised situation that has no reason to occur in practice. On the contrary, it is likely are that some tuples from R

will be more or less present than others. However, we may assume that after the join the attribute value dependencies implied by our factorisation remain valid within each relation. We call this the *attribute value dependency preservation* assumption. The idea is that if attributes A, B and C are dependent in a certain way in relation R, then there is not much reason to believe that these dependencies will disappear once R is joined with S. Although this may not necessarily always occur in practice, it is still a much softer assumption than those usually made by cost models.

To illustrate, let us consider a toy database composed of the following relations: customers with attributes $\{nationality, hair, salary\}$, shops with attributes $\{name, city, size\}$, purchases with attributes $\{day\ of\ week\}$. Moreover, assume that the purchases relation has two foreign keys, one that references the primary key of customers and another with that of shops. The purchases relation can thus be seen as a fact table whilst customers and shops can be viewed as a dimension table. In what follows we will use the shorthand C the customers relation, S for the shops relation, and P for the purchases relation.

In the customers relation, there are Swedish customers and a lot of them have blond hair. We might capture this property in a Bayesian network with the conditional distribution $P_C(hair \,|\, nationality)$, which indicates that hair colour is influenced by nationality. We could suppose that the fact that Swedish people have blond hair is still true once the customers relation is joined with the purchases relation. In other words, the hair colour shouldn't change the rate at which Swedish customers make purchases. However, we may rightly assume that the number of purchases will change according to the nationality of each customer. Mathematically, we are saying the following:

$$P_{C \bowtie P}(hair, nationality) = P_C(hair \,|\, nationality) \times P_{C \bowtie P}(nationality) \tag{3.18}$$

In other words, because we assume that $P_C(hair \,|\, nationality) = P_{C \bowtie P}(hair \,|\, nationality)$, then we know $P_{C \bowtie P}(hair, nationality)$ – i.e., we assume that their conditional distribution remains unchanged after the join. An immediate consequence is that we get to know the $P_{C \bowtie P}(hair)$ distribution for free. Indeed, by summing over the nationalities, we obtain:

$$P_{C \bowtie P}(hair) = \sum_{nationality} P_C(hair \,|\, nationality) \times P_{C \bowtie P}(nationality) \tag{3.19}$$

To demonstrate why our assumption is useful for the purpose of selectivity estimation, let us use the example data in tables 3.16 and 3.17.

Customer	Nationality	Hair
1	Swedish	Blond
2	Swedish	Blond
3	Swedish	Brown
4	American	Blond
5	American	Brown

Table 3.16: Customers relation

Purchase	Shop	Customer
1	1	1
2	1	1
3	2	1
4	2	1
5	3	2
6	4	3
7	5	5

Table 3.17: Purchases relation, which contains a foreign key that is related to the primary key of the customers relation

Let us say we wish to know how many purchases are made by customers who are both blond and Swedish. The straightforward way to do this is to count the number of times "Blond" and "Swedish" appear together in the join $C \bowtie P$:

$$P_{C \bowtie P}(hair = Blond, nationality = Swedish) = \frac{5}{7} \tag{3.20}$$

The fraction $\frac{5}{7}$ is the true amount of purchases that were made by Swedish customers with blond hair – said otherwise this is the selectivity of the query. Obtaining it requires scanning the rows resulting from the join of C with P. In practice this can be very burdensome, especially when queries involve many relations. If we assume that the join uniformity assumption holds – in other words we assume that the value distributions of *nationality* and *hair* do not change – then we can simply reuse the Bayesian network of the customers relation:

$$\begin{aligned} P_{C \bowtie P}(Blond, Swedish) &\simeq P_C(Blond \,|\, Swedish) \times P_C(Swedish) \\ &\simeq \frac{2}{3} \times \frac{3}{5} \\ &\simeq \frac{2}{5} \end{aligned} \tag{3.21}$$

In this case, making the join uniformity independence assumption makes us underestimate the true selectivity by 44% ($1 - \frac{2}{5} \times \frac{7}{5}$). Some of this error is due to the fact that

the nationality attribute values are not distributed in the same way once C and P are joined – indeed in this toy example Swedish customers make more purchases than American ones. However, if we know the distribution of the nationality attribute values, i.e., $P_{C \bowtie P}(nationality)$, then we can enhance our estimate in the following manner:

$$\begin{aligned} P_{C \bowtie P}(Blond, Swedish) &\simeq P_C(Blond \,|\, Swedish) \times P_{C \bowtie P}(Swedish) \\ &\simeq \frac{2}{3} \times \frac{6}{7} \\ &\simeq \frac{4}{7} \end{aligned} \tag{3.22}$$

Now our underestimate has shrunk to 20%. The only difference with the previous equation is that we have replaced $P_C(Swedish)$ with $P_{C \bowtie P}(Swedish)$. Note that we did not have to precompute $P_{C \bowtie P}(Blond, Swedish)$. Indeed, we assumed that the dependency between nationality and hair doesn't change once C and P are joined, which stems from our dependency preservation assumption. Note that, in our toy example, the assumption is slightly wrong because blond customers have a higher purchase rate than brown haired ones, regardless of the nationality. Regardless, this assumption is still much softer than the join uniformity and attribute value independence assumptions.

The newly introduced assumption is softer than the join uniformity assumption because it allows attribute value distributions to change after a join. Statistically speaking, instead of assuming that tuples appear in a join following a uniform distribution, we are saying that the distribution of the tuples is conditioned on a particular attribute (e.g., the nationality of the customers dictates the distribution of the customers in the join between shops and customers). We also assume that attribute value dependencies with each relation are preserved through joins (e.g., hair colour is still dependent on nationality). The insight is that in a factorised distribution, the top-most attribute is part of any query. For instance, in the distribution $P(A \,|\, B) \times P(B \,|\, C) \times P(C)$, every query involving any combination of A, B, and C will necessarily involve $P(C)$. We will now see how our newly introduced attribute value dependency preservation assumption can be used to link Bayesian networks from different relations together, and as such relax the join uniformity and attribute value independence assumptions at the same time.

3.4.3 Linking Bayesian networks

As explained in the previous subsection, if a purchases relation has a foreign key that references a primary key of another relation named customers, then the distribution of purchases' attribute values will not change after joining customers and purchases. However the distribution of customers' attribute values will change if purchases' foreign key is skewed, which is always the case to some degree. If we use Bayesian networks independently, then the Bayesian network built on customers would not be accurate when estimating selectivities for queries involving customers and purchases. This is because it would assume the distributions of the attribute values from customers are preserved after the join, which is a consequence of the join uniformity assumption. Moreover, because of the AVI assumption, we would not be capturing the existing dependencies between customers's attributes and purchases's attributes because their respective attributes are assumed to be independent with those of the opposite relation. On the other hand, if we join customers and purchases and build a Bayesian network on top of the join then we will capture the cross-relation attribute value dependencies, but at too high a computational cost [Getoor et al., 2001, Tzoumas et al., 2011]. Up to now, we have only mentioned the case where there one join occurs, but the same kind of issues occur for many-way joins – including chain-joins and star-joins [Lindsay et al., 1999].

If the attribute value distributions of customers and purchases are estimated using Bayesian networks that possess a tree structure, then we only have to include the dependencies of a subset customers's attributes with those of purchases. Specifically, we only have to include the root attribute of customers's Bayesian network into that of purchases. Indeed, because customers's Bayesian network is a tree, then all of its nodes are necessarily linked to the root. If we know the distribution of the root attribute's values *after* customers is joined with purchases, then, by making the attribute value dependency preservation assumption earlier introduced, we automatically obtain the distribution of the rest of customers's attribute. In other words, if the distribution of an attribute's values is modified when the relation it belongs to is joined with another relation, then we assume that all the attributes that depend on it have their value distributions modified in the exact same manner. This is another way of saying that the conditional distributions remain the same.

Let us now see how this works on our toy database consisting of relations customers, shops, and purchases. Following the methodology from the previous section, we would have built one Bayesian network per relation. Each Bayesian network would necessarily have been a tree as a consequence of using the Chow-Liu algorithm [Chow and Liu, 1968]. Depending on the specifics of the data, we might have obtained the Bayesian networks shown in figure 3-7.

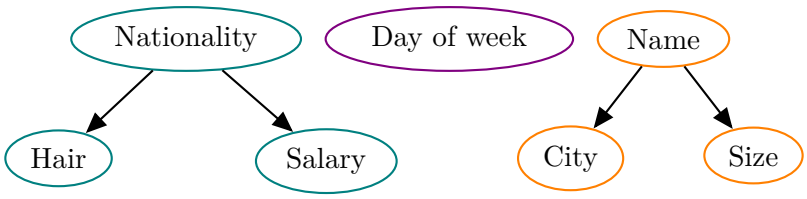

Figure 3-7: Separate Bayesian networks of customers, shops, and purchases

Furthermore, let us consider the following SQL query:

```
SELECT *
FROM customers, shops, purchases
WHERE customers.id = purchases.customer_id
AND shops.id = purchases.shop_id
AND customers.nationality = 'Japanese'
AND customers.hair = 'Dark'
AND shops.name = 'Izumi'
```

If we were to estimate the amount of tuples that satisfy the above query using the Bayesian networks from figure 3-7, then we would estimate the query selectivity in the following manner:

$$\begin{aligned} P(Dark, Japanese, Izumi) = P_C(Dark \mid Japanese) \\ \times P_C(Japanese) \\ \times P_S(Izumi) \end{aligned} \tag{3.23}$$

On the one hand, the conditional distribution $P_C(Dark \mid Japanese)$ captures the fact that Japanese people tend to have dark hair inside the customers relation. Graphically this is represented by the arrow that points from the "Nationality" node to the "Hair"

node in figure 3-7. On the other hand, our estimate ignores the fact that shops in Japan, including "Izumi", are mostly frequented by Japanese people. The reason why is that we have one Bayesian network per relation, instead of a global network spanning all relations, and are thus not able to capture this dependency. Regardless of the missed dependency, this simple method is still more accurate than assuming total independence. Indeed the AVI assumption would neglect the dependency between *hair* and *nationality*, even though both attributes are part of the same relation. Meanwhile assuming relational independence is convenient because it only requires capturing dependencies within relations, but it discards the dependency between *nationality* and *city*. We propose to capture said dependency by adding nodes from the Bayesian networks of customers and shops to the Bayesian network of purchases. Specifically, for reasons that will become clear further on, we add the roots of the Bayesian networks of customers and shops (i.e., *nationality* and *name*) to the Bayesian network of purchases. This results in the linked Bayesian network shown in figure 3-8.

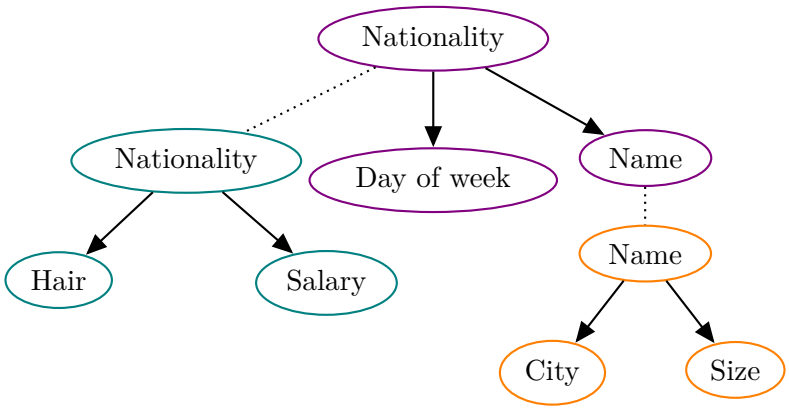

Figure 3-8: Linked Bayesian network of customers, shops, and purchases

In this new configuration, we still have one Bayesian network per relation. The difference is that the Bayesian network of purchases includes the root attributes of both customers and shops's Bayesian networks. In other words, we have joined the purchases relation with the customers and shops and we have then built a Bayesian network for purchases that now includes attributes from customers and shops. A linked Bayesian network is thus a set of separate Bayesian networks where some of the attributes are duplicated in two related networks. In practice, this means that we now know the distribution of the *nationality* and *name* attribute values once the relations they belong to have been joined with purchases. Meanwhile, we also know their distributions when these relations are not

joined with purchases. In other words, we store two distributions for each root attribute, one before the join and one afterwards. The distribution of a root attribute in a Bayesian network is nothing more than a one-dimensional histogram. This means that storing two distributions for each root attribute doesn't incur any significant memory burden.

The configuration shown in figure 3-8 has two immediate benefits over the one presented in figure 3-7. First of all, we are now able to determine if the percentage of Japanese in the purchases relation is different from the one in the customers relation. Indeed, we do not have to assume the distribution remains the same after the join now that we know the distribution of *nationality*'s values when customers is joined with purchases. A key observation is that we get to know something about the distribution of the *hair* attribute values when customers is joined with purchases. That is to say, because we know how the distribution of *nationality* attribute values changes after the join, then we also know something about the *hair* attribute values because both attributes are dependent within the customers relation. This stems from the fact that we assume that the conditional distribution $P(hair \,|\, nationality)$ is preserved after the join. Mathematically this translates to:

$$P_{C \bowtie P}(hair, nationality) = P_C(hair \,|\, nationality) P_{C \bowtie P}(nationality) \tag{3.24}$$

Although, in practice, we expect the dependency preservation assumption to not always be verified, we argue that it is a much weaker assumption than assuming total relational independence. The second benefit is that we can now take into account the fact the Japanese people typically shop in Japanese shops, even though the involved attributes belong to relations that are not directly related. This happens because the *name* attribute is now part of purchases's Bayesian network as well as that of shops. Formally the query selectivity can now be expressed as so:

$$\begin{aligned} P_{C \bowtie P \bowtie S}(Dark, Japanese, Izumi) = {} & P_C(Dark \,|\, Japanese) \\ & \times P_{P \bowtie S}(Izumi \,|\, Japanese) \\ & \times P_{C \bowtie P}(Japanese) \end{aligned} \tag{3.25}$$

Let us now consider the following SQL query where the only difference with the previous query is that are filtering by *city* instead of by *name*:

```
SELECT *
FROM customers, shops, purchases
WHERE customers.id = purchases.customer_id
AND shops.id = purchases.shop_id
AND customers.nationality = 'Japanese'
AND customers.hair = 'Dark'
AND shops.city = 'Osaka'
```

In this case, our linked Bayesian network would estimate the selectivity as so:

$$\begin{aligned} P(Dark, Japanese, Osaka) = {} & P_C(Dark \mid Japanese) \\ & \times \sum_{name} P_{P \bowtie S}(Osaka \mid name) P_P(name \mid Japanese) \\ & \times P_{C \bowtie P}(Japanese) \end{aligned} \tag{3.26}$$

This is a simple application of Bayesian network arithmetic [Jensen et al., 1996]. The reason why there is a sum is that we have to take into account all the shops that are located in Osaka because none of them in particular has been specified in the SQL query. Note that our linked Bayesian network is still capable of estimating selectivities when only a single relation is involved. For example, we only need to use $P_P(nationality)$ when the customers relation is joined with purchases relation. If only the customers relation is involved in a query, then we can simply use $P_C(nationality)$ instead of $P_P(nationality)$. We discuss these two points in further detail in subsection 3.4.5.

Linked Bayesian networks thus combine the benefits of independent Bayesian networks, while having the benefit of softening the join uniformity assumption as well as the attribute value independence assumption. We will now discuss how one may obtain a linked Bayesian network in an efficient manner.

3.4.4 Building linked Bayesian networks

A linked Bayesian network is essentially a set of Bayesian networks. Indeed, our method consists in taking individual Bayesian networks and linking them together in order to obtain one single Bayesian network. This linking process is detailed in the next subsection. In our case, by only including the root attribute of each relation into the Bayesian network of its parent relation, we ensure that the final network necessarily has a tree topology. Performing inference on a Bayesian network with a tree topology can be done in linear time using the sum-product algorithm [Kschischang et al., 2001]. Building a linked Bayesian network involves building the Bayesian networks of each relation in a particular order. Indeed, in our example, we first have to build the Bayesian networks of the customers and shops relations in order to determine the roots that are to be included in the Bayesian network of the purchases relation. To build the purchases Bayesian network, we first have to join the root attributes (i.e., *nationality* and *name*) of the first two Bayesian networks (i.e., customers and shops) with the purchases relation. Naturally, performing joins incurs an added computational cost. However, we argue that joins are unavoidable if one is to capture attribute value dependencies across relations. Indeed, if joins are disallowed whatsoever, then there is basically no hope of measuring dependencies between attributes of different relations. Our methodology requires performing one left-join per primary/foreign key relationship, whilst only requiring to include one attribute per join, which is as cost-effective as possible.

The specifics of the procedure we used to build the linked Bayesian network are given in algorithm 2. We assume the algorithm is given a set of relations. In addition, the algorithm is provided with the set of primary/foreign key relationships in the database (e.g., purchases has a foreign key that references customers' primary key and another that references shops's primary key). This set of primary/foreign key relationships can easily be extracted from any database's metadata. The idea is to go through the set of relations and check if the Bayesian networks of the dependent relations have been built. In this implementation a `while` loop is used to go through the relations in their topological order, from bottom to top. The Bayesian networks are built using the $BuildBN$ function, which is in effect the algorithm used for the Bayesian networks of the previous section. In short, the $BuildBN$ function works in three steps:

1. Build a fully-connected, undirected weighted graph, where each node is an attribute and each vertex's weight is the *mutual information* between two attributes.

2. Find the *maximum spanning tree* (MST) of the graph.

3. Orient the MST in order to obtain a tree by choosing a root.

The *BuildBN* function produces a Bayesian network with a *Chow-Liu tree* structure. This tree has the property of being the tree which stores the maximum amount of information out of all the legal trees. In our algorithm, the first pass of the `while` loop will build the Bayesian networks of the relations that have no dependencies whatsoever (e.g., those who's primary key isn't referenced by any foreign key). The next pass will build the Bayesian networks of the relations that contain primary keys referenced by the foreign keys of the relations covered in the first pass. The algorithm will necessarily terminate once each relation has an associated Bayesian network; it will take as many steps as there are relations in the database.

Algorithm 2 Linked Bayesian networks construction

```
1:  function BuildLinkedBN(relations, relationships)
2:      lbn ← {}
3:      built ← {}                          ▷ Records which relations have been processed
4:      while |lbn| < |relations| do
5:          queue ← relations \ built       ▷ Relations which don't have a BN
6:          for each relation ∈ queue do
7:              if relationships[relation] \ built = ∅ then
8:                  for each child ∈ relationships[relation] do
9:                      relation ← relation ⋈ child.root
10:                 end for
11:             end if
12:             lbn ← lbn ∪ BuildBN(relation)
13:             built ← built ∪ relation
14:         end for
15:     end while
16:     return lbn
17: end function
```

3.4.5 Selectivity estimation

The algorithm for producing selectivity estimates using linked Bayesian networks is extremely similar to the variable elimination procedure used for single Bayesian networks. The key insight is that we can fuse linked Bayesian networks into a single Bayesian network. Indeed, in our building process we have to make sure to include the root attribute of each relation's Bayesian network into it's parent Bayesian's network. This allows to link each pair of adjacent Bayesian networks together via their shared attribute. In figure 3-8, these implicit links are represented with dotted lines. The purchases and customers relation have in common the *nationality* attribute, whereas the shops and purchases relations have in common the *name* attribute. The resulting "stiched" network is necessarily a tree because each individual Bayesian network is a tree and each shared attribute is located at the root of each child network.

Algorithm 3 Selectivity estimation using a linked Bayesian network

```
1: function InferSelectivity(lbn, query)
2:     relations ← ExtractRelations(query)
3:     relevant ← PruneLinkedBN(lbn, relations)
4:     linked ← LinkNetworks(relevant)
5:     selectivity ← ApplySumProduct(linked)
6:     return selectivity
7: end function
```

The pseudocode for producing selectivity estimates is given in algorithm 3. The first step of the selectivity estimation algorithm is to identify which relations are involved in a given query. Indeed each SQL query will usually involve a subset of relations, and thus we only need to use the Bayesian networks that pertain to said subset. The *PruneLinkedBN* thus takes care of removing the unnecessary Bayesian networks from the entire set of Bayesian networks. Naturally, in practice, and depending on implementation details, this may involve simply loading in memory the necessary Bayesian networks. In any case, the next step is to connect the networks into a single one. This necessitates looping over the Bayesian networks in topological order – in the same exact fashion as algorithm 2 – and linking them along the way. Linking two Bayesian networks together simply involves replacing the attribute they have in common with the child Bayesian network. For instance, in figure

3-8, the *nationality* attribute from the purchases Bayesian network will be replaced by the customers Bayesian network. This is because we are interested in the distribution of the attributes after the join, not before. The resulting tree thus approximates the distribution of attribute values inside the $(customers \bowtie purchases \bowtie shops)$ join instead of estimating selectivities inside each relation independently, as is done in textbook cost models. The result of this linking process is exemplified in figure 3-9, which shows the unrolled version of the linked Bayesian network shown in figure 3-8. Finally, once the Bayesian networks have been linked together, the sum-product algorithm [Kschischang et al., 2001] can be used to output the desired selectivity.

Our method for estimating selectivities is very efficient. The main reason is because we only have to apply the sum-product algorithm once, whereas the methodology from the previous section requires applying the algorithm once for each relation involved in the query at hand. This difference is made clear when comparing equations 3.12 and 3.14. Furthermore, the sum-product algorithm is much more efficient in the case of trees than the clique tree algorithm from [Tzoumas et al., 2011]. We confirm these insights in the benchmarks section.

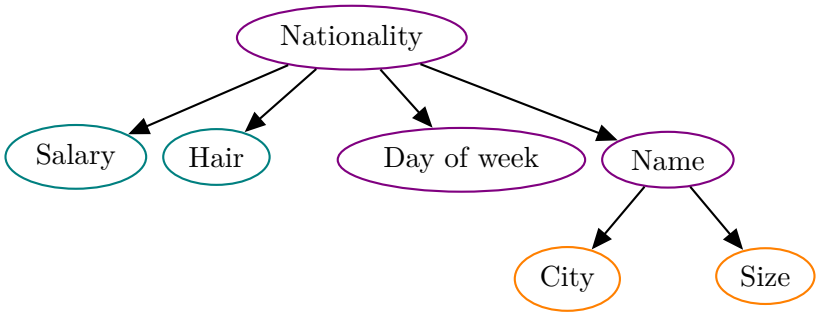

Figure 3-9: Unrolled version of figure 3-8

3.4.6 Including more than just the roots

Our model assumes that the dependencies between attribute values within a relation are preserved when a join occurs. Indeed we assume that tuples are uniformly distributed inside a join *given* each value in the root attribute. One may wonder why we have to stop at the root. Indeed, it turns out that we can include more attributes in addition to the root of each child Bayesian network when building a parent Bayesian network. For example, consider the linked Bayesian network shown in figure 3-10. In this configuration we include the *salary*

attribute as well as the *nationality* attribute in the Bayesian network of the purchases relation. By doing so we obtain a new conditional distribution $P(salary|nationality)$ which tells us the dependence between *salary* and *nationality* after customers has been joined with purchases.

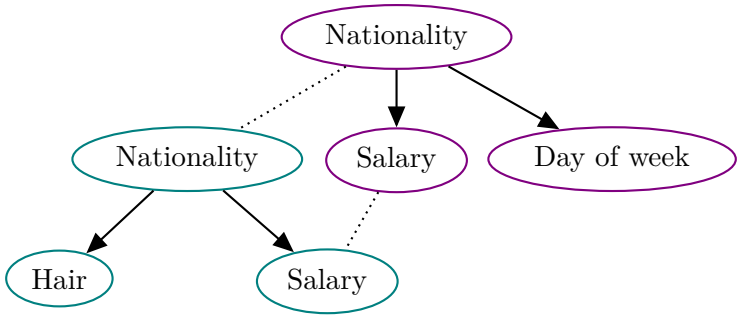

Figure 3-10: Linked Bayesian network of customers and purchases

The linked Bayesian network shown in 3-10 is valid because we can unroll it in order to obtain a single tree, just as we did earlier on when we only included the *nationality* attribute. However, the *salary* attribute can be included in the purchases Bayesian network only because of the fact that the *nationality* attribute is included as well. Indeed, if the *nationality* attribute was not included, then linking customers and purchases together would have resulted in a Bayesian network which would not necessarily have been a tree. In this case, we would not have been able to compute $P(salary \mid nationality)$ in purchases's Bayesian network. In other words, a node can be included in a parent Bayesian network only if all of it's conditioning attributes are included as well. Assuming a child Bayesian network has n nodes then we can include a number $k \in \{0, \ldots, n\}$ of it's nodes in the parent Bayesian network. If $k = 0$, then we simply keep each Bayesian network separate, which brings us back to using independent Bayesian networks. If $k = 1$, then we only include the root of each child Bayesian network, which is the case we have discussed up to now. If $k = n$, then we will include all the child's attributes in the parent BN, which is somewhat similar to the global methods presented in [Getoor et al., 2001] and [Tzoumas et al., 2011]. On the one hand, increasing k will produce larger parent Bayesian networks that capture more attribute value dependencies but also incur a higher computational cost. On the other hand, lower values of k will necessitate less computation but will assume more strongly that dependencies are preserved through joins. The k parameter is thus a practical parameter for compromising between selectivity estimation accuracy and computational requirements.

Notice that different values of k can be used for each pair of relations. For instance, we might want to increase k if we notice that the cost model makes very bad estimates for a certain relation. This can be decided upon as deemed fit, be it manually or via automated DBA [Van Aken et al., 2017].

3.4.7 Summary

The method we propose attempts to generalise existing selectivity estimation methods based on Bayesian networks. Following the methodology from the first section, we build one Bayesian network per relation using Chow-Liu trees. The only difference is that we include a set of attributes from the child relations into the Bayesian network associated with each parent relation. The set of included attributes depends on a chosen parameter k and the structure of each child relation's Bayesian network. Many distributions can be obtained for free because of the fact that each Bayesian network is a tree in which the root attribute conditions the rest of the attributes. This requires assuming that attribute value dependencies are preserved through joins. This assumption, although not always necessarily true, is much softer than the join uniformity as well as the attribute value independence assumptions. The resulting Bayesian networks are thus able to capture attribute value dependencies across relations, as well as inside individual relations. The downside of our method is that requires it requires performing joins between pairs of relations, albeit with only a couple of attributes at a time. Note however that such joins are performed offline during database downtime, and therefore have no impact on the runtime performance. Moreover, we argue that joins have to be performed if one is to capture dependencies across relations, and therefore are unavoidable. The major benefit of our method is that it only requires including a single attribute per join, and yet it brings a great deal of information for free through transitivity thanks to our newly introduced assumption. Moreover, our method can still benefit from the efficient selectivity estimation procedure presented in the first section because of the preserved tree structure. Finally, our method is able to generalise existing methods based on Bayesian networks through a single parameter which determines the amount of dependency to measure between the attributes of relations that share a primary/foreign key relationship.

3.5 Experimental results

We evaluate our proposal on an extensive workload derived from the JOB benchmark [Leis et al., 2015]. The JOB benchmark consists of 113 SQL queries, along with an accompanying dataset extracted from the IMDb website. The dataset therefore consists of non-synthetic data, whereas other benchmarks such as TPC-DS [Poess et al., 2002] are based on synthetic data. The dataset is challenging because it contains skewed distributions and exhibits many correlations between attributes, both across and inside relations. The JOB benchmark is now an established and reliable standard for evaluating and comparing cost models. The dataset and the queries are publicly available [1].

During the query optimisation phase, the cost model has to estimate the selectivity of each query execution plan (QEP) enumerated by the query optimiser. Query optimisers usually build QEPs in a bottom-up fashion [Chaudhuri, 1998]. Initially, the cost model will have to estimate selectivities for simple QEPs that involve a single relation. It will then be asked to estimate selectivities for larger QEPs involving multiple joins and predicates. We decided to mimic this situation by enumerating all the possible sub-queries for each of the JOB benchmark's queries, as detailed in [Chaudhuri et al., 2009]. For example, if a query pertains to 4 relations, we will enumerate all the possible sub-queries involving 1, 2, 3, and all 4 relations. We also enumerate through all the combinations of filter conditions. To do so, we represented each query as a graph with each node being an attribute and each edge a join. We then simply had to retrieve all the so-called *induced subgraphs*, which are all the subgraphs that can be made up from a given graph. Each induced subgraph was then converted back to a valid SQL statement. This procedure only takes a few minutes and yields a fairly large amount of queries; indeed a total of 5,122,790 subqueries can be generated for the JOB benchmark's 113 queries. Tables 3.18 and 3.19 provide an overview of the contents of our workload.

The general goal of our experiments is to detail the pros and cons of our method with respect to the textbook approach from [Selinger et al., 1979] and some state-of-the-art methods that we were able to implement. Most industrial databases still resort to using textbook approaches, which are thus important to be compared with. Specifically our experiments solely focus on the selectivity estimation module, not on the final query execution

[1] JOB dataset and queries: https://github.com/gregrahn/join-order-benchmark/

Joins	Amount
0	889
1-5	177,309
6-10	1,175,120
11-15	2,060,614
16-20	1,320,681
21-25	388,177

Table 3.18: Query spread per number of join conditions

Filters	Amount
1	261,440
2	763,392
3	1,301,840
4	1,380,329
5	923,481
6	384,285
7	94,855
8	12,496
9	672

Table 3.19: Query spread per number of filter conditions

time. We assume that improving the selectivity estimates will necessarily have a beneficial impact on the accuracy of the cost model and thus on the query execution time. Naturally, the estimation has to remain reasonable. This seems to be a view shared by many in the query optimisation community [Leis et al., 2015]. Indeed, many papers that deal with selectivity estimation, both established and new, do not measure the impact on the final query execution [Chen and Roussopoulos, 1994, Poosala et al., 1996, Poosala and Ioannidis, 1997, Tzoumas et al., 2011, Vengerov et al., 2015, Dutt et al., 2019].

We compared our proposal with a few promising state-of-the-art methods as well as the selectivity estimation module from the PostgreSQL database system. PostgreSQL's selectivity estimation module is a fair baseline as it is a textbook implementation of the decades old ideas from [Selinger et al., 1979]. We used version 10.5 of PostgreSQL and did not tinker with the default settings. Additionally, we did not bother with building indexes, as these have no consequence on the selectivity estimation module. A viable selectivity estimation method should be at least as accurate as PostgreSQL, without introducing too much of a computational cost increase. We implemented basic random sampling [Olken and Rotem, 1986], which consists in executing a given query on a sample of each relation in order to extrapolate a selectivity estimate. Basic random sampling is simple to implement, but isn't suited for queries that involve joins because of the empty-join problem, as explained in section 2. However many sampling methods that take into account the empty-join problem have been proposed. We implemented one such method, namely *correlated sampling* [Vengerov et al., 2015]. Correlated sampling works by hashing related primary and foreign

keys and discards the tuples of linked relation where the hashes disagree. We also implemented MSCN, which is the deep learning method that is presented in [Kipf et al., 2019b]. Finally we implemented the Bayesian network approach from [Tzoumas et al., 2011]. This method differs from ours in that it is a global approach that builds one single Bayesian networks over the entire set of relations. Although a global approach is able to capture more correlations than ours, it require more computation. We compared our method with different values for the k parameter presented in section 3.4.6. Note that choosing $k = 0$ is equivalent to using independent Bayesian networks. Increasing k is expected to improve the accuracy of the selectivity estimates but deteriorates the computational performance. The k parameter can thus be used to trade between accuracy and computational resources depending on the use case and the constraints of the environment.

3.5.1 Selectivity estimation accuracy

We first measured the accuracy of the selectivity estimates for each method by comparing their estimates with the true selectivity. The true selectivity can be obtained by executing the query and counting the number of tuples in the result. The appropriate metric for such a comparison is called the q-error [Moerkotte et al., 2009, Leis et al., 2018], and is defined as so:

$$q(y, \hat{y}) = \frac{max(y, \hat{y})}{min(y, \hat{y})} \tag{3.27}$$

where y is the true selectivity and $\hat{y}$ is the estimated selectivity. The q-error thus simply measures the multiplicative error between the estimate and the truth. The q-error has the property of being symmetric, and will thus be the same whether $\hat{y}$ is an underestimation or an overestimation. Moreover the q-error is scale agnostic (e.g., $\frac{8}{3} = \frac{24}{9}$), which helps in comparing errors over results with different scales.

Figure 3-11 shows the q-errors made by each method for all the queries of the workload derived from the JOB benchmark. The y axis represents the q-error associated with each query. Meanwhile the x axis denotes the amount of queries that have less than a given q-error. For instance, PostgreSQL managed to estimate the selectivity of two million queries with a q-error of less than 10 for each query. The curves thus give us a detailed view into the distribution of the q-errors for each method. While the curves seem to exhibit a linear

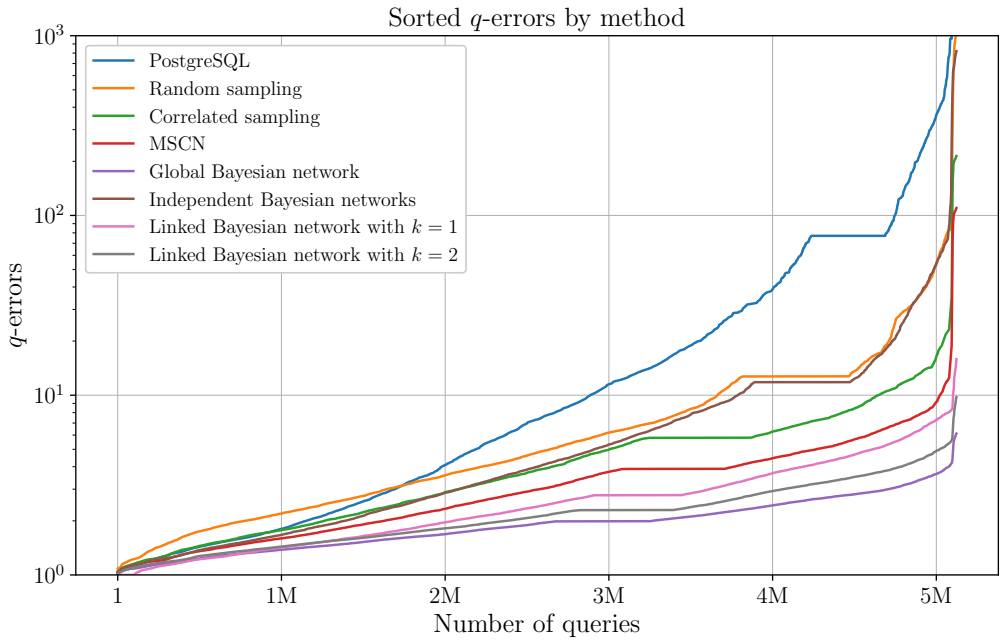

Figure 3-11: Sorted q-errors for all queries by method on the JOB workload

trend, one must note that the scale of the y axis is logarithmic. The figure gives us a global idea of the accuracy of each method in comparison with the others. The mean, maximum, and meaningful quantiles of the q-errors are given in table 3.20.

Table 3.20: q-error statistics for each method on the JOB workload

	median	90th	95th	99th	max	average
PostgreSQL	7.32	77.01	185.84	707.21	10906.17	77.01
Sampling	4.79	16.45	33.17	81.34	1018.43	12.71
Correlated sampling	3.83	9.63	12.63	22.72	214.1	5.79
MSCN	2.99	6.12	7.47	12.49	110.56	3.89
Global BN	1.95	2.92	3.22	4.01	7.45	1.99
Independent BN	4.0	15.36	32.9	76.91	820.46	11.82
Linked BN $k = 1$	2.41	5.03	6.15	8.07	21.09	2.79
Linked BN $k = 2$	2.13	3.7	4.26	5.23	12.6	2.3

The overall worst method is the cost model used by PostgreSQL. This isn't a surprise, as it assumes total independence between attributes, both within and between relations. It is interesting to notice that the q-errors made by PostgreSQL's cost model can be extremely high, sometimes even reaching the tens of thousands. In this case, the query optimiser

is nothing short from blind because the selectivity estimates are extremely unreliable. Although this doesn't necessarily mean that the query optimiser will not be able to find a good query execution plan, it does imply that finding a good execution plan would be down to luck [Leis et al., 2018]. One may even wonder if estimating a selectivity by picking a random number between 0 and 1 might do better. Using our method with k equal to 0 is equivalent to the methodology proposed in the previous sections. Indeed, if no attributes are shared by the Bayesian networks of each relation, then it is as if we considered attribute value dependencies within each relation but not between relations. As expected, the performance is similar to that of random sampling because both methods capture dependencies within a relation but not between relations. Correlated sampling performs a bit better because it is a join-aware sampling method. However, the rest of the implemented methods seems to be more precise by an order of magnitude. The deep learning method, MSCN, outperforms correlated sampling, but it isn't as performant as the Bayesian networks. However, it can probably reach a better level of performance by tuning some of the many parameters that it exposes. Meanwhile, the method we proposed with $k = 1$ means that we include the root attribute of each child relation within the Bayesian network of each parent relation. This brings to the table the benefits detailed in section 3. If $k = 2$, then an additional attribute from each child relation is included with the Bayesian network of each parent relation. We can see on figure 3-12 that the global accuracy increases with k, which is what one would expect. The most accurate method overall is the global Bayesian network presented in [Tzoumas et al., 2011]. However, our method with $k = 2$ is not far off. This makes the case that our attribute value dependency preservation assumption is a realistic one.

We have also benchmarked the methods on the TPC-DS benchmark. In contrast to the IMDb dataset used in the JOB benchmark, the TPC-DS dataset is synthetic. By nature, it contains less attribute dependencies than would be expected in a realistic use case. The TPC-DS dataset is therefore less realistic than the JOB benchmark. To produce a workload as we did for the JOB benchmark, we have taken the 30 first queries that are provided with the TPC-DS dataset and have generated all possible sub-queries. This led to a total 1,414,593 queries. The amount of joins went from 2 to 15. The overall results are shown in table 3.21.

As expected, the q-errors for the TPC-DS benchmark are better across the board because the dataset exhibits less correlations between attributes. Nonetheless, the rankings between

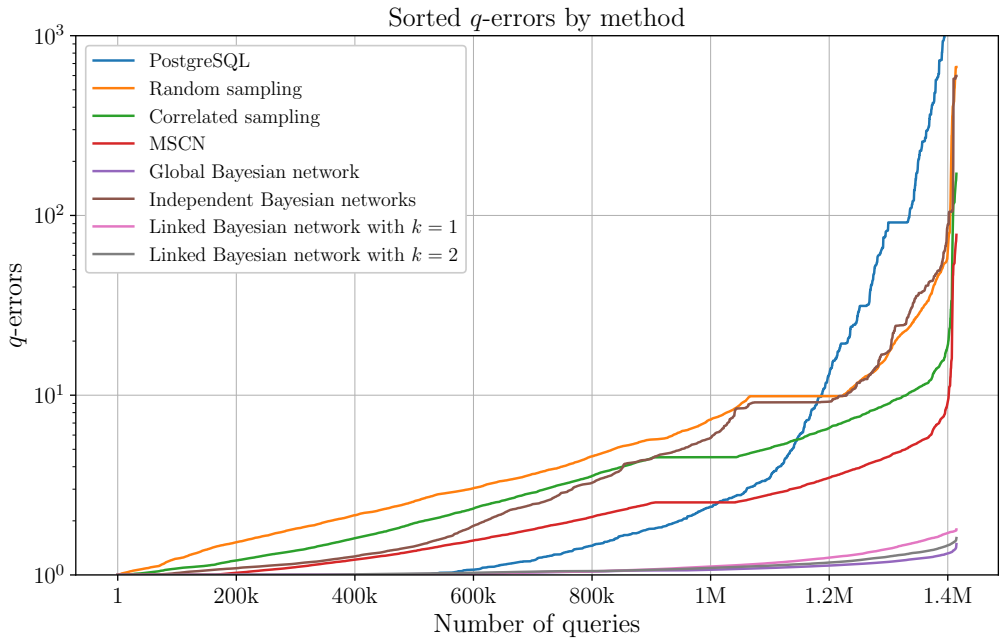

Figure 3-12: Sorted q-errors for all queries by method on the TPC-DS workload

Table 3.21: q-error statistics for each method on the TPC-DS workload

	median	90th	95th	99th	max	average
PostgreSQL	1.23	43.4	138.05	1025.49	82898.28	91.46
Sampling	3.69	13.39	26.34	66.83	669.58	9.87
Correlated sampling	2.89	8.24	10.63	19.32	170.63	4.51
MSCN	1.82	4.23	5.32	9.24	78.01	2.54
Global BN	1.03	1.17	1.23	1.33	1.49	1.06
Independent BN	2.5	13.59	33.87	89.22	597.0	9.12
Linked BN $k = 1$	1.04	1.38	1.54	1.74	1.94	1.11
Linked BN $k = 2$	1.05	1.26	1.35	1.47	1.6	1.08

the methods remains somewhat the same. Our method very slightly outperforms the global Bayesian network, but we believe that this is just an implementation artifact. In any case, our method is much more accurate than any method that assumes independence between attributes of different relations. Even so, a viable selectivity estimation method also has to be able to produce estimates in a very short amount of time, which is a point we will now discuss.

3.5.2 Inference time

Naturally, we next sought to measure how fast each method was at producing selectivity estimates. In a high throughput environment, the query optimiser isn't allowed to spend much time searching for an efficient QEP. In addition to using the cost model, the query optimiser also has to enumerate potential query execution plans and pick one of them [Chaudhuri, 1998]. Thus, only a fraction of the short amount of time allocated to the query optimiser can actually be consumed by the cost model. This means that any viable selectivity estimation has to be extremely efficient, and is probably the main reason why current cost models are kept simple. We call the amount of time necessary to produce a selectivity estimate the *inference time*. During our experiments we recorded the inference time for each query and for each model. The results shown in table 3.22 show the average inference time for each method, aggregated by the number of joins present in each query.

Table 3.22: Average inference time in milliseconds for each method with respect to the number of joins on the JOB workload

	No joins	1 join	2 to 5 joins	6 joins or more
PostgreSQL	2.3 ± 1.1	2.6 ± 1.4	3.6 ± 1.3	8.4 ± 3.1
Sampling	19.6 ± 5.4	36.2 ± 6.8	120.2 ± 5.9	268.4 ± 8.7
Correlated sampling	20.4 ± 4.9	155.7 ± 3.2	280.6 ± 7.1	493.4 ± 9.9
Global BN	84.3 ± 2.1	116.1 ± 2.9	145.8 ± 4.4	236.1 ± 3.8
Independent BN	8.3 ± 1.8	10.9 ± 1.3	12.6 ± 2.4	12.1 ± 3.2
Linked BN $k = 1$	9.5 ± 1.9	12.8 ± 1.6	14.1 ± 2.8	15.2 ± 3.4
Linked BN $k = 2$	10.1 ± 1.4	12.9 ± 1.5	14.3 ± 2.1	16.4 ± 2.9

It is important to mention that the inference time measured for PostgreSQL is simply the time it takes the database to execute the `ANALYZE` statement for each query. This thus includes the optimisation time, on top of the time spent at estimating selectivities. Even though they are already by far the best, the numbers displayed in our benchmark for PostgreSQL are pessimistic and are expected to be much lower in practice. It is also worth mentioning that we implemented the rest of the methods in Python, which is an interpreted language and thus slower than compiled languages such as C, in which PostgreSQL is written. If these methods were implemented in optimised C they would naturally be much faster. However, what matters here is the relative differences between each method, not the absolute ones.

We can clearly see from the results in table 3.22 that the global Bayesian network loses in speed what it gains in accuracy. This is because it uses a complex inference method called the *clique-tree algorithm*, which is the standard approach for Bayesian networks with arbitrary topologies. Although it is the most accurate method, it is much slower than our method, regardless of the k parameter we use. What's more, the inference time of our method doesn't increase dramatically when the number of joins increases. This is due to the variable elimination algorithm. The inference algorithm scales well because we are able to merge the Bayesian networks of each relation into a single tree. We can also see that correlated sampling is the slowest method, even though it's accuracy is competitive as shown in the previous subsection. We argue that even though our method is not as accurate as the method proposed by [Tzoumas et al., 2011], it is much faster and is thus more likely to be used in practice. Naturally, we also have to take into account the amount of time it requires to build our method, as well as how much storage space it requires.

3.5.3 Construction time and space

The cost model uses metadata that is typically obtained when the database isn't being used. This is done in order not to compute it in real time during the query optimisation phase. This metadata has to be refreshed every so often in order for the cost model to use relevant figures. Typically, the metadata has to be refreshed when the underlying data distributions change significantly. For instance, if attributes become correlated when new data is inserted, then the metadata has to be refreshed to take this into account. Therefore, the amount of time it takes to collect the necessary information is rather important, as ideally we would like to refresh the metadata as often as possible. Additionally, any viable selectivity estimation method crucially has to make do with a little amount of storage space. Indeed, spatial complexity is a major reason why most methods proposed in the literature are not being used in practice. These two computational requirements highlight the dilemma that cost models have to face: they have to be accurate whilst running with a very low footprint. Most multidimensional methods that have been proposed are utterly useless when it comes to their performance in this regard.

Table 3.23 summarises the computational requirements of the methods we compared. The results explain why PostgreSQL – and most database engines for that matter – stick to using simplistic methods. Indeed, in our measurements PostgreSQL is both the fastest

Table 3.23: Computational requirements of the construction phase per method on the JOB workload

	Construction time	Storage size
PostgreSQL	5 seconds	12KB
Sampling	7 seconds	276MB
Correlated sampling	32 seconds	293MB
Global BN	24 minutes 45 seconds	429KB
Independent BN	55 seconds	217KB
Linked BN $k = 1$	2 minutes 3 seconds	322KB
Linked BN $k = 2$	2 minutes 8 seconds	464KB

method as well the lightest one. PostgreSQL's cost model makes many simplifying assumptions and thus only has to build and store one-dimensional histograms, which can be done extremely rapidly. The sampling methods are quite fast in comparison with the methods based on Bayesian networks. This isn't surprising, as they only require sampling the relations and then storing the samples. Indeed, most of the building time involves persisting the samples on the disk. On the other hand, sampling methods require a relatively large amount of space because they do not apply any summarising whatsoever (note that their storage size are given in terms of megabytes, not kilobytes). Correlated sampling takes more time than basic sampling because it has to scan primary and foreign keys in order to avoid the empty join problem.

All of the methods based on Bayesian networks take more time to build than the two sampling methods, which is as expected. They make up in storage requirements and in inference time. The global Bayesian network takes a very large amount of time to build, which is in accordance with the results from [Tzoumas et al., 2011]. In comparison, our method is much faster. This is a logical consequence of the fact that we only build one Bayesian network per relation. Additionally, each Bayesian network has a tree topology, which means that each conditional probability distribution we need to store is a two-way table. The sudden jump in building time between $k = 0$ and $k = 1$ is due to the need to compute joins when $k > 0$. However, note that the jump is much smaller between $k = 1$ and $k = 2$. The reason is that the joins don't have to be repeated for each additional attribute included in every parent Bayesian network.

3.6 Conclusion

During the query optimisation phase, a cost model is invoked by the query optimiser to estimate the cost of query execution plans. In this context, the selectivity of operators is a crucial input to the cost model [Leis et al., 2015]. Inaccurate selectivity estimates lead to bad cost estimates which in turn have a negative impact on the overall running time of a query. Moreover, errors in selectivity estimation grow exponentially throughout a query execution plan [Ioannidis and Christodoulakis, 1991]. Selectivity estimation is still an open research problem, even though many proposals have been made. This is down to the fact that the requirements in terms of computational resources are extremely tight, and one thus has to compromise between accuracy and efficiency.

Our method is based on Bayesian networks, which are a promising way to solve the aforementioned compromise. Although the use of Bayesian networks for selectivity estimation isn't new, previous propositions entail a prohibitive building cost and inference time. In order to address these issues, we provide a way to link Bayesian networks build on separate relations. We show how we can soften the relational independence assumption without requiring an inordinate amount of computational resources. We validate our method by comparing it with other methods on an extensive workload derived from the JOB [Leis et al., 2015] and the TPC-DS [Poess et al., 2002] benchmarks. Our results show that our method is only slightly less accurate than the global Bayesian network from [Tzoumas et al., 2011], whilst being an order of magnitude less costly to build and perform inference with. Additionally, our method is more accurate than join-aware sampling, whilst requiring significantly less storage and computational requirements. In comparison with other methods which make more simplifying assumptions, our method is notably more accurate, whilst offering very reasonable guarantees in terms of computational time and space. Naturally, there is still a lot of space for improvement. For instance, when a new attribute is added to a relation, we do not have any elegant method for adding said attribute to an already existing Bayesian network. Instead, we have to reconstruct the Bayesian network from scratch. Another worthy next step would be to implement our method in a database query optimiser to measure the impact on the overall query response time, as perceived by a query issuer.

Chapter 4

Correcting selectivities with machine learning

4.1 Preliminaries

4.1.1 Motivation

The previous chapter explored the use of Bayesian networks for selectivity estimation. Bayesian networks can be viewed as a density estimation method, which falls under the umbrella of unsupervised learning. In the latter paradigm, the goal is essentially to model a probability distribution of the data. This then allows us to answer probabilistic queries, which is basically what selectivity estimation is all about. Prior to this, we made the case that unsupervised learning was more adequate than supervised learning. Indeed, with supervised learning, the data is labelled and the goal is to predict outcomes for previously unseen data. The main justification for our preference towards unsupervised learning is that selectivity estimation fundamentally boils down to density estimation, which is a subcase of unsupervised learning. The goal of density estimation is to determine the amount of data that satisfies certain conditions, and is thus closely related to selectivity estimation. Indeed, a density estimation model has been fitted to a batch of data, it is capable of answering any probabilistic query. Density estimation is unsupervised, as the goal is not to predict the outcome of each sample. On the contrary, the goal is to determine the frequency of samples. Crudely put, the goal is to predict how likely a person is to be Swedish, rather than determining if the person is a Swede in the first place. The point is that selectivity estimation can

be seen as a series of probabilistic queries. The quality of the answers is a function of the model's correctness. The latter is essentially bound by how much resources we are willing to allocate to the model. On the one hand, we can construct high-dimensional distributions in order to capture dependencies between many attributes. On the other hand, we have to make a compromise in order to alleviate the computational burden of such a distribution. Bayesian networks provide a natural framework for tuning this compromise. As already said, the advantage is that unsupervised learning is data-based, and doesn't rely on user queries to function. In other words, we can train an unsupervised model on a database without having to use a historical log of queries issued by users. Meanwhile, a supervised learning approach would consist in directly mapping user queries to selectivities. This is supervised in the sense that there is an explicit target to predict, which is the selectivity. This challenge is to be able to generalise to unseen queries.

Although a query optimiser may encounter a wide array of query typologies which highly differ with each other, some recurring patterns can be observed. One of the most compelling observations is that cost models very often *underestimate* the selectivity of a query execution plan [Lohman, 2014]. Although it may happen that they overestimate selectivities, the usual case is that instead they underestimate them. Indeed, queries tend to pertain to attribute values that have share traits. The underestimation is thus a direct consequence of the many simplifying assumptions that are made by the cost model. For instance, for two attributes A and B, the attribute value independence (AVI) assumption leads to the estimate $P(A, B) \simeq P(A) \times P(B)$. If A and B are even remotely positively correlated, then $P(A) \times P(B) < P(A, B)$, which is simple consequence of probability theory and independence. Indeed, recall that $P(A, B) = P(A \mid B)P(B)$. If A is more likely to occur when B takes place, then the previous inequality holds. By making this assumption, cost models are *always* underestimating selectivities. Naturally, if A and B are negatively correlated, then the inequality is reversed. In statistical terms, they are said to be *biased*. We have discussed this issue at large throughout this manuscript. The main way to soften the independence assumption is to develop more sophisticated models that take into account attribute dependencies. For instance, this is the whole purpose of using a Bayesian network. However, the fact is that industrial databases don't use these more advanced models because, seemingly, their computational burden is too high. Therefore, a practical research question is to make do without relaxing these assumptions. In such a case, if we are not able to change the

internals of the cost model, then the only leeway we have is to modify its outputs. For instance, if the cost models is biased and is constantly underestimating selectivities, then good sense tells us that it might be worth multiplying the selectivities by a correction factor. The question, therefore, is how much correction to apply.

A small body of query optimisation research has devoted itself to correcting the outputs of a cost model in order to account for its inherent bias. This can be seen as a "downstream" approach, whilst the method we have discussed in the previous chapter can be regarded as an "upstream" approach. Indeed, instead of building a cost model from scratch, we want to take an existing cost model and tutor it so as to correct its outputs. The most notable method in this approach is LEO [Stillger et al., 2001], which works by memorising the selectivities of past query execution plans. However, it is a bit of an outlier because it doesn't really "correct" the cost models estimates, but rather replaces them by the ground truth. In this regard, it doesn't help whatsoever when having to estimate the selectivity of previously unseen plans. Aside from the memorising approach, the main other approach has been to use a *heuristic formula*. For instance, the exponential-backoff formula [Müller et al., 2018], consists in replacing the product formula induced by the AVI assumption with the following approximation:

$$P(X_1, \ldots, X_n) \simeq \prod_{i=1}^{n} P(X_i)^{\frac{1}{i}}, \tag{4.1}$$

where the $P(X_i)$s are ordered in decreasing order. In other words $P(X_n)$ is the selectivity of most selective attribute. Note that blindly following the AVI assumption would simply suppress each exponent. Recall that a selectivity is bounded between 0 and 1, as it is simple a percentage. Therefore, applying a square root moves the product closer towards 1. Intuitively, the goal of the above formula is to dampen the effect of the most selective attributes. Arguably, this makes more sense than assuming total independence. However, it doesn't seem to rest on any theoretical background. Instead, it is very much a heuristic. Its legitimacy comes from the fact that it works well in practice. The rumour has it that database systems such as Microsoft's SQL Server apply it, but alas there is no certainty, as industrial cost models are very much shrouded in secret.

A natural step forward from heuristic rules would be to *learn* (better) rules via a machine learning algorithm. In fact, this methodology is being (re)discovered and adopted in many

other fields other than database query optimisation. The umbrella term *machine learning systems* [Dean, 2017] has in fact been coined to encompass methodologies that aim to replace hand-craft rules by parameter-dependent rules that are tuned by a machine learning algorithm from feedback. The term *Software 2.0*[1] is also regularly mentioned in a similar context.

Heuristics are aplenty throughout computer systems. Applications possess many knobs that can be tuned in order to improve a measure of performance. For instance, page sizes in relational databases [Weisberg and Wiseman, 2009], virtual machine replication amounts [Goudarzi and Pedram, 2012], cache sizes in storage systems [Oh et al., 2012], and backoff amounts for re-transmits in networking [Kwak et al., 2005], are all parameters that are chosen somewhat heuristically. A significant amount of work has explored the idea of replacing these heuristics with rules found by a machine learning algorithm. For instance, [Mirhoseini et al., 2017] proposed a reinforcement learning method for optimising device placement in a heterogeneous distributed environment which outperformed human experts. Meanwhile in [Kraska et al., 2018], the authors proposed to replace index structures such as B-trees with what they coined as *learned indexes*.

4.1.2 Supervised selectivity estimation

Once a QEP has been picked by the query optimiser, it is executed and the results are returned to the user. Once the execution is finished, the true selectivity at each stage of the QEP is made available. Therefore, it is possible to measure the error of each selectivity estimate every time a query is processed. In the case of [Chen and Roussopoulos, 1994], they use the feedback to update an iterative curve-fitting formula for summarising the distribution of a single attribute. Meanwhile, in [Aboulnaga and Chaudhuri, 1999], the authors propose to exploit the feedback to tune the parameters of both univariate and multivariate *query-driven histograms*. While innovative, these methods are confined to single-table predicates. Furthermore, the methodology from [Aboulnaga and Chaudhuri, 1999] still requires the construction of multi-dimensional histograms, which is prohibitively expensive when done at scale. Furthermore, synopses such as histograms only work when estimating the selectivity of specific types of predicates. As mentioned above, an alternative

[1] Andrej Karpathy on Software 2.0: https://medium.com/@karpathy/software-2-0-a64152b37c35

method is to memorise selectivity estimates. This can be seen as a cheap form of learning, as it doesn't allow to generalise to unseen queries.

The issue with a memorising approach is that it uses the explicit representation of a QEP. If a new QEP differs even by a little from each past QEP, then we are unable to guess the correct selectivity. A more sophisticated approach is to project QEPs into a new domain wherein similar QEPs have similar traits. In practical terms, the idea is to represent a QEP by a set of *features*. The goal is to define features that are correlated with the query selectivities. Then, a machine learning algorithm can be taught to map the features to the selectivities. In the case of selectivity estimation, many models have been proposed to directly map the features to the selectivity. For instance, *linear models* have been proposed [Malik et al., 2007, Wu et al., 2018], as well as support vector machines (SVM) [Akdere et al., 2012], neural networks [Boulos et al., 2014, Liu et al., 2015], gradient boosting over decision trees [Ivanov and Bartunov, 2017], and mixture models [Park et al., 2018]. Whatever machine learning method is used, the challenge is to find meaningful features which help in accurately predicting the selectivity of a QEP. Historically, designing good features for machine learning purposes has been done manually by domain experts. On top of being a tedious process, it is difficult to assert that features which work for one particular workload will work for another. In recent times, there has been a surge of proposals that aim to automatically learn these features via a *deep learning* approach. We discussed many of these proposals in the related work chapter. Overall, learning to estimate selectivities is a promising approach that is yet to be fully understood by the query optimisation community. Machine learning approaches, including deep learning ones, seem to regularly outperform established methods in traditional cases, or so it may seem according to the literature. This is no surprise, as said methods are extremely simple. The only reason why they are used in the first place is because they are very efficient, produce estimates in a short amount of time, and do not require an expensive learning phase.

Meanwhile, machine learning models usually imply heavier algorithms that require a training phase where all the training data has to be accessible at a single point in time. Moreover, traditional batch models cannot be dynamically updated. Indeed, they assume that the training queries they are shown is representative of the queries that will be seen in the future. Therefore, if a new attribute is appended to a relation, they produce useless results. For instance, if an attribute is added to a relation, then the one-hot encoding

schemes of [Kipf et al., 2018] and [Woltmann et al., 2019] won't be usable anymore. Finally, batch models do not adapt to concept drift, which is to say that they do not take into account changes in the distributions of the attributes when tuples are inserted, removed, or modified. We argue that batch machine learning is thus ill-fitting for the purpose of selectivity estimation. Indeed, because of the streaming nature of the problem, we believe that using an online model which can handle such a stream is a much more adequate solution. Models that can do so are part of a sub-field of machine learning named *online machine learning*. We will now give an overview of the latter, before framing the selectivity estimation problem online.

4.1.3 The benefits of online machine learning

Machine learning is the task of teaching a computer program to learn to perform a task by showing it examples. This program can be equally referred to as a *model*. In supervised learning, the model is shown pairs of samples (x, y), and is tasked with learning a function that is able to predict y (called the target) given x (called the features). In a regression task, which is a special case of supervised learning, the computer program has to learn a function $f : \mathbb{R}^p \to \mathbb{R}$. That is, for a given set of p features $x = \{v_1, \ldots, v_j, \ldots, v_p\}$, the program is capable to output a prediction $\hat{y} = f(x)$. The performance of f can be measured by comparing the prediction $\hat{y}$ with the true outcome y, which, in a live environment, is unknown at the time of making the prediction. Note that selectivity estimation can therefore be frame as a regression task, which is what is being proposed by a significant amount of the recent literature query optimisation. In our case, x would be features derived from a query execution plan, whilst y would be it's selectivity.

Machine learning models traditionally function in a batch setting, whereby a bunch of n feature sets $X = \{x_1, \ldots, x_i, \ldots, x_n\}$ and n known outcomes $Y = \{y_1, \ldots, y_i, \ldots, y_n\}$ are available. During the training phase, the program learns a function f from the so-called training set (X, Y). During the prediction phase, the function f is applied to new sets of features, which we can call the test set. A strong assumption which is usually made is that the training set and the test set originate from the same statistical distribution. A "good" model's performance on test data will be similar to its performance on the training set. If the performance on the test data is significantly lower, then the model is said to *overfit*, which essentially means that it has memorised instead of generalising.

The main issue with a batch learning model is that it has to be retrained from scratch in order to learn from new data. Indeed, the predictive function f produced by a batch learner is set in stone and cannot be updated incrementally with a new pair (x, y). However, in a live environment such as a database system, a never-ending stream of training data is constantly being produced. In such a scenario, f is immediately sub-optimal because it isn't exploiting the new observations. This is even more so the case when *concept drift* [Gama et al., 2014] occurs, which implies that the non-stationary condition of the underlying data distribution is being violated. In this case the model's performance is susceptible to plummet. In practice, batch learning models are periodically retrained. The choice of the period between training sessions is a compromise between the cost of the training and the regret in terms of missed opportunity. In the software engineering community, periodically retraining a model offline and making predictions online is often referred to as the *lambda architecture* [Kiran et al., 2015]. The lambda architecture is a loose term to describe a system where the model is treated as a static element, and the incoming stream of data is fed through the model. Each of the machine learning methods we have discussed until now follow the lambda architecture. Thus, they are not able to update themselves when new QEPs are executed and new information is made available. The lambda architecture opposes itself to the *kappa architecture* [Lin, 2017], whereby everything is handled in a streaming fashion. Models are updated every time a new sample arrives, instead of being periodically retrained from scratch. This allows them to stay-up-date with the latest trends in the data. The downside is that models that are capable to update themselves with one element at a time are rarer and not as popular as their batch counterparts.

In contrast to batch machine learning model, an online machine learning model learns from a potentially infinite stream of observations (x_t, y_t), where t denotes a timestamp. In such a setting, the predictive function f can be updated with a single observation. It also has the *anytime* property, meaning that it can make predictions at any moment in it's lifetime. Online machine learning brings many benefits to the table. First of all, once the learning algorithm has updated itself with an observation, the later can be discarded. In other words, a training set does not have to be stored aside. This strongly simplifies things from a data engineering point of view. Secondly, an online learning model is *dynamic*, in that the number of features it takes as an input may be altered on-the-fly. Indeed, an online learning model is resilient and will still work if features are added or are not available.

Usually, features that are added in such a manner are called *emerging features* [Yu et al., 2015]. Thirdly, if well-designed, an online learning model is *adaptive*, which means that it can adjust itself when concept drift occurs. Note that not all online models have this property, indeed some assume that the data generation process is stationary and that drift will never happen. Last but not least, the performance of the model can be measured in real-time. Indeed we can update a performance metric by comparing the model's prediction with the true outcome once it is available, *before* updating the model. This is called *progressive validation* [Blum et al., 1999] and is extremely practical because it means all the data can be used as a validation or test set, instead of a subset of the data as is done in traditional cross-validation [McMahan et al., 2013]. Another consequence of this interleaved prediction and learning process is that online models do not overfit. In fact, overfitting becomes a non-issue in online learning. Indeed online models remain constantly up-to-date with the latest information and implicitly forget the older data – which is often less relevant and not as representative of the newer incoming data. On the contrary, in batch learning, the challenge is to ensure that the model doesn't memorise the training set, and instead is able to generalise to unseen data. With online learning we have the ability to constantly update the model, and therefore are essentially immune to issues pertaining to overfitting. Of course there is a trade-off. Usually, for a given dataset, online models are less performant than their batch counterparts. However, their ability to update themselves means they do not play by the same rules, and therefore the performance discrepancy on a fixed dataset is a moot point.

Online machine learning is a perfect fit for the selectivity estimation task because of the streaming nature of database queries. Indeed, the workload of a database consists of queries emitted by clients, and is therefore continuously ongoing. Therefore, new query patterns may emerge, and it is therefore important to be able to keep track of the latter and have at our disposal model that can incorporate the latest information. We will now formalise our approach and describe our methodology in detail. We will then explain how we converted QEPs to features and enumerate the online machine learning models we considered.

4.2 Methodology

4.2.1 Learning to adjust selectivity estimates online

Let us start by introducing a simple heuristic to alleviate the tendency a cost model may have to underestimate selectivities. We denote by y the true selectivity, which is available *after* the query has been successfully executed. Meanwhile, $\hat{y}$ denotes the estimated selectivity, which is produced by the cost model *before* the query is executed. Note that this could be any cost model, including one that many simplifying assumptions, but also including Bayesian networks and other sophisticated methods. On average, the cost model will underestimate the selectivity by a factor c, whose definition is:

$$c = \frac{1}{n} \sum_{i=1}^{n} \frac{y_i}{\hat{y}_i} \tag{4.2}$$

In other words, c is the average multiplicative error made by the cost model. If c were equal to 1, then the cost model would be unbiased – i.e. it wouldn't have a preference for underestimating or overestimating. It follows that if we were able to multiply each selectivity estimate $\hat{y}$ by c, then we could "unbias" the model. This works because the cost model is not well calibrated. It constantly underestimates the selectivity of conjunctive predicates because it makes the attribute value independence assumption. Obviously, we can't do this because we can only obtain c once the queries have been executed. As an alternative, we could however compute c on past queries and apply to new ones. Therefore, we can multiply each selectivity estimate by this estimate in order to correct the model.

A simple heuristic rule is thus to keep track of the average multiplicative error of the cost model and multiply each subsequent selectivity by said error. We can refine this heuristic rule by increasing it's granularity. For instance, we can segment queries according to the number of joins they require and keep track of the multiplicative error inside each segment. Therefore, instead of having a single correction factor c, we would have one correction factor c_j for each number of joins j. Intuitively, such a method is more accurate than having a single correction factor c. The only source of doubt may come from the fact some c_j may estimated with small sample sizes. In such a case, c_j might be incorrect and may in fact worsen our estimates. Later on, we will discuss this issue some more and see how we can alleviate. Note that we prefer a multiplicative term to an additive term because

it removes the scale issues that would arrive from an additive formulation. However, we haven't necessarily validated this intuition experimentally.

[Wu et al., 2018] took the idea of refining the correction factor a step further. Indeed, they proposed to choose the constant as a function of the QEP. They consider many more attributes on top of the number of joins. Additionally, instead of just computing a running average for each case, they train a supervise learning model to learn the correction factor with a statistic model. In other words, they extract features from each QEP, build a historical dataset, and train a machine learning model to learn to predict the multiplicative error. In the case of [Wu et al., 2018], they use a batch model that is retrained periodically.

We argue that extending this approach with online machine learning reaps many benefits. Indeed, in the context of selectivity estimation, queries are constantly being issued to the database, therefore there is a never ending flow of queries to learn from. Whenever a query arrives, we extract features x from it. We then predict the multiplicative error we expect the cost model to make. Once the cost model has made its estimate – which can be done concurrently – we multiply it with the inverse of our error estimate. Further along, once the query optimiser has picked an execution plan and has finished executing it, we obtain the true selectivity. We can then determine how wrong the cost model was, and consequently correct our model with the features we extracted and the error made by the cost model. Note that their is no need to store data, other than keep x aside whilst the associated query is being executed.

4.2.2 Extracting useful features from a plan

A query execution plan as such cannot be processed by a machine learning model. Indeed, most machine learning models expect to be provided with numeric features. This stems from the way they were designed. Some models may take as input categorical data, such as dates and strings, but that is far from being the norm. As per usual in machine learning, the goal is find meaningful features that are correlated with the output. Ideally features should be uncorrelated with each other so as to avoid duplicate information. The key advantage of neural network approaches, such as that of [Kipf et al., 2018], is that they do not have to define features because these are automatically inferred from the one-hot encoded version of the QEPs. As we show in the next subsection, this is akin to using a field-aware factorisation machine [Juan et al., 2016].

Our feature extraction process was defined by the observation that there are two types of QEPs may encounter. On the one hand, a QEP might already have been seen. This is not usual, and is in fact quite common. Indeed, many times queries are repeated by users. Moreover, QEPs can also be part of larger QEPs (think of a relation which is included in more than one join). This redundancy was observed and exploited in DB2's LEO optimiser [Stillger et al., 2001]. On the other hand, some QEPs might never have been seen before. In this case memory-based approaches such as LEO are useless. From this observation, we defined two sets of features.

The first set of features is meant to address QEPs which have never been seen before. In this case we use features that express general information: 1) the number of joins, 2) the number of involved relations 3) the number of `WHERE` statements, 4) the maximum number of `WHERE` clauses on a single relation. The second set of features is aimed at exploiting the fact that QEPs have more often that not already been seen. One way would be store a hash table mapping QEPs to selectivities, but this would potentially grow too large and not be practical. Instead, we calculate average selectivity estimation errors by grouping over various information from each QEP. In machine learning, this encoding scheme is called *target encoding*. The idea is to extract information such as the relations, joins, attributes, attribute values, and replace them by the average of the selectivity estimation errors encountered in the past. For example, if a QEP contains a filter condition on a `country` attribute, then we would look at the average selectivity estimation error of queries which involved it. We also can do target encoding on combinations, such as on pair of attributes, and thus capture dependencies. This can be done in a streaming fashion by noticing that an average can updated online:

$$\bar{x}_{i+1} = \bar{x}_i + \frac{x - \bar{x}_i}{i + 1} \tag{4.3}$$

where $\bar{x}_i$ is the current average and x is the new value (in our case, a selectivity). The main issue of target encoding is that it's outputs are somewhat unreliable early on. This steps from the fact that the average of a small sample if often unreliable. Indeed, if a particular attribute has only been used in one particular QEP before, then the error that was made for that QEP will be the only one included in the average. To alleviate this issue, one may use a Bayesian average, which requires choosing a prior average and a weighting

term in order to compute a weighted average between the observed average and the prior average. For instance, assuming an empirical average $\bar{x}_i$ computed on a sample of size n, a prior average μ, and a weighting factor m, a Bayesian average $x^\star$ can be computed as so:

$$x^\star = \frac{n \times \bar{x}_i + m \times \mu}{n + m} \tag{4.4}$$

Note that although the above formulation is quite common, it is prone to numeric overflow because it multiplies unbounded numbers. The following formulation is more stable:

$$x^\star = \mu + \frac{n}{n + m} \times (\bar{x}_i - \mu) \tag{4.5}$$

Although this might not seem very important, but it gives an insight into the philosophy of online machine learning. Essentially, the goal is to design algorithms that are able to hold supposedly for ever. For such a purpose, it is therefore important to take into account the finite capacity of computer registers by constructing algorithms around it.

Bayesian averages are often used in applied machine learning because of their ability to reduce the variance of statistical estimates. A detailed treatment can be found in [Micci-Barreca, 2001].

4.2.3 Choice of online learning models

Machine learning models are traditionally designed with a batch setting in mind. This is mainly for historical reasons, but also because it is a less constrained setting than the online setting. Indeed, in an online setting, the rule of the game is that we can process one sample at a time, and are not allowed to revisit said sample. For a comprehensive review of the different available approaches, we refer readers to [Gepperth and Hammer, 2016], [Benczúr et al., 2018], as well as [Hoi et al., 2018], and the references found within each one. For our experiments, we decided to use well-established models with available open-source implementations. Indeed, our main motivation in this chapter is to motivate the use of online machine learning, whereas the exact details are more of an afterthought.

Linear models

The most common approach to online machine learning is to use models that minimise an empirical loss function. For instance, a common objective in regression is to find the parameters θ of a model m which minimise the squared loss between the training set outputs y and the predicted outputs $\hat{y} = m(x)$ made over the training set inputs x:

$$\min_{\theta} \sum_{i=1}^{n} (y_i - m(x_i))^2 \tag{4.6}$$

If m can be differentiated with respect to it's inputs, then for each pair (x_i, y_i) a gradient g_i can be obtained. The gradient can then be used to update θ. Because only one pair is being processed at a time, this is called *stochastic gradient descent* [Robbins and Monro, 1951]. In the special case of a linear model, θ corresponds to the weights assigned to each input x_i. Meanwhile in the case of a neural network, θ corresponds to the weights and biases of each neuron within each layer.

There are different flavors of stochastic gradient descent (i.e., updating θ given a gradient g_i). The simplest way is to multiply the gradient by a learning rate η and subtract the result from θ. The learning rate determines the magnitude of parameter updates, whilst the gradient gives the best expected update direction. In order to obtain convergence, a common procedure is to define a schedule which reduces the learning rate as time goes by. Although learning rate schedules are commonly used in practice, they are ill-suited for online learning. Indeed, if we expect to observe drift, then lowering the learning rate means that the model will not be able to adapt as fast as we would like. In our experiments we thus decided to use a constant learning rate. Note that another possible solution would have been to use a cyclic learning rate [Smith, 2017]. Many stochastic gradient descent methods have been proposed, and we refer to [Ruder, 2016] for an overview. In addition to the plain and simple method we just mentioned, we also explored the use of AdaGrad [Duchi et al., 2011], Adam [Kingma and Ba, 2014], FTRL [McMahan et al., 2013]. Note that we considered Online Newton Step (ONS) [Hazan et al., 2007], which is a second-order method that approximates the Hessian of the loss function, but discarded it because in a truly online setting it is too slow.

Many models can be framed this way. The simplest one is linear regression, which is just a dot product between a set of p weights w_i and a set of p input features x_i:

$$\hat{y} = \sum_{j=1}^{p} w_{ij} x_{ij} \tag{4.7}$$

Although linear regression is a simplistic method, it has the advantage of being interpretable, easy to debug, and fast.

Neural networks

Neural networks (including deep learning) can also be trained via stochastic gradient descent, and in fact almost always are. In our experiments we only considered the use of standard feed-forward networks. These can be seen as a succession of dot-products f interleaved by non-linear activation functions a (e.g., sigmoid, tanh, ReLU).

$$\hat{y} = a(f(a(f \dots a(f(x))))) \tag{4.8}$$

Remark that linear models can be seen as a special of neural networks whereby a single layer is used. In a neural network, the gradient can be obtained by working backwards from the loss function and accumulating the individual derivatives at each step along the way. This process is the well-known *backpropagation* algorithm [Linnainmaa, 1970]. Neural networks can be preferable to linear regression because they take into account interactions between features. The downside is that require tuning many more parameters, and are therefore more difficult to train.

Factorization machines

Another way to take into account feature interactions is to compute polynomial combinations of said features. For a given parameter $k > 1$, one can thus obtain $\sum_i^k p^i$ additional features, where p is the cardinality of the features. This provides a simple way to give non-linear capacity to a linear model [Chang et al., 2010]. The downside is that the number of additional features can grow extremely large, which in practice is a strong deterrent. Moreover, some feature interactions might occur very rarely, which may lead to uncertain parameter estimation and over-fitting. Factorisation machines (FM) [Rendle, 2010] have been proposed as to circumvent these issues. Instead of explicitly computing parameter

for each combination of features, a FM stores *latent parameters* for each feature. Each feature x_j is thus associated with a latent vector v_j of length k. The interaction between two features is then computed as the dot product between their associated latent vector.

$$\hat{y}_i = w_0 + \sum_{j=1}^{p} w_j x_j + \sum_{j=1}^{p} \sum_{k=i+1}^{p} \langle v_j, v_k \rangle x_j x_k \tag{4.9}$$

where $\langle \cdot, \cdot \rangle$ is the dot product:

$$\langle v_i, v_j \rangle = \sum_{f=1}^{k} v_{i,f} \times v_{j,f} \tag{4.10}$$

We can now store $k \times p$ weights instead of p^2. To quote the original FM paper, "*instead of using an own model parameter $w_{i,j} \in \mathcal{R}$ for each interaction, the FM models the interaction by factorising it*". Even if two features have never observed together in a single observation, their interaction parameter can be estimated from their respective latent vectors. FMs have been used with great success for high-dimensional problems such as click-through rate prediction [Pan et al., 2018]. We identified them as potential solutions to the selectivity estimation task because of the high number of attributes, values, relations, and joins present in a database schema. Indeed, we view FMs as an elegant way to identify attribute dependencies.

Tree learners

Decision trees are efficient models that are popular in practice. They are based on a simple, which is to partition the space of observations into boxes, whereby the observations contained inside each box are homogeneous. The idea is recursively partition the data by finding split rules which minimise a heterogeneity criterion, such as the entropy for classification or the mean squared error for regression. Decision trees are typically trained in a batch context [Breiman, 2017]. However, there have been a few proposals to train them online. The most established method is called the Hoeffding tree [Domingos and Hulten, 2000]. Because all the data is not available at once, the idea is to maintain summary statistics $P(X_i \mid y_j)$ where x_i are features and y_j are labels. The summary statistics are stored inside each leaf of the tree and are updated each time an observation is sorted into a leaf. The tree starts off as a single leaf. Every so often, the summary statistics are used

to evaluate possible split rules. A leaf is split into two leaves, and thus becomes a branch, once the gain from the heterogeneity criterion surpasses a certain threshold, which is called the Hoeffding bound.

A Hoeffding tree is guaranteed to find the same structure as a batch decision tree as long the underlying data distribution is stationary. There also exists an adaptive version of Hoeffding tree [Hulten et al., 2001] which can cope with concept drift, but we didn't try it out because we found there to be a lack of sufficient detail that would permit on an implementation on our part. Finally, Mondrian trees [Lakshminarayanan et al., 2014] have also been proposed. In contrast to Hoeffding trees, they take the surprising approach of picking splits at random. Indeed, the splitting process is based on the Mondrian process [Roy et al., 2008], which is an infinite stochastic process that can be used to partition a hyper-rectangle. Essentially, they do not use the target value to inform their split decisions. Instead, they record the average of the target values seen within each leaf and use these estimates to produce predictions. They add some sophistication to this process, such as introducing prior target distributions within each leaf. They also some form of apply averaging so that each leaf can benefit from the information contained in the path that leads to it from the root. Although novel and interesting, we didn't find Mondrian trees to be very performant in practice. Very recently, an improvement called "aggregated Mondrian forests" has been proposed [Mourtada et al., 2019]. Its performance is vastly superior to its eponymous parent, and in fact is competitive with batch random forests. However, as of writing this manuscript, it hasn't been peer-reviewed, and we have thus not included it in our results. We do however believe that it seems like a prominent candidate for becoming a mainstream tool for online machine learning.

4.2.4 Drift-resilient Bayesian linear regression

The methods we have described in the previous subsection are not novel. In fact, the emphasis of this chapter isn't so much on the choice of the models as much as this is on the basic principle of using online machine learning. Therefore, we have resorted to using methods that are established in the statistical learning community. However, during the PhD we have stumbled on an interesting twist of Bayesian linear regression which we have judged somewhat worthy of being included in this manuscript. We will purposefully skip some details. For information we refer to chapter 3 of [Bishop, 2006].

Bayesian modeling is a framework for mixing prior knowledge with observed evidence. On a side-note, note that this has very little to do with Bayesian networks. Although both methodologies make use of Bayes' rule, they are fundamentally different in nature. As an example, in the case of linear regression, we can impose a prior distribution on the weights. We can denote this prior distribution as $P(\theta)$. A typical parametrisation choice is to use a multivariate normal distribution centered in 0. In such a case, it may seem at first that the prior is *uninformative* because it is vague and doesn't contain any subjective information. We will soon see that this prior becomes useful in a streaming context.

A Bayesian model can be updated with observed samples. The goal is to adjust the parameters of the model according to the observed information, whilst taking into account the current knowledge. In some sense, this goal is shared with that of online machine learning. The advantage of Bayesian modeling is that it offers update formulas which are consistent with the rules of probability, and are in fact optimal under the latter. Given a new sample (x_t, y_t), Bayesian modeling gives us a mechanism for obtaining a new parameter distribution $p(\theta_{t+1} \mid \theta_t, x_t, y_t)$, which is therefore conditioned on the current distribution and the new sample. This probability distribution is obtained via Bayes' rule, as so:

$$p(\theta_{t+1} \mid \theta_t, x_t, y_t) = \frac{p(y_t \mid x_t, \theta_t) \times p(\theta_t)}{p(\theta_t, x_t, y_t)} \tag{4.11}$$

The left-hand side of the numerator is the likelihood of observing y_t given the current parameter distribution θ_t and the features x_t. The right-hand side is the current parameter distribution. The denominator is the distribution, which isn't in fact known. However, because it isn't dependent on θ, it can be simplified depending on the chosen parametrisation. In fact, the mathematical details work nicely when the prior distribution and the likelihood are said to be *conjugate*. The latter is a mathematical term that describes the fact that two distributions can be fused into a new distribution with updated parameters. When this isn't the case, then one has to resort to approximate Bayesian inference, which is beyond the scope of this discussion. One way to see it is that we are interested in the "old-school" way of doing Bayesian modeling, whereby distributions are conjugate to each other, which

leads to analytical formulas that are well suited to online machine learning. To keep things general, we will simply write down:

$$p(\theta_{t+1} \mid \theta_t, x_t, y_t) \propto p(y_t \mid x_t, \theta_t) p(\theta_t) \tag{4.12}$$

The previous statement simply expresses the fact that the posterior distribution of the model parameters is proportional to the product of the likelihood and the prior distribution. In other words, in can be obtained using an analytical formula that is specific to the chosen likelihood and prior distribution. If we're being pragmatic, then what we're really interested in is to obtain the *predictive distribution*, which is obtained by marginalising over the model parameters θ_t:

$$p(y_t \mid x_t) = \int p(y_t \mid \mathbf{w}, x_t) p(\mathbf{w}) d\mathbf{w} \tag{4.13}$$

Again, this isn't analytically tractable, except if the likelihood and the prior are conjugate to each other. The equation does make sense though, because essentially we're computing a weighted average of the potential y_i values, by considering the whole distribution of model parameters $\mathbf{w}$.

$$p(y_t \mid x_t) \propto p(y_t \mid x_t, \theta_t) p(\theta_t) \tag{4.14}$$

In short, the predictive distribution can be obtained by mixing the predictive distribution and the current parameter distribution. Again, this isn't analytically tractable, except if the likelihood and the prior are conjugate to each other. For the purpose of online machine learning, what matters to us is that we can update the distribution of the parameters when a new pair (x_t, y_t) arrives:

$$p(\theta_{t+1} \mid \theta_t, x_t, y_t) \propto p(x_t, y_t \mid \theta_t) p(\theta_t) \tag{4.15}$$

Before any data comes in, the model parameters follow the initial distribution we picked, which is $p(\theta_0)$. At this point, if we're asked to predict y_0, then it's predictive distribution would be obtained as so:

$$p(y_0 \mid x_0) \propto p(y_0 \mid x_0, \theta_0) p(\theta_0) \tag{4.16}$$

Next, once the first observation (x_0, y_0) arrives, we can update the distribution of the parameters:

$$p(\theta_1 \mid \theta_0, x_0, y_0) \propto p(x_0, y_0 \mid \theta_0) p(\theta_0) \tag{4.17}$$

The predictive distribution, given a set of features x_1, is thus:

$$p(y_1 \mid x_1) \propto p(y_1 \mid x_1, \theta_1) \underbrace{p(\theta_1 \mid \theta_0, x_0, y_0)}_{p(\theta_1)} \tag{4.18}$$

The previous equations expresses the fact that the prior of the weights for the current iteration is the posterior of the weights at the previous iteration. Once the second pair (x_1, y_1) is available, the distribution of the model parameters is updated in the same way as before:

$$p(\theta_2 \mid \theta_1, x_1, y_1) \propto p(y_1 \mid x_1, \theta_1) \underbrace{p(y_0 \mid x_0, \theta_0) p(\theta_0)}_{p(\theta_1)} \tag{4.19}$$

When the pair (x_2, y_2) arrives, the distribution of the weights can be obtained once again:

$$p(\theta_3 \mid \theta_2, x_2, y_2) \propto p(y_2 \mid x_2, \theta_2) \underbrace{p(y_1 \mid x_1, \theta_1) \underbrace{p(y_0 \mid x_0, \theta_0) p(\theta_0)}_{p(\theta_1)}}_{p(\theta_2)} \tag{4.20}$$

By now, it might be clear that there is recursive relationship that links each iteration: the posterior distribution at step t becomes the prior distribution at step $t+1$. This simple fact is the reason why analytical Bayesian inference can naturally be used as an online machine learning algorithm. Indeed, we only need to store the current distribution of the weights to make everything work.

Up until now we didn't give any useful example. We will now see how to perform linear regression by using Bayesian inference. In a linear regression, the model parameters θ_t are just weights w_t that are linearly applied to a set of features x_t:

$$y_t = w_t x_t^\intercal + \epsilon_t \tag{4.21}$$

Each prediction is the scalar product between p features x_t and p weights w_t. The trick here is that we're going to assume that the noise ϵ_i follows a given distribution. In particular, we will be boring and use the Gaussian ansatz, which implies that the likelihood function is a Gaussian distribution:

$$p(y_t \mid x_t, w_t) = \mathcal{N}(w_t x_t^\intercal, \beta^{-1}) \tag{4.22}$$

Christopher Bishop calls β the "noise precision parameter". In statistics, the precision is inversely related to the noise variance as so: $\beta = \frac{1}{\sigma^2}$. Basically, it translates our belief on how noisy the target distribution is. Both concepts coexist mostly because statisticians can't agree on a common Bible. There are ways to tune this parameter automatically from the data, however for the sake of simplicity we will treat it as known constant. In any case, the appropriate prior distribution for the above likelihood function is the multivariate Gaussian distribution:

$$p(w_0) = \mathcal{N}(m_0, S_0) \tag{4.23}$$

m_0 is the mean vector of the distribution while S_0 is its covariance matrix. Initially, their initial values will be:

$$m_0 = (0, \ldots, 0) \tag{4.24}$$

$$S_0 = \begin{pmatrix} \alpha^{-1} & \ldots & \ldots \\ \ldots & \alpha^{-1} & \ldots \\ \ldots & \ldots & \alpha^{-1} \end{pmatrix} \tag{4.25}$$

α is a hyperparameter that needs to be provided. From our experience, its influence is very small in an online scenario and therefore its value does not matter very much. We can now determine the posterior distribution of the weights:

$$p(w_{t+1} \mid w_t, x_t, y_t) = \mathcal{N}(m_{t+1}, S_{t+1}) \tag{4.26}$$

$$S_{t+1} = (S_t^{-1} + \beta x_t^\intercal x_t)^{-1} \tag{4.27}$$

$$m_{t+1} = S_{t+1}(S_t^{-1} m_t + \beta x_t y_t) \tag{4.28}$$

Note that $x_t^\intercal x_t$ is the outer product of x_t with itself. There are also a set of formulas that can be used to obtain the predictive distribution:

$$p(y_t) = \mathcal{N}(\mu_t, \sigma_t) \tag{4.29}$$

$$\mu_t = w_t x_t^\intercal \tag{4.30}$$

$$\sigma_t = \frac{1}{\beta} + x_t S_t x_t^\intercal \tag{4.31}$$

All of the above formulas are quite common and can be found in many introductions to Bayesian inference. One of the issues with this formulation is that the data is assumed to be stationary. Indeed, the more data we show the model, the more it will be confident about its parameter estimate. However, we could like to it to be able to be robust to concept drift and impose some sort of adversarial setting. In the case of plain linear regression this isn't so much an issue because we use a constant learning rate that allows the weights to be updated whenever drift occurs. The solution we found was to change the update formulas of the covariance matrix and the mean vector in the following manner:

$$S_{t+1} = (\gamma S_t^{-1} + (1 - \gamma)\beta x_t^\intercal x_t)^{-1} \tag{4.32}$$

$$m_{t+1} = S_{t+1}(\gamma S_t^{-1} m_t + (1 - \gamma)\beta x_t y_t) \tag{4.33}$$

In the above equations, γ acts a smoothing parameter which controls how much the model "forgets" its current state and sticks to the new data. In the case where $\gamma = 1$, the model doesn't learn and sticks to the prior distribution. On the contrary, when $\gamma = 0$, the model memorises the latest sample and forgets what it has seen up to there. These

formulas are very much heuristic and we have not taken the time to give them a thorough analytical treatment. We have considered them to be worthy of inclusion in the manuscript because they have provided us with very good experimental results, as will be show in the next section. As far as we can tell, they haven't been used in published literature. They are however, not complex and resemble exponential weighted moving averages. Note that a similar formulation can be found in [Russac et al., 2019], although the context and the premises are somewhat different.

4.3 Experimental results

We evaluated our approach using the IMDb dataset from the JOB benchmark [Leis et al., 2015]. The IMDb dataset contains real-word data pertaining to the movie industry and contains many correlations (for instance French actors usually play in French movies). It contains 21 relations and weighs 3.6 GB. We simulated query workloads by sampling from the queries that accompany each dataset. In the case of the IMDb dataset, there are 113 available queries. For each sampled query, we asked PostgreSQL to generate an execution plan and sampled sub-plans from each of these execution plans. For each sub-plan, we tasked each machine learning model we benchmarked to predict the correction factor we introduced previously. We then multiplied each predicted correction factor with the selectivity estimate made by PostgreSQL's cost model. Finally, we calculated the q-error [Moerkotte et al., 2009], which measures the multiplicative difference between predictions and ground truths. The q-error is generally used for evaluating selectivity estimation modules.

We sampled the queries in three different manners. First of all, we sampled the queries completely at random. Secondly, we split the queries into three buckets and sampled from a particular bucket. We made sure that each bucket contained queries that were similar with other, whilst being different to the queries from other buckets. At certain points in time, we swap the buckets in order to simulate a hard drift. Our goal with this approach is to measure how well a method is able to cope with a change in query patterns. Finally, we created an environment with a slow drift, where we slowly transitioned from one bucket to the other. To perform such a simulation, we define the probability of sampling a bucket to

be a function of the current time step. There are many ways to define such a function. In our experiments we chose the following formula:

$$P(b,t) = \frac{exp(-\frac{(t-t_b)^2}{d})}{\sum_{b_i} exp(-\frac{(t-t_{b_i})^2}{d})} \tag{4.34}$$

where $P(b,t)$ is the probability of sampling bucket b at time step t. Meanwhile, t_b is an arbitrary time step assigned to bucket b, and w is a parameter which controls the abruptness of the drift. For our experiments we arbitrarily settled on a value for w of 3.

We benchmarked the following online machine learning models:

1. Linear regression trained via stochastic gradient descent, with a constant learning rate of 0.1.

2. Bayesian linear regression, which α set to 10. Note that in practice the α doesn't seem to have a great impact on the model's convergence.

3. Hoeffding tree with a patience of 200 and a maximum depth of 5. This means that we attempt to split nodes every 200 samples, and stop whenever a leaf has reached a depth of 5.

4. Feed-forward neural network with 2 hidden layers of 30 neurons trained with the Adam optimiser and a constant learning rate of 0.01.

5. FM with 10 components trained via stochastic gradient, with a constant learning rate of 0.1. We provided it with one-hot encoded versions of the used attributes, attribute values, relations, and joins.

Note that the parameters we propose are those that have us the best results experimentally. For the sake of simplicity we haven't the detail of all of our results. As a comparison, we included the selectivity estimates made by PostgreSQL's cost model. We also benchmarked the following batch learning methods:

1. Standard linear regression fitted with maximum likelihood estimation (MLE). This provides a baseline of what a simple batch learning method is able to do.

2. LightGBM [Ke et al., 2017], which is an ensemble method that combines boosting [Freund et al., 1996] with decision trees. In short, this involves a step-wise process

where each decision tree attempts to correct the mistakes of its predecessors. This is widely considered to be the best off-the-shelf supervised (batch) learning model, and is therefore interesting to compare against. Note that we used the default parameters proposed in the reference implementation.

3. MSCN from [Kipf et al., 2019b], which we trained for 50 epochs. We chose this method in particular because it is a representative candidate of the recent trend from the query optimisation community towards deep learning models.

We trained each batch method on 100,000 execution plans prior to conducting the benchmark. Meanwhile the online methods do not require this warm-up phase because they are trained online. The evaluation phase samples 500,000 query execution plans and obtains a prediction from each method. Figure 4-1 shows the q-errors for each method along time. Note that the online models are represented with dotted lines whereas the batch models are show with solid lines. As can be expected, the batch methods initially outperform the online methods because they have has a warm-up phase. However, online learning methods eventually outperform their batch counterparts. The best performing method is FM, which attains an average q-error of 4.5. As a comparison, PostgreSQL has an average q-error of 25. Naturally, the batch learning can be retrained and thus may outperform the online methods, but that requires storing observations and defining a training schedule, which is something we are tried to avoid.

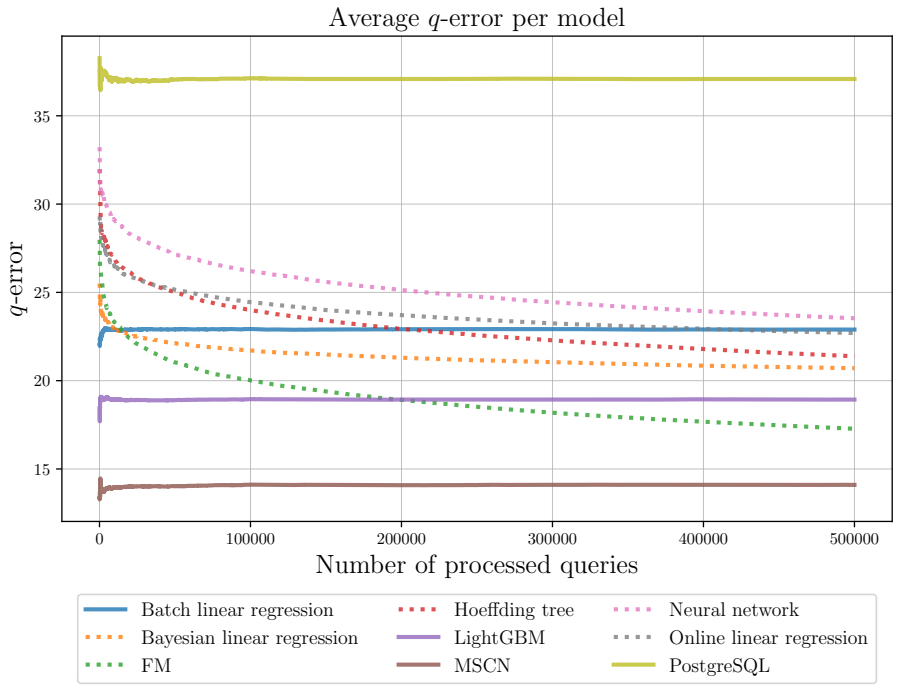

Figure 4-1: q-errors for each method when sampling queries at random

The second sampling manner simulates a concept drift. We randomly sample 200,000 execution plans from a pool of queries, and then repeat this twice for two other query pools. The results are shown in figure 4-2. Up until the first hard drift, which occurs at the 200,000 marker, the curves follow somewhat the same ranking as in figure 4-1. Most of the models experience a drop in performance once the query patterns changes. For instance, batch linear regression goes from an average q-error of around 12 to 26. The online models also take a hit from the drift, but are for the most part able adapt. Meanwhile, the Hoeffding tree, which is not designed to handle concept drift, is not able to reattain its initial performance. The reason why is that it is not able to revisit the decision splits it has made, which therefore become irrelevant.

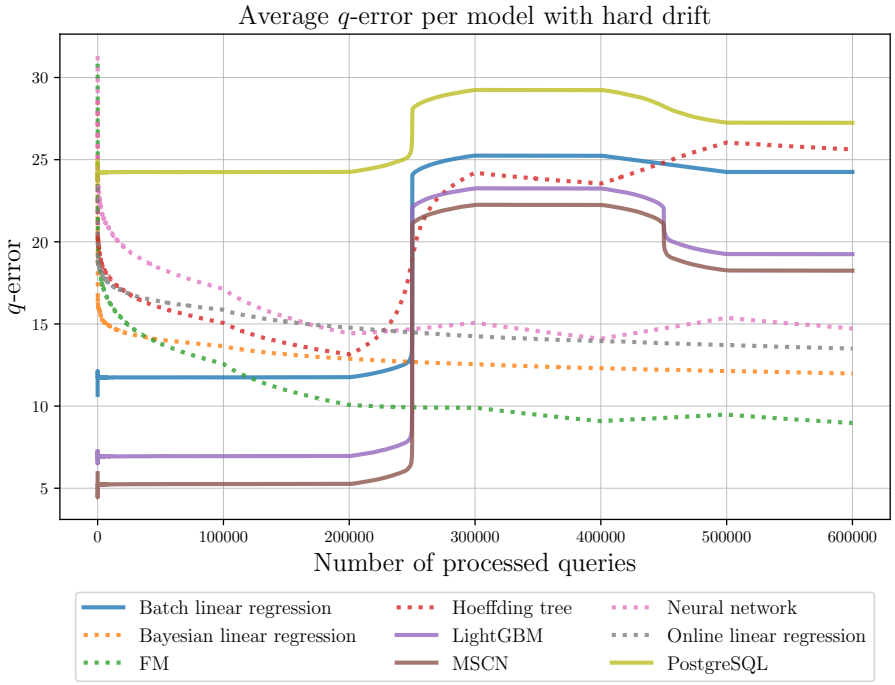

Figure 4-2: q-errors for each method when simulating concept drift

Our final sampling schema incurs a slow drift, which is represented in 4-3. Effectively, we start by sampling from the first pool of queries, and slowly transition to the second bucket. After some time we also transition towards the first bucket. Towards the middle of the process, we are essentially sampling from all three buckets at random. This is a strongly adversarial environment to cope with for any machine learning model, be it online or not. As can be seen in figure 4-3, the average q-error are worse all across the board. The performance of the online learning models seems somewhat constant, which highlights the fact that the underlying data distribution keeps changing. Meanwhile, the batch models do not adapt, as they have been trained on a workload which is similar to the first bucket. Therefore their performant plummets as time goes on.

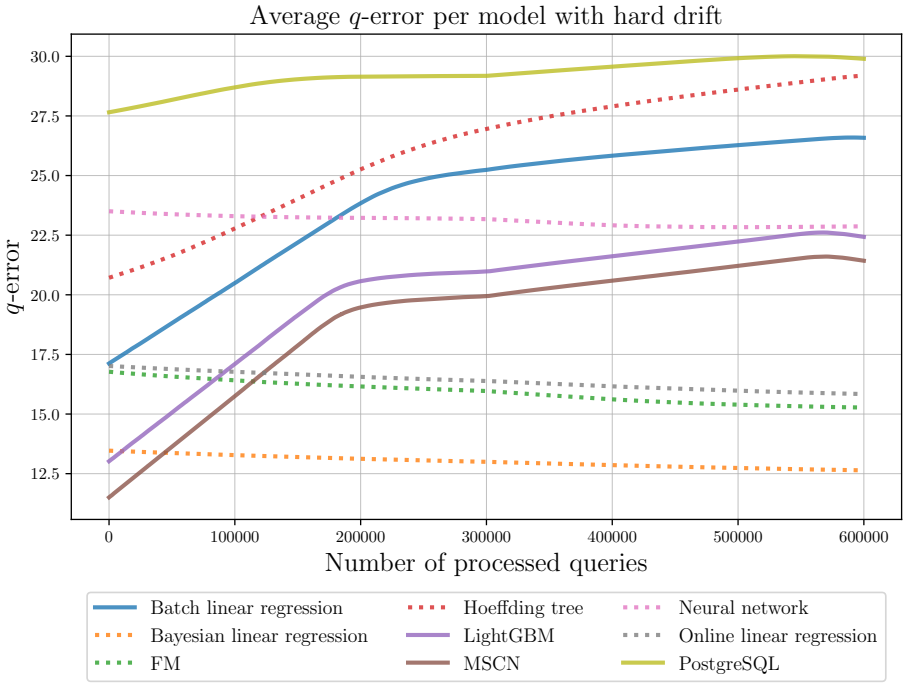

Figure 4-3: q-errors for each method when simulating concept drift

4.4 Conclusion

In this chapter, we have looked at correcting an existing selectivity estimation model, instead of designing one from scratch as we did in the previous chapter. Specifically, we resorted to correct selectivity estimates with a multiplicative factor. Essentially, our goal has been to suppress the inherent tendency of simplistic cost models to underestimate selectivities. In particular, we have advocated the use of online machine learning for this purpose. Indeed, although a growing body of query optimisation literature is busying itself with applying supervised learning in many of its shapes and colours, we argue that the batch learning approach they surmise isn't particularly well suited to the task. On the contrary, we believe that in light of the fact that query optimisers receive a never-ending flow of queries, it makes more sense to use an online approach. First and foremost online machine learning allows to stay up to date with the latest data and can thus result in better performance overall. Aside from this benefit, an online paradigm doesn't require having to store data, and thus presents somewhat less of an engineering challenge. We have attempted to make our case both through qualitative arguments, as well as experimental results. The main point we wish to make is that the methodology we propose is simpler than methods based

on batch learning. With online machine learning, it is possible to learn from the stream of execution plans on-the-fly. Therefore, there is no need to store a historical training set and retrain the model periodically. Another benefit is that online machine learning may significantly reduce the amount *technical debt* that is known to plague machine learning software [Sculley et al., 2014]. We believe that online machine learning can and probably should be used for many tasks that fall under the umbrella of *machine learning for systems* [Dean, 2017], not just selectivity estimation. The challenge, of course, is to think in terms of online learning and design the resulting system through the lens of online computation. For instance, features need to be able to be extracted online. Likewise, the necessary data structures are not the same as those used with batch methods. Finally, we stress that our approach is complementary to our contribution from the previous chapter, as it is agnostic to which cost model is being corrected. Therefore, the contributions we have made in both chapters are complementary and not opposed.

Chapter 5

Conclusion

The goal and topic of this PhD has been to explore the use of machine learning for the purpose of selectivity estimation. We began by describing selectivity estimation and discussed the performance requirements that it usually entails. Selectivity estimation is critical because it is the main component of the cost model, which is the basis on which the query optimiser relies to build query execution plans. Selectivity estimation is tricky because one has to account for correlations between attributes, both within and between relations. The difficulty comes from the fact that there are strict performance guarantees that we would like to satisfy. Therefore, complex methods that have a high computational cost are not usable and are ignored in practice. In this sense, there is no optimal solution to the problem of selectivity estimation. Instead, a compromise has to be made between efficiency and accuracy. As of now, and ever since the early works on query optimisation, databases have strongly erred towards simple models that trade accuracy in favour of efficiency.

We proceeded to enumerate existing methods, and categorised them according to their style of approach. As we mentioned, there are many ways to skin a cat. The ontology we chose is not universal, and is fact closer to the statistics point of view than it is to the query optimisation literature. A first approach is to execute given queries on a sample of the database. The obtained selectivity can then be extrapolated to the whole database. The advantage is that this is a straightforward approach, and there are thus many sampling tools that are very well understood. The main issue is that is difficult to build samples across multiple relations whilst avoiding the so-called "empty-join problem". Additionally,

executing a query on a sample is still a costly operation. Therefore isn't a satisfying solution when having to estimate the selectivities for queries that involve multiple relations.

A second approach is to learn to predict selectivities with supervised learning. A supervised learning model takes as input a query execution plan and outputs a selectivity. Typically, the model learns by examining a historical set of queries and tuning a set of parameters that work best. There are many flavors to this approach, and a lot of attention is currently being devoted to it. In particular, the popularity of deep learning has encouraged researchers to dust off old methods and revisit them with new tools and new computational settings. The main issue with a supervised learning approach is that it learns from the queries. It therefore does not exploit the information from the database, such as the distributions of attribute value and their correlation. It therefore performs density estimation by taking a detour, which is difficult to justify and has been criticised. One of the particularities of selectivity estimation is that there is domain closure as regards to the underlying data. We are not in the typical scenario where the data we have is a sample of a bigger universe. Supervised learning methods are not available to make use of this property.

The third and final approach we have discussed is unsupervised learning. The latter encompasses every method that attempts to model the distribution of the database's attribute values. In statistical jargon, this is called *density estimation*. Many of the old and venerable methods that have stood the test of time, such as histograms, are in fact cases of density estimation. Instead of looking at the queries, they look directly at the data and summarise its distribution into data structures that possess varying amounts of sophistication. Of the latter, we identified Bayesian networks as the most relevant method, and decided to examine its use for the purpose of selectivity estimation. We believe that the computational cost required by existing proposals is too high to be used in practice. We therefore sought to design a methodology based on Bayesian networks that offered a better compromise between selectivity estimation accuracy and computational cost.

We began by tailoring Bayesian networks so that they could be used to estimate selectivities within individual relations. Our goal was therefore to soften the ubiquitous attribute value independence assumption. In particular, we made precise choices so as to ensure that each Bayesian networks was 1) able to be built in a timely manner through the use of Chow-Liu trees, 2) able to provide an upper-bound on the required amount of storage, 3) structured in a way that permits fast inference. We found that our approach was a solid

first step towards improving selectivity estimation. However, it did not allow us to capture dependencies that span over two or more relations, which are the bane of cost models. We therefore proceeded to extend our methodology so as to allow it to capture some amount of dependency across more than one relation. We made a point to keep this methodology simple and not fall into the combinatorial explosion trap. We introduced a new assumption, which we coined the *attribute dependency preservation* assumption. Our assumption is that the dependencies that hold within one relation are preserved if the latter is joined with one or more other relations. In the meantime, we do not assume that an attribute's value distribution does not change. In fact, our methodology allows this, contrary to past proposals. We are able to do so by exploiting the simple fact that since our models are trees, the root attribute has an influence on the rest of the factorised distribution that the tree represents. Therefore, by estimating the distribution of the root attribute once it is joined with another relation, we are able to infer the distribution of other attributes virtually for free. We have validated our method on common selectivity estimation benchmarks. It is able to compete with complex methods that model the distribution of the entire database, but requires a much lower cost, both in terms of spatial and temporal complexity.

To complement our research, we looked downstream and attempted to correct an existing cost model via supervised learning. The justification for this is that, in practice, cost models make many simplifying assumptions that lead them to consistently underestimate selectivities. Therefore, a seemingly reasonable approach is to adjust the cost model by multiplying each selectivity it produces by a correcting factor. A simple heuristic is to define each selectivity by a constant factor. A sophistication could be to choose a value that minimises the error on some historical dataset. An even more involved idea is to formulate it as a function of features that can be derived from each query execution plan. In other words, one may frame this as a supervised learning task. We explored this method in our last chapter. In particular, we made the case that formulating this as an online supervised learning is a better fit than a batch approach. Indeed, learning online allows us to exploit the never-ending stream of queries and seamlessly adapt to concept drift. In fact, we were able to show in our experiments that such a methodology is competitive, and actually outperforms the more traditional batch methods. The latter include the MSCN neural network, which is representative of the recent trend from the query optimisation community towards deep learning.

Throughout our two methods, we have managed to approach the selectivity estimation problem from different angles. On the one hand, our Bayesian network is built from the ground up and can be used to answer many kinds of selectivity estimation queries. On the other hand, our second method works by correcting an existing selectivity estimation method, and is therefore complementary to our first method. Meanwhile, we have also used two different flavors of machine learning, namely unsupervised learning and supervised learning. As can be expected, there remain a few research directions that we have not pursued as much as we would have liked to. First of all, on a practical note, it is clear that the methods we have proposed require further exploration and validation. For example, it could be worthwhile to integrate some flavor of them into a database system in order to assess their added value on the query execution duration, as perceived by the user. Secondly, we believe it would be worthwhile to consider robust query optimisation, which is a paradigm where the uncertainty of each selectivity estimate is taken into account. The idea is that if a query optimiser can make a more informed decision and potentially avoid catastrophic decisions if it is provided with a distribution of values rather than a point-wise estimate. Bayesian networks are not designed to produce uncertainty estimates because their structure is assumed to be an exact proxy of the underlying data generation process. However, they do allow to incorporate uncertainty in their parameter estimates, but this isn't of any use to us because of domain closure. Indeed, given a structure, there is no uncertainty as to the values of each conditional distribution because we are working with an bounded dataset. Meanwhile, the supervised learning methods discussed in the last chapter do permit to integrate uncertainty. Finally, we believe that like most work pertaining to selectivity estimation, our research could be extended to other applications, such as workload forecasting and resource allocation in database systems. Indeed, the ability to determine the cost of a query without executing is, without a doubt, and for the foreseeable future, of immense use.

Bibliography

[Aboulnaga and Chaudhuri, 1999] Aboulnaga, A. and Chaudhuri, S. (1999). Self-tuning histograms: Building histograms without looking at data. *ACM SIGMOD Record*, 28(2):181–192.

[Acharya et al., 1999] Acharya, S., Gibbons, P. B., Poosala, V., and Ramaswamy, S. (1999). Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 275–286.

[Akdere et al., 2012] Akdere, M., Çetintemel, U., Riondato, M., Upfal, E., and Zdonik, S. B. (2012). Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering*, pages 390–401. IEEE.

[Armbrust et al., 2015] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. (2015). Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM.

[Barbar'a et al., 1997] Barbar'a, D., DuMouchel, W., Faloutsos, C., Haas, P. J., Hellerstein, J. M., Ioannidis, Y., Jagadish, H., Johnson, T., Ng, R., Poosala, V., et al. (1997). The new jersey data reduction report. In *IEEE Data Engineering Bulletin*. Citeseer.

[Begoli et al., 2018] Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M. J., and Lemire, D. (2018). Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, pages 221–230. ACM.

[Bellman, 2015] Bellman, R. E. (2015). *Adaptive control processes: a guided tour*. Princeton university press.

[Benczúr et al., 2018] Benczúr, A. A., Kocsis, L., and Pálovics, R. (2018). Online machine learning in big data streams. *arXiv preprint arXiv:1802.05872*.

[Bishop, 2006] Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

[Blohsfeld et al., 1999] Blohsfeld, B., Korus, D., and Seeger, B. (1999). A comparison of selectivity estimators for range queries on metric attributes. In *ACM SIGMOD Record*, volume 28, pages 239–250. ACM.

[Blum et al., 1999] Blum, A., Kalai, A., and Langford, J. (1999). Beating the hold-out: Bounds for k-fold and progressive cross-validation. In *COLT*, volume 99, pages 203–208.

[Boulos et al., 2014] Boulos, J., le Bretonneux, M., and Viémont, Y. (2014). Selectivity estimation using neural networks. *provided on search dated*, pages 1–21.

[Breiman, 2017] Breiman, L. (2017). *Classification and regression trees*. Routledge.

[Bruno et al., 2001] Bruno, N., Chaudhuri, S., and Gravano, L. (2001). Stholes: a multidimensional workload-aware histogram. In *Acm Sigmod Record*, volume 30, pages 211–222. ACM.

[Chakrabarti et al., 2001] Chakrabarti, K., Garofalakis, M., Rastogi, R., and Shim, K. (2001). Approximate query processing using wavelets. *The VLDB Journal—The International Journal on Very Large Data Bases*, 10(2-3):199–223.

[Chang et al., 2010] Chang, Y.-W., Hsieh, C.-J., Chang, K.-W., Ringgaard, M., and Lin, C.-J. (2010). Training and testing low-degree polynomial data mappings via linear svm. *Journal of Machine Learning Research*, 11(Apr):1471–1490.

[Chaudhuri, 1998] Chaudhuri, S. (1998). An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM.

[Chaudhuri et al., 1999] Chaudhuri, S., Motwani, R., and Narasayya, V. (1999). On random sampling over joins. *ACM SIGMOD Record*, 28(2):263–274.

[Chaudhuri et al., 2009] Chaudhuri, S., Narasayya, V., and Ramamurthy, R. (2009). Exact cardinality query optimization for optimizer testing. *Proceedings of the VLDB Endowment*, 2(1):994–1005.

[Chen and Roussopoulos, 1994] Chen, C. M. and Roussopoulos, N. (1994). Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 161–172.

[Chen and Yi, 2017] Chen, Y. and Yi, K. (2017). Two-level sampling for join size estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 759–774. ACM.

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

[Cooper, 1990] Cooper, G. F. (1990). The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2-3):393–405.

[Cowell et al., 2006] Cowell, R. G., Dawid, P., Lauritzen, S. L., and Spiegelhalter, D. J. (2006). *Probabilistic networks and expert systems: Exact computational methods for Bayesian networks*. Springer Science & Business Media.

[Dean, 2017] Dean, J. (2017). Machine learning for systems and systems for machine learning. In *Presentation at 2017 Conference on Neural Information Processing Systems*.

[Deshpande et al., 2001] Deshpande, A., Garofalakis, M., and Rastogi, R. (2001). Independence is good: Dependency-based histogram synopses for high-dimensional data. *ACM SIGMOD Record*, 30(2):199–210.

[Domingos and Hulten, 2000] Domingos, P. and Hulten, G. (2000). Mining high-speed data streams. In *Kdd*, volume 2, page 4.

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

[Dutt et al., 2019] Dutt, A., Wang, C., Nazi, A., Kandula, S., Narasayya, V., and Chaudhuri, S. (2019). Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment*, 12(9):1044–1057.

[El-Helw et al., 2007] El-Helw, A., Ilyas, I. F., Lau, W., Markl, V., and Zuzarte, C. (2007). Collecting and maintaining just-in-time statistics. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 516–525. IEEE.

[Estan and Naughton, 2006] Estan, C. and Naughton, J. F. (2006). End-biased samples for join cardinality estimation. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 20–20. IEEE.

[François-Lavet et al., 2018] François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., Pineau, J., et al. (2018). An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354.

[Freund et al., 1996] Freund, Y., Schapire, R. E., et al. (1996). Experiments with a new boosting algorithm. In *icml*, volume 96, pages 148–156. Citeseer.

[Friedman et al., 1999] Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. (1999). Learning probabilistic relational models. In *IJCAI*, volume 99, pages 1300–1309.

[Gama et al., 2014] Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., and Bouchachia, A. (2014). A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):44.

[Ganguly et al., 1996] Ganguly, S., Gibbons, P. B., Matias, Y., and Silberschatz, A. (1996). Bifocal sampling for skew-resistant join size estimation. In *ACM SIGMOD Record*, volume 25, pages 271–281. ACM.

[Gelfand and Smith, 1991] Gelfand, A. E. and Smith, A. F. (1991). Gibbs sampling for marginal posterior expectations. *Communications in Statistics-Theory and Methods*, 20(5-6):1747–1766.

[Gepperth and Hammer, 2016] Gepperth, A. and Hammer, B. (2016). Incremental learning algorithms and applications.

[Getoor et al., 2001] Getoor, L., Taskar, B., and Koller, D. (2001). Selectivity estimation using probabilistic models. In *ACM SIGMOD Record*, volume 30, pages 461–472. ACM.

[Goudarzi and Pedram, 2012] Goudarzi, H. and Pedram, M. (2012). Energy-efficient virtual machine replication and placement in a cloud computing system. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 750–757. IEEE.

[Gu et al., 2012] Gu, Z., Soliman, M. A., and Waas, F. M. (2012). Testing the accuracy of query optimizers. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, pages 1–6.

[Gunopulos et al., 2005] Gunopulos, D., Kollios, G., Tsotras, J., and Domeniconi, C. (2005). Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal—The International Journal on Very Large Data Bases*, 14(2):137–154.

[Haas et al., 1996] Haas, P. J., Naughton, J. F., Seshadri, S., and Swami, A. N. (1996). Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences*, 52(3):550–569.

[Hayek and Shmueli, 2019] Hayek, R. and Shmueli, O. (2019). Improved cardinality estimation by learning queries containment rates. *arXiv preprint arXiv:1908.07723*.

[Hayek and Shmueli, 2020] Hayek, R. and Shmueli, O. (2020). Nn-based transformation of any sql cardinality estimator for handling distinct, and, or and not. *arXiv preprint arXiv:2004.07009*.

[Hazan et al., 2007] Hazan, E., Agarwal, A., and Kale, S. (2007). Logarithmic regret algorithms for online convex optimization. *Machine Learning*, 69(2-3):169–192.

[Heckerman et al., 1998] Heckerman, D. et al. (1998). A tutorial on learning with bayesian networks. *Nato Asi Series D Behavioural And Social Sciences*, 89:301–354.

[Heckerman et al., 1995] Heckerman, D., Geiger, D., and Chickering, D. M. (1995). Learning bayesian networks: The combination of knowledge and statistical data. *Machine learning*, 20(3):197–243.

[Heimel et al., 2015] Heimel, M., Kiefer, M., and Markl, V. (2015). Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1477–1492. ACM.

[Hellerstein and Stonebraker, 2005] Hellerstein, J. M. and Stonebraker, M. (2005). *Readings in database systems*. MIT Press.

[Hilprecht et al., 2020] Hilprecht, B., Bang, T., El-Hindi, M., Hättasch, B., Khanna, A., Rehrmann, R., Röhm, U., Schmidt, A., Thostrup, L., Ziegler, T., et al. (2020). Dbms fitting: Why should we learn what we already know?

[Hilprecht et al., 2019] Hilprecht, B., Schmidt, A., Kulessa, M., Molina, A., Kersting, K., and Binnig, C. (2019). Deepdb: Learn from data, not from queries! *arXiv preprint arXiv:1909.00607*.

[Hoi et al., 2018] Hoi, S. C., Sahoo, D., Lu, J., and Zhao, P. (2018). Online learning: A comprehensive survey. *arXiv preprint arXiv:1802.02871*.

[Huang et al., 2019] Huang, D., Yoon, D. Y., Pettie, S., and Mozafari, B. (2019). Joins on samples: A theoretical guide for practitioners. *arXiv preprint arXiv:1912.03443*.

[Hulten et al., 2001] Hulten, G., Spencer, L., and Domingos, P. (2001). Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 97–106. ACM.

[Hwang et al., 1992] Hwang, F. K., Richards, D. S., and Winter, P. (1992). *The Steiner tree problem*, volume 53. Elsevier.

[Ilyas et al., 2004] Ilyas, I. F., Markl, V., Haas, P., Brown, P., and Aboulnaga, A. (2004). Cords: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 647–658. ACM.

[Ioannidis and Christodoulakis, 1991] Ioannidis, Y. E. and Christodoulakis, S. (1991). *On the propagation of errors in the size of join results*, volume 20. ACM.

[Ioannidis and Christodoulakis, 1993] Ioannidis, Y. E. and Christodoulakis, S. (1993). Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Transactions on Database Systems (TODS)*, 18(4):709–748.

[Ivanov and Bartunov, 2017] Ivanov, O. and Bartunov, S. (2017). Adaptive cardinality estimation.

[Jaakkola et al., 2010] Jaakkola, T., Sontag, D., Globerson, A., and Meila, M. (2010). Learning bayesian network structure using lp relaxations. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 358–365.

[Jensen et al., 1996] Jensen, F. V. et al. (1996). *An introduction to Bayesian networks*, volume 210. UCL press London.

[Juan et al., 2016] Juan, Y., Zhuang, Y., Chin, W.-S., and Lin, C.-J. (2016). Field-aware factorization machines for ctr prediction. In *Proceedings of the 10th ACM Conference on Recommender Systems*, pages 43–50. ACM.

[Ke et al., 2017] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154.

[Kiefer et al., 2017] Kiefer, M., Heimel, M., Breß, S., and Markl, V. (2017). Estimating join selectivities using bandwidth-optimized kernel density models. *Proceedings of the VLDB Endowment*, 10.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Kipf et al., 2019a] Kipf, A., Freitag, M., Vorona, D., Boncz, P., Neumann, T., and Kemper, A. (2019a). Estimating filtered group-by queries is hard: Deep learning to the rescue. *Proceedings AIDB*.

[Kipf et al., 2018] Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P., and Kemper, A. (2018). Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*.

[Kipf et al., 2019b] Kipf, A., Vorona, D., Müller, J., Kipf, T., Radke, B., Leis, V., Boncz, P., Neumann, T., and Kemper, A. (2019b). Estimating cardinalities with deep sketches. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1937–1940.

[Kiran et al., 2015] Kiran, M., Murphy, P., Monga, I., Dugan, J., and Baveja, S. S. (2015). Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2785–2792. IEEE.

[Kooi, 1980] Kooi, R. P. (1980). The optimization of queries in relational databases.

[Kraska et al., 2018] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. (2018). The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM.

[Krishnan et al., 2018] Krishnan, S., Yang, Z., Goldberg, K., Hellerstein, J., and Stoica, I. (2018). Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*.

[Kruskal, 1956] Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50.

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

[Kwak et al., 2005] Kwak, B.-J., Song, N.-O., and Miller, L. E. (2005). Performance analysis of exponential backoff. *IEEE/ACM Transactions on Networking (TON)*, 13(2):343–355.

[Lakshminarayanan et al., 2014] Lakshminarayanan, B., Roy, D. M., and Teh, Y. W. (2014). Mondrian forests: Efficient online random forests. In *Advances in neural information processing systems*, pages 3140–3148.

[Lee et al., 1999] Lee, J.-H., Kim, D.-H., and Chung, C.-W. (1999). Multi-dimensional selectivity estimation using compressed histogram information. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 205–214.

[Leis et al., 2015] Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., and Neumann, T. (2015). How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215.

[Leis et al., 2017] Leis, V., Radke, B., Gubichev, A., Kemper, A., and Neumann, T. (2017). Cardinality estimation done right: Index-based join sampling. In *CIDR*.

[Leis et al., 2018] Leis, V., Radke, B., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., and Neumann, T. (2018). Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal*, 27(5):643–668.

[Lin, 2017] Lin, J. (2017). The lambda and the kappa. *IEEE Internet Computing*, (5):60–66.

[Lindsay et al., 1999] Lindsay, B. G., Lohman, G. M., Pirahesh, M. H., Shekita, E. J., Simmen, D. E., and Urata, M. S. (1999). Star/join query optimization. US Patent 5,960,428.

[Linnainmaa, 1970] Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7.

[Lipton and Naughton, 1990] Lipton, R. J. and Naughton, J. F. (1990). Query size estimation by adaptive sampling. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 40–46. ACM.

[Liu et al., 2015] Liu, H., Xu, M., Yu, Z., Corvinelli, V., and Zuzarte, C. (2015). Cardinality estimation using neural networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, pages 53–59. IBM Corp.

[Lohman, 2014] Lohman, G. (2014). Is query optimization a "solved" problem. In *Proc. Workshop on Database Query Optimization*, volume 13. Oregon Graduate Center Comp. Sci. Tech. Rep.

[Malik et al., 2007] Malik, T., Burns, R. C., and Chawla, N. V. (2007). A black-box approach to query cardinality estimation. In *CIDR*, pages 56–67.

[Marcus et al., 2019] Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., and Tatbul, N. (2019). Neo: A learned query optimizer. *Proceedings of the VLDB Endowment*, 12(11):1705–1718.

[Marcus and Papaemmanouil, 2018] Marcus, R. and Papaemmanouil, O. (2018). Towards a hands-free query optimizer through deep learning. *arXiv preprint arXiv:1809.10212*.

[Marcus and Papaemmanouil, 2019] Marcus, R. and Papaemmanouil, O. (2019). Plan-structured deep neural network models for query performance prediction. *Proceedings of the VLDB Endowment*, 12(11):1733–1746.

[Matias et al., 1998] Matias, Y., Vitter, J. S., and Wang, M. (1998). Wavelet-based histograms for selectivity estimation. In *ACM SIGMoD Record*, volume 27, pages 448–459. ACM.

[McMahan et al., 2013] McMahan, H. B., Holt, G., Sculley, D., Young, M., Ebner, D., Grady, J., Nie, L., Phillips, T., Davydov, E., Golovin, D., et al. (2013). Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM.

[Micci-Barreca, 2001] Micci-Barreca, D. (2001). A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems. *ACM SIGKDD Explorations Newsletter*, 3(1):27–32.

[Mirhoseini et al., 2017] Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. (2017). Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2430–2439. JMLR. org.

[Moerkotte et al., 2009] Moerkotte, G., Neumann, T., and Steidl, G. (2009). Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment*, 2(1):982–993.

[Mourtada et al., 2019] Mourtada, J., Gaïffas, S., and Scornet, E. (2019). Amf: Aggregated mondrian forests for online learning. *arXiv preprint arXiv:1906.10529*.

[Müller et al., 2018] Müller, M., Moerkotte, G., and Kolb, O. (2018). Improved selectivity estimation by combining knowledge from sampling and synopses. *Proceedings of the VLDB Endowment*, 11(9):1016–1028.

[Muralikrishna and DeWitt, 1988] Muralikrishna, M. and DeWitt, D. J. (1988). Equi-depth multidimensional histograms. In *ACM SIGMOD Record*, volume 17, pages 28–36. ACM.

[Murphy, 2001] Murphy, K. P. (2001). Active learning of causal bayes net structure.

[Muthukrishnan et al., 1999] Muthukrishnan, S., Poosala, V., and Suel, T. (1999). On rectangular partitionings in two dimensions: Algorithms, complexity and applications. In *International Conference on Database Theory*, pages 236–256. Springer.

[Nešetřil et al., 2001] Nešetřil, J., Milková, E., and Nešetřilová, H. (2001). Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete mathematics*, 233(1-3):3–36.

[Oh et al., 2012] Oh, Y., Choi, J., Lee, D., and Noh, S. H. (2012). Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, volume 12.

[Olken, 1993] Olken, F. (1993). *Random sampling from databases*. PhD thesis, University of California, Berkeley.

[Olken and Rotem, 1986] Olken, F. and Rotem, D. (1986). Simple random sampling from relational databases.

[Ortiz et al., 2018] Ortiz, J., Balazinska, M., Gehrke, J., and Keerthi, S. S. (2018). Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, pages 1–4.

[Ortiz et al., 2019] Ortiz, J., Balazinska, M., Gehrke, J., and Keerthi, S. S. (2019). An empirical analysis of deep learning for cardinality estimation. *arXiv preprint arXiv:1905.06425*.

[Pan et al., 2018] Pan, J., Xu, J., Ruiz, A. L., Zhao, W., Pan, S., Sun, Y., and Lu, Q. (2018). Field-weighted factorization machines for click-through rate prediction in display advertising. In *Proceedings of the 2018 World Wide Web Conference*, pages 1349–1357.

[Park et al., 2018] Park, Y., Zhong, S., and Mozafari, B. (2018). Quicksel: Quick selectivity learning with mixture models. *arXiv preprint arXiv:1812.10568*.

[Pearl, 1982] Pearl, J. (1982). *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science, University of California, Los Angeles.

[Piatetsky-Shapiro and Connell, 1984] Piatetsky-Shapiro, G. and Connell, C. (1984). Accurate estimation of the number of tuples satisfying a condition. *ACM Sigmod Record*, 14(2):256–276.

[Poess et al., 2002] Poess, M., Smith, B., Kollar, L., and Larson, P. (2002). Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 582–587.

[Poosala et al., 1996] Poosala, V., Haas, P. J., Ioannidis, Y. E., and Shekita, E. J. (1996). Improved histograms for selectivity estimation of range predicates. In *ACM Sigmod Record*, volume 25, pages 294–305. ACM.

[Poosala and Ioannidis, 1997] Poosala, V. and Ioannidis, Y. E. (1997). Selectivity estimation without the attribute value independence assumption. In *VLDB*, volume 97, pages 486–495.

[Prim, 1957] Prim, R. C. (1957). Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401.

[Rendle, 2010] Rendle, S. (2010). Factorization machines. In *2010 IEEE International Conference on Data Mining*, pages 995–1000. IEEE.

[Riondato et al., 2011] Riondato, M., Akdere, M., Çetintemel, U., Zdonik, S. B., and Upfal, E. (2011). The vc-dimension of sql queries and selectivity estimation through sampling. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 661–676. Springer.

[Robbins and Monro, 1951] Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407.

[Robertson and Seymour, 1986] Robertson, N. and Seymour, P. D. (1986). Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322.

[Roy et al., 2008] Roy, D. M., Teh, Y. W., et al. (2008). The mondrian process. In *NIPS*, pages 1377–1384.

[Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

[Russac et al., 2019] Russac, Y., Vernade, C., and Cappé, O. (2019). Weighted Linear Bandits for Non-Stationary Environments. In *NeurIPS 2019 - 33rd Conference on Neural Information Processing Systems*, Vancouver, Canada.

[Russell and Norvig, 2002] Russell, S. and Norvig, P. (2002). Artificial intelligence: a modern approach.

[Sculley et al., 2014] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., and Young, M. (2014). Machine learning: The high interest credit card of technical debt.

[Selinger et al., 1979] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. (1979). Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM.

[Sethi et al., 2019] Sethi, R., Traverso, M., Sundstrom, D., Phillips, D., Xie, W., Sun, Y., Yegitbasi, N., Jin, H., Hwang, E., Shingte, N., et al. (2019). Presto: Sql on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813. IEEE.

[Silverman, 1986] Silverman, B. W. (1986). *Density estimation for statistics and data analysis*, volume 26. CRC press.

[Smith, 2017] Smith, L. N. (2017). Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE.

[Srivastava et al., 2006] Srivastava, U., Haas, P. J., Markl, V., Kutsch, M., and Tran, T. M. (2006). Isomer: Consistent histogram construction using query feedback. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 39–39. IEEE.

[Stillger et al., 2001] Stillger, M., Lohman, G. M., Markl, V., and Kandil, M. (2001). Leo-db2's learning optimizer. In *VLDB*, volume 1, pages 19–28.

[Stonebraker and Rowe, 1986] Stonebraker, M. and Rowe, L. A. (1986). *The design of Postgres*, volume 15. ACM.

[Sun and Li, 2019] Sun, J. and Li, G. (2019). An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment*, 13(3):307–319.

[To et al., 2013] To, H., Chiang, K., and Shahabi, C. (2013). Entropy-based histograms for selectivity estimation. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1939–1948.

[Traverso, 2013] Traverso, M. (2013). Presto: Interacting with petabytes of data at facebook. *Retrieved February*, 4:2014.

[Tzoumas et al., 2011] Tzoumas, K., Deshpande, A., and Jensen, C. S. (2011). Lightweight graphical models for selectivity estimation without independence assumptions. *Proceedings of the VLDB Endowment*, 4(11):852–863.

[Tzoumas et al., 2013] Tzoumas, K., Deshpande, A., and Jensen, C. S. (2013). Efficiently adapting graphical models for selectivity estimation. *The VLDB Journal*, 22(1):3–27.

[Tzoumas et al., 2008] Tzoumas, K., Sellis, T., and Jensen, C. S. (2008). A reinforcement learning approach for adaptive query processing. *History*.

[Van Aken et al., 2017] Van Aken, D., Pavlo, A., Gordon, G. J., and Zhang, B. (2017). Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM.

[Vengerov et al., 2015] Vengerov, D., Menck, A. C., Zait, M., and Chakkappen, S. P. (2015). Join size estimation subject to filter conditions. *Proceedings of the VLDB Endowment*, 8(12):1530–1541.

[Wang et al., 2016] Wang, W., Zhang, M., Chen, G., Jagadish, H., Ooi, B. C., and Tan, K.-L. (2016). Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record*, 45(2):17–22.

[Wang et al., 2020] Wang, Y., Xiao, C., Qin, J., Cao, X., Sun, Y., Wang, W., and Onizuka, M. (2020). Monotonic cardinality estimation of similarity selection: A deep learning approach. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1197–1212.

[Weisberg and Wiseman, 2009] Weisberg, P. and Wiseman, Y. (2009). Using 4kb page size for virtual memory is obsolete. In *2009 IEEE International Conference on Information Reuse & Integration*, pages 262–265. IEEE.

[Weiss and Freeman, 2000] Weiss, Y. and Freeman, W. T. (2000). Correctness of belief propagation in gaussian graphical models of arbitrary topology. In *Advances in neural information processing systems*, pages 673–679.

[Wilson et al., 2019] Wilson, D., Hou, W.-C., and Yu, F. (2019). Scalable correlated sampling for join query estimations on big data. In *Proceedings of 28th International Conference*, volume 64, pages 41–50.

[Woltmann et al., 2019] Woltmann, L., Hartmann, C., Thiele, M., Habich, D., and Lehner, W. (2019). Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, page 5. ACM.

[Wu et al., 2018] Wu, C., Jindal, A., Amizadeh, S., Patel, H., Le, W., Qiao, S., and Rao, S. (2018). Towards a learning optimizer for shared clouds. *Proceedings of the VLDB Endowment*, 12(3):210–222.

[Wu et al., 2013] Wu, W., Chi, Y., Zhu, S., Tatemura, J., Hacigümüs, H., and Naughton, J. F. (2013). Predicting query execution time: Are optimizer cost models really unusable? In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1081–1092. IEEE.

[Yu et al., 2015] Yu, K., Ding, W., Simovici, D. A., Wang, H., Pei, J., and Wu, X. (2015). Classification with streaming features: An emerging-pattern mining approach. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 9(4):1–31.

[Zaheer et al., 2017] Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R., and Smola, A. J. (2017). Deep sets. In *Advances in neural information processing systems*, pages 3391–3401.

[Zhao et al., 2018] Zhao, Z., Christensen, R., Li, F., Hu, X., and Yi, K. (2018). Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1525–1539.