DeepDB: Learn from Data, not from Queries!

Benjamin Hilprecht
TU Darmstadt, Germany
benjamin.hilprecht@tu-darmstadt.de

Andreas Schmidt
KIT & Hochschule Karlsruhe, Germany
andreas.schmidt@kit.edu

Moritz Kulessa
TU Darmstadt, Germany
moritz.kulessa@tu-darmstadt.de

Alejandro Molina
TU Darmstadt, Germany
alejandro.molina@tu-darmstadt.de

Kristian Kersting
TU Darmstadt, Germany
kristian.kersting@tu-darmstadt.de

Carsten Binnig
TU Darmstadt, Germany
carsten.binnig@tu-darmstadt.de

ABSTRACT

The typical approach for learned DBMS components is to capture the behavior by running a representative set of queries and use the observations to train a machine learning model. This workload-driven approach, however, has two major downsides. First, collecting the training data can be very expensive, since all queries need to be executed on potentially large databases. Second, training data has to be recollected when the workload or the database changes. To overcome these limitations, we take a different route and propose a new data-driven approach for learned DBMS components which directly supports changes of the workload and data without the need of retraining. Indeed, one may now expect that this comes at a price of lower accuracy since workload-driven approaches can make use of more information. However, this is not the case. The results of our empirical evaluation demonstrate that our data-driven approach not only provides better accuracy than state-of-the-art learned components but also generalizes better to unseen queries.

PVLDB Reference Format:
B. Hilprecht et al.. DeepDB: Learn from Data, not from Queries!. *PVLDB*, 13(7): 992-1005, 2020.
DOI: https://doi.org/10.14778/3384345.3384349

1. INTRODUCTION

Motivation. Deep Neural Networks (DNNs) have not only been shown to solve many complex problems such as image classification or machine translation, but are applied in many other domains, too. This is also the case for DBMSs, where DNNs have successfully been used to replace existing DBMS components with learned counterparts such as learned cost models [16, 42] as well as learned query optimizers [27], or even learned indexes [17] or query scheduling and query processing schemes [24, 39].

Proceedings of the VLDB Endowment, Vol. 13, No. 7
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3384345.3384349

The predominant approach for learned DBMS components is that they capture the behavior of a component by running a representative set of queries over a given database and use the observations to train the model. For example, for learned cost models such as [16, 42] different query plans need to be executed to collect the training data, which captures the runtime (or cardinalities), to then learn a model that can estimate costs for new query plans. This observation also holds for the other approaches such as learned query optimizers or the learned query processing schemes, which are also based on collected training data that requires the execution of a representative workload.

A major obstacle of this workload-driven approach is that collecting the training data is typically very expensive since many queries need to be executed to gather enough training data. For example, approaches like [16, 42] have shown that the runtime of hundreds of thousands of query plans is needed for the model to provide a high accuracy. Still, the training corpora often only cover a limited set of query patterns to avoid even higher training costs. For example, in [16] the training data covers only queries up to two joins (three tables) and filter predicates with a limited number of attributes.

Moreover, the training data collection is not a one-time effort since the same procedure needs to be repeated over and over if the workload changes or if the current database is not static and the data is constantly being updated as it is typical for OLTP. Otherwise, without collecting new training data and retraining the models for the characteristics of the changing workload or data, the accuracies of these models degrade with time.

Contributions. In this paper, we take a different route. Instead of learning a model over the workload, we propose to learn a purely data-driven model that captures the joint probability distribution of the data and reflects important characteristics such as correlations across attributes but also the data distribution of single attributes. Another important difference to existing approaches is that our data-driven approach supports direct updates; i.e., inserts, updates, and deletes on the underlying database can be absorbed by the model without the need to retrain the model.

As a result, since our model captures information of the data it can not only be used for one particular task but supports many different tasks ranging from query answering, over cardinality estimation to potential other more so-

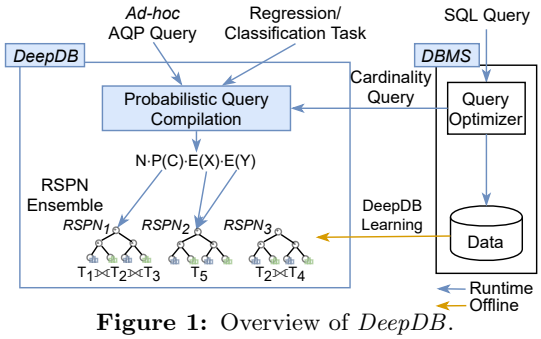


Figure 1: Overview of *DeepDB*.

phisticated tasks such as in-DBMS machine learning inference. One could now think that this all comes at a price and that the accuracy of our approach must be lower since the workload-driven approaches get more information than a pure data-driven approach. However, as we demonstrate in our experiments, this is not the case. Our approach actually outperforms many state-of-the-art workload-driven approaches and even generalizes better.

However, we do not argue that data-driven models are a silver bullet to solve all possible tasks in a DBMS. Instead, we think that data-driven models should be combined with workload-driven models when it makes sense. For example, a workload-driven model for a learned query optimizer might use the cardinally estimates of our model as input features. This combination of data-driven and workload-driven models provides an interesting avenue for future work but is beyond the scope of this paper.

To summarize, the main contributions of this paper are:(1) We developed a new class of deep probabilistic models over databases: Relational Sum Product Networks (RSPNs), that can capture important characteristics of a database. (2) To support different tasks, we devise a probabilistic query compilation approach that translates incoming database queries into probabilities and expectations for RSPNs. (3) We implemented our data-driven approach in a prototypical DBMS architecture, called *DeepDB*, and evaluated it against state-of-the-art learned and non-learned approaches.

Outline. The remainder of the paper is organized as follows. In Section 2 we first present an overview of *DeepDB* and then discuss details of our models and the query compilation in Sections 3 and 4. Afterwards, we explain further extensions of *DeepDB* in Section 5 before we show an extensive evaluation comparing *DeepDB* against state-of-the art approaches for various tasks. Finally, we iterate over related work in Section 7 before concluding in Section 8.

2. OVERVIEW AND APPLICATIONS

Overview. As shown in Figure 1, the main idea of *DeepDB* is to learn a representation of the data offline. An important aspect of *DeepDB* is that we do not aim to replace the original data with a model. Instead, a model in *DeepDB* augments a database similar to indexes to speed-up queries and to provide additional query capabilities while we can still run standard SQL queries over the original database.

To optimally capture relevant characteristics of relational data in *DeepDB*, we developed a new class of models called *Relational Sum Product Networks* (RSPNs). In a nutshell, RSPNs are a class of deep probabilistic models that capture the joint probability distribution over all attributes in a database that can then be used at runtime to provide the answer for different user tasks.

While RSPNs are based on Sum Product Networks (SPNs) [35, 28], there are significant differences: (1) While SPNs support only single tables and simple queries (i.e., no joins and no aggregation functions), RSPNs can be built on arbitrary schemata and support complex queries with multi-way joins and different aggregations (COUNT, SUM, AVG). Moreover, RSPNs also go beyond the idea of other recent learned data models that need to know join paths a priori such as [25, 51] since RSPNs allow true ad-hoc joins by combining RSPN models. (2) Another major difference is that RSPNs support direct updates, i.e., if the underlying database changes the RSPN can directly ingest the updates without the need to retrain the model. (3) RSPNs also include a set of database-specific extensions such as NULL-value handling and support for functional dependencies.

Once the RSPNs are created offline, they can be leveraged at runtime for a wide variety of different applications, ranging from user-facing tasks (e.g., to provide fast approximate answers for SQL queries) to system-internal tasks (e.g., to provide estimates for cardinalities). In order to support these tasks, *DeepDB* provides a new so called *probabilistic query compilation* procedure that translates a given task into evaluations of expectations and probabilities on RSPNs. We now give a brief overview of the applications currently supported by the query compilation engine of *DeepDB*.

Cardinality Estimation. The first task *DeepDB* supports is cardinality estimation for a query optimizer. Cardinality estimation is needed to provide cost estimates but also to find the correct join order during query optimization. A particular advantage of *DeepDB* over existing learned approaches for cardinality estimation [16, 42] is that we do not have to create dedicated training data, i.e. pairs of queries and cardinalities. Instead, since RSPNs capture the characteristics of the data independent of a workload, we can support arbitrary join queries without the need to train a model for a particular workload. Moreover, RSPNs can be kept up to date at low costs similar to traditional histogram-based approaches, which is different from other workload-driven learned approaches for cardinality estimation such as [16, 42] which require retraining.

Approximate Query Processing (AQP). The second task we currently support in *DeepDB* is AQP. AQP aims to provide approximate answers to support faster query response times on large datasets. The basic idea of how a query on a single table is executed inside *DeepDB* is simple: for example, an aggregate query $\texttt{AVG}(X)$ with a where condition C is equal to the conditional expectation $\mathbb{E}(X \mid C)$ which can be approximated with RSPNs. In *DeepDB*, we implement a more general AQP procedure that leverages the fact that RSPNs can support joins of multiple tables. A major difference to other learned approaches for AQP such as [25, 44] is again that *DeepDB* supports ad-hoc queries and is thus not limited to the query types covered by the training set.

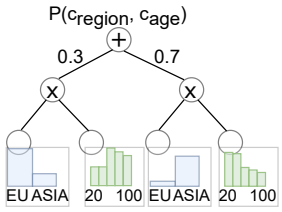
Other Applications. While the applications above show the potential of *DeepDB*, we believe *DeepDB* is not limited to those applications. For example, machine learning inference tasks such as regression and classification can be answered by RSPNs. However, discussing these opportunities in detail is beyond the scope of this paper.

c_id	c_age	c_region
1	80	EU
2	70	EU
3	60	ASIA
4	20	EU
...	...	...
998	20	ASIA
998	25	EU
999	30	ASIA
1000	70	ASIA

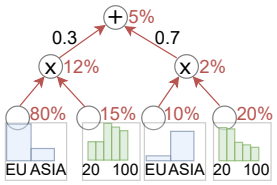
(a) Example Table

c_age	c_region
80	EU
70	EU
60	ASIA
20	EU
...	...
...	...
20	ASIA
25	EU
30	ASIA
70	ASIA

(b) Learning with Row/Column Clustering



(c) Resulting SPN



(d) Probability of European Customers younger than 30

Figure 2: Customer Table and corresponding SPN.

3. LEARNING A DEEP DATA MODEL

In this section, we introduce Relational Sum Product Networks (RSPNs), which we use to learn a representation of a database and, in turn, to answer queries using our query engine explained in the next section. We first review Sum Product Networks (SPNs) and then introduce RSPNs. Afterwards, we describe how an ensemble of RSPNs can be created to encode a given database multiple tables.

3.1 Sum Product Networks

Sum-Product Networks (SPNs) [35] learn the joint probability distribution $P(X_1, X_2, \ldots, X_n)$ of the variables X_1, $X_2, \ldots, X_n$ in the dataset. They are an appealing choice because probabilities for arbitrary conditions can be computed very *efficiently*. We will later make use of these probabilities for our applications like AQP and cardinality estimation.

For the sake of simplicity, we restrict our attention to Tree-SPNs, i.e., trees with sum and product nodes as internal nodes and leaves. Intuitively, sum nodes split the population (i.e., the rows of dataset) into clusters and product nodes split independent variables of a population (i.e., the columns of a dataset). Leaf nodes represent a single attribute and approximate in the present paper the distribution of that attribute either using histograms for discrete domains or piecewise linear functions for continuous domains [29]. For instance, in Figure 2c, an SPN was learned over the variables *region* and *age* of the corresponding *customer* table in Figure 2a. The top sum node splits the data into two groups: The left group contains 30% of the population, which is dominated by older European customers (corresponding to the first rows of the table), and the right group contains 70% of the population with younger Asian customers (corresponding to the last rows of the table). In both groups, region and age are independent and thus split by a product node each. The leaf nodes determine the probability distributions of the variables *region* and *age* for every group.

Learning SPNs [10, 29] works by recursively splitting the data in different clusters of rows (introducing a sum node) or clusters of independent columns (introducing a product node). For the clustering of rows, a standard algorithm such as *KMeans* can be used or the data can be split according to a random hyperplane. To make no strong assumptions about the underlying distribution, Randomized Dependency Coefficients (RDC) are used for testing independence of different columns [23]. Moreover, independence between all columns is assumed as soon as the number of rows in a cluster falls below a threshold n_{min}. As stated in [35, 28], SPNs in general have polynomial size and allow inference in linear time w.r.t. the number of nodes. However, for the configurations we use in our experiments, we can even bound the size of the SPNs to linear complexity w.r.t. the number of columns in a dataset since we set $n_{min} = n_s/100$ (i.e. relative to the sample size), which turned out to be a robust configuration.

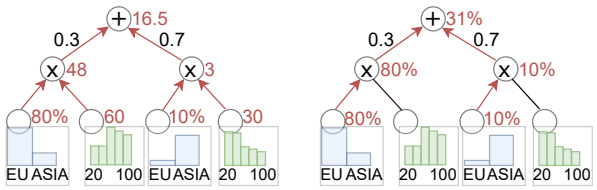
With an SPN at hand, one can compute probabilities for conditions on arbitrary columns. Intuitively, the conditions are first evaluated on every relevant leaf. Afterwards, the SPN is evaluated bottom up. For instance in Figure 2d, to estimate how many customers are from Europe and younger than 30, we compute the probability of European customers in the corresponding blue *region* leaf nodes (80% and 10%) and the probability of a customer being younger than 30 (15% and 20%) in the green *age* leaf nodes. These probabilities are then multiplied at the product node level above, resulting in probabilities of 12% and 2%, respectively. Finally, at the root level (sum node), we have to consider the weights of the clusters, which leads to $12\% \cdot 0.3 + 2\% \cdot 0.7 = 5\%$. Multiplied by the number of rows in the table, we get an approximation of 50 European customers who are younger than 30.

3.2 Relational Sum-Product Networks

One important issue with SPNs is that they can only capture the data of single tables but they also lack other important features needed for *DeepDB*. To alleviate these problems, we now introduce RSPNs.

Extended Inference Algorithms. The first and most important extension is that for many queries such as AVG and SUM expectations are required (e.g., to answer a SQL aggregate query which computes an average over a column). In order to answer these queries, RSPNs allows computing expectations over the variables on the leaves to answer those aggregates. To additionally apply a filter predicate, we still compute probabilities on the leaves for the filter attribute and propagate both values up in the tree. At product nodes, we multiply the expectations and probabilities coming from child nodes whereas on sum nodes the weighted average is computed. In Figure 3, we show an example how the average age of European customers is computed. The ratio of both terms yields the correct conditional expectation. A related problem is that SPNs do not provide confidence intervals. We also developed corresponding extensions on SPNs in Section 5.1.

Database-specifics. Finally, SPNs lack support for important database specifics: (1) First, SPNs do not provide mechanisms for handling NULL values. Hence, we developed an extension where NULL values are represented as a dedicated value for both discrete and continuous columns at the leaves during learning. Furthermore, when computing conditional probabilities and expectations, NULL val-



(a) $\mathbb{E}(\texttt{c\_age} \cdot 1_{\texttt{c\_region='EU'}})$ **(b)** $P(\texttt{c\_region='EU'})$

Figure 3: Process of computing $\mathbb{E}(\texttt{c\_age} \mid \texttt{c\_region='EU'})$.

ues must be handled according to the three-valued logic of SQL. (2) Second, SPNs aim to generalize the data distribution and thus approximate the leaf distribution, abstracting away specifics of the dataset to generalize. For instance, in the leaf nodes for the age in Figure 2c, a piecewise linear function would be used to approximate the distribution [29]. Instead, we want to represent the data as accurate as possible. Hence, for continuous values, we store each individual value and its frequency. If the number of distinct values exceeds a given limit, we also use binning for continuous domains. (3) Third, functional dependencies between non-key attributes $A \rightarrow B$ are not well captured by SPNs. We could simply ignore these and learn the RSPN with both attributes A and B, but this often leads to large SPNs since the data would be split into many small clusters (to achieve independence of A and B). Hence, we allow users to define functional dependencies along with a table schema. If a functional dependency $A \rightarrow B$ is defined, we store the mapping from values of A to values of B in a separate dictionary of the RSPN and omit the column B when learning the RSPN. At runtime, queries with filter predicates for B are translated to queries with filter predicates for A.

Updatability. Finally, a last important extensions of RSPNs over SPNs is the direct updatability of the model. If the underlying database tables are updated, the model might become inaccurate. For instance, if we insert more young European customers in the table in Figure 2a, the probability computed in Figure 2d is too low and thus the RSPN needs to be updated. As described before, an RSPN consists of product and sum nodes, as well as leaf nodes, which represent probability distributions for individual variables. The key-idea to support direct updates of an existing RSPN is to traverse the RSPN tree top-down and update the value distribution of the weights of the sum-nodes during this traversal. For instance, the weight of a sum node for a subtree of younger European customers could be increased to account for updates. Finally, the distributions in the leaf-nodes are adjusted. The detailed algorithm of how to directly update RSPNs is discussed in Section 5.2.

3.3 Learning Ensembles of RSPNs

In order to support ad-hoc join queries one could naively learn a single RSPN per table as we discuss in Section 4. However, in this case potential correlations between tables might be lost and lead to inaccurate approximations. For learning an ensemble of RSPNs for a given database with multiple tables, we thus take into account if tables of a schema are correlated.

In the following, we describe our procedure that constructs a so called *base ensemble* for a given database scheme. In this procedure, for every *foreign key→primary key* relationship we learn an RSPN over the corresponding full outer join of two tables if there is a correlation between attributes of these two tables. Otherwise, RSPNs for the single tables will be learned. For instance, if the schema consists of a `Customer` and an `Order` table as shown in Figure 4, we could either learn two independent RSPNs (one for each table) or a joint RSPN (over the full outer join).

In order to test independence of two tables and thus to decide if one or two RSPNs are more appropriate, we check for every pair of attributes from these tables if they can be considered independent or not. In order to enable an efficient computation, this test can be done on a small random sample. As a correlation measure that does not make major distributional assumptions, we compute RDC values [23] between two attributes, which are also used in the SPN learning algorithm [29]. If the maximum pairwise RDC value between all attributes of two tables exceeds a threshold (where we use the standard thresholds of SPNs), we assume that two tables are correlated and learn an RSPN over the join.

In the base ensemble only correlations between two tables are captured. While in our experiments, we see that this already leads to highly accurate answers, there might also be correlations not only between directly neighboring tables. Learning these correlations helps to further improve the accuracy of queries that span more than two tables. For instance, if there was an additional `Product` table that can be joined with the `Orders` table and the product prize is correlated with the customers region, this would not be taken into account in the *base ensemble*. In Section 5.3, we thus extend our basic procedure for ensemble creation to take dependencies among multiple tables into account.

4. QUERY COMPILATION

The main challenge of probabilistic query compilation is to translate an incoming query into an inference procedure against an ensemble of RSPNs. The class of SQL queries that *DeepDB* currently supports are of the form:

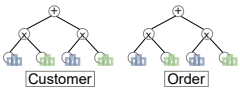
```
Q_D: SELECT AGG
     FROM T_1 JOIN T_2 ON ... JOIN T_n ON ...
     WHERE T_i.a OP LITERAL AND/OR ...
     (GROUP BY ...);
```

where `AGG` is one of the aggregations `COUNT`, `SUM`, or `AVG` over a numerical attribute, the joins are acyclic equi-joins and the filter in the `WHERE` clause are either a conjunction of filters or a disjunction. While conjunctions are supported natively by RSPNs, disjunctions are realized using the principle of inclusion and exclusion. In the filters, `OP` is one of the operators `<,>,=, <=,>=,IN`. Finally, there is an optional `GROUP BY` clause on one or several attributes.

Most importantly, in *DeepDB* the queries are supported ad-hoc, i.e. an RSPN ensemble is learned once and then arbitrary queries of the above form can be answered using our probabilistic query compilation procedure. In the following, we first describe how this procedure works for COUNT queries without grouping which is sufficient for cardinality estimation. We then show the extensions to support a broader set of aggregate queries for AQP including other aggregates (AVG and SUM) as well as grouping.

4.1 Simple COUNT Queries

In this section, we explain how we can translate COUNT queries with and without filter predicates over single tables or over joins of multiple tables using inner joins (equi-joins). These types of queries can be used already for cardinality



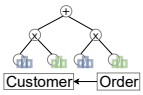
Customer

c_id	c_age	c_region	$\mathcal{F}_{C\leftarrow O}$
1	20	EU	2
2	50	EU	0
3	80	ASIA	2

Order

o_id	c_id	o_channel
1	1	ONLINE
2	1	STORE
3	3	ONLINE
4	3	STORE

(a) Ensemble with Single Tables



Customer ⟗ Order

$\mathcal{N}_C$	c_id	c_age	c_region	$\mathcal{F}'_{C\leftarrow O}$	$\mathcal{N}_O$	o_id	o_channel
1	1	20	EU	2	1	1	ONLINE
1	1	20	EU	2	1	2	STORE
1	2	50	EU	1	0	NULL	NULL
1	3	80	ASIA	2	1	3	ONLINE
1	3	80	ASIA	2	1	4	STORE

(b) Ensemble with Full Outer Join

Figure 4: Two RSPN Ensembles for the same Schema. Additional (blue) columns are also learned by the RSPNs.

estimation but also cover some cases of aggregate queries for AQP. For answering the simple COUNT queries, we distinguish three cases of how queries can be mapped to RSPNs: (1) an RSPN exists that exactly matches the tables of the query, (2) the RSPN is larger and covers more tables, and (3) we need to combine multiple RSPNs since there is no single RSPN that contains all tables of the query.

Case 1: Exact matching RSPN available. The simplest case is a single table COUNT query with (or without) a filter predicate. If an RSPN is available for this table and N denotes the number of rows in the table, the result is simply $N \cdot P(C)$. For instance, the query

```
Q1: SELECT COUNT(*) FROM CUSTOMER C
      WHERE c_region='EU';
```

can be answered with the `CUSTOMER` RSPN in Figure 4a. The result is $|\texttt{C}| \cdot \mathbb{E}(\mathbf{1}_{\texttt{c\_region='EU'}}) = 3 \cdot \frac{2}{3} = 2$. Note that 1_C denotes the random variable being one if the condition C is fulfilled and thus $\mathbb{E}(\mathbf{1}_C) = P(C)$. While a conjunction in a filter predicates is directly supported, a disjunction could be realized using the inclusion-exclusion principle.

A natural extension for COUNT queries over joins could be to learn an RSPN for the underlying join and use the formula $|J| \cdot P(C)$ where the size of the joined tables without applying a filter predicate is $|J|$. For instance, the query

```
Q2: SELECT COUNT(*) FROM CUSTOMER C
      NATURAL JOIN ORDER O
      WHERE c_region='EU' AND
      o_channel='ONLINE';
```

could be represented as $|\texttt{C} \bowtie \texttt{O}| \cdot P(\texttt{o\_channel='ONLINE'} \cap \texttt{c\_region='EU'})$ which is $4 \cdot \frac{1}{4} = 1$.

However, joint RSPNs over multiple tables are learned over the full outer join. By using full outer joins we preserve all tuples of the original tables and not only those that have one or more join partner in the corresponding table(s). This way we are able for example to answer also single table queries from a joint RSPN, as we will see in Case 2. The additional NULL tuples that result from a full outer join must be taken into account when answering an inner join query. For instance, the second customer in Figure 4b does not have any orders and thus should not be counted for query Q_2. To make it explicit which tuples have no join partner and thus would not be in the result of an inner join, we add an additional column $\mathcal{N}_T$ for every table such as in the ensemble in Figure 4b. This column is also learned by the RSPN and can be used as an additional filter column to eliminate tuples that do not have a join partner for the join query given. Hence, the complete translation of query Q_2 for the RSPN learned over the full outer join in Figure 4b is $|\texttt{C} \fojoin \texttt{O}| \cdot P(\texttt{o\_channel='ONLINE'} \cap \texttt{c\_region='EU'} \cap \mathcal{N}_O = 1 \cap \mathcal{N}_C = 1) = 5 \cdot \frac{1}{5} = 1$.

Case 2: Larger RSPN available. The second case is that we have to use an RSPN that was created on a set of joined tables, however, the query only needs a subset of those tables. For example, let us assume that the query Q_1 asking for European customers is approximated using an RSPN learned over a full outer join of customers and orders such as the one in Figure 4b. The problem here is that customers with multiple orders would appear several times in the join and thus be counted multiple times. For instance, the ratio of European customers in the full outer join is 3/5 though two out of three customers in the dataset are European.

To address this issue, for each foreign key relationship $S \leftarrow P$ between tables P and S we add a column $\mathcal{F}_{S\leftarrow P}$ to table S denoting how many corresponding join partners a tuple has. We call these *tuple factors* and later use them as correction factor. For instance, in the customer table in Figure 4a for the first customer the tuple factor is two since there are two tuples in the order table for this customer. It is important to note that tuple factors have to be computed only once per pair of tables that can be joined via a foreign key. In *DeepDB*, we do this when the RSPNs for a given database are created and our update procedure changes those values as well. Tuple factors are included as additional column and learned by the RSPNs just as usual columns. When used in a join, we denote them as $\mathcal{F}'_{S\leftarrow P}$. Since we are working with outer joins, the value of $\mathcal{F}'$ is at least 1.

We can now express the query counting European customers as $|\texttt{C} \fojoin \texttt{O}| \cdot \mathbb{E}(1/\mathcal{F}'_{C\leftarrow O} \cdot \mathbf{1}_{\texttt{c\_region='EU'}} \cdot \mathcal{N}_C)$ which results in $5 \cdot \frac{1/2+1/2+1}{5} = 2$. First, this query both includes the first customer (who has no orders) because the RSPN was learned on the full outer join. Second, the query also takes into account that the second and third customer have two orders each by normalizing them with their tuple factor $\mathcal{F}'_{C\leftarrow O}$. In general, we can define the procedure to compile a query requiring only a part of an RSPN as follows:

THEOREM 1. *Let Q be a `COUNT` query with a filter predicate C which only queries a subset of the tables of a full outer join J. Let $\mathcal{F}'(Q, J)$ denote the product of all tuple factors that cause result tuples of Q to appear multiple times in J. The result of the query is equal to:*

$$|J| \cdot \mathbb{E}\left(\frac{1}{\mathcal{F}'(Q,J)} \cdot 1_C \cdot \prod_{T \in Q} \mathcal{N}_T\right)$$

For an easier notation, we write the required factors of query Q as $\mathbf{F}(Q)$. The expectation $\mathbb{E}(\mathbf{F}(Q))$ of theorem 1 can be computed with an RSPN because all columns are learned.

Case 3: Combination of multiple RSPNs. As the last case, we handle a `COUNT` query that needs to span over multiple RSPNs. We first handle the case of two RSPNs and extend the procedure to n RSPNs later. In this case, the query can be split into two subqueries Q_L and Q_R, one for each RSPN. There can also be an overlap between Q_L and Q_R which we denote as Q_O (i.e., a join over the shared common tables). The idea is first to estimate the result of Q_L using the first RSPN. We then multiply this result by the ratio of tuples in Q_R vs. tuples in the overlap Q_O. Intuitively, this expresses how much the missing tables not in Q_L increase the COUNT value of the query result.

For instance, there is a separate RSPN available for the `Customer` and the `Order` table in Figure 4a. The query Q_2, as shown before, would be split into two queries Q_L and Q_R, one against the RSPN built over the `Customer` table and the other one over the RSPN for the `Order` table. Q_O is empty in this case. The query result of Q_2 can thus be expressed using all these sub-queries as:

$$|\texttt{C}| \cdot \underbrace{\mathbb{E}(\mathbf{1}_{\texttt{c\_region='EU'}} \cdot \mathcal{F}_{C \leftarrow O})}_{Q_L} \cdot \underbrace{\mathbb{E}(\mathbf{1}_{\texttt{o\_channel='ONLINE'}})}_{Q_R}$$

which results in $3 \cdot \frac{2+0}{3} \cdot \frac{2}{4} = 1$. The intuition of this query is that the left-hand side that uses Q_L computes the orders of European customers while the right-hand side computes the fraction of orders that are ordered online out of all orders.

We now handle the more general case that the overlap is not empty and that there is a foreign key relationship $S \leftarrow T$ between a table S in Q_O (and Q_L) and a table T in Q_R (but not in Q_L). In this case, we exploit the tuple factor $\mathcal{F}_{S \leftarrow T}$ in the left RSPN. We now do not just estimate the result of Q_L but of Q_L joined with the table T. Of course this increases the overlap which we now denote as Q'_O. As a general formula for this case, we obtain Theorem 2:

THEOREM 2. *Let the filter predicates and tuple factors of $Q_L \setminus Q_O$ and $Q_R \setminus Q_O$ be conditionally independent given the filter predicates of Q_O. Let $S \leftarrow T$ be the foreign key relationship between a table S in Q_L and a table T in Q_R that we want to join. The result of Q is equal to*

$$|J_L| \cdot \mathbb{E}\left(\mathbf{F}(Q_L) \cdot \mathcal{F}_{S \leftarrow T}\right) \cdot \frac{\mathbb{E}\left(\mathbf{F}(Q_R)\right)}{\mathbb{E}\left(\mathbf{F}(Q'_O)\right)}.$$

Independence across RSPNs is often given since our ensemble creation procedure preferably learns RSPNs over correlated tables as discussed in Section 3.

Alternatively, we can start the execution with Q_R. In our example query Q_2 where Q_R is the query over the orders table, we can remove the corresponding tuple factor $\mathcal{F}_{C \leftarrow O}$ from the left expectation. However, we then need to normalize Q_L by the tuple factors to correctly compute the fraction of customers who come from Europe. To that end, the query Q_2 can alternatively be computed using:

$$|\texttt{O}| \cdot \mathbb{E}(\mathbf{1}_{\texttt{o\_channel='ONLINE'}}) \cdot \frac{\mathbb{E}\left(\mathbf{1}_{\texttt{c\_region='EU'}} \cdot \mathcal{F}_{C \leftarrow O} \mid \mathcal{F}_{C \leftarrow O}\right)}{\mathbb{E}\left(\mathcal{F}_{C \leftarrow O} \mid \mathcal{F}_{C \leftarrow O} > 0\right)}$$

Execution Strategy. If multiple RSPNs are required to answer a query, we have several possible execution strategies. Our goal should be to handle as many correlations between filter predicates as possible because predicates across RSPNs are considered independent. For instance, assume we have both the `Customer`, `Order` and `Customer-Order` RSPNs of Figure 4 in our ensemble, and a join of customers and orders would have filter predicates on `c_region`, `c_age` and `o_channel`. In this case, we would prefer the `Customer-Order` RSPN because it can handle all pairwise correlations between filter columns (`c_region-c_age`, `c_region-o_channel`, `c_age-c_channel`). Hence, at runtime we greedily use the RSPN that currently handles the filter predicates with the highest sum of pairwise RDC values. We also experimented with strategies enumerating several probabilistic query compilations and using the median of their predictions. However, this was not superior to our RDC-based strategy. Moreover, the RDC values have already been computed to decide which RSPNs to learn. Hence, at runtime this strategy is very compute-efficient.

The final aspect is how to handle joins spanning over more than two RSPNs. To support this, we can apply Theorem 2 several times.

4.2 Other Aggregate Queries

So far, we only looked into COUNT queries without group-by statements. In the following, we first discuss how we extend our query compilation to also support AVG and SUM queries before we finally explain group-by statements as well as outer joins.

AVG Queries. We again start with the case that we have an RSPN that exactly matches the tables of a query and later discuss the other cases. For this case, queries with `AVG` aggregates can be expressed as conditional expectations. For instance, the query

```
Q3: SELECT AVG(c_age) FROM CUSTOMER C
      WHERE c_region='EU';
```

can be formulated as $|\texttt{C}| \cdot \mathbb{E}(\texttt{c\_age} \mid \texttt{c\_region='EU'})$ with the ensemble in Figure 4a.

However, for the case that an RSPNs spans more tables than required, we cannot directly use this conditional expectation because otherwise customers with several orders would be weighted higher. Again, normalization by the tuple factors is required. For instance, if the RSPN spans customers and orders as in Figure 4b for query Q_3 we use

$$\frac{\mathbb{E}\left(\frac{\texttt{c\_age}}{\mathcal{F}'_{C \leftarrow O}} \mid \texttt{c\_region='EU'}\right)}{\mathbb{E}\left(\frac{1}{\mathcal{F}'_{C \leftarrow O}} \mid \texttt{c\_region='EU'}\right)} = \frac{20/2 + 20/2 + 50}{1/2 + 1/2 + 1} = 35.$$

In general, if an average query for the attribute A should be computed for a join query Q with filter predicates C on an RSPN on a full outer join J, we use the following expectation to answer the average query:

$$\mathbb{E}\left(\frac{A}{\mathcal{F}'(Q,J)} \mid C\right) / \mathbb{E}\left(\frac{1}{\mathcal{F}'(Q,J)} \mid C\right).$$

The last case is where the query needs more than one RSPN to answer the query. In this case, we only use one RSPN that contains A and ignore some of the filter predicates that are not in the RSPN. As long as A is independent of these attributes, the result is correct. Otherwise, this is just an approximation. For selecting which RSPN should be used, we again prefer RSPNs handling stronger correlations between A and P quantified by the RDC values. The RCDs can also be used to detect cases where the approximation would ignore strong correlations with the missing attributes in P.

SUM Queries. For handling `SUM` queries we run two queries: one for the `COUNT` and `AVG` queries. Multiplying them yields the correct result for the `SUM` query.

Group-by Queries. Finally, a `group by` query can be handled also by several individual queries with additional filter predicates for every group. This means that for n groups we have to compute n times more expectations than for the corresponding query without grouping. In our experimental evaluation, we show that this does not cause performance issues in practice if we compute the query on the model.

Outer Joins. Query compilation can be easily extended to support outer joins as well (left/right/full). The idea is that we only filter out tuples that have no join partner for all inner joins (case 1 and 2 in Section 4.1) but not for outer joins (depending on the semantics of the outer join). Moreover, in case 3, the tuple factors $\mathcal{F}$ with value zero have to be handled as value one to support the semantics of the corresponding outer join.

5. DEEPDB EXTENSIONS

We now describe important extensions of our basic framework presented before.

5.1 Support for Confidence Intervals

Especially for AQP confidence intervals are important. However, SPNs do not provide those. After the probabilistic query compilation the query is expressed as a product of expectations. We first describe how to estimate the uncertainty for each of those factors and eventually how a confidence interval for the final estimate can be derived.

First, we split up expectations as a product of probabilities and conditional expectations. For instance, the expectation $\mathbb{E}(X \cdot 1_C)$ would be turned into $\mathbb{E}(X \mid C) \cdot P(C)$. This allows us to treat all probabilities for filter predicates C as a single binomial variable with probability $p = \prod P(C_i)$ and the amount of training data of the RSPN as $n_{samples}$. Hence, the variance is $\sqrt{n_{samples} p(1-p)}$. For the conditional expectations, we use the Koenig-Huygens formula $\mathbb{V}(X \mid C) = \mathbb{E}(X^2 \mid C) - \mathbb{E}(X \mid C)^2$. Note that also squared factors can be computed with RSPNs since the square can be pushed down to the leaf nodes. We now have a variance for each factor in the result.

For the combination we need two simplifying assumptions: (i) the estimates for the expectations and probabilities are independent, and (ii) the resulting estimate is normally distributed. In our experimental evaluation, we show that despite these assumptions our confidence intervals match those of typical sample-based approaches.

We can now approximate the variance of the product using the independence assumption by recursively applying the standard equation for the product of independent random variables: $\mathbb{V}(XY) = \mathbb{V}(X)\mathbb{V}(Y) + \mathbb{V}(X)\mathbb{E}(Y)^2 + \mathbb{V}(Y)\mathbb{E}(X)^2$. Since we know the variance of the entire probabilistic query compilation and we assume that this estimate is normally distributed we can provide confidence intervals.

5.2 Support for Updates

The intuition of our update algorithm is to regard RSPNs as indexes. Similar to these, insertions and deletions only affect subtrees and can be performed recursively. Hence, the updated tuples recursively traverse the tree and passed weights of sum nodes and the leaf distributions are adapted. Our approach supports *insert* and *delete* operations, where an *update*-operation is mapped to a pair of *delete* and *insert* operations.

Algorithm 1 Incremental Update

```
1:  procedure UPDATE_TUPLE(node, tuple)
2:      if leaf-node then
3:          update_leaf_distribution(node, tuple)
4:      else if sum-node then
5:          nearest_child ← get_nearest_cluster(node, tuple)
6:          adapt_weights(node, nearest_child)
7:          update_tuple(nearest_child, tuple)
8:      else if product-node then
9:          for child in child_nodes do
10:             tuple_proj ← project_to_child_scope(tuple)
11:             update_tuple(child, tuple_proj)
```

The update algorithm is depicted in Algorithm 1. Since it is recursive, we have to handle sum, product and leaf nodes. At sum nodes (line 4) we have to identify to which child node the inserted (deleted) tuple belongs to determine which weight has to be increased (decreased). Since children of sum nodes represent row clusters found by *KMeans* during learning [29], we can compute the closest cluster center (line 5), increase (decrease) its weight (line 6) and propagate the tuple to this subtree (line 7). In contrast, product nodes (line 8) split the set of columns. Hence, we do not propagate the tuple to one of the children but split it and propagate each tuple fragment to the corresponding child node (lines 9-11). Arriving at a leaf node, only a single column of the tuple is remaining. We now update the leaf distribution according to the column value (line 2).

This approach does not change the structure of the RSPN, but only adapts the weights and the histogram values. If there are new dependencies as a result of inserts they are not represented in the RSPN. As we show in Section 6.1 on a real-word dataset, this typically does not happen, even for high incremental learning rates of 40%. Nevertheless, in case of new dependencies the RSPNs have to be rebuilt. This is solved by checking the database cyclically for changed dependencies by calculating the pairwise RDC values as explained in Section 5.3 on column splits of product nodes. If changes are detected in the dependencies, the affected RSPNs are regenerated. As for traditional indexes, this can be done in the background.

5.3 Ensemble Optimization

As mentioned before, we create an ensemble of RSPNs for a given database. The base ensemble contains either RSPNs for single tables or they span over two tables connected by a foreign key relationship if they are correlated. Correlations occurring over more than two tables are ignored so far since they lead to larger models and higher training times. In the following, we thus discuss an extension of our ensemble creation procedure that allows a user to specify a training budget (in terms of time or space) and *DeepDB* selects the additional larger RSPNs that should be created.

To quantify the correlations between tables, as mentioned already before, we compute the pairwise RDC values for every pair of attributes in the schema. For every pair of tables, we define the maximum RDC value between two columns $\max_{c \in T_i, c' \in T_j} rdc(c, c')$ as the dependency value. The dependency value indicates which tables should appear in the same RSPN and which not. For every RSPN the goal is to achieve a high mean of these pairwise maximal RDC values.

This ensures that only tables with high pairwise correlation are merged in an RSPN.

The limiting factor (i.e., the constraint) for the additional RSPN ensemble selection should be the budget (i.e., extra time compared to the base ensemble) we allow for the learning of additional RSPNs. For the optimization procedure, we define the maximum learning costs as a factor B relative to the learning costs of the base ensemble C_{Base}. Hence, a budget factor $B = 0$ means that only the base ensemble would be created. For higher budget factors $B > 0$, additional RSPNs over more tables are learned in addition. If we assume that an RSPN r among the set of all possible unique RSPNs R has a cost $C(r)$, then we could formulate the optimization problem as a minimization of $\sum_{r \in \mathcal{E}} \{\max_{c \in T_i, c' \in T_j} rdc(c, c') \mid T_i, T_j \in r\}$ subject to $\sum_{r \in \mathcal{E}} C(r) \leq B \cdot C_{Base}$.

However, estimating the real cost $C(r)$ (i.e., time) to build an RSPN r is hard and thus we can not directly solve the optimization procedure. Instead, we estimate the relative cost to select the RSPN r that has the highest mean RDC value and the lowest relative creation cost. To model the relative creation cost, we assume that the costs grow quadratic with the number of columns $cols(r)$ since the RDC values are created pairwise and linear in the number of rows $rows(r)$. Consequently, we pick the RSPN r with highest mean RDC and lowest cost which is $cols(r)^2 \cdot rows(r)$ as long as the maximum training time is not exceeded.

6. EXPERIMENTAL EVALUATION

In this Section, we show that *DeepDB* outperforms state-of-the-art systems for both cardinality estimation and AQP. The RSPNs we used in all experiment were implemented in Python as extensions of SPFlow [30]. As hyperparameters, we used an RDC threshold of 0.3 and a minimum instance slice of 1% of the input data, which determines the granularity of clustering. Moreover, we used a budget factor of 0.5, i.e. the training of the larger RSPNs takes approximately 50% more training time than the base ensemble. We determined these hyperparameters using a grid-search, which gave us the best results across different datasets.

6.1 Experiment 1: Cardinality Estimation

Workload and Setup. As in [16, 19], we use the JOB-light benchmark as workload for all approaches (*DeepDB* and baselines). The benchmark uses the real-world IMDb database and defines 70 queries. Furthermore, we additionally defined a synthetic query set of 200 queries were joins from three to six tables and one to five filter predicates appear uniformly on the IMDb dataset. We use this query set to compare the generalization capabilities of the learned approaches.

As baselines, we used the following learned and traditional approaches: First we trained a Multi-Set Convolutional Network (MCSN) [16] as a learned baseline. MCSNs are specialized deep neural networks using the join paths, tables and filter predicates as inputs. As representative of a synopsis-based technique, we implemented an approach based on wavelets [5]. The main idea of [5] is that one wavelet is built per table. Moreover, query operators (e.g., joins) can be executed directly on the wavelet representation. We have chosen this approach because it is similar to *DeepDB* since the tables that are joined by queries do not have to be known beforehand. We also implemented an approach called *Perfect Selectivities*. In this approach, we use an oracle that returns the true cardinalities for single tables. This approach can be seen as the best case for any synopsis-based approach that supports ad-hoc queries by combining "perfect" synopsis on single tables. Finally, we use the standard cardinality estimation of Postgres 11.5 as well as online random sampling and Index-Based Join Sampling (IBJS) [20] as a non-learned baselines. Similar to *DeepDB*, IBJS considers potential correlations across tables when sampling. For *DeepDB*, we use the hyper-parameters discussed before and a sample size of 10M samples for constructing RSPNs if not noted otherwise.

Table 1: Estimation Errors for the JOB-light Benchmark

	median	90th	95th	max
DeepDB	1.34	2.50	3.16	39.63
DeepDB (Storage Opt.)	1.32	4.14	5.74	72.00
Perfect Selectivities	2.08	9	11	33
MCSN	3.22	65	143	717
Wavelets	7.64	9839	15332	564549
Postgres	6.84	162	817	3477
IBJS	1.67	72	333	6949
Random Sampling	5.05	73	10371	49187

Training Time and Storage Overhead. In contrast to other learned approaches for cardinality estimation [16, 42], no dedicated training data is required for *DeepDB*. Instead, we just learn a representation of the data. The training of the base ensemble takes 48 minutes. The creation time includes the data preparation time to sample and compute the tuple factors as introduced in Section 4.1. In contrast, for the MCSN [16] approach, 100k queries need to be executed to collect cardinalities resulting in 34 hours of training data preparation time (when using Postgres). Moreover, the training of the neural network takes only about 15 minutes on a Nvidia V100 GPU. As we can see, our training time is much lower since we do not need to collect any training data for the workload. Another advantage is that we do not have to re-run the queries once the database is modified. Instead, we provide an efficient algorithm to update RSPNs in *DeepDB* as discussed in Section 3.2.

Another dimension is the storage footprint needed for the different approaches. While the sampling-based approaches, i.e., IBJS and random sampling, do not incur a storage overhead, their limiting factor is the number of samples which is determined by the latency. All other approaches require only a few KB to MB of storage for the IMDb database of the JOB-light benchmark (which uses 3.7 GB disk space). The storage overhead of *DeepDB* is 28.9MB vs. 2.6 MB for MSCN and just 60kB for Postgres that uses histograms with just 100 buckets by default (however with the lowest accuracy as we show next). For the wavelet approach we used $20k$ wavelet coefficients to allow as much storage as the standard version of *DeepDB* requires. In addition, we also created a storage-optimized version of *DeepDB*, which has a similar storage footprint as MCSNs by reducing the number of samples. In contrast to *DeepDB*, allowing a larger storage overhead for MSCNs by for instance adding hidden layers does not improve the performance since we already use the optimized hyperparameters of [16]. As we show next, the storage-optimized version of *DeepDB* can provide accuracies that are still significantly better than all other baselines including MCSN. Furthermore, while there has been a line of research optimizing the storage footprint of DNNs

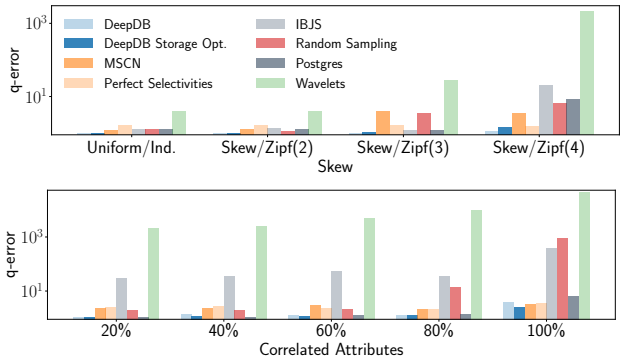


Figure 5: Mean Estimation Errors for Synthetic Data.

there are no comparable approaches for SPNs. We believe that future research will reduce the storage requirements for *DeepDB* even further. However, we think that even a few MB of storage for an entire database of several GB is still acceptable for more accurate cardinality estimates.

Estimation Quality. The prediction quality of cardinality estimators is usually evaluated using the q-error, which is the factor by which an estimate differs from the real execution join size. For example, if the real result size of a join is 100, the estimates of 10 or 1k tuples both have a q-error of 10. Using the ratio instead of an absolute or quadratic error captures the intuition that for making optimization decisions only relative differences matter. In Table 1, we depict the median, 90-th and 95-th percentile and max q-errors for the JOB-light benchmark of our approach compared to the other baselines. We additionally provide the q-errors for a storage-optimized version of *DeepDB*, which relies only on a base ensemble and 100k samples per RSPN. As we can see, both *DeepDB* and the storage-optimized version outperform the best competitors often by orders of magnitude. While IBJS provides a low q-error in the median, the advantage of learned MCSNs is that they outperform traditional approaches by orders of magnitude for the higher percentiles and are thus more robust. *DeepDB* not only outperforms IBJS in the median, but provides additional robustness having a 95-th percentile for the q-errors of 3.16 vs. 143 (MCSN). The q-errors of both Postgres and random sampling are significantly larger both for the medians and the higher percentiles. Finally, wavelets have the highest error since they suffer from the curse of dimensionality (as we show later in Figure 12). While *Perfect Selectivities* which is based on an oracle provides errors better than wavelets it is still worse than *DeepDB* since it does not take correlations across tables into account.

Synthetic Data. In order to further investigate the trade-offs of the different approaches, we implemented a synthetic data generator for the IMDb schema (such that we can then run the JOB-light benchmark). First, we generated data with uniform distributions without any correlations. Second, we varied the characteristics that make cardinality estimation hard in reality; i.e., we used skewed distributions and correlations between different columns. We then used the same approaches as before to provide cardinality estimates for the original 70 JOB-light queries and report the mean q-errors of queries not having a cardinality of zero because otherwise the q-error is not defined. Figure 5 shows the mean q-errors (log-scale) for varying degrees of skew

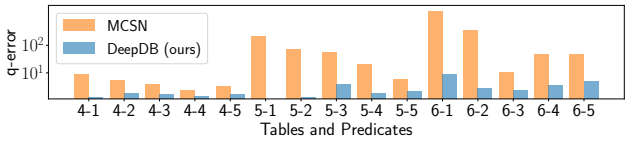


Figure 6: Median q-errors (logarithmic Scale) for different Join Sizes (4,5,6) and Number of Filter Predicates (1-5).

(upper plot) and varying degrees of correlation (lower plot). We can see that *DeepDB* and the storage optimized version can both outperform all other baselines. While on uniform/independent data, *DeepDB* provides no significant advantage even over simple techniques such as random sampling or Postgres (as expected), *DeepDB* outperforms the other baselines for higher degrees of skew/correlation. For higher degrees of skew/correlation, the approaches based on sampling (random sampling, IBJS) as well as Postgres all degrade significantly. Compared to those approaches, MSCN can handle skew/correlation much better but still degrades which we attribute again to the coverage of the training queries. Finally, wavelets again provide the lowest accuracy on all configurations since they suffer from the curse of dimensionality similar to the real-world data in Figure 1.

Generalization Capabilities. Especially for learned approaches, the question of generalization is important, i.e., how well the models perform on previously unseen queries. For instance, by default the MCSN approach is only trained with queries up to three joins because otherwise the training data generation would be too expensive [16]. Similarly in our approach, in the ensemble only few RSPNs with large joins occur because otherwise the training would also be too expensive. However, both approaches support cardinality estimates for unseen queries.

To compare both learned approaches, we randomly generated queries for joins with four to six tables and one to five selection predicates for the IMDb dataset. In Figure 6, we plot the resulting median q-errors for both learned approaches: *DeepDB* and MCSN [16]. The median q-errors of *DeepDB* are orders of magnitude lower for larger joins. Additionally, we can observe that, for the MCSN approach, the estimates tend to become less accurate for queries with fewer selection predicates. One possible explanation is that more tuples qualify for such queries and thus higher cardinalities have to be estimated. However, since there are at most three tables joined in the training data such higher cardinality values are most likely not predicted. Thus, using RSPNs leads to superior generalization capabilities.

Updates. In this experiment, we show the update capabilities of RSPNs. The easy and efficient updateability is a clear advantage of *DeepDB* compared to deep-learning based approaches for cardinality estimation [16, 42]. To show the effects of updates on the accuracy, we first learn the base RSPN ensemble on a certain share of the full IMDb dataset and then use the remaining tuples to update the database.

To ensure a realistic setup, we split the IMDb dataset based on the production year (i.e., newer movies are inserted later). As depicted in Table 2 the q-errors do not change significantly for updated RSPNs even if the update fraction increases; i.e., if we split on earlier production years. For building the RSPNs, we use zero as the budget factor to demonstrate that even a base RSPN ensemble provides good estimates after updates. This is also the reason why

Table 2: Estimation Errors for JOB-light after Updates.

Temporal Split	< 2019 (0%)	< 2011 (4.7%)	< 2009 (9.3%)	< 2004 (19.7%)	< 1991 (40.1%)
Median	1.22	1.28	1.31	1.34	1.41
90th	3.45	3.17	3.23	3.60	4.06
95th	4.77	4.30	3.83	4.07	4.35

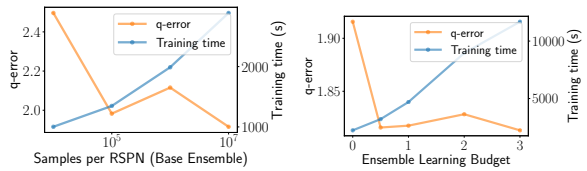


Figure 7: Q-errors and Training Time (in s) for varying Budget Factors and Sample Sizes.

the estimation errors slightly deviate from Table 1. Our results in Table 2 show that with a higher fraction of updates, the accuracy drops only slightly. The reason is that the structure of the RSPN tree is not changed by updates, but only the parameters of the RSPNs which might not be the optimal structure anymore if the data distributions/correlations change due to the updates. In case the accuracy drops beyond a threshold, *DeepDB* can still decide to recreate the RSPN offline based on the new data.

Parameter Exploration. Finally, in the last experiment, we explore the tradeoff between ensemble training time and prediction quality of *DeepDB*. We first vary the budget factor used in the ensemble selection between zero (i.e. learning only the base ensemble with one RSPN per join of two tables) and B=3 (i.e. the training of the larger RSPNs takes approximately three times longer than the base ensemble) while using 10^7 samples per RSPN. We then use the resulting ensemble to evaluate 200 queries with three to six tables and one to five selection predicates. The resulting median q-errors are shown in Figure 7. For higher budget factors the means are improving but already saturate at $B = 0.5$. This is because there are no strong correlations in larger joins that have not already been captured in the base ensemble.

Moreover, we evaluate the effect of the sampling to reduce the training time. In this experiment we vary the sample size from 1000 to 10 million. We observe that while the training time increases, the higher we choose this parameter, the prediction quality improves (from 2.5 to 1.9 in the median). In summary, the training time can be significantly reduced if slight compromises in prediction quality are acceptable. When minimization of training time is the more important objective we can also fall back and only learn RSPNs for all single tables and no joins at all. This reduces the ensemble training time to just five minutes. However, even this cheap strategy is still competitive. For JOB-light this ensemble has a median q-error of 1.98, a 90-th percentile of 5.32, a 95-th percentile of 8.54 and a maximum q-error of 186.53. Setting this in perspective to the baselines, this ensemble still outperforms state of the art for the higher percentiles and only Index Based Join Sampling is slightly superior in the median. This again proves the robustness of RSPNs.

Latencies. The estimation latencies for cardinalities using *DeepDB* are currently in the order of μs to *ms* which suffices for complex join queries that often run for multiple seconds on larger datasets. If more complex predicates spanning over several columns are used or more tables are involved in the join the latencies increase. In Figure 8 we investigate

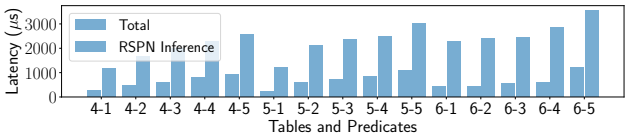


Figure 8: Latencies of *DeepDB* for different Join Sizes (4,5,6) and Number of Filter Attributes (1-5).

this effect in more detail. We report both the latency required for the RSPN inference and the total time including the overhead of translating the queries to expectations and probabilities using our probabilistic query compilation procedure. The RSPN inference is efficient because C++ code is compiled automatically for the trained RSPNs similar to [40]. As we see, while the latencies increase for more complex predicates and joins, they are still around 3ms in the worst case and in the range of μs for easier queries. In future, we plan to optimize not just RSPN inference but also the overhead of query translation to bring the total latency even closer to only the RSPN inference.

6.2 Experiment 2: AQP

Workload and Setup. We evaluated the approaches on both a synthetic dataset and a real-world dataset. As synthetic dataset, we used the Star Schema Benchmark (SSB) [32] with a scale factor of 500 with the standard queries (denoted by S1.1-S4.3). As the real-world dataset, we used the Flights dataset [1] with queries ranging from selectivities between 5% an 0.01% covering a variety of group by attributes, `AVG`, `SUM` and `COUNT` queries (denoted by F1.1-F5.2). To scale the dataset up to 1 billion records we used IDEBench [9].

As baselines we used VerdictDB [33], Wander Join/XDB [21] and the Postgres `TABLESAMPLE` command (using random samples). VerdictDB is a middleware that can be used with any database system. It creates a stratified and a uniform sample for the fact tables to provide approximate queries. For VerdictDB, we used the default sample size (1% of the full dataset) for the Flights dataset. For the SSB benchmark, this led to high query latencies and we thus decided to choose a sample size such that the query processing time was two seconds on average. Wander Join is a join sampling algorithm leveraging secondary indexes to generate join samples quickly. We set the time bound also to two seconds for a fair comparison and only evaluated this algorithm for datasets with joins. To this end, we created all secondary indexes for joins and predicates. For `TABLESAMPLE` we chose a sample size such that the queries take two seconds on average. For *DeepDB*, we use a sample size of 10M samples for the Flights dataset and 1M samples for the SSB dataset to construct RSPNs.

Training Time and Storage Overhead. The training took just 17 minutes for the SSB dataset and 3 minutes for the Flights dataset. The shorter training times compared to the IMDb dataset are due to fewer cross-table correlations and hence fewer large RSPN models in the ensemble. For VerdictDB, uniform and stratified samples have to be created from the dataset. This took 10 hours for the flights dataset and 6 days for the SSB benchmark using the standard setup of VerdictDB.For wander join, secondary indexes had to be created also requiring several hours for the SSB dataset.

For the Flights dataset the model size of *DeepDB* is 2.2 MB (vs. 11.4 MB for VerdictDB) and for the SSB dataset

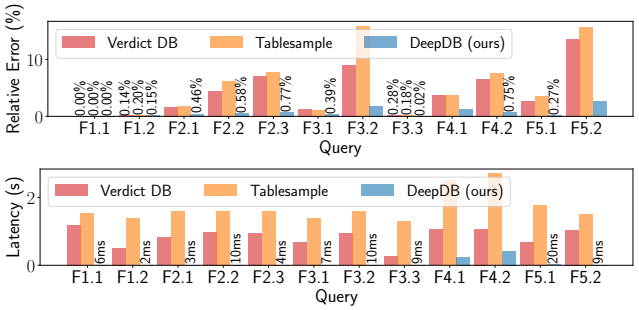


Figure 9: Average Relative Error and Latencies for the Flights dataset.

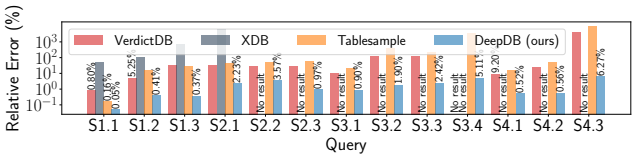


Figure 10: Average Relative Error for SSB dataset. Note the logarithmic Scale for the Errors.

DeepDB requires 34.4 MB (vs. 30.7 MB for VerdictDB). In contrast to *DeepDB* and VerdictDB, XDB and Postgres `TABLESAMPLE` compute samples online and thus do not have any additional (offline) storage overhead.

Accuracy and Latency. For AQP two dimensions are of interest: the quality of the approximation and the runtime of queries. For reporting the quality of the approximation we use the relative error which is defined as $\frac{|a_{true} - a_{predicted}|}{a_{true}}$ where a_{true} and $a_{predicted}$ are the true and predicted aggregate function, respectively. If the query is a group by query, several aggregates have to be computed. In this case, the relative error is averaged over all groups.

For the Flights dataset, as shown in Figure 9 we can observe that *DeepDB* always has the lowest average relative error. This is often the case for queries with lower selectivities where sample-based approaches have few tuples that satisfy the selection predicates and thus the approximations are very inaccurate. In contrast, *DeepDB* does not rely on samples but models the data distribution and leverages the learned representation to provide estimates. For instance, for query 11 with a selectivity of 0.5% VerdictDB and the `TABLESAMPLE` strategy have an average relative error of 15.6% and 13.6%, respectively. In contrast, the average relative error of *DeepDB* is just 2.6%.

Moreover, the latencies for both `TABLESAMPLE` and VerdictDB are between one and two seconds on average. In contrast, *DeepDB* does not rely on sampling but on evaluating the RSPNs. This is significantly faster resulting in a maximum latency of 31ms. This even holds true for queries with several groups where more expectations have to be computed (at least one additional per different group).

The higher accuracies of *DeepDB* are even more severe for the SSB benchmark. The queries have even lower selectivities between 3.42% and 0.0075% for queries 1 to 12 and 0.00007% for the very last query. This results in very inaccurate predictions of the sample-based approaches. Here, the average relative errors are orders of magnitude lower for *DeepDB* always being less than 6%. In contrast, VerdictDB, Wander Join and the `TABLESAMPLE` approach often have average relative errors larger than 100%. Moreover, for some

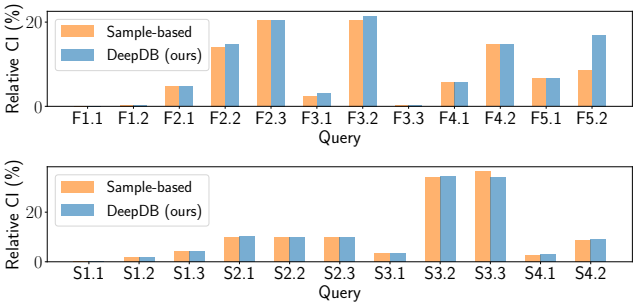


Figure 11: True and predicted relative length of the Confidence Intervals.

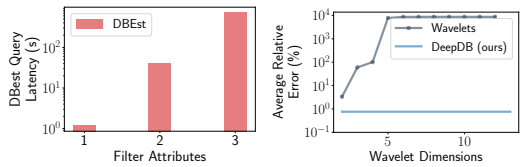


Figure 12: Microbenchmarks for non ad-hoc Approaches.

queries no estimate can be given at all because no samples are drawn that satisfy the filter predicates. However, while the other approaches take two seconds to provide an estimate, *DeepDB* requires no more than 293ms in the worst case. In general latencies are lower for queries with fewer groups because less expectations have to be computed.

Confidence Intervals. In this experiment, we evaluate how accurate the confidence intervals predicted by *DeepDB* are. To this end, we measure the relative confidence interval length defined as: $\frac{a_{predicted} - a_{lower}}{a_{predicted}}$, where $a_{predicted}$ is the prediction and a_{lower} is the lower bound of the confidence interval. This relative confidence interval length is compared to the confidence interval of a sample-based approach. For this we draw 10 million samples (as many samples as our models use for learning in this experiment) and compute estimates for the average, count and sum aggregates. We then compute the confidence intervals of these estimates using standard statistical methods. The resulting confidence interval lengths can be seen as ground truth and are compared to the confidence intervals of our system in Figure 11. Note that we excluded queries where less than 10 samples fulfilled the filter predicates. In these cases the estimation of a standard deviation has itself a too high variance.

In all cases, the confidence intervals of *DeepDB* are very good approximations of the true confidence intervals. The only exception is query F5.2 for the Flights dataset which is a difference of two `SUM` aggregates. In this case, assumption (i) of Section 5.1 does not hold: the probabilities and expectation estimates cannot be considered independent. This is the case because both `SUM` aggregates contain correlated attributes and thus the confidence intervals are overestimated. However, note that in the special case of the difference of two sum aggregates the AQP estimates are still very precise as shown in Figure 9 for the same query F5.2. Such cases can easily be identified and only occur when arithmetic expressions of several aggregates should be estimated.

Non ad-hoc Approaches. We now compare the accuracy of *DeepDB* against approaches that either require a priori information about the workload or can make use of it to provide better accuracies. The results of *DeepDB* and the other approaches on the Flights dataset are shown in Figure 13.

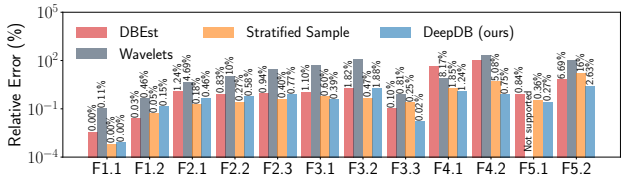


Figure 13: Relative Error for *DeepDB* vs. Approaches with a priori Knowledge.

First, we compared *DeepDB* to the wavelet approach [5] that we used before. As discussed, this approach does not require a priori information but a priori knowledge can be used to optimize wavelets since they do not scale to a large number of dimensions. Thus, if the required column combinations of all queries are known beforehand, we can construct optimal (minimal-sized) wavelets on a per-query basis. However, as shown in Figure 13, even when using a priori information for wavelets, *DeepDB* still outperforms them. We investigated this effect further and found that the accuracy of wavelets significantly drops even for a small number of dimensions as shown in Figure 12 (right).

Second, we also compared *DeepDB* to other approaches that require a priori information: (1) stratified sampling as used in BlinkDB [2] which mitigates effects of skew and (2) a recent learned AQP approach called DBEst [25]. It is important to note that different from *DeepDB* these approaches cannot answer ad-hoc queries with column combinations not covered in the a priori information which is possible in *DeepDB* by combining multiple RSPNs using our probabilistic query compilation approach as discussed in Section 4. In the following, we discuss the results if a priori information for all queries on the Flights dataset is available. As shown in Figure 13 for these queries, stratified sampling can provide accuracies comparable to *DeepDB*. However, interestingly if a query has highly selective filter conditions on one of the stratification columns (as in query F5.2 or F4.2), *DeepDB* is still superior. Moreover, while for DBEst the accuracies are also comparable to *DeepDB* we noticed that for some queries it takes up to 20s to provide an answer. We investigated this effect closer in Figure 12 (left) finding that the latency is exponential w.r.t. the number of filter conditions on numeric columns. This is inevitable since the approach relies on an integration over the domains of numeric columns.

7. RELATED WORK

Learned Cardinality Estimation. The problem of selectivity estimation for single tables is a special case of cardinality estimation. There is a large body of work applying different ML approaches including probabilistic graphical models [45, 12, 11], neural networks [18, 22] and deep learning density models [14] to this problem. Recently, Dutt et al. [8] suggested using lightweight tree-based models in combination with log-transformed labels. The first works applying ML to cardinality estimation including joins used simple regression models [3, 26]. More recently, Deep Learning was specifically proposed to solve cardinality estimation end-to-end [16, 42]. Woltmann et al. [48] also separate the problem of cardinality estimation on a large schema by learning models similar to [16] for certain schema sub-parts. However, two models for schema sub-parts cannot be combined to provide estimates for a larger join. Other techniques exploit learned models for overlapping subgraph templates for recurring cloud workloads [49]. All these models need a workload to be executed and used as training data which is different from our data-driven approach.

Learned AQP. Early work [38] suggests to approximate OLAP cubes by mixture models based on clusters in the data. Though greatly reducing the required storage, the approximation errors are relatively high. *FunctionDB* [43] constructs piecewise linear functions as approximation. In contrast to *DeepDB*, only continuous variables are supported. *DBEst* [25] builds models for popular queries. However, in contrast to *DeepDB* slight variations of those popular queries and no ad-hoc queries are supported. Park et al. suggested *Database Learning* [34] which builds a model from query results that is leveraged to provide approximate results for future queries. In contrast, *DeepDB* is data-driven and does not require past query results. Moreover, specialized generative models were suggested to draw samples for AQP [44]. However, this technique does not work for joins.

Probabilistic Databases. Similar to our work, probabilistic databases have used graphical models to represent joint probability distributions [47, 15, 36, 37] to overcome the tuple-independence assumption that early approaches relied on [6, 31, 41]. For instance, Markov Logic Networks (MLNs) were used to explicitly specify correlations in probabilistic databases [15, 13]. In contrast to *DeepDB*, correlations need to be manually specified and are not learned. Moreover, Rekatsinas et al. [36] instead use factor graphs to model correlations in probabilistic databases and construct annotated arithmetic circuits (AACs) which also encode a probability distribution with sum and product nodes similar to SPNs. Different from *DeepDB*, additional representations (lineage-AACs) have to be constructed on a per-query basis whereas RSPNs are data-driven and thus workload-independent. Finally, related to the idea of computing a data-driven model is generally the field of knowledge compilation where an expensive offline phase creates a representation for instance for evaluating Boolean formulas [7, 46, 4]. However, non of these approaches targets the complexity of SQL queries supported in *DeepDB*.

8. CONCLUSION AND FUTURE WORK

In this work we have proposed *DeepDB* which is a data-driven approach for learned database components. We have shown that our approach is general and can be used to support various tasks including cardinality estimation and approximate query processing. We believe our data-driven learning approach can also be used for other DBMS components. For instance, it has already been shown that column correlations can be exploited to improve indexing [50]. In addition, SPNs naturally provide a notion of correlated clusters that can also be used for suggesting using interesting patterns in data exploration.

9. ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their helpful feedback. The work partly grew out of discussions within and support of the missions “Federated Machine Learning” and “ML for Vulnerability Detection” of the National Research Center ATHENE, the AI lighthouse BMWi project SPAICER (01MK20015E), and the DFG Training Group “Adaptive Preparation of Information from Heterogeneous Sources” (AIPHES, GRK 1994/1).

10. REFERENCES

[1] Flights dataset. https://www.kaggle.com/usdot/flight-delays. Accessed: 2019-06-30.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

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

[4] S. Bova. Sdds are exponentially more succinct than obdds. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[5] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. *The VLDB Journal*, 10, 12 2000.

[6] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, Oct. 2007.

[7] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[8] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. R. Narasayya, and S. Chaudhuri. Selectivity estimation for range predicates using lightweight models. *PVLDB*, 12(9):1044–1057, 2019.

[9] P. Eichmann, C. Binnig, T. Kraska, and E. Zgraggen. Idebench: A benchmark for interactive data exploration, 2018.

[10] R. Gens and P. Domingos. Learning the Structure of Sum-Product Networks. In *International Conference on Machine Learning*, pages 873–880, 2013.

[11] L. Getoor and L. Mihalkova. Learning statistical models from relational data. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1195–1198, New York, NY, USA, 2011. ACM.

[12] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 461–472, New York, NY, USA, 2001. ACM.

[13] E. Gribkoff and D. Suciu. Slimshot: in-database probabilistic inference for knowledge bases. *PVLDB*, 9(7):552–563, 2016.

[14] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. Multi-attribute selectivity estimation using deep learning. *CoRR*, abs/1903.09999, 2019.

[15] A. Jha and D. Suciu. Probabilistic databases with markoviews. *PVLDB*, 5(11):1160–1171, 2012.

[16] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[17] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, New York, NY, USA, 2018. ACM.

[18] M. S. Lakshmi and S. Zhou. Selectivity estimation in extensible databases - a neural network approach. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 623–627, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[19] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[20] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[21] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, pages 615–629. ACM, 2016.

[22] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality estimation using neural networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 53–59, Riverton, NJ, USA, 2015. IBM Corp.

[23] D. Lopez-Paz, P. Hennig, and B. Schölkopf. The randomized dependence coefficient. In *Advances in neural information processing systems*, pages 1–9, 2013.

[24] Q. Ma and P. Triantafillou. Dbest: Revisiting approximate query processing engines with machine learning models. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, pages 1553–1570, 2019.

[25] Q. Ma and P. Triantafillou. Dbest: Revisiting approximate query processing engines with machine learning models. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1553–1570, New York, NY, USA, 2019. ACM.

[26] T. Malik, R. Burns, and N. Chawla. A black-box approach to query cardinality estimation. In *CIDR*, 2007.

[27] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *CoRR*, abs/1904.03711, 2019.

[28] A. Molina, S. Natarajan, and K. Kersting. Poisson Sum-Product Networks: A Deep Architecture for Tractable Multivariate Poisson Distributions. In *AAAI*, 2017.

[29] A. Molina, A. Vergari, N. D. Mauro, S. Natarajan, F. Esposito, and K. Kersting. Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains. In *AAAI*, 2018.

[30] A. Molina, A. Vergari, K. Stelzner, R. Peharz, P. Subramani, N. D. Mauro, P. Poupart, and K. Kersting. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks, 2019.

[31] D. Olteanu, J. Huang, and C. Koch. Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases. In *2009 IEEE 25th International Conference on Data Engineering*, pages 640–651, March 2009.

[32] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009.

[33] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1461–1476, New York, NY, USA, 2018. ACM.

[34] Y. Park, A. S. Tajik, M. Cafarella, and B. Mozafari. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 587–602, New York, NY, USA, 2017. ACM.

[35] H. Poon and P. Domingos. Sum-product networks: A New Deep Architecture. In *2011 IEEE International Conference on Computer Vision Workshops*, pages 689–690, November 2011.

[36] T. Rekatsinas, A. Deshpande, and L. Getoor. Local structure and determinism in probabilistic databases. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 373–384, New York, NY, USA, 2012. ACM.

[37] P. Sen, A. Deshpande, and L. Getoor. Prdb: Managing and exploiting rich correlations in probabilistic databases. *The VLDB Journal*, 18(5):1065–1090, Oct. 2009.

[38] J. Shanmugasundaram, U. Fayyad, P. S. Bradley, et al. Compressed data cubes for olap aggregate query approximation on continuous dimensions. In *ACM SIGKDD*, 1999.

[39] Y. Sheng, A. Tomasic, T. Zhang, and A. Pavlo. Scheduling OLTP transactions via learned abort prediction. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*, pages 1:1–1:8, 2019.

[40] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch. Automatic mapping of the sum-product network inference problem to fpga-based accelerators. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 350–357, 2018.

[41] D. Suciu and C. Re. Efficient top-k query evaluation on probabilistic data, Oct. 12 2010. US Patent 7,814,113.

[42] J. Sun and G. Li. An end-to-end learning-based cost estimator. *CoRR*, abs/1906.02560, 2019.

[43] A. Thiagarajan and S. Madden. Querying continuous functions in a database system. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 791–804. ACM, 2008.

[44] S. Thirumuruganathan, S. Hasan, N. Koudas, and G. Das. Approximate query processing using deep generative models. *CoRR*, abs/1903.10000, 2019.

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

[46] G. Van den Broeck and A. Darwiche. On the role of canonicity in knowledge compilation. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[47] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein. Bayesstore: Managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, 1(1):340–351, 2008.

[48] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner. Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '19, pages 5:1–5:8, New York, NY, USA, 2019. ACM.

[49] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a learning optimizer for shared clouds. *PVLDB*, 12(3):210–222, 2018.

[50] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1223–1240, New York, NY, USA, 2019. ACM.

[51] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *PVLDB*, 13(3):279–292, 2019.