The VLDB Journal (2013) 22:3–27
DOI 10.1007/s00778-012-0293-7

SPECIAL ISSUE PAPER

Efficiently adapting graphical models for selectivity estimation

Kostas Tzoumas · Amol Deshpande · Christian S. Jensen

Received: 14 December 2011 / Revised: 29 July 2012 / Accepted: 3 September 2012 / Published online: 7 November 2012

Abstract Query optimizers rely on statistical models that succinctly describe the underlying data. Models are used to derive cardinality estimates for intermediate relations, which in turn guide the optimizer to choose the best query execution plan. The quality of the resulting plan is highly dependent on the accuracy of the statistical model that represents the data. It is well known that small errors in the model estimates propagate exponentially through joins, and may result in the choice of a highly sub-optimal query execution plan. Most commercial query optimizers make the attribute value independence assumption: all attributes are assumed to be statistically independent. This reduces the statistical model of the data to a collection of one-dimensional synopses (typically in the form of histograms), and it permits the optimizer to estimate the selectivity of a predicate conjunction as the product of the selectivities of the constituent predicates. However, this independence assumption is more often than not wrong, and is considered to be the most common cause of sub-optimal query execution plans chosen by modern query optimizers. We take a step towards a principled and practical approach to performing cardinality estimation without making the independence assumption. By carefully using concepts from the field of graphical models, we are able to factor the joint probability distribution over all the attributes in the database into small, usually two-dimensional distributions, without a significant loss in estimation accuracy. We show how to efficiently construct such a graphical model from the database using only two-way join queries, and we show how to perform selectivity estimation in a highly efficient manner. We integrate our algorithms into the PostgreSQL DBMS. Experimental results indicate that estimation errors can be greatly reduced, leading to orders of magnitude more efficient query execution plans in many cases. Optimization time is kept in the range of tens of milliseconds, making this a practical approach for industrial-strength query optimizers.

Keywords Graphical models · Selectivity estimation · Query optimization

Electronic supplementary material The online version of this article (doi:10.1007/s00778-012-0293-7) contains supplementary material, which is available to authorized users.

K. Tzoumas (✉)
Technische Universität Berlin, Berlin, Germany
e-mail: kostas.tzoumas@tu-berlin.de

A. Deshpande
University of Maryland, College Park, MD, USA
e-mail: amol@cs.umd.edu

C. S. Jensen
Aarhus University, Aarhus, Denmark
e-mail: csj@cs.au.dk

1 Introduction

Query optimizers use estimates of intermediate relation sizes to determine the best query execution plan. The task of cardinality estimation consists of efficiently and accurately estimating intermediate result sizes at compile time. Typically, statistical summaries of the data are constructed off-line, and are used during query compilation to estimate selectivities of join and selection predicates.

Query optimizers typically make simplifying assumptions about the statistical properties of the data. Specifically, aiming at simplicity and low overhead in cost estimation, the original System R optimizer [35] made three simplifying assumptions:

1. **Uniform distribution assumption (Uniform):** The values of an attribute X of a relation R, $R.X$, are uni-

formly distributed across the attribute's active domain $\mathrm{Dom}(R.X)$. This allows the approximation

$$\Pr(R.X = x) \approx 1/|\mathrm{Dom}(R.X)|.$$

2. **Attribute value independence assumption (AttrInd):** All attributes in a relation are independent of each other. For attributes $R.X$ and $R.Y$ of relation R, this allows the approximation

$$\Pr(R.X = x, R.Y = y) \approx \Pr(R.X = x)\Pr(R.Y = y).$$

3. **Join predicate independence assumption (JoinInd):** Join predicates are independent of each other, and of selection predicates. For join predicates $R.A = S.A$ and $S.B = T.B$, this assumption allows the approximation

$$\Pr(R.A = S.A, S.B = T.B) \approx \Pr(R.A = S.A) \Pr(S.B = T.B).$$

Further, for a join predicate $R.A = S.A$ and an attribute $R.X$, the assumption allows the approximation

$$\Pr(R.A = S.A, R.X = x) \approx \Pr(R.A = S.A) \Pr(R.X = x).$$

Note that **JoinInd** is implied by **AttrInd**. We discuss it separately due to its importance.

It is well known that these simplifying assumptions can lead to substantial estimation errors in base selectivity estimates [18,20]. Such errors typically propagate exponentially through joins [21], and can result in multiple orders of magnitude errors for queries with many joins, causing the query optimizer to choose a sub-optimal plan [30]. Thus, a great body of research has been devoted to avoiding these assumptions, while keeping the overhead of selectivity estimation reasonable. Since errors propagate exponentially through joins, it is especially important to capture correlations between join selectivities and selection predicates.

Uniform was assumed by early relational DBMSs [35]. Due to a wealth of research on histograms (see, e.g., [19,20,22–24,29] and more recently [26]), modern systems do not typically make this assumption for selection predicates over base relations. Instead, they use one-dimensional statistical summaries to model the skew in attribute distributions. These summaries, termed *attribute-level synopses* [36], aim to estimate the real distribution of an attribute $P(X)$ in limited space using another distribution $\hat{P}(X)$. Modern DBMSs are thus able to accurately estimate the selectivity of a selection predicate over a single attribute. However, modern DBMSs typically do not reason about skew in determining the selectivity of a join, but use instead the principle of inclusion [37]. The bulk of the estimation errors in modern DBMSs stem from assumptions **AttrInd** and **JoinInd** [18].

AttrInd allows the optimizer to ignore statistical correlations between attributes and to estimate the selectivity of a conjunction of predicates as the product of their constituent selectivities. Therefore, this assumption restricts the statistical model of the data to a collection of one-dimensional histograms. **AttrInd** persists in most commercial optimizers. Several *table-level synopses* such as multi-dimensional histograms have been proposed [5,15,31,33]. However, the curse of dimensionality quickly renders most techniques ineffective for wide tables. For large numbers of attributes, one can either resort to sampling or *decompose* the joint probability distribution by identifying the most important correlations in the database [11]. The latter is typically done by using concepts from the field of graphical models, which also forms the theoretical foundation of our work.

The **JoinInd** assumption is not adequately addressed by table-level synopses. Assume a join predicate $R.A = S.B$ between relations R and S and a selection predicate on an attribute $R.X = x$. In principle, one could construct a three-dimensional synopsis that approximates the joint distribution $P(R.X, R.A, S.B)$ and estimate the joint selectivity of the join and the selection $R.X = x$ correctly (in case they are correlated). This approach suffers from the problem that join attributes are typically keys with large domains, which yield distributions that are hard to approximate. One solution is to use binary random variables, called *join indicators*, and to construct a (decomposed) probability distribution on all the attributes and join indicators in the database [14]. This is the approach we follow in this paper.

Our work falls in the category of *schema-level synopses*. A statistical summary is constructed that approximates the joint probability distribution of all attributes and join predicates in the database. So far, few research efforts have proposed such synopses [2,14,18,36,38], and only two efforts [36,38] can provide selectivity estimates for general database schemas and workloads without making the uniformity assumption.

To summarize, the commercial state-of-the-art is able to discover and exploit the distribution skew of attributes during query optimization. The same does not hold for statistical correlations between attributes. The research state of the art is especially lacking methods that can efficiently capture correlations between attributes of different relations and are general enough to support practical (e.g., cyclic) database schemas.

1.1 A detailed example

Before describing our approach and key contributions, we delve into the details of an example query and explicate the effects of the above assumptions on its execution. Consider the following query on the TPC-H schema:

```
select c_name,c_address
from   orders,lineitem,customer
where  o_okey = l_okey and o_totalprice in [x]
       and l_extendedprice in [y] and
       o_ckey = c_ckey and c_acctbal in [z]
```

The query finds the names and addresses of customers that have placed orders within a given price range and with items within a given price range, with a further selection on the customer balance. Since an order's price is calculated using the prices of the items it comprises, the attributes `o_totalprice` and `l_extendedprice` are correlated for all tuples from `lineitem` and `orders` that match using the join predicate `o_okey=l_okey`. This positive correlation causes the selectivity

$$\Pr(\texttt{o\_okey} = \texttt{l\_okey}, \texttt{o\_totalprice}\,\texttt{in}\,[\mathrm{x}], \texttt{l\_extendedprice}\,\texttt{in}\,[\mathrm{y}])$$

to be much higher than the product of the constituent selectivities

$$\Pr(\texttt{o\_okey} = \texttt{l\_okey}) \times \Pr(\texttt{o\_totalprice}\,\texttt{in}\,[\mathrm{x}]) \times \Pr(\texttt{l\_extendedprice}\,\texttt{in}\,[\mathrm{y}]).$$

By making the **JoinInd** assumption, the PostgreSQL optimizer severely underestimates the cardinality of the join `lineitem` $\bowtie$ `orders` after the two selection predicates, and places it first in the join plan. The query plan picked by PostgreSQL is[1]

$(\texttt{orders} \bowtie_{HJ} \texttt{lineitem}) \bowtie_{NLJ} \texttt{customer}$,

which has several orders of magnitude slower execution time than the alternative plan

$(\texttt{customer} \bowtie_{HJ} \texttt{orders}) \bowtie_{HJ} \texttt{lineitem}$,

which is picked if we equip PostgreSQL with the selectivity estimation machinery described in this paper.

1.2 Approach and contributions

Performing selectivity estimation without making **AttrInd** or **JoinInd** is a crucial and timely problem. The challenge lies in carefully choosing the kind of dependencies that the statistical model can encode so that the curse of dimensionality does not exponentially increase the query optimization overhead quickly. Graphical models [10] provide a theoretical foundation for exploiting conditional independence to factor a joint probability distribution. They can thus serve as a basis for a principled solution to the problem. As in previous work [14], we use graphical models to factor the complete joint distribution of all attributes and join selectivities in the database. Our foci are on efficiency during query optimization and on integration into a DBMS query optimizer. The contributions are the following:

1. We carefully restrict the space of possible graphical models that can be used to model the database. The resulting models can be stored using two-dimensional histograms only in most cases. Although this design decision also restricts the kinds of statistical correlations that can be modeled, it is instrumental in keeping the overhead during selectivity estimation low. Further, as has been noted before [18], the reduction in estimation errors diminishes significantly when moving from two-variable to three-variable synopses (and the reduction is orders of magnitude when moving from one-variable to two-variable synopses).
2. The fixed model structure we choose allows for a scalable and efficient model construction algorithm, that issues join queries over only two tables at a time.
3. We propose two algorithms for selectivity estimation. The first is a variation of the junction tree propagation algorithm [10]. The second is a custom algorithm that minimizes the number of histogram multiplications by exploiting the order of requests made by a bottom-up dynamic programming query optimizer.
4. We include a treatment of cyclic schemas. Contrary to previous work that can be used only in tree schemas with key-foreign key joins [2,14], our work is applicable to cyclic schemas and arbitrary θ-joins.
5. We propose *query-specific* modeling where, in essence, a new graphical model is chosen for each query. This new approach results in better estimates and improved optimization times.
6. We implement the above scheme and algorithms inside the PostgreSQL query optimizer. This is, to the best of our knowledge, the first implementation of graphical models inside a DBMS kernel.

The rest of this paper is organized as follows. Section 2 presents background material on graphical models. Section 3 discusses the design decisions that lead to our fixed model structure. Section 4 presents the model construction algorithm. Section 5 presents the algorithms for selectivity estimation. Section 6 discusses alternative extensions to the model needed to support cyclic schemas. Section 7 presents our approach to query-specific modeling. Section 8 discusses our implementation, and Sect. 9 reports on an experimental study. Finally, Sect. 10 discusses previous work, and Sect. 11 concludes and offers research directions.

[1] $\bowtie_{HJ}$ and $\bowtie_{NLJ}$ denote hash and nested loop joins. The right operand of $\bowtie$ in the plans is the inner relation.

2 Background: graphical models

Our work builds on the theory of graphical models from the statistics and machine learning communities [10]. The following brief overview of graphical models focuses on the concepts used in the paper.

Preliminaries: Assume a probability distribution P_{D} over the set of random variables $\mathsf{X} = \{X_1, \ldots, X_n\}$. Our goal is to approximate this distribution by another distribution P_{M} that can be expressed as a product of *factors* $\Phi = \{\phi_1, \ldots, \phi_m\}$

$$P_{\mathsf{M}} = \frac{1}{Z} \prod_{\phi \in \Phi} \phi,$$

where a factor is a function $\phi_i : \mathsf{X}_i \subset \mathsf{X} \to \mathbb{R}^+$ and Z is a normalization constant. The benefits of the approximation $P_{\mathsf{D}} \simeq P_{\mathsf{M}}$ are substantial. Denote by $\mathrm{Dom}(X_i)$ the domain of the variable X_i, and assume $\forall X_i$ that $|\mathrm{Dom}(X_i)| = d$. Then, assuming a tabular representation, the space needed to store the full joint distribution P_{D} grows exponentially as d^n. However, the space needed to store P_{M} is just $\sum_i d^{|\mathsf{X}_i|}$. So, as long as each factor ϕ_i contains only a few variables, the exponential blow-up of storage space no longer applies. In addition, extracting marginal probability distributions over subsets of random variables becomes computationally tractable.

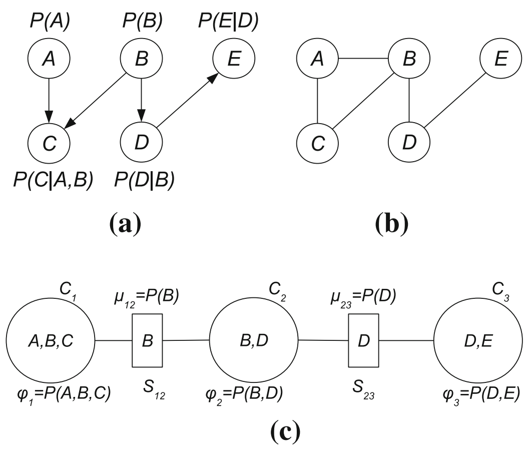
To factor the distribution, the notion of *conditional independence* is used. Two variables X, Y are conditionally independent given the variable Z (denoted $X \perp Y | Z$) iff

$$\begin{aligned}&\forall x, y, z : \Pr(X = x, Y = y | Z = z)\\ &\quad = \Pr(X = x | Z = z) \Pr(Y = y | Z = z).\end{aligned}$$

Intuitively, once we know the value of Z, the knowledge of the value of Y does not convey additional information about X. An implication of $X \perp Y | Z$ is that their joint distribution can be factored as:

$$P(X, Y, Z) = \frac{P(X, Z) P(Y, Z)}{P(Z)}.$$

Bayesian networks: Bayesian networks (BNs), a class of graphical models, correspond to a class of such factorizations, and they offer a graphical representation of the conditional independencies implied by them. A Bayesian network $\mathsf{BN}(\mathsf{S}, \theta)$ consists of a graphical, qualitative component S and a quantitative component θ. S is a directed acyclic graph $\mathsf{G}(\mathsf{V}, \mathsf{E})$ that contains one vertex per random variable in X. Figure 1a shows an example Bayesian network graph for five binary random variables A, B, C, D, E. An edge $X_i \to X_j \in \mathsf{E}$ in a Bayesian network graph denotes that the value of X_j is (stochastically) influenced by the value of X_i. In the example in the figure, A directly influences C. The absence of an edge from one variable to another does not imply independence. Consider the chain of two edges $B \to D \to E$ in the example. Here, B influences E indirectly through D. Once D is known, B and E become independent. On the other hand, if D is not known, an interaction between B and E exists. In fact, the chain of interactions represents the conditional independence $B \perp E | D$.



a	b	c	$P(a,b,c)$
a_1	b_1	c_1	0.25
a_1	b_1	c_2	0.32
a_1	b_2	c_1	0.01
a_1	b_2	c_2	0.12
a_2	b_1	c_1	0.08
a_2	b_1	c_2	0.04
a_2	b_2	c_1	0.1
a_2	b_2	c_2	0.08

b	d	$P(b,d)$
b_1	d_1	0.4
b_1	d_2	0.3
b_2	d_1	0.15
b_2	d_2	0.15

d	e	$P(d,e)$
d_1	e_1	0.7
d_1	e_2	0.1
d_2	e_1	0.05
d_2	e_2	0.15

(d)

Fig. 1 A small graphical model of five binary random variables A, B, C, D, E **a** Bayesian network. **b** Moral graph. **c** Junction tree. **d** Clique potentials

Denote by $\mathrm{Pa}(X)$ the parents of X in G, and denote by $\mathrm{NonDesc}(X)$ the *non-descendants* of X, i.e., the nodes that are not reachable by X following the directed edges in E. Then, the Bayesian network encodes the set of conditional independences:

$$X \perp \mathrm{NonDesc}(X) | \mathrm{Pa}(X) \quad \forall X.$$

Therefore, the Bayesian network of Fig. 1a implies the following independences:

$$\begin{aligned}&A \perp \{B, D, E\} | \emptyset; \quad && B \perp A | \emptyset; \quad && C \perp \{D, E\} | A, B;\\ &D \perp \{A, C\} | B; && E \perp \{A, B, C\} | D.\end{aligned}$$

A Bayesian network BN induces a probability distribution P_{BN} over $\mathsf{X} = \{X_1, \ldots, X_n\}$ that, due to the encoded conditional independences, can be factorized as:

$$P_{\mathsf{BN}}(\mathsf{X}) = \prod_{i=1,\ldots,n} P_{\mathsf{D}}(X_i | \mathrm{Pa}(X_i)).$$

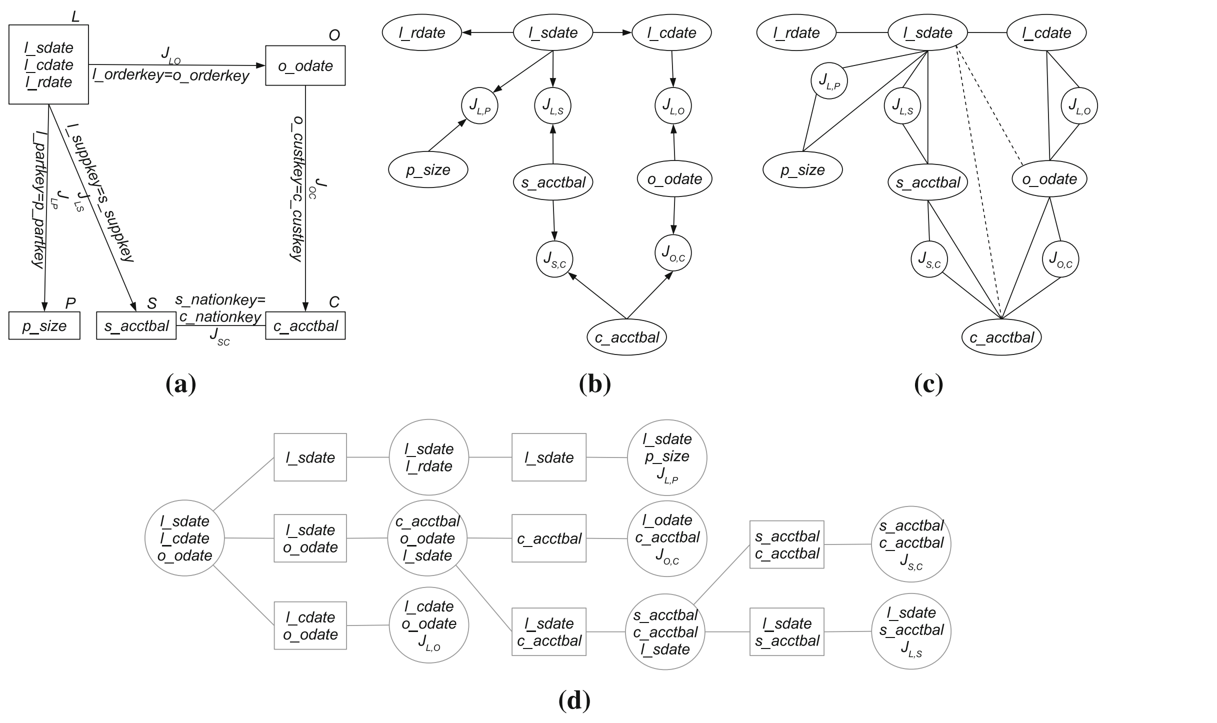


Fig. 2 A model constructed on a subset of the TPC-H schema graph along with seven descriptive attributes. The Bayesian network, the moral graph, and the junction tree representations of the model are shown. The dotted edges in (**c**) are added during triangulation. **a** Schema graph and descriptive attributes. **b** Bayesian network. **c** Moral graph (excluding the dotted edges). **d** Junction tree

For the example of Fig. 1a, we obtain the factorization

$$P_{\mathsf{BN}}(A, B, C, D, E) = P(E|D)P(D|B)P(C|A, B)$$
$$P(A)P(B).$$

The quantitative part θ of the Bayesian network is the set of distributions $\{P(X_i|\text{Pa}(X_i))|i = 1, \ldots, n\}$. Each of these distributions is defined over $|\text{Pa}(X_i)| + 1$ variables. As long as these numbers are low, the desired decomposition of P_{D} is achieved, by approximating it with P_{BN}.

Inference: Having achieved a factorization of P_{D}, the *inference problem* is to efficiently compute marginal probability distributions (e.g., of the form $P(X_i)$ or $P(X_i, X_j)$) or conditional probability distributions (e.g., $P(X_i|X_j = x_j)$). Later we discuss in detail how selectivity estimation queries are equivalent to such queries. We use the *junction tree approach* [25] for inference, which, compared to the other approaches like *variable elimination* [10], offers much lower online inference times, and thus selectivity estimation times. The cornerstone of this approach is the junction tree data structure, which pre-computes a collection of appropriate marginals to limit the amount of work that needs to be done at inference time.

Junction trees: A junction tree is constructed in three steps. First, the directed graph G of the Bayesian network is converted to an undirected graph, called the *moral graph* G^m. G^m has the same vertices as G and has an undirected edge between X and Y if G contains either the edge $X \rightarrow Y$ or the edge $Y \rightarrow X$. In addition, G^m contains an edge between any two nodes that share a child in G (hence the name "moral"). Figure 1b shows the moral graph obtained from the Bayesian network of Fig. 1a. It contains the same (undirected) edges as the Bayesian network, and the extra edge $A - B$ because A and B share the child vertex C.

Second, the moral graph G^m is triangulated if it is not already. A graph is triangulated (also called *chordal*) if it does not contain a cycle of length more than 3 without a *chord*, i.e., an edge between at least two non-consecutive vertices on the cycle. For example, the more complex Bayesian network of Fig. 2b represents a model of a subset of the TPC-H schema using the techniques presented in this paper. Figure 2c shows the obtained moral graph, if the dotted edges are excluded. That moral graph contains a 5-cycle among the vertices `l_sdate`, `l_cdate`, `o_odate`, `c_acctbal`, and `s_acctbal`. The new (dotted) edge between `l_sdate` and `c_acctbal` becomes a chord for that cycle. However, we need to add a second chord to address the 4-cycle among vertices `l_sdate`, `l_cdate`, `o_odate`, and `c_acctbal`, which was formed by after adding the first chord. After adding the second chord (dotted edge), the graph becomes triangulated. Triangulating a graph by adding minimum number of edges is NP-hard in general [3]. Let G^{mt} denote the triangulated graph.

Third, the junction tree T is constructed. The junction tree contains a node for every maximal clique in G^{mt}. Two cliques C_i and C_j are connected via an undirected edge only if they contain a common variable (but not all cliques with common variables are directly connected). The edge $C_i - C_j$ is annotated with a *separator node* S_{ij} that contains all the common variables of C_i and C_j. Figure 1c shows a valid junction tree for the moral graph of Fig. 1b (and Fig. 2d shows a junction tree for the moral graph of Fig. 2c). To be valid, a junction tree must satisfy two properties.

Family preservation: For every $P_i(X_i|\text{Pa}(X_i)) \in \theta$, there exists a clique C_j that contains all the variables $\{X_i\}\cup\text{Pa}(X_i)$.
Running intersection: For every pair of cliques C_1, C_2, the variables belonging to their intersection $C_1 \cap C_2$ are contained in every node on the path from C_1 to C_2 in the junction tree T.

Junction tree construction is not deterministic: a given triangulated graph G^{mt} may correspond to multiple equivalent valid junction trees.

The final step in junction tree construction is *calibration*, which is typically realized using a message passing algorithm. The purpose of calibration is to create the clique and separator marginal distributions, to be stored with the cliques and separators and to be used during the subsequent inference process to compute required marginal or conditional distributions. Specifically with every clique X_i, we store a clique probability distribution

$$\phi_i(\mathsf{X}_i) = \sum_{\mathsf{X}-\mathsf{X}_i} P_{\mathsf{D}}(\mathsf{X});$$

and with every separator X_{ij}, we store a separator distribution

$$\mu_{ij}(\mathsf{X}_{ij}) = \sum_{\mathsf{X}-\mathsf{X}_{ij}} P_{\mathsf{D}}(\mathsf{X}).$$

In our work, we are able to bypass the message passing algorithm typically used to calibrate junction trees. Instead we construct the clique and separator potentials directly by scanning the data. However, the selectivity estimation algorithm presented in Sect. 5 proceeds very similarly to the message passing algorithm, and we discuss it in further detail there.

The resulting junction tree also reflects a factorization of P_{D}. The only distributions that need to be kept are the marginals of the cliques and separators, called the clique and separator *potentials*. Consider the junction tree of Fig. 1c. The probability distributions that need to be stored are the marginals of the cliques

$$\phi_1 = P(A, B, C); \quad \phi_2 = P(B, D); \quad \phi_3 = P(D, E)$$

and the marginals of the separators

$$\mu_{12} = P(B); \quad \mu_{23} = P(D).$$

The marginals of the cliques for the junction tree of Fig. 1c are shown in Fig. 1d. The distribution P_{M} induced by the junction tree (that approximates P_{D}) can be expressed as the product of marginals divided by the product of the separator marginals:

$$P_{\mathsf{M}} = \frac{\prod_{\mathbf{C}} P(\mathbf{C})}{\prod_{\mathbf{S}} P(\mathbf{S})}.$$

For the example junction tree, the factorization is

$$P_{\mathsf{M}}(A, B, C, D) = \frac{P(A, B, C)P(B, D)P(D, E)}{P(B)P(D)}.$$

Inference in junction trees: Next we discuss how to answer inference queries using a junction tree. It is easy to extract a marginal distribution from a junction tree if all variables belong to the same clique. To extract the marginal $P(A, C)$ from the junction tree of Fig. 1c, we just need to sum out B from the clique potential ϕ_1:

$$\begin{aligned} P(A, C) &= \sum_B P(A, B, C) \\ &= P(A, b_1, C) + P(A, b_2, C) + \cdots . \end{aligned}$$

If the variables do not belong to the same clique, we need to multiply clique distributions. For example, to extract the marginal $P(A, D)$, we need to multiply the clique potentials ϕ_1 and ϕ_2 and sum out the unnecessary variables. When multiplying two clique potentials, it is necessary to divide by the separator potential:

$$P(A, D) = \sum_{B,C} \frac{P(A, B, C)P(B, D)}{P(B)}.$$

In our work, we need to extract probabilities as well as marginal distributions. We define three operations on potentials: marginalization, substitution, and multiplication. We will use these operations to extract probabilities from a junction tree in Sect. 5.

Marginalization: Consider a factor $f : \mathsf{X} \to \mathbb{R}^+$, where $\mathsf{X} = \{X_1, \ldots, X_n\}$, and $\mathsf{Y} = \{X_1, \ldots, X_k\} \subset \mathsf{X}$. Then, the result of the marginalization $\sum_{\mathsf{Y}} f$ is a factor $f' : \mathsf{X} - \mathsf{Y} \to \mathbb{R}^+$ where

$$f'(x_{k+1}, \ldots, x_n) = \sum_{X_1,\ldots,X_k} f(x_1, \ldots, x_n).$$

Substitution: Assume a factor $f : \mathsf{X} \to \mathbb{R}^+, \mathsf{X} = \{X_1, \ldots, X_n\}$, and a variable $X_i \in \mathsf{X}$, and a predicate ψ on the variable X_i (e.g., $\psi \equiv X_i < x, x \in \text{Dom}(X_i)$). Then, the result of the substitution $f[\psi]$ is a factor $f' : \mathsf{X}-\{X_i\} \to \mathbb{R}^+$ where $f'(\mathsf{X} - \{X_i\}) = \sum_{X_i} f''(\mathsf{X})$ and

$$f''(x_1, \ldots, x_n) = \begin{cases} f(x_1, \ldots, x_n) & \text{if } \psi(x_i) = \mathbf{T} \\ 0 & \text{otherwise.} \end{cases}$$

Substitution is generalized in an obvious way to several variables.

Multiplication: Assume two factors $f_1 : \mathsf{X} \cup \mathsf{Z} \rightarrow \mathbb{R}^+, f_2 : \mathsf{Y} \cup \mathsf{Z} \rightarrow \mathbb{R}^+$, where $\mathsf{X} = \{X_1, \ldots, X_n\}$, $\mathsf{Y} = \{Y_1, \ldots, Y_m\}$, and where $\mathsf{Z} = \{Z_1, \ldots, Z_k\}$ denotes the intersection of the domains of the two factors. The result of the multiplication $f_1 f_2$ is a factor $f' : \mathsf{X} \cup \mathsf{Y} \cup \mathsf{Z} \rightarrow \mathbb{R}^+$ where

$$f'(x_1, \ldots, x_n, y_1, \ldots, y_m, z_1, \ldots, z_k) \\ = f_1(x_1, \ldots, x_n, z_1, \ldots, z_k) f_2(y_1, \ldots, y_m, z_1, \ldots, z_k).$$

3 Modeling a database

3.1 Preliminaries

We first provide a series of definitions needed for a formal definition of the problem. Given a database and possibly a query workload, the goal is to formally define a probability distribution. Then, our task reduces to approximating this distribution using graphical models. First, we define the schema graph of a database. Then, we define two kinds of random variables that will be included in the probability distribution, and we define the probability space.

First we define the schema graph $\mathsf{G}(\mathsf{R}, \mathsf{J})$ of a database. The vertex set R of the schema graph $\mathsf{G}(\mathsf{R}, \mathsf{J})$ of a database consists of the relations in the database. Each vertex is annotated with a subset of the attributes of its relation. An edge exists between two vertices if the relations that the vertices encode may be joined in a query—the join predicate itself can be arbitrary. Figure 2a shows an example schema graph for a subset of the TPC-H schema. Consider the node L that represents the `lineitem` relation. It is annotated with three attributes (`l_shipdate`, `l_commitdate`, and `l_receiptdate`), and it is connected to three nodes (O, P, S representing `orders`, `part`, and `supplier` respectively). The three edges are annotated with the corresponding join predicates. Note that the directed edges are used for key-foreign key joins, while undirected edges are used for many-to-many joins (e.g., the join between `supplier` and `customer` on `nationkey`). This is only for visual convenience; our approach works with any type of join predicate, including non-equality and user-defined predicates. The endpoints of an edge could be the same vertex, indicating a self-join, and there can be multiple edges between two vertices, indicating multiple predicates that can be used to join two relations.

The schema graph can be derived from a database schema that includes foreign key constraints. It can be also enriched using a query workload. A workload can help to limit the number of attributes per relation to include in the schema graph by indicating which attributes appear in selection predicates, and it can identify further ways in which relations can be joined that do not follow from the database schema. In addition, using a workload to construct a schema graph is essential for cases where the DBA has not declared foreign key constraints, e.g., for performance reasons.

Next, we define the database probability distribution that can be used to compute selectivity estimates, and that we aim to approximate using a graphical model. Two kinds of random variables form the database probability distribution: descriptive attributes and join indicators. A *descriptive attribute* is defined for every attribute that exists in a vertex annotation in the schema graph. The marginal distribution of a descriptive attribute is derived from the contents of the database. For example, for attribute A of relation R, we define the descriptive attribute $\mathbf{A}$ as a random variable with domain

$$\mathrm{Dom}(\mathbf{A}) \equiv \texttt{select distinct}(A)\ \texttt{from}\ R$$

and probability distribution

$$\Pr(\mathbf{A} = a) \equiv \frac{1}{|R|} \left| \begin{array}{l} \texttt{select count}(*)\ \texttt{from}\ R \\ \texttt{where}\ A = a \end{array} \right| .$$

Note that we define our variables, and thus our model, using the frequentist definition of probability, which is appropriate for selectivity estimation [13]. The contents of the database represent the "ground truth," and there is no need for the model to hold for any other database instance (in fact, this would make the model less accurate).

A *join indicator* is a binary random variable that captures the event that two tuples from two relations join. A join indicator is defined for every possible join predicate, i.e., every edge in the schema graph. Consider the equality join $R.A = S.A$ between relations R and S. We define the join indicator J_{RS} as a random variable with domain $\{\mathbf{T}, \mathbf{F}\}$ and distribution

$$\Pr(J_{RS} = \mathbf{T}) \equiv \frac{1}{|R||S|} \left| \begin{array}{l} \texttt{select count}(*)\ \texttt{from}\ R, S \\ \texttt{where}\ R.A = S.A \end{array} \right| ,$$
$$\Pr(J_{RS} = \mathbf{F}) \equiv 1 - \Pr(J_{RS} = \mathbf{T}).$$

This definition can easily be generalized to non-equi joins. The idea of using join indicators is attributed to Getoor et al. [13,14]. It is a very interesting alternative to using histograms to compute join selectivities. Instead of constructing (and joining) histograms on attributes that are used in join predicates only, the selectivity of the join is precomputed. Then, instead of keeping the distribution of the key value, which is typically hard to approximate using a histogram, one only needs to store two values for the $\mathbf{F}$ and $\mathbf{T}$ values of the join indicator. This is one of the observations that enables our approach to guarantee 2-dimensional histograms only. In addition, using join indicators our method can support arbitrary θ-join predicates.

Consider a database with a schema graph $\mathsf{G}(\mathsf{R}, \mathsf{J})$, where $\mathsf{R} = \{R_1, \ldots, R_n\}$, and $\mathsf{J} = \{J_1, \ldots, J_m\}$ and denote by A

the set of attributes that annotate the vertices of the schema graph. We construct the *universal relation* U in two steps. First, we take the Cartesian product $\mathsf{C} = R_1 \times \cdots \times R_n$ of all relations in the database. We add all join indicators to relation C[2], and we set their values appropriately. Denote this relation by CJ. Finally, we project CJ on $\mathsf{A} \cup \mathsf{J}$ using bag semantics thus obtaining the relation U.

For example, consider a small database of two relations $R(X, Y, A)$ and $S(Z, W, B)$. Assume that in the schema graph, vertex R is annotated with the attributes X, Y and vertex S is annotated with the attributes Z, W. The only edge in the schema graph between R and S is annotated with the predicate $R.A = S.B$. The descriptive attributes are then $\mathsf{A} = \{X, Y, Z, W\}$, and the join indicators are $\mathsf{J} = \{J_{RS} \equiv (R.A = S.B)\}$. Assuming the following contents of the relations, the constructed universal relation is shown (note the bag semantics used in the construction of U):

$R =$

X	Y	A
a	c	1
a	c	2
b	c	2
a	d	3
b	d	5

$S =$

B	Z	W
1	e	g
2	f	g
6	e	h
3	f	g

$\Rightarrow \mathsf{U} =$

X	Y	J_{RS}	Z	W
a	c	**T**	e	g
a	c	**F**	f	g
a	c	**F**	e	h
a	c	**F**	f	g
a	c	**F**	e	g
a	c	**T**	f	g
a	c	**F**	e	h
a	c	**F**	f	g
b	c	**F**	e	g
b	c	**T**	f	g
b	c	**F**	e	h
b	c	**F**	f	g
a	d	**F**	e	g
a	d	**F**	f	g
a	d	**F**	e	h
a	d	**T**	f	g
b	d	**F**	e	g
b	d	**F**	f	g
b	d	**F**	e	h
b	d	**F**	f	g

.

The universal relation defines the probability distribution induced by the database; it defines with perfect accuracy the joint probability distribution of all descriptive attributes and join indicators. We denote this distribution by P_{U}. To be more specific, each tuple in the universal relation U is associated with a probability $1/|U|$. Thus, we get that $P_{\mathsf{U}}(X = a, Y = c, J_{RS} = \mathbf{T}, Z = e, W = g) = 1/|U|$, whereas $P_{\mathsf{U}}(X = a, Y = c, J_{RS} = \mathbf{F}, Z = f, W = g) = 2/|U|$ (since there are two tuples corresponding to the latter values). To obtain, e.g., a joint probability $P_{\mathsf{U}}(X = a, J_{RS} = \mathbf{T}$, we can (conceptually) issue the query

`select count(*) from` U `where` $X = a$ `and` J_{RS}

and divide the result by $|U|$. It is easy to see that, the marginal distributions obtained in this manner are identical to the ones computed earlier in the section. The problem we are solving in this paper can be stated as:

Given a schema graph, approximate distribution P_{U} using another distribution P_{M} which takes less space and allows for faster selectivity estimation.

3.2 Relational independences

From the construction of the universal relation, the following independences hold regardless of the database instance:

Theorem 1 *Consider the relations $R \neq S \neq T \neq U \in \mathsf{R}$. The following independences hold in P_{U}:*

1. *Descriptive attributes belonging to different relations are independent: $P_{\mathsf{U}}(R.X, S.Y) = P_{\mathsf{U}}(R.X)P_{\mathsf{U}}(S.Y)$*
2. *Join indicators are independent if they do not share a relation: $P_{\mathsf{U}}(J_{RS}, J_{TU}) = P_{\mathsf{U}}(J_{RS})P_{\mathsf{U}}(J_{TU})$*
3. *A descriptive attribute and a join indicator are independent if they do not share a relation: $P_{\mathsf{U}}(J_{RS}, T.X) = P_{\mathsf{U}}(J_{RS})P_{\mathsf{U}}(T.X)$*

Proof The proof uses simply the frequentist definition of probability:

$$\Pr(R.X = x) \equiv \frac{|\sigma_{R.X=x}(R)|}{|R|}, \Pr(J_{RS} = \mathbf{T}) \equiv \frac{|R \bowtie S|}{|R \times S|},$$
$$\Pr(J_{RS} = \mathbf{F}) \equiv \frac{|R \times S - R \bowtie S|}{|R \times S|} \equiv \frac{|R \not\bowtie S|}{|R \times S|};$$

and it uses the fact that selection commutes over the Cartesian product: $\sigma_{R.X=x}(R \times S) = \sigma_{R.X=x}(R) \times S$.

1. Assume arbitrary values $x \in \mathrm{Dom}(R.X)$ and $y \in \mathrm{Dom}(S.Y)$. Then,

$$\begin{aligned}\Pr(R.X{=}x, S.Y{=}y) &\equiv \frac{|\sigma_{R.X=x \wedge S.Y=y}(R \times S)|}{|R \times S|} \\ &= \frac{|\sigma_{R.X=x}(R)||\sigma_{S.Y=y}(S)|}{|R||S|} \\ &\equiv \Pr(R.X = x)\Pr(S.Y{=}y).\end{aligned}$$

2. Assume $J_{RS} \equiv (R.A = S.A)$ and $J_{TU} \equiv (T.C = U.C)$. We need to prove four cases ($\{\mathbf{T}, \mathbf{T}\}$, $\{\mathbf{T}, \mathbf{F}\}$, $\{\mathbf{F}, \mathbf{T}\}$, $\{\mathbf{F}, \mathbf{F}\}$) for the possible values of the two join indicators. For example, for $J_{RS} = \mathbf{T}$ and $J_{TU} = \mathbf{F}$ we have:

$$\begin{aligned}&\Pr(J_{RS} = \mathbf{T}, J_{TU} = \mathbf{F}) \\ &\equiv \frac{|\sigma_{R.A=S.A \wedge T.C \neq U.C}(R \times S \times T \times U)|}{|R \times S \times T \times U|} \\ &\equiv \Pr(J_{RS} = \mathbf{T})\Pr(J_{TU} = \mathbf{F}).\end{aligned}$$

The remaining cases are proven similarly.

3. Proven similarly. □

[2] In the rest of this paper, and when the meaning is clear from the context, we will denote a descriptive attribute $\mathbf{X}$ and the attribute X that it represents with the same symbol X. Similarly, we will denote by J_{RS} both the random variable and the predicate that defines it.

These independences may seem counter-intuitive at first. For example, consider the example database with two relations R, S of Sect. 3.1. The first independence implies $X \perp Z$. This does not mean that our model cannot capture cross-relation dependencies (e.g., the dependence between `o_orderprice` and `l_extendedprice` from TPC-H). Dependencies between attributes of different relations exist only if we have some information about whether the tuples of the relations join, i.e., an instantiation of J_{RS}. Once J_{RS} is instantiated, X and Z become dependent (conditioned on the event that e.g., $J_{RS} = \mathbf{T}$). For the TPC-H schema example,

$$\texttt{o\_orderprice} \perp \texttt{l\_extendedprice},$$

but

$$\texttt{o\_orderprice} \not\perp \texttt{l\_extendedprice} | J_{LO} = \mathbf{T}.$$

Note that the relational independences are a result of using the frequentist definition of probability, and they do not hold in domains where generalization is important, e.g., probabilistic databases. The important implication of the relational independences is that any model that does not imply them is overly complicated for the task of selectivity estimation. Recall that a Bayesian network is an encoding of a set of (conditional) independences. If the relational independences are not a subset of these encoded independences, then the particular Bayesian network contains more edges than needed (and thus requires more space to be stored, and the complexity to perform selectivity estimation is higher than needed).

3.3 Fixed model structure

Contrary to previous approaches [14] we restrict the space of possible graphical models that we consider. In particular, we restrict ourselves to the space of Bayesian networks with the following properties:

1. The subset of the Bayesian network that corresponds to the descriptive attributes of one relation is a directed tree.
2. A join indicator has at most two parents, which are descriptive attributes from the relations it joins.

This class of models has four interesting properties:

1. It directly models the correlation between a join selectivity and the attributes that are most strongly correlated with it.
2. Through chains of dependencies, it indirectly models the correlation between a join selectivity and all attributes of the two corresponding relations.
3. If the schema graph is acyclic, the model can be stored using two-dimensional histograms only. If the schema graph is cyclic, this is not a hard guarantee, but the chance of needing a three-dimensional distribution is slim and can be further avoided. See Sect. 6 for details. Further, if the schema graph contains cycles but the query graph is acyclic, only two-dimensional histograms are needed (see Sect. 7).
4. The model can be constructed efficiently, by issuing two-table join queries to the database (and never a three-table join or higher).

On the other hand, the fixed model structure limits the kinds of correlations that can be modeled. In particular, it cannot model complex three-way correlations, and it cannot model direct dependence of one join selectivity on another. The first limitation does not lead to serious performance degradation for most data sets [18], and the second limitation is intuitive. Intuitively, join selectivities are the "result" and not the "cause" of a dependency. Further, if we were to allow an edge between two join indicators, we would need arbitrarily complex join queries to construct the model, or alternatively we would need to restrict to acyclic key-foreign key schema graphs [2,14]. Other kinds of edges, e.g., edges between descriptive attributes of different relations would make the model more complex than needed, since it would not imply the relational independences. Allowing more than two parents per join indicator would create three-dimensional histograms, and would only be helpful if two attributes jointly influence the join selectivity (and the query contains the one with less influence). To summarize, the fixed model structure is a heuristic design decision which captures, however, the most important correlations, and greatly restricts the size of the model, the time to construct the model, and the complexity needed to perform selectivity estimation.

We briefly explain how the model can guarantee two-dimensional histograms. Figure 2b shows a possible Bayesian network for the TPC-H schema graph that conforms to the fixed structure, and Fig. 2c shows the corresponding moral graph (ignore the dotted edges for now). Within a relation, there is no "common effect" structure where a variable has two parents in the Bayesian network. Therefore, no edges are added during moralization. The only added edges are between two descriptive attributes that are common parents to a join indicator. Assume for now that the schema graph is acyclic. Then, the moral graph contains cliques of two variables for the descriptive attributes of every relation, and a clique of three variables for every join indicator. Therefore, any junction tree that corresponds to an acyclic schema graph contains two kinds of nodes:

1. Nodes with two descriptive attributes, which can be stored using a 2-dimensional histogram.
2. Nodes with two descriptive attributes and one join indicator, which can be stored as two 2-dimensional histograms, one for the **T** and one for the **F** value of the join indicator.

Note that the junction tree shown in Fig. 2d originated from a cyclic schema, and contains also nodes with three descriptive attributes; we cover cyclic schemas in Sect. 6.

We now prove that our fixed model structure indeed implies the relational independences.

Theorem 2 *The fixed model structure respects the relational independences and produces a valid Bayesian network.*

Proof The proof follows from d-separation and the structure of the model. It is easy to prove that any Bayesian network that conforms to our fixed model structure is valid (acyclic). First, since the "local" Bayesian network in each relation is a directed tree, it cannot contain a cycle. Second, since a join indicator has no children, it cannot be a part of a directed cycle.

For the first part of the theorem, we need to prove that any BN that conforms to our fixed model structure encodes the relational independences. Recall that the conditional independences encoded by a Bayesian network are

$$X \perp \text{NonDesc}(X)|\text{Pa}(X)\forall X.$$

We need to prove three cases:

1. Two descriptive attributes from different relations are independent. Consider the attribute X from relation R, and the attribute Y from relation S. The local Bayesian network of the relation R is a directed tree. Consider the path from the root of that tree, X_1, to X: $X_1 \rightarrow X_2 \rightarrow \cdots \rightarrow X_n \rightarrow X$. Due to the fixed model structure, Y is a non-descendant of all attributes $X_1, \ldots, X_n, X$. Since X_1 has no parent, we obtain from the Bayesian network that

   $$X_1 \perp Y|\emptyset \Rightarrow P(Y|X_1) = P(Y).$$

   Using the above result, and that X_2 is independent of Y given X_1 we obtain that

   $$X_2 \perp Y|X_1 \Rightarrow P(X_2, Y|X_1) = P(X_2|X_1)P(Y|X_1) \Rightarrow P(X_2, Y|X_1) = P(X_2|X_1)P(Y).$$

   For the joint distribution of X_2 and Y we have:

   $$\begin{aligned} P(X_2, Y) &= \sum_{X_1} P(X_1, X_2, Y) \\ &= \sum_{X_1} P(X_2, Y|X_1)P(X_1)) \\ &= \sum_{X_1} P(X_2|X_1)P(Y)P(X_1)) \\ &= \sum_{X_1} P(X_2, X_1)P(Y) \\ &= P(X_2)P(Y). \end{aligned}$$

   Therefore, $X_2 \perp Y$. In the same way, by following the chain from X_2, we can deduce $X \perp Y$.
2. A descriptive attribute is independent from a join indicator that does not involve the attribute's relation. Consider the attribute X of relation R and the join indicator J_{ST}. J_{ST} has at most two parents, one descriptive attribute from relation S, and one descriptive attribute from relation T. Therefore, J_{ST} is a non-descendant of X. We prove that $X \perp J_{ST}$ in the same way as in case 1 by substituting Y with J_{RS}.
3. Two join indicators that do not involve a common relation are independent. Consider the join indicators J_{RS} and J_{TU}. J_{TU} is a non-descendant of J_{RS}. If J_{RS} does not have any parents, $J_{RS} \perp J_{TU}|\emptyset \Rightarrow J_{RS} \perp J_{TU}$. If J_{RS} has one parent X from relation R: $J_{RS} \perp J_{TU}|X$. From case, 2 we have $X \perp J_{TU}$, from which we easily obtain that $J_{RS} \perp J_{TU}$. If J_{RS} has two parents,$R.X$ and $S.Y$, it holds that $J_{RS} \perp J_{TU}|X, Y$. From case 2 we have $X \perp J_{TU}$ and $Y \perp J_{TU}$. From these we easily obtain that $J_{RS} \perp J_{TU}$. □

4 Efficient model construction

The fixed model structure described in the previous section enables an efficient and scalable model construction algorithm. The construction algorithm is given the schema graph as input, and constructs a junction tree that will be stored in the DBMS catalog. Note that for our fixed model structure, moralization is invertible. That is, given the moral graph, there is only one Bayesian network that can produce it which conforms to our fixed structure. Therefore, our construction algorithm operates on the moral graph directly.

Due to the fixed model structure, the model construction algorithm has two tasks: first, for each relation it needs to find the best moral graph that is a tree, and for each join indicator the two descriptive attributes from the relations it joins that have the highest degree of correlation using some measure of dependence.

The construction algorithm proceeds as follows (Algorithm 1 provides pseudo-code). Given the schema graph $\mathsf{G} = (\mathsf{R}, \mathsf{J})$ it finds the best "local" moral graph MG_R for each relation $R \in \mathsf{R}$, and the best two predictors for each join indicator $J \in \mathsf{J}$. The concatenation of the "local" moral graphs and the edges between the join indicators and the descriptive attributes form the final moral graph. The measure of dependence we use is *mutual information*. We chose mutual information because of its close connection to the Kullback–Leibler distance [9], but we could also use other dependence measures like mean-square contingency coefficient [18] instead. The mutual information between two random variables X, Y is

Algorithm 1 Construction of the moral graph

```
1: function CONSTRUCT-MG(A,J,R)
2:     MG = $(\mathsf{A} \cup \mathsf{J}, \{\})$
3:     for $R$ in R do
4:         $\mathsf{MG}_R = (\mathsf{A}_R, \{\})$
5:         for $A_i$ in $\mathsf{A}(R)$ do
6:             for $A_j$ in $\mathsf{A}(R)$ do
7:                 Add $(A_i - A_j)$ to $\mathsf{MG}_R$ with weight $I(A_i : A_j)$
8:         $\mathsf{MG}_R$ =MAXIMUM-SPANNING-TREE($\mathsf{MG}_R$)
9:         Add edges of $\mathsf{MG}_R$ to MG
10:    for $J_{ij}$ in J do
11:        $A_i^{best} = \arg\max_{A_i \in \mathsf{A}(R_i) \wedge I(A_i : J_{ij}) > 0} I(A_i : J_{ij})$
12:        $A_j^{best} = \arg\max_{A_j \in \mathsf{A}(R_j) \wedge I(A_j : J_{ij}) > 0} I(A_j : J_{ij})$
13:        if $A_i^{best} \neq$ null then
14:            Add $(A_i^{best} - J_{ij})$ to MG
15:        if $A_j^{best} \neq$ null then
16:            Add $(A_j^{best} - J_{ij})$ to MG
17:        if $A_i^{best} \neq$ null $\wedge A_j^{best} \neq$ null then
18:            Add $(A_i^{best} - A_j^{best})$ to MG
19:    return MG
```

$$I(X;Y) = \sum_x \sum_y P(x,y) \log \left(\frac{P(x,y)}{P(x)P(y)} \right).$$

To construct MG_R, we need to test every pair (A_i, A_j) of attributes of R for dependence, and determine their dependence measure. We can extract the joint distribution $P(A_i, A_j)$ by normalizing the result of the SQL query

`select` $A_i, A_j,$ `count(*) from` R `group by` A_i, A_j.

Having done this for every (A_i, A_j) pair, we have a complete graph with the attributes of R as vertices, where the edge $A_i - A_j$ is annotated with $I(A_i; A_j)$. The maximum spanning tree of this complete graph (also called the Chow–Liu tree [9]) is the desired MG_R.

For each join indicator J_{RS} that connects relations R and S, the construction algorithm must test dependence for every triple (A_i, J_{RS}, B_j) where A_i is a descriptive attribute of relation R and B_j is a descriptive attribute of relation S. We can retrieve the distribution $P(A_i, B_j, J_{RS})$ in two steps. First, by dividing the result of the query

`select` $A_i, B_j,$ `count(*) from` R, S
`where` J_{RS} `group by` A_i, B_j

by the size of the Cartesian product $|R \times S|$, we obtain the part of the distribution for the **T** value of the join indicator, $P(A_i, B_j, J_{RS} = \mathbf{T})$. For the rest of the distribution, it holds

$$P(A_i, B_j, J_{RS} = \mathbf{F}) = P(A_i, B_j) - P(A_i, B_j, J_{RS} = \mathbf{T}).$$

Note that by Theorem 1 attributes A_i and B_j are independent since they belong to different relations:

$$P(A_i, B_j, J_{RS} = \mathbf{F}) = P(A_i)P(B_j) - P(A_i, B_j, J_{RS} = \mathbf{T}).$$

We can estimate $P(A_i)$ and $P(B_j)$ by normalizing the result of the queries

`select` A_i, `count(*) from` R `group by` A_i,

`select` B_j, `count(*) from` S `group by` B_j.

Therefore, we are able to estimate $P(A_i, B_j, J_{RS})$ without forming the Cartesian product $R \times S$.

The work done to form the joint probability distributions $P(A_i, A_j)$, and $P(A_i, B_j, J_{RS})$ in order to measure mutual information is not lost. Many of these sets of random variables will likely form cliques in the junction tree. Therefore, we cache the distributions of the pairs and triples of random variables, and no database access is needed to form the final junction tree.

If the moral graph is not chordal, a triangulation procedure adds edges to make it chordal. We defer the discussion on triangulation until Sect. 6. We note that if the input schema graph is a tree, the moral graph is always chordal, and triangulation is not needed (c.f. Sect. 6). Triangulation may need further database access for the new cliques it creates. Finally, the junction tree is constructed in the standard way [10], and stored in the DBMS catalog.

How exactly the junction tree is stored is DBMS-specific, and multiple implementations are possible. We briefly describe our choices based on PostgreSQL. The PostgreSQL catalog is itself stored as relational tables. For example, the `pg_statistics` relation stores all PostgreSQL statistics on attributes. We add four new relations to the PostgreSQL catalog:

1. `pg_descattr` contains all the descriptive attributes, and pointers to the attribute entries of the PostgreSQL catalog.
2. `pg_joinind` contains information about the join indicators.
3. `pg_clique` contains the cliques of the junction tree and their inter-connections. The junction tree root is chosen at random.
4. `pg_probdistr` contains the joint probability distributions kept. The distributions are kept in different relations, because a clique that is created by triangulation may contain several distributions (see Sect. 6 for details), and different separators may have the same distribution.

Due to our fixed model structure, the model construction algorithm needs to check only pairs of attributes for dependence, and needs only to issue two-table joins. Due to these features, our algorithm issues the same queries to the database as CORDS [18], a highly efficient previous approach to discovering correlations. CORDS captures pairwise correlations using a very similar algorithm to the one described above. It uses a different dependence measure, namely mean-square contingency coefficient instead of mutual information. In CORDS the joint distributions are not used to provide selectivity estimates. Instead, lightweight "Column-Group statistics" are used to indicate presence of a correlation. During the correlation discovery phase, the same queries are issued to the database since both methods discover pairwise correlations using two-table joins. By using samples of tables, the overhead of the CORDS construction algorithm was shown to be independent of the size of the database, while still resulting in high quality estimates. That result applies to our model construction algorithm as well, and a small sample is usually enough to capture most distributions (we discuss this further in Sect. 9).

5 Selectivity estimation

Once the junction tree of the database has been constructed and stored in the DBMS catalog, it can be used for selectivity estimation. Consider a query that contains selection predicates and join predicates

```
select * from $R_1, R_2, \ldots, R_n$
where $A_1 = a_1$ and ... $A_m = a_m$ and
      $B_1 = C_1$ and ... $B_k = C_k$.
```

We detect the descriptive attribute that corresponds to every selection predicate $A_i = a_i$, and the join indicator that corresponds to every join forming the sets of random variables A_q and J_q. Denote by ϕ_A the predicate that corresponds to the descriptive attribute A. The selectivity of the query is equal to

$$\Pr\left(\bigwedge_{A\in\mathsf{A}_q}\phi_A, \bigwedge_{J\in\mathsf{J}_q} J = \mathbf{T}\right).$$

During query optimization, the plan enumerator asks for cardinalities of intermediate result sizes. Equivalently, it will ask for probabilities of the form

$$\Pr\left(\bigwedge_{A\in\mathsf{A}}\phi_A, \bigwedge_{J\in\mathsf{J}} J = \mathbf{T}\right).$$

where $\mathsf{A} \subset \mathsf{A}_q$, and $\mathsf{J} \subset \mathsf{J}_q$ (depending on heuristics used during plan enumeration, only a subset of all possible combinations will be queried). Note that this includes the possibility of performing selection after a join (for example, for expensive predicates [8,17]). Our task is to compute such *probability queries*.

We solve this problem using a junction tree representation of our statistical model in two steps. First, the so-called Steiner tree, the minimal sub-tree that contains A_q and J_q, is extracted from the junction tree. Second, a selectivity estimation algorithm substitutes the values of the query variables, and eliminates the remaining variables in the tree. The final unnormalized number is returned as the result of the query.

Algorithm 2 Steiner tree computation

```
1: procedure CONSTRUCT-STEINER-TREE($\mathsf{T}$,$\mathsf{V}$)
2:   $C_{root}$ = root of $\mathsf{T}$
3:   CLIQUE-INCLUDE($C_{root}$,$\mathsf{V}$)
4:   return cliques with $C$.include = $\mathbf{T}$
5: procedure CLIQUE-INCLUDE($C$,$\mathsf{V}$)
6:   if $C$ is leaf then
7:     $C$.varsInSubtree = $\mathsf{V} \cap C$.vars
8:     if $C$.varsInSubtree $\neq \emptyset$ then
9:       $C$.include = $\mathbf{T}$
10:  else
11:    localVarsInSubtree = $\mathsf{V} \cap C$.vars
12:    $C$.varsInSubtree = localVarsInSubtree
13:    for $c$ in $C$.children do
14:      CLIQUE-INCLUDE($c$,$\mathsf{V}$)
15:      if localVarsInSubtree $\supset c$.varsInSubtree then
16:        $c$.include = $\mathbf{F}$
17:      $C$.varsInSubtree = $C$.varsInSubtree $\cup$ $c$.varsInSubtree
18:    if $C$.varsInSubtree $\neq \emptyset$ then
19:      $C$.include = $\mathbf{T}$
20:    if $C$.varsInSubtree = $\mathsf{V}$ then
21:      Stop the algorithm
```

Steiner tree computation is in our setting different from the classical problem, because we need to select the minimal subtree that contains a set of variables, and not a set of nodes. We discuss it first in Sect. 5.1. Then, in Sect. 5.2 we describe the algorithm for selectivity estimation we use, and in Sect. 5.3 we describe an alternative algorithm that has better performance for Steiner trees of low degree (e.g., for chain queries).

5.1 Steiner tree computation

Given a rooted junction tree T and a set of variables $\mathsf{V}_q = \mathsf{A}_q \cup \mathsf{J}_q$, our goal is to extract the minimal connected junction tree that contains these variables. For each clique of the tree C, we maintain a Boolean variable "C.include" that indicates whether the clique is included in the resulting tree, and we maintain a set "C.varsInSubtree" that contains the subset of V that is included in the subtree rooted at C. The algorithm, shown in Algorithm 2, sets the "include" value for each clique. If a clique is a leaf, it is included if it contains variables in the query. A non-leaf clique is included if the variables contained in the sub-tree originating from the clique contain query variables. The fine point of the algorithm is that we do not want to include a clique whose parent already covers the variables of interest. Lines 15 and 16 of the algorithm set the include bit of a clique's child to false if the parent clique already covers the needed variables.

5.2 Selectivity estimation via junction tree propagation

Assume a rooted Steiner tree T, a set of random variables V, and associated predicates whose joint selectivity we want

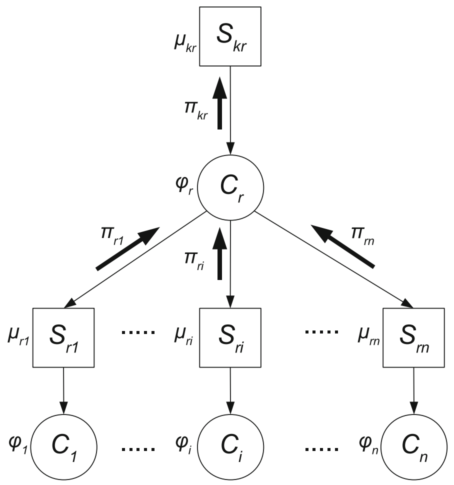


Fig. 3 Selectivity estimation algorithm

Algorithm 3 Basic propagation algorithm.

```
1: function COMPUTE-SELECTIVITY($\mathsf{T}, \mathsf{A}_q, \mathsf{J}_q$)
2:     return COMPUTE-PROB-REC($\mathsf{T}$.root, $\mathsf{A}_q, \mathsf{J}_q$)
3: function COMPUTE-PROB-REC($C_r, \mathsf{A}_q, \mathsf{J}_q$)
4:     $\Pi = \emptyset$
5:     for $i = 0$ to $n$ do
6:         $\pi_{ri}$ =COMPUTE-PROB-REC($C_r$.children[i], $\mathsf{A}_q, \mathsf{J}_q$)
7:         $\Pi = \Pi \cup \pi_{ri}$
8:     $\phi_r^* = \phi_r[J_R = \mathbf{T}, \bigwedge_{A \in \mathsf{A}_{qr}} A = a]$
9:     $\mu_{kr}^* = \mu_{kr}[\bigwedge_{A \in \mathsf{A}_{qkr}} A = a]$
10:    for $i = 1$ to $n$ do
11:        $\phi_r^* = \frac{\phi_r^* \pi_{ri}}{\mu_{ri}^*}$
12:        $\phi_r^* = \sum_{C_r \cup C_i \cup \S_{ri} - \mathsf{A}_q - S_{kr}^*} \phi_r^*$
13:    return $\phi_r^*$
```

to evaluate. Selectivity estimation proceeds as junction tree propagation [10]. Messages are sent from the root to the leaves, and then returned from the leaves to the root. When the root has received messages from all its children, the algorithm terminates.

Assume the clique C_r shown in Fig. 3 with children $C_q, \ldots, C_n$, parent C_k, and upwards separator S_{kr}. The entry point of the recursive call in C_r is C_k asking C_r for a message π_{kr}. Denote by $\mathsf{V}_r = \mathsf{V} \cap C_r$ ($\mathsf{V}_{kr} = \mathsf{V} \cap S_{kr}$) the subset of query variables that C_r (S_{kr}) contains, and by $\Psi_r = \{\psi_i(V_i) | V_i \in \mathsf{V}_r\}$ ($\Psi_{kr} = \{\psi_i(V_i) | V_i \in \mathsf{V}_{kr}\}$) the set of query predicates on variables V_r (V_{kr}).

If $n = 0$ (i.e., C_r is a leaf in the Steiner tree), C_r will substitute the predicates ψ_i in its potential ϕ_r, and then sum out all variables that do not belong to the upwards separator S_{kr}. The resulting factor (or real number) is the message sent to the parent clique C_k:

$$\pi_{kr} = \sum_{C_r - S_{kr}} \phi_r[\Psi_r](C_r).$$

By substituting the predicates Ψ_r into the clique potential ϕ_r, we get an unnormalized factor, whose sum is the joint probability of all predicates Ψ_r. Variables that do not belong to the separator S_{kr} are not needed in the upwards sub-tree, so they can be safely eliminated without implying an independence assumption that is not already encoded in the model.

If C_r is an intermediate clique, it will first call the selectivity estimation algorithm on its children $C_1, \ldots, C_n$, obtaining the messages $\pi_{ri} | i = 1, \ldots, n$. Then, it will substitute the predicates Ψ_r in its potential, and also substitute the predicates Ψ_{ri} in the potential of every separator S_{ri}:

$$\phi_r^* = \phi_r[\Psi_r]$$
$$\mu_{ri}^* = \mu_{ri}[\Psi_{ri}], \quad i = 1, \ldots, n$$

Then, it will iteratively multiply ϕ_r^* with π_{ri} dividing by the modified separator potential μ_{ri}^*, while eliminating all unnecessary variables. The unnecessary variables at step i of the iteration are those that do not appear in the upwards separator S_{kr}, or the separators S_{rj}, $j > i$. Let $\hat{\mathsf{V}}_r = C_r \cup C_1 \cup \cdots \cup C_n$. Then, the iteration can be written as

$$\phi_r^* \leftarrow \sum_{\hat{\mathsf{V}}_r - S_{kr} - \cup_{j>i} S_{rj}} \frac{\phi_r^* \pi_{ri}}{\mu_{ri}^*}.$$

By eliminating variables while multiplying the cliques, we ensure that we never carry around a factor with higher cardinality than needed, while not making any independence assumptions (other than the ones implied by the model) along the way. Algorithm 3 provides pseudo-code for the selectivity estimation algorithm based on propagation.

We conclude this section with an example. Assume that we are asked to estimate the selectivity of the query

```
select * from lineitem, supplier, customer
where l_suppkey = s_suppkey
  and s_nationkey = c_nationkey
  and c_acctbal = a
  and l_shipdate = b
```

The equivalent probability query is

$$\Pr(J_{LS} = \mathbf{T}, J_{SC} = \mathbf{T}, \texttt{c\_acctbal} = a, \texttt{l\_shipdate} = b).$$

Assume the junction tree of Fig. 2d. The Steiner tree for the query is shown in Fig. 4 where the cliques have been numbered. Assume that C_1 is the root. C_1 will first call the algorithm recursively on its children C_2, C_3. C_2 is a leaf clique, so it will substitute the related predicates in its potential, and sum out unneeded variables:

$$\phi_2^*(\texttt{s\_acctbal}) = \phi_2[J_{SC} = \mathbf{T}, \texttt{c\_acctbal} = a].$$

Clique C_2 will then return back to C_1 the message $\pi_{12} = \phi_2^*(\texttt{s\_acctbal})$. Note that no marginalization was performed by C_2. Variable `s_acctbal` cannot be eliminated

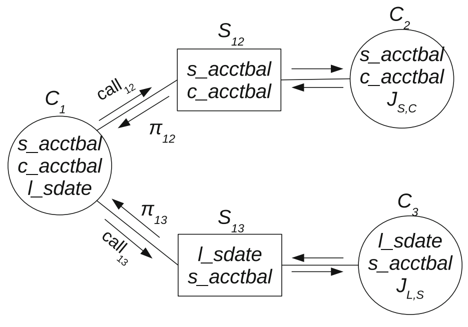


Fig. 4 Selectivity estimation via propagation

because it appears in the separator S_{12} and does not appear in the query. Were C_2 to eliminate s_acctbal, we would lose the indirect dependency between J_{SC} and l_shipdate through s_acctbal, i.e., we would be making an ad-hoc independence assumption that is not implied by the model.

Similarly, C_3 will compute its message π_{13} as

$$\phi_3^*(\texttt{s\_acctbal}) = \phi_3[J_{LS} = \mathbf{T}, \texttt{l\_shipdate} = b].$$

The control flow returns to C_1 once the clique has collected the messages π_{12}, π_{13} from its children. C_1 will first substitute the relevant predicates to its potential and the potential of the downwards separators, while eliminating unneeded variables:

$$\phi_1^*(\texttt{s\_acctbal}) = \phi_1[\texttt{c\_acctbal} = a, \texttt{l\_shipdate} = b],$$
$$\mu_{12}^*(\texttt{s\_acctbal}) = \mu_{12}[\texttt{c\_acctbal} = a],$$
$$\mu_{13}^*(\texttt{s\_acctbal}) = \mu_{13}[\texttt{l\_shipdate} = b].$$

The clique will then multiply ϕ_1^* with the messages received from its children. Assume that the multiplication with π_{12} happens first resulting to the intermediate factor ϕ_1':

$$\phi_1'(\texttt{s\_acctbal}) = \frac{\phi_1^*(\texttt{s\_acctbal})\pi_{12}(\texttt{s\_acctbal})}{\mu_{12}^*(\texttt{s\_acctbal}}.$$

Then, the intermediate factor will be combined with π_{13}, and the variable s_acctbal can be finally eliminated:

$$\phi_1''() = \sum_{\texttt{s\_acctbal}} \frac{\phi_1'(\texttt{s\_acctbal})\pi_{13}(\texttt{s\_acctbal})}{\mu_{13}^*(\texttt{s\_acctbal})}.$$

The factor $\phi_1''()$ is a real number, and it is returned as the selectivity of the query.

5.3 Selectivity estimation via dynamic programming

Algorithm 3 described in the previous section adapts junction tree propagation to perform selectivity estimation. The algorithm performs a full propagation of the Steiner tree for every selectivity estimate asked by the query optimizer during plan enumeration. This is unnecessary; we can reduce the number of multiplications (but, as will be evident later, not the running time in all cases) by exploiting the series of

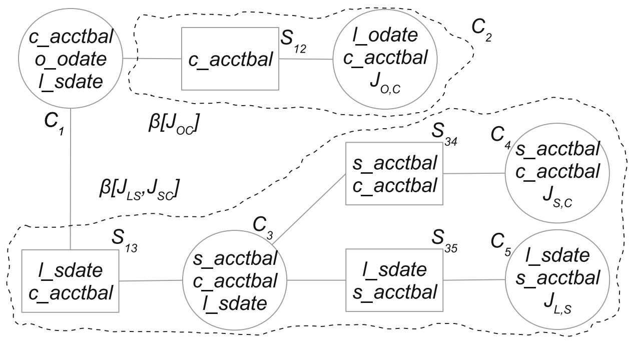


Fig. 5 Selectivity estimation via dynamic programming

requests made by the query optimizer. Assume a dynamic-programming, bottom-up plan enumerator and the 3-join query

```
select * from lineitem,orders,customer,supplier
where l_suppkey = s_suppkey
  and s_nationkey = c_nationkey
  and o_custkey = c_custkey
  and l_shipdate = a
```

The Steiner tree for the query is shown in Fig. 5. Clique C_1 is the root. The equivalent probability query is

$$s_1 = \Pr(J_{LS} = \mathbf{T}, J_{SC} = \mathbf{T}, J_{OC} = \mathbf{T}, \texttt{l\_shipdate}{=}a).$$

The basic idea that we exploit is that before the optimizer asks for a joint selectivity on a set of predicates, it has already asked (and received estimates for) the selectivities of some subsets of these predicates using the junction tree of Fig. 5. For example, when the optimizer considers the expressions $\sigma_{\texttt{l\_shipdate}=a}(L) \bowtie S \bowtie C$, and $O \bowtie C$, it receives estimates for

$$s_2 = \Pr(J_{LS} = \mathbf{T}, J_{SC} = \mathbf{T}, \texttt{l\_shipdate} = a),$$
$$s_3 = \Pr(J_{OC} = \mathbf{T}).$$

To estimate s_2, we need to marginalize from clique C_2:

$$s_2 = \sum_{\texttt{l\_orderdate},\texttt{c\_acctbal}} \phi_2[J_{OC} = \mathbf{T}]].$$

To estimate s_3, we need to multiply cliques C_3, C_4, C_5:

$$s_3 = \sum_{\substack{\texttt{s\_acctbal},\\ \texttt{c\_acctbal}}} \frac{\phi_3[\texttt{l\_shipdate} = a]\phi_4[J_{SC} = \mathsf{T}] \\ \phi_5[\texttt{l\_shipdate} = a, J_{LS} = \mathbf{T}]}{\mu_{34}\mu_{35}[\texttt{l\_shipdate} = a]}.$$

Even after having estimates for s_2 and s_3, in order to estimate s_1 (which computes the selectivity of the whole query), we would need to perform propagation in the whole junction tree, thereby multiplying five cliques. Define the factors

$$\beta[J_{LS}, J_{SC}](\texttt{c\_acctbal}) \equiv \sum_{\texttt{s\_acctbal}} \frac{\phi_3[\texttt{l\_shipdate} = a]\phi_4[J_{SC} = \mathrm{T}] \; \phi_5[\texttt{l\_shipdate} = a, J_{LS} = \mathbf{T}]}{\mu_{34}\mu_{35}[\texttt{l\_shipdate} = a]},$$

$$\beta[J_{OC}](\texttt{c\_acctbal}) \equiv \sum_{\texttt{l\_orderdate}} \phi_3[J_{OC} = \mathbf{T}].$$

We are forming $\beta[\Pr(J_{LS}, J_{SC})]$ ($\beta[J_{LP}]$) while computing s_2 (s_3), just before eliminating `c_acctbal`. Had we stored these factors, we could compute s_1 using only two multiplications as

$$s_1 = \sum_{\substack{\texttt{c\_acctbal},\\ \texttt{o\_orderdate}}} \frac{\phi_1[\texttt{l\_shipdate}{=}a]\beta[J_{LS}, J_{SC}]\beta[J_{OC}]}{\mu_{12}\mu_{13}[\texttt{l\_shipdate}{=}a]}.$$

We generalize this process as follows. We assume that predicates are pushed under joins. If this does not hold (e.g., for expensive predicates), our method can be easily generalized. For each *join expression* $R_1 \bowtie \cdots \bowtie R_n$ considered by the plan enumerator, we extract the set of join indicators in the expression, J, and define the β-factor $\beta[\mathsf{J}]$ as the multiplication of all the cliques C that contain J, marginalized to all the variables that belong to separators that connect C to the rest of the Steiner tree. The factor is computed as the result of the selectivity estimate for the join expression before eliminating some variables. The β-factor is stored in a hash table indexed by the join indicator identifiers. Then, the β-factor for $\mathsf{J}_1 \cup \mathsf{J}_2$ can be computed using the stored β-factor for J_1 and J_2 as

$$\beta[\mathsf{J}_1 \cup \mathsf{J}_2] = \sum_{\mathsf{V}} \frac{\beta[\mathsf{J}_1]\beta[\mathsf{J}_2]C_1 \ldots C_n}{\prod \mu_{ij}},$$

where $C_1, \ldots, C_k$ are the cliques that belong to the path between $\beta[\mathsf{J}_1]$ and $\beta[\mathsf{J}_2]$.

This algorithm minimizes the number of multiplications, but this does not always result in better optimization time. The reason is that the cardinalities of the stored β-entries can become large, and one multiplication can dominate the time needed to perform selectivity estimation. Therefore, the dynamic programming algorithm is suitable for Steiner trees of low fanout, and in particular for chain-shaped Steiner trees.

6 Coping with cyclic schemas

A junction tree can be constructed only if the moral graph G^m is *chordal*. A graph is chordal if it does not contain a cycle with more than three nodes without a "chord," an edge that connects two non-adjacent nodes in the cycle.

For the case that the moral graph is not chordal, an extra step before creating the junction tree is needed. Triangulation is a process that adds edges to the moral graph until a chordal graph has been constructed. It is usually realized as a process of node elimination [3]. A node is eliminated by connecting all its neighbors, and removing the node and its edges from the graph. A node is called simplicial if it can be eliminated without introducing extra edges (i.e., all its neighbors are connected).

Algorithm 1 may create a moral graph that is not chordal. This can happen if the schema graph of the database contains cycles, but it cannot happen in a tree-shaped schema graph. In our fixed model structure, join indicators are simplicial nodes because they have at most two parents that are connected during moralization (e.g., in the moral graph shown in Fig. 2, we can see that the join indicator nodes $J_{LP}, J_{LS}, \ldots$ are all simplicial nodes). Thus, triangulation will not add edges involving join indicators. It will only add edges that contain descriptive attributes (e.g., the edge between `l_sdate` and `o_odate` in Fig. 2). This can only happen in the case of a cyclic schema graph, where different descriptive attributes are parents to join indicators of the same relation. Disallowing the latter is an easy solution to guarantee that all histograms are 2-dimensional. Alternatively, one can choose to allow the possibility of higher-dimensionality histograms.

The latter is acceptable in most cases because, even with cliques containing 3 or more descriptive attributes, we often only need to store at most 2-dimensional histograms. For example, consider the clique (`s_acctbal`, `c_acctbal`, `l_sdate`) created during triangulation in Fig. 2. The three variables in this clique are independent of each other since they belong to different relations. Hence, this 3-dimensional probability distribution can be stored compactly as a collection of three 1-dimensional distributions. Although this seems counter-intuitive, it is actually expected; the junction tree encodes fewer conditional independences than the original Bayesian network (because it contains more edges). On the other hand, the clique (`l_sdate`, `l_cdate`, `o_odate`) requires us to store a 2-dimensional histogram on (`l_sdate`, `l_cdate`). In future work, we would like to investigate if it is possible to construct a custom elimination sequence that always produces a chordal graph that needs only 2-dimensional distributions.

7 Query-specific modeling

In this section we present an alternative way to construct a junction tree for a query, which we term query-specific junction tree. Constructing a query-specific junction tree improves both estimate accuracy and optimization time compared to the method presented so far. Further, it eliminates the need for triangulation as long as the queries are acyclic, even if the schema graph contains cycles. Therefore, the method in this section provides a formal guarantee that only two-dimensional histograms are needed for acyclic queries on possibly cyclic schemas. Finally, query-specific modeling provides a formal mapping from the size of the query to the size of the resulting junction tree. For example, for a query that joins n relations and contains one selection predicate per relation, the resulting junction tree contains exactly $n-1$ cliques.

Query-specific modeling solves an important problem of junction trees, namely that inference complexity is guided by the distance between random variables in the precomputed junction tree. Two queries of equal complexity (same number of join and selection predicates) may yield drastically different optimization times depending on the position of the random variables in the junction tree. By computing query-specific junction trees, we are able to overcome this problem and provide a mapping from query complexity to optimization time. The price we need to pay for query-specific modeling is increased model space. For n relations with m attributes per relation and k join predicates we need to keep $O((n+2k)m^2)$ 2-dimensional histograms.

Example: We explain the basic idea of query-specific modeling with an example. Consider a relation $R(X_1, X_2, X_3, X_4)$. According to traditional modeling, which we have followed so far, the best junction tree is constructed off-line. In order to construct the junction tree, we test every pair of attributes for dependence by forming all the distributions $P(X_i, X_j)$ and determining the mutual information $I(X_i; X_j), \forall X_i, X_j$. Assume that the best junction tree contains cliques $C_1 = (X_1, X_2), C_2 = (X_2, X_3)$, and $C_3 = (X_3, X_4)$ connected in a chain $C_1 - C_2 - C_3$. During model construction we store in the DBMS catalog the distributions $P(X_1, X_2)$, $P(X_2, X_3)$, and $P(X_3, X_4)$, discarding all other distributions $P(X_i, X_j)$ that were formed to test dependence. Then, the probability query $q = \Pr(X_1 = x_1, X_4 = x_4)$ needs two clique multiplications to be computed:

$$q = \sum_{X_3, X_4} \frac{P(X_1 = x_1, X_2)P(X_2, X_3)P(X_3, X_4 = x_4)}{P(X_2)P(X_3)}.$$

Consider now the alternative of not forming a junction tree up-front, but storing all the probability distributions that we use for testing dependence. In this scenario, all distributions $P(X_i, X_j)$ are stored in the DBMS catalog, but the junction tree is formed on demand, when a query is received. Then, we can compute query q directly by using the stored clique $P(X_1, X_4)$. This method results in both better accuracy and faster selectivity estimation, at the price of storing six instead of three 2-dimensional distributions.

In query-specific modeling, the off-line model construction algorithm returns a list of cliques C that are created while testing random variables for dependence. For each relation R with attributes $X_1, \ldots, X_n$ we store the cliques $C_{ij}^R = (R.X_i, R.X_j), \forall j \neq i$. For each join indicator J_{RS} that contains a join predicate between relations $R(X_1, \ldots, X_n)$ and $S(Y_1, \ldots, Y_m)$ we store the cliques $C_{ij}^{RS} = (X_i, Y_j, J_{RS}) \forall i, j$. Note that this does not incur overhead during model construction; we need to form all these distributions to test for dependence anyway. The added overhead of writing the distributions to the DBMS catalog is very small, and is compensated by the lack of the junction tree construction phase, leading to overall faster model construction (see Sect. 9.5 for an experimental justification).

Constructing a valid junction tree is deferred until runtime, when the query optimizer receives a query. The problem is then, given a set of stored cliques and a query that contains several descriptive attributes and join indicators, to create the best junction tree for the query. The optimization metric now consists of (i) the number of cliques in the junction tree that we need to minimize and (ii) as a secondary goal, the mutual information which we need to maximize. The construction of the best moral graph, the construction of the junction tree, and the Steiner tree are now merged into one step.

Algorithm sketch: Assume a query q on relations R_q that contains predicates on descriptive attributes A_q and join indicators J_q. We decompose the task of creating the best query-specific junction tree into three tasks:

1. Selecting a "local" junction tree for each relation $R \in \mathsf{R}_q$
2. Selecting the best clique for each join indicator $J_{RS} \in \mathsf{J}_q$
3. Connecting the above local junction trees and join indicator cliques so that they form a valid junction tree.

We consider the three tasks in turn. For each relation $R(X_1, \ldots, X_n) \in \mathsf{R}_q$, we need to find the local junction tree JT_R that "covers" the descriptive attributes of the relation that are in the query $\mathsf{A}_{Rq} = \mathsf{A}_q \cap \{X_1, \ldots, X_n\}$ using the minimal number of nodes. An important observation is that cliques that contain at least one descriptive attribute not in the query need not be considered during junction tree construction, since they would needlessly increase the number of cliques in the junction tree. There are three cases we need to consider, depending on the number of query attributes that belong to relation R:

1. $|\mathsf{A}_{Rq}| = 1$: The query contains only one descriptive attribute from R. Assuming that there will is a join indicator J_{RS} involving R (i.e., the query does not contain a Cartesian product), we can cover this attribute in the same clique that contains J_{RS}. JT_R in this case is empty.
2. $|\mathsf{A}_{Rq}| = 2$: The query contains two descriptive attributes from R, X and Y. JT_R in this case consists of the single clique (X, Y)
3. $|\mathsf{A}_{Rq}| = n > 2$: The query contains several descriptive attributes from R. First, we collect the available cliques C_{Rq} that contain only descriptive attributes from A_{Rq}. Note that this step can be implemented efficiently by indexing the catalog table that contains all cliques on the clique attributes. Also note that at this step the actual clique distributions do not need to be retrieved. Each

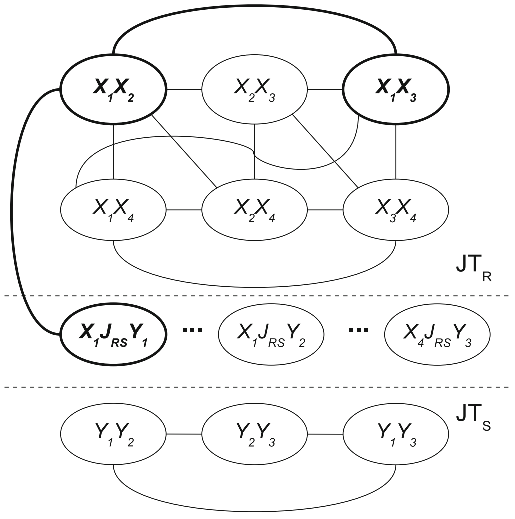


Fig. 6 Query-specific modeling. The bold lines indicate the cliques and edges that form the selected junction tree

clique $C \in \mathsf{C}_{Rq}$ with attributes X, Y is associated with a weight $w_C = I(X; Y)$. We then find the Chow–Liu tree over these attributes, i.e., the maximum-weight spanning tree of a complete graph on A_{Rq}, with weight on an edge X, Y being $I(X; Y)$.

Assume, for example the relations $R(X_1, X_2, X_3, X_4)$ and $S(Y_1, Y_2)$ connected by the join indicator J_{RS}. Figure 6 shows the complete graph for all the possible cliques of R and S. For example, the three attributes of S can form three possible cliques (Y_1, Y_2), (Y_2, Y_3), and (Y_1, Y_3) all connected with each another. Cliques can be connected if they share a variable. Assume now the query $\Pr(X_1 = x_1, X_2 = x_2, X_3 = x_3, J_{RS} = \mathbf{T}, Y_1 = y_1)$. For relation R, we have $\mathsf{A}_{Rq} = \{X_1, X_2, X_3\}$ (case 3 above). We collect all cliques that contain only attributes from A_{Rq}: $(X_1, X_2), (X_2, X_3), (X_1, X_3)$. We then select cliques (X_1, X_2) and (X_1, X_3) using the Chow–Liu technique (hard-coded and optimized for small numbers of descriptive attributes). For relation $S, \mathsf{A}_{Sq} = \{Y_1\}$ (case 1 above). Therefore, the junction tree for S is empty.

In the general case, for each join indicator J_{RS}, we need to select one clique of the form (X, J_{RS}, Y), where X and Y are descriptive attributes from R and S respectively. We have again three cases depending on the uncovered attributes of R and S:

1. There exist descriptive attributes X and Y from R, S that have not been covered. This is the result of case 1 above, when $|\mathsf{A}_{Rq}| = 1$ and $|\mathsf{A}_{Sq}| = 1$. Then, we select the clique (X, J_{RS}, Y).
2. There exists an attribute X of R that is uncovered. Then, we select the clique (X, J_{RS}, Y_i) that has the maximal weight, where $Y_i \in \mathsf{A}_{Sq}$. The weight of a clique $C = (X_i, J_{RS}, Y_j)$ is measured as $w_C = I(X_i, J_{RS}) + I(Y_j, J_{RS})$.
3. There are no uncovered attributes. Then we select the clique (X_i, J_{RS}, Y_j) with the maximal weight, where $X_i \in \mathsf{A}_{Rq}, Y_j \in \mathsf{A}_{Sq}$.

Once we have JT_R for each relation R in the query, and a clique for each join indicator J_{RS}, it is trivial to connect these in a valid junction tree. For each clique (X, J_{RS}, Y), we identify a clique of JT_R, (X, X') that contains X, and a clique of S, (Y, Y') that contains Y, and we add the edges $(X, X') - (X, J_{RS}, Y) - (Y, Y')$.

In the example of Fig. 6, we are in case 1 above, since attribute Y_1 is not covered. We can choose any join indicator clique of the form $(X_i, J_{RS}, Y_1), i = 1, \ldots, 4$. In this case, we choose the clique (X_1, J_{RS}, Y_1), and connect it with (X_1, X_2) as shown in the figure.

Theorem 3 *The produced junction tree is valid.*

Proof We prove that the produced junction tree is a tree, and has the running intersection property. The family preservation property is not relevant in this case, since the junction tree is not produced by a Bayesian network. The running intersection property states (see Sect. 2) that for every pair of cliques C_1, C_2, the variables belonging to their intersection $C_1 \cap C_2$ are contained in every node in the path from C_1 to C_2 in the junction tree JT.

Assume that the tree-shaped join query q is over the relations $\mathsf{R}_q = \{R_1, \ldots, R_n\}$ and it contains join indicators $\mathsf{J}_q = \{J_{R_iR_j}\}, |\mathsf{J}_q| = n - 1$. The resulting junction tree consists of the local junction trees JT_{R_i}, and a clique $C_{R_iR_j}$ for each join indicator $J_{R_iR_j}$. Assume the join tree of the query $\mathsf{T}_q = (\{R_1, \ldots, R_n\}, E_q)$. The vertices of T_q are the relations, and the edges are the join predicates: $E_q = \{(R_i, R_j) | \exists J_{R_iR_j} \in \mathsf{J}_q\}$. Assume now the tree T with vertices $V_\mathsf{T} = \mathsf{JT}_{R_i}$ and an edge between JT_{R_i} and JT_{R_j} if there exists a clique $C_{R_iR_j}$. It is obvious that T_q and T are isomorphic, therefore T is a tree. Recall that every local junction tree JT_{R_i} is a tree since it is constructed using standard techniques. Therefore, the produced junction tree is a tree.

To prove the running intersection property, we have three cases:

1. Consider two cliques $C_1, C_2 \in \mathsf{JT}_R$ for $R \in \mathsf{R}_q$. Since JT_R is constructed using standard techniques, the running intersection property for C_1 and C_2 holds.
2. Consider $C_1 \in \mathsf{JT}_R, C_2 \in \mathsf{JT}_S$. Since the cliques contain descriptive attributes from different relations, $C_1 \cap C_2 = \emptyset$, and the running intersection property trivially holds. The same holds if C_2 is a join indicator clique that does not involve R.
3. Consider $C_1 = (X, Z) \in \mathsf{JT}_R, C_2 = (X, J_{RS}, Y)$. Assume the clique $C_3 \in \mathsf{JT}_R$ that is connected with

C_2. From case 1 above, the running intersection property holds for C_1 and C_3. From the construction algorithm, C_3 must contain X. Therefore, the running intersection property holds for C_1 and C_3. □

We are not aware of prior work in that literature that has proposed techniques similar to this. There are however several properties that are satisfied by the restricted class of graphical models that we consider here, that are needed to prove the correctness of this technique, and that may not be satisfied by arbitrary graphical models. We plan to investigate the generality of our approach in more depth in future work.

8 Implementation

We have implemented a graphical model foundation and the proposed selectivity estimation algorithms in PostgreSQL. To the best of our knowledge, this is the first implementation of graphical models-based selectivity estimation in the kernel of a real DBMS. Our implementation consists of two parts: the model construction prototype and the selectivity estimation prototype. Model construction (the implementation of Algorithm 1) is done outside the DBMS. It is written in Java, and it accesses the database using SQL queries. The resulting junction tree structure is stored as four relational tables in the PostgreSQL catalog as described at the end of Sect. 4.

The selectivity estimation part is implemented in the PostgreSQL backend. When the optimizer is called, the clique catalog table is scanned, and the junction tree structure is created in the backend. The clique potentials are not read from disk at this point, since that would incur significant and unnecessary overhead. Then the Steiner tree for the query is created with an implementation of Algorithm 2. Only then are the probability distributions in the much smaller, query-specific junction tree read from the catalog table. The startup overhead of loading the junction tree is very small: it takes between 1 and 3 milliseconds in all our experiments.

Selectivity estimation implements Algorithms 3 and the dynamic programming algorithm of Sect. 5.3 as operations on a junction tree structure. We implemented cliques and probability distributions as plan nodes in the PostgreSQL class system, including algorithms to multiply, divide, and marginalize multi-dimensional probability distributions [10]. We used simple equi-width histograms for multi-dimensional probability distributions. Implementing and studying clique multiplication for more advanced types of multidimensional histograms is an interesting future direction, which would enable further benefits [11]. For the experiments in Sects. 9.2, and 9.3 we modified PostgreSQL to use equi-width histograms in order to focus on the impact of missed correlations on result accuracy and execution time. In Sect. 9.4 we use the vanilla PostgreSQL summaries in order to showcase the competitiveness of our approach against a complete open-source DBMS. In every case, we used histogram sizes so that the space required by a typical histogram of our method (which contains two descriptive attributes and one join indicator) is the same as the space required by a PostgreSQL histogram. The result is that vanilla PostgreSQL can capture the marginal distributions with better accuracy than our technique (because it has more buckets allocated to them), but does not capture correlated distributions.

9 Experimental results

In this section, we cover the results of a comprehensive experimental evaluation performed using our implementation and three datasets. We first describe the datasets and the query workloads.

9.1 Data sets and workloads

We used three data sets in the experiments. First, we modified the TPC-H data generator to introduce pairwise correlations between attributes. Second, we used the IMDB data set. Finally the Accidents dataset is a synthetic data set whose goal was to approximate the statistics characteristics of a real database from a Department of Motor Vehicles [18,28].

1. **IMDB:** This data set is approximately 800 MB, and consists of 21 tables, a total of 101 columns, and 51,400,459 rows. It contains information about movies, actors, directors, etc., and was extracted from the IMDB website (http://www.imdb.com).
2. **TPC-H:** To obtain the TPC-H data set, we changed the data generator to introduce additional pairwise correlations between attributes. Unless mentioned otherwise, we use a scale factor of $s = 0.1$, a zipf factor of $z = 3$, and a Pearson correlation parameter of $r = 0.9$ in our experiments.
3. **Accidents:** The Accidents dataset [18] contains six relations, 23 columns that were used as descriptive attributes, and six join predicates that were used as join indicators. It is a synthetic data set, that aims to model a real data set of a Department of Motor Vehicles [28]. It contains several meaningful correlations, e.g., a correlation between attributes `model` and `make` of a table `Cars`. The generator includes a query generator that generates select-project-join-aggregate queries with random values in the selection predicates.

Queries. In Sect. 9.2 we use a suite of 8 queries over the IMDB and TPC-H datasets that contain joins whose selectivities are correlated with specific values of the descriptive attributes. The queries, denoted *TPCH-1 – TPCH-4,* *IMDB-1 – IMDB-4*, can be found in the electronic supplementary material that accompanies the paper.[3]

In Sect. 9.3 we use a random workload of selection and join queries over the restricted TPC-H schema graph shown in Fig. 2a. The experiments in this data set exhibit the benefits of our approach in a setting of very high correlations. We use 21 descriptive attributes and introduced two predicates per attribute. Figure 2d shows the junction tree created for a subset of those attributes. By taking all possible combinations, we generate 80 one-join queries. By randomly combining queries with one join, we generate 80 two-join queries, and so on until 400 queries with one to five join predicates are created. Finally, in Sect. 9.4, we use a large workload of generated queries against the Accidents database.

9.2 The impact of missed correlations

We proceed to offer insight into how missed correlation can affect the plan chosen by a query optimizer, and subsequently the time needed to execute a query. We use a skewed TPC-H data set with zipf factor $z = 3$. Correlations are inherent in the TPC-H schema. Consider the following query (denoted TPCH-1):

```
select c_name,c_address
from   lineitem,orders,customer
where  l_orderkey = o_orderkey and
       o_custkey = c_custkey and
       o_totalprice = x and
       l_extendedprice = y and c_acctbal = z
```

We seek to find customers that have placed orders with a particular combination of total price and prices of items, and with a further selection on the customer's account balance.

The attributes `l_extendedprice` and `o_totalprice` are correlated for tuples of `lineitem` and `orders` that join. Specifically, the total price of an order is a function of the price of its items, and their tax and discount. This positive correlation causes the selectivity $\Pr(J_{LO}, \phi_L, \phi_O)$ to be much higher than the product of the selectivities $\Pr(J_{LO})$, $\Pr(\phi_L)$, and $\Pr(\phi_O)$. PostgreSQL cannot capture such a correlation and therefore underestimates the selectivity $\Pr(J_{LO}, \phi_L, \phi_O)$ by a factor of 20 in our setting. This causes the optimizer to place the join $L \bowtie O$ before the join $O \bowtie C$ in the query plan. Further, it causes PostgreSQL to use a nested loop join for the final join with C, using the join $L \bowtie O$ as the inner relation. The plan picked by the optimizer using the default PostgreSQL estimates is $A : (O \bowtie_{HJ} L) \bowtie_{NLJ} C$, where the right operand of the $\bowtie$ operator is pipelined. In contrast, using our improved estimates, the plan picked by the optimizer is $B : (C \bowtie_{HJ} O) \bowtie_{HJ} L$. Thus, both the join order and the join algorithms picked are different. The execution time difference between these two plans is huge. While plan B (the one picked using our selectivity estimates) takes less than 2 s to execute in a cold state, plan A takes more than 40 min. After instructing PostgreSQL to not use a nested loop join, plan A is executed in 4 s. Therefore, the wrong join order can result in a 2× execution time penalty, and the nested loop join increases the execution time by orders of magnitude. Both decisions were made due to the missed correlation between the attributes `l_extendedprice`, `o_totalprice`, and the join indicator J_{LO}. By capturing these correlations using a graphical model, the optimizer can pick a better query plan.

Figure 7 shows results for eight queries on the TPC-H and IMDB data sets. All axes are in logarithmic scale. The supplementary material (see footnote 3) provides the SQL code of the queries and offers details on the data sets. All queries show how correlations lead to wrong join orders, and indeed the PostgreSQL optimizer chooses different plans using the default estimates versus using the estimates by our method. Our plans are better in all cases. Figure 7a shows the plan cost (measured as the number of intermediate tuples generated), as estimated by the query optimizer, and as measured after executing the query. Figure 7b shows the actual execution times for these queries.

PostgreSQL underestimates the cost of query TPCH-1. As discussed, this causes the execution time to spiral (Fig. 7b) due to a nested loop join. This is also the case for query TPCH-2. For both queries, our estimates are very close to the actual values. Queries TPCH-3 and TPCH-4 are examples where our estimates resulted in overestimation. Here, a negative correlation causes a join to produce zero tuples. Using the default estimates, the optimizer misses the opportunity to place the join first. The plan chosen using the default estimates was worse by a factor of 1.5 for these two queries. In these queries, although the default estimates are more accurate for their resulting plan, an overestimation by our method can guide the optimizer to a completely different plan than the default estimates.

In the IMDB data set, correlations are present, but they are not as extreme as in the synthetic data set. Queries IMDB-1–IMDB-4 are all examples of underestimation by the PostgreSQL default selectivity estimates. The resulting differences in execution time can vary from very small (IMDB-2) to more than a factor of 2 (IMDB-1, IMDB-3, IMDB-4).

The time needed for selectivity estimation is significantly higher than that of PostgreSQL (Fig. 7c). This is expected, given the complexity of performing propagation in a junction tree compared to simply checking an 1-dimensional histogram. However, due to our optimizations, the selectivity estimation time is on the order of tens of milliseconds. The

[3] See supplementary material associated with the online version of this article on the journal's web site.

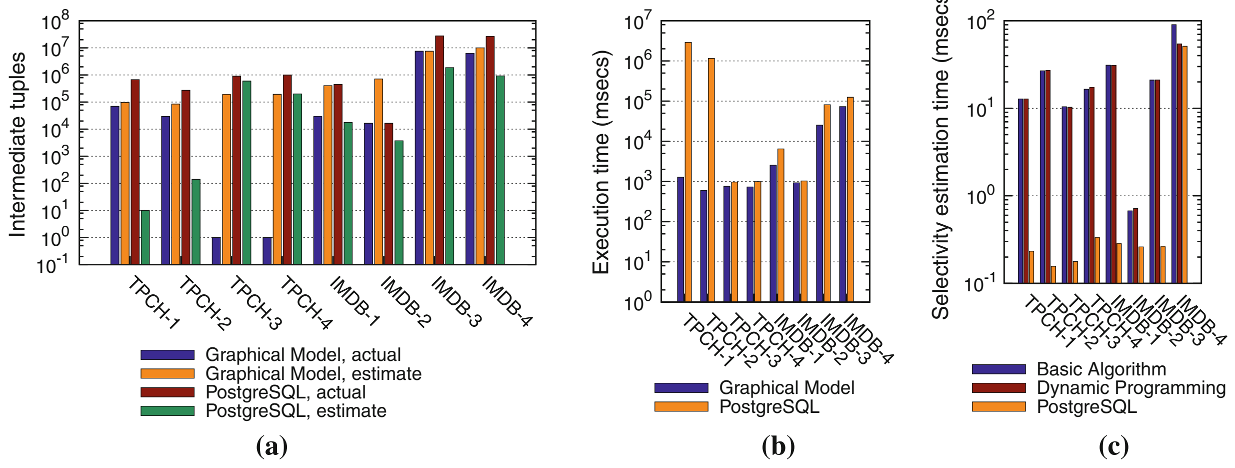


Fig. 7 2-join queries on the TPC-H and IMDB data sets. In all queries, PostgreSQL chose different plans using our versus the default estimates. **a** Optimizer cost, as number of intermediate tuples generated. Estimated and actual costs are shown. **b** Execution time. **c** Time for selectivity estimation

IMDB-4 query is a 3-join chain query. There, the dynamic programming algorithm can reduce selectivity estimation time to about half, and it approaches the time needed by PostgreSQL. The rest of the queries are 2-join queries, and the time needed by the basic selectivity estimation algorithm and the dynamic programming algorithm are similar.

9.3 Measuring estimate accuracy

Although what matters in practice is the execution time of the resulting plan (as shown in the previous section), using this as the sole metric for selectivity estimation can be misleading. Due to the complexity of query optimizers and the fine points of each individual optimizer, a better estimate does not always result in a better query plan. For example, a large overestimation may be better than a slight underestimation for certain queries, since it may result in a more "conservative" plan. In order to place equal emphasis on overestimation and underestimation, we use the multiplicative error metric, which is the most appropriate for query optimization [11,30]. Assume that a relation Q produced during query execution has cardinality $|r|$ and that the cardinality estimate is $|\hat{r}|$. Then, the multiplicative error is defined as $\max(|\hat{r}|, |r|)/\min(|\hat{r}|, |r|)$.

Given a query Q, we define the *average multiplicative error* as the geometric mean of the multiplicative errors for all estimates that are provided to the query optimizer during the optimization of the query:

$$avg\text{-}err(Q) = \left(\prod_{i=1,\dots,n} \frac{\max(|\hat{q}_i|, |q_i|)}{\min(|\hat{q}_i|, |q_i|)} \right)^{1/n},$$

where q_i is a relation whose cardinality estimate was requested by the optimizer.

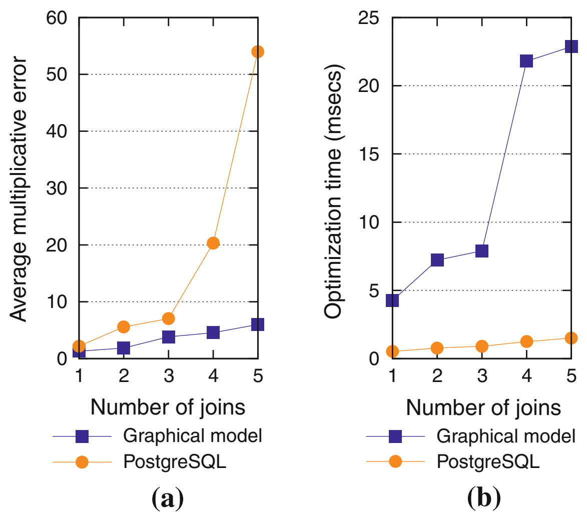


Fig. 8 Results on a workload of 400 queries on the TPC-H data set. **a** Average multiplicative error. **b** Average optimization time.

We generated a workload of 400 queries on a cyclic subset of the TPC-H data set. Figure 8a shows the average multiplicative error of the default PostgreSQL and our selectivity estimates, averaged over queries with the same number of joins. Our estimates are better across all queries; and for 5-join queries, we can achieve a tenfold reduction of the multiplicative error. Figure 8b shows the optimization time. The basic selectivity estimation algorithm is used. The penalty for our better estimates is an increase in optimization time. However, optimization time is always on the order of tens of milliseconds, which is an acceptable overhead considering the reduced estimation errors. Note that the optimization time does not depend linearly on the number of joins. Rather, optimization time depends on the number of cliques in the Steiner tree, which in turn depends on the position of

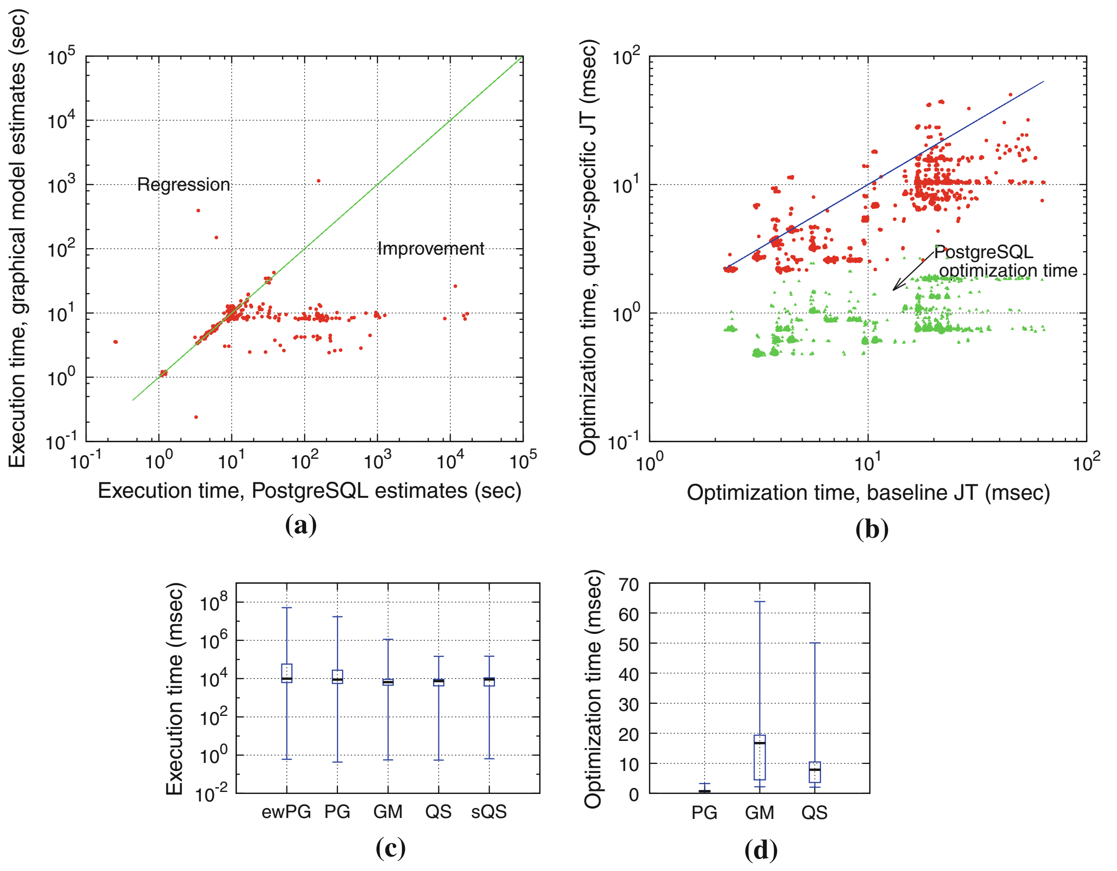


Fig. 9 Results on Accidents database. **a** and **b** Scatter plots of execution and optimization times. **c** and **d** Box plot of execution and optimization times for modified PostgreSQL with equi-width histograms (ewPG), PostgreSQL (PG), Graphical Model (GM), Query-specific graphical model (QS), and Query-specific graphical model created using samples (sQS). A workload of 400 queries was used when running queries against vanilla PostgreSQL to measure execution time [(**a**), (**c**)], otherwise a workload of 4,000 queries was used

the query variables in the junction tree. Our query-specific modeling technique solves this problem for acyclic queries, by selecting a query-specific junction tree with the minimal number of cliques.

9.4 Results on Accidents database

We measured the savings in terms of execution time and optimization time overhead using the Accidents database. We use a large query workload in order to explore the robustness of our solution.

Figure 9a is a scatter plot that compares the execution times of about 400 queries when optimized using the default PostgreSQL and the graphical model selectivity estimates (note that both axes are in logarithmic scale). The queries contain 1–5 joins, random selections, and group by clauses. All queries are issued against a PostgreSQL instance in cold state. For certain queries, different estimates may lead to different plans, which can greatly affect the execution time. For other queries, better estimates do not lead to different plans. As discussed above, due to the complexity of the optimizer, the quality of estimates has varying effects on execution time; better estimates may lead to a worse plan. Therefore, we anticipate cases of regression. We did not create any indexes in the database; therefore, the only reasons for execution time difference are a different join order, a different choice of inner and outer relation, and a different choice of join algorithm, usually either hash or sort-merge join.

In Fig. 9a, we observe improvement in execution time for almost all queries. The improvement can vary from being minimal to 4 orders of magnitude. Figure 9c is a box plot of the execution time of the workload as a whole. As shown, the median time is slightly reduced, and the third quartile is reduced by a factor of 3 compared to vanilla PostgreSQL. The difference between the medians of vanilla PostgreSQL and the version with modified equi-width histograms is one second. We conclude that the improved estimates due to the use of the graphical model can make a query optimizer more robust. Instead of nearly five orders of magnitude variation (the x axis of Fig. 9a) for the execution time of the queries, we obtain a two orders of magnitude variation for most queries. Finally, there is a substantial bottom-line improvement.

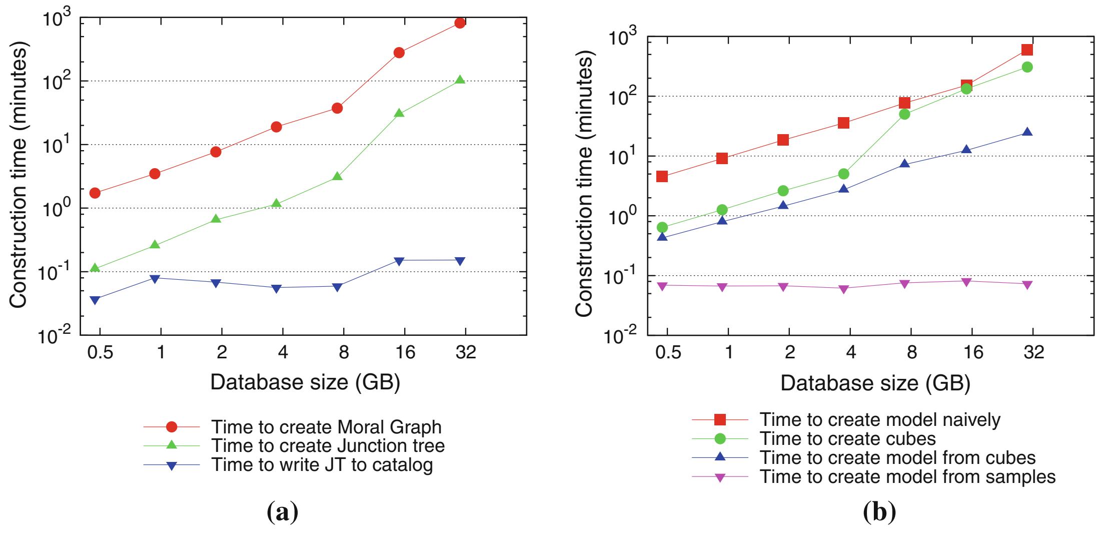


Fig. 10 Model construction results. **a** Scalability of model construction algorithm. **b** Optimization using cubes and samples

For the workload as a whole, we observed a 20× speedup in execution time compared to vanilla PostgreSQL.

We also investigated the effect of sampling on the execution time. Instead of using the full database to create the junction tree marginals, we used samples of 10,000 tuples over the base tables and over the 2-relation joins defined by the join indicators. The sample size used adequately captures the accuracy needed, and at the same time keeps the model construction time low. It takes about 4 s to construct the model using the sampled data (see Fig. 10b). Figure 9c shows the execution time using the estimates built with samples and the query-specific model construction method that provides better plan quality ("sQS" in the figure). We observed a slight increase in the median and the third quartile (by 1 s each). As expected, a small sample of the database is usually enough to capture distributions similar to the ones obtained using the complete data set. Prior work offers techniques to calculate good sample sizes [6]. We note that PostgreSQL uses a reservoir sampling scheme internally, but does not expose a sampling interface to the SQL layer; therefore, we create our samples with queries of the form

```
select * from R order by random() limit 10000,
```

which need a full scan and sort of the relations. To create samples of two-table joins, we sample after fully executing the join, as joining samples provides unacceptable results in practice [2]. We note that more advanced techniques can be used [7].

Figure 9b is a scatter plot of the optimization time needed by the baseline propagation method, and the improved query-specific modeling from Sect. 7. We used a large workload of 4,000 queries that is a superset of the workload used for execution time experiments. Note again the logarithmic scale on both axes. The figure also shows the optimization time needed by vanilla PostgreSQL, measured on the y axis. These times are in the range from 0.5 to 2.5 ms. Optimization with graphical models cannot match such a low overhead, which is due to the complexity of performing multiplications on two-dimensional histograms. The overhead is one order of magnitude higher, usually from 2.5 to 50 ms. However, this overhead is modest, and it pays off in terms of execution time savings. The query-specific modeling method gives an overall speedup of 1.7× for the workload as a whole. Perhaps more importantly, it reduces the standard deviation of the optimization time by a factor of 2. This is expected; query-specific modeling results in smaller junction trees and therefore fewer clique multiplications. The size of the junction tree is no longer random, but depends on the number of predicates (selections and joins) in the query. We expect the speedup to be higher for databases that have more relations and attributes, and that then have larger junction trees if query-specific modeling is not used. As shown in the box plot of Fig. 9d, both the median value and the third quartile are reduced by almost a factor of 2 using query-specific modeling.

9.5 Model construction scalability

We study the scalability and efficiency of our model construction algorithm (Algorithm 1). Recall that the construction algorithm first identifies the best moral graph of the database, then creates a junction tree from the moral graph, and finally writes the junction tree to the PostgreSQL catalog. Figure 10a shows the time needed for these three steps when the database size varies from 500 MB to 32 GB (note the logarithmic scale on both axes). All experiments were conducted on the Accidents database. The total time needed is the sum of the three phases. The time needed to construct the moral graph always accounts for more than 90 % of the total time. This is expected; constructing the moral graph includes querying the database for the probability distributions of all pairs of attributes and join indicators using the SQL queries described in Sect. 4.

The construction time scales well with the size of the database. To understand the behavior better, consider that the bulk of time needed to construct the model is spent on evaluating queries of the form

```
select R.X, R.Y, count(*) from R
group by R.X, R.Y
```

for computing $P(X, Y)$, and queries of the form

```
select R.X, S.Y, count(*) from R, S
where R.A = S.B group by R.X, S.Y
```

for computing $P(X, Y, J_{RS})$. The PostgreSQL optimizer chose hash joins and aggregates for these queries in our setting.

Figure 10a indicates that the overhead of constructing the model is quite high (about 15 h for a 32 GB database). We briefly comment on techniques that can lower this overhead.

First, samples can be used to create distributions of similar accuracy. For our approach, we need to sample over single tables and two-table joins. In fact, our model construction algorithm issues exactly the same queries to the database as CORDS [18], where it was shown that a small sample size can give excellent accuracy, rendering the model construction time independent of the size of the database. We obtained empirical evidence that supports this finding. Using samples of 10,000 tuples, we observed similar execution times (Fig. 9c), with the time needed to build the model from the samples being about 4 s (Fig. 10b).

Second, if the samples are still large enough to cause performance concerns, we note that the workload resulting from the model construction algorithm can benefit from fast scans, columnar storage, parallel processing, and the techniques of Bravo et al. [4].

Another possible optimization is to pre-compute or leverage existing data cubes for each relation and for each two-relation join for which a join indicator exists, and then answer the queries using the data cubes. Experimental results of using this technique are shown in Fig. 10b. We extracted the query workload resulting from our model construction and issued it to PostgreSQL. This corresponds to the naive model construction line in the figure. Then, we rewrote these queries to be on pre-computed data cubes for each relation, and for each two-relation join that corresponds to a join indicator. While the creation of the data cubes has the same overhead as the naive construction of the model for large databases, the availability of pre-computed cubes can reduce the model construction overhead by an order of magnitude. For the 32 GB database, this reduces the overhead for constructing the model from 15 h to 25 min.

The space needed to store the model is on the order of 64 kilobytes, and 128 kilobytes when query-specific modeling is used. A final observation from Fig. 10a is that the time to construct the junction tree is much higher than the time to write it to the catalog. This means that the query-specific modeling algorithm also offers lower model construction overhead; it skips the step of constructing the junction tree, while it needs to write more cliques to the catalog.

9.6 Summary of experimental results

We conclude this section with a summary of our experimental findings and offer guidelines to implementors. We have found that our proposed fixed model structure can greatly improve the estimates and the running times of a wide class of queries with very few observed regressions. We suggest the query-specific modeling as the model construction method of choice because it provides a direct mapping from query size to query optimization overhead for acyclic queries, making the optimization overhead predictable.

For selectivity estimation, while the dynamic programming algorithm can greatly reduce the number of clique multiplications, its overhead depends on the shape of the Steiner tree, and it can become unacceptable for high treewidths. Thus, we recommend the propagation algorithm for all but queries that result in chain Steiner trees. This can be easily decided at compile time. We note that this recommendation is dependent on the types of histograms used to store the junction tree cliques. If more compact histograms are used, the dynamic programming algorithm may become beneficial for a wider variety of cases. This tradeoff is a promising topic that we plan to address in future work.

For model construction, we have found that relatively small samples over base relations and 2-relation joins can provide high quality estimates, and capture most correlations. If larger samples are used, pre-computing cubes can drive the model construction time down.

10 Related work

A recent survey [16] offers a comprehensive coverage of related work on discovering and exploiting statistical properties for query optimization. It classifies proposed methods into two categories: proactive methods that use the database to create a model and reactive methods that rely on query feedback. Our work falls into the first category but is also classified as workload-aware since it makes limited use of a workload in order to define the random variables of the model.

To relax the attribute value independence assumption and to model data correlations, one cannot avoid the need to approximate multi-dimensional probability distributions.

Multi-dimensional histograms attempt to directly approximate a distribution using a fixed number of buckets. While practical one-dimensional histograms can be built in polynomial time [24], the same problem is NP-hard even for two dimensions [32]. Therefore, research has focused on providing good heuristics. Early solutions focus on equi-depth [31] and recursive hierarchical partitioning [33], and later works allowed bucket overlap [15] and "holes" [5], and exploited query feedback [1,5]. All of these methods are only applicable to few dimensions. Therefore, they cannot be used to model the distribution of a real-world database. They can be used to approximate the model factors after a decomposition has been achieved.

Dependency-based histograms [11] use graphical models to decompose the distribution of all attributes within a relation. Reference [11] also includes implementations of factor multiplication and marginalization when factors are implemented as hierarchically decomposed histograms [33].

We build on Probabilistic Relational Models (PRMs) [13,14], the first work that approximates the distribution of a database as a whole including join indicators, by using Bayesian networks (BNs). PRMs are limited to tree-shaped query graphs and key-foreign key join relationships that induce a topological order in all relations. Further, they allow arbitrary BNs, which can lead to high-dimensional distributions, that can cause performance to degrade significantly.

CORDS [18] provides a cost-effective schema-level synopsis that makes the uniform distribution assumption, but not the attribute and join independence assumptions. CORDS discovers correlated attributes, but does not create the corresponding probability distributions. Further, it is limited to pairs of attributes. Similarly to CORDS, we also consider only pairs of attributes in order to create the model efficiently. However, we avoid all three assumptions at once. Further, by organizing the dependencies in a junction tree, we can model indirect dependencies between attributes via chains or trees of dependencies. Our model construction algorithm has roughly the same complexity as CORDS.

Our work can be seen as a bridge between CORDS and PRMs. Like PRMs, we do not make the attribute value independence or the uniformity assumptions, and we are able to estimate selectivities of arbitrary conjuncts of predicates and joins. Unlike PRMs, we are not restricted to a stratified schema graph [14]. Like CORDS, we only consider pairwise correlations to limit the overheads of selectivity estimation and model construction.

A different approach to the problem is to estimate cardinalities from samples. Join synopses [2] is the first work on sampling over joins. Like PRMs, they are limited to an acyclic schema graph of key-foreign key joins that defines a topological order on the relations. We overcome this problem through our fixed model structure that needs only two-table joins. Graph-based synopses [36] choose a subset of the complete tuple graph of a database and use that subset to perform selectivity estimation. That approach encodes independence assumptions in the heuristic model construction algorithm. In contrast, our approach makes the (conditional) independence assumptions explicit by means of a graphical model. Although the approaches have different origins (graphical models and XML graph summarization), it would be interesting to experimentally compare their efficiency in the future.

Cardinality estimation via maximum entropy [27,34] addresses the problem of finding the best way to estimate the selectivity of a conjunct of predicates given that many equivalent ways of computing this selectivity are possible. Maximum entropy considers the wider space log-linear models that subsume graphical models [11]. On the other hand, many estimates need an expensive application of an iterative scaling algorithm, which can be avoided using our model. Finally, these works do not consider the problem of selecting the underlying summaries.

11 Conclusions and future work

Missed statistical correlations is one of the most frequent sources of estimation errors in modern optimizers, and they very often lead to severely sub-optimal plan selection. We provide a principled approach to selectivity estimation that does not make the attribute value independence assumption. Our approach is theoretically founded on graphical models. To balance expressive power with low query optimization overhead, we restrict the space of possible models to an interesting class of models that in most cases require only two-dimensional histograms.

A very important piece of our work is integrating graphical models for selectivity estimation in a real DBMS kernel. We provide two techniques for using a junction tree to perform selectivity estimation, and we provide a novel technique that defers junction tree construction until a query is optimized, which renders the size of the junction tree both minimal and predictable.

We report on extensive experiments with our prototype, showcasing the scalability, efficiency, and robustness of our approach. A model that belongs to the proposed reduced class of models can be constructed efficiently, since the construction algorithm issues only one-way joins. Use of the new selectivity estimates enables reductions of the multiplicative estimation error of up to a factor of 10. The reduced errors lead to more efficient plans and increased robustness of the query optimizer. Optimization time is kept low, in the order of tens of milliseconds, often an acceptable overhead for modern systems.

One direction for future work is to enable incremental and efficient model updates when the underlying data changes. We can possibly exploit techniques from the graphical model literature [12]. A more challenging direction is to provide error guarantees. Recently, a connection between multiplicative error bounds and plan optimality has been established [30], and one-dimensional histograms that provide these bounds have been proposed [26]. Designing multi-dimensional histograms that can offer similar guarantees, and can be efficiently combined with each other using our techniques is a promising direction for increasing the robustness of query optimizers.

Acknowledgments We thank the anonymous reviewers for comments that greatly improved the content and presentation of this paper.

References

1. Aboulnaga, A., Chaudhuri, S.: Self-tuning histograms: Building histograms without looking at data. In: SIGMOD, pp. 181–192 (1999)
2. Acharya, S., Gibbons, P.B., Poosala, V., Ramaswamy, S.: Join synopses for approximate query answering. In: SIGMOD, pp. 275–286 (1999)
3. Berry, A., Blair, J.R.S., Heggernes, P., Peyton, B.W.: Maximum cardinality search for computing minimal triangulations of graphs. Algorithmica **39**(4), 287–298 (2004)
4. Bravo, H.C., Ramakrishnan, R.: Optimizing MPF queries: Decision support and probabilistic inference. In: SIGMOD, pp. 701–712 (2007)
5. Bruno, N., Chaudhuri, S., Gravano, L.: STHoles: A multidimensional workload-aware histogram. In: SIGMOD, pp. 211–222 (2001)
6. Chaudhuri, S., Motwani, R., Narasayya, V.R.: Random sampling for histogram construction: How much is enough? In: SIGMOD, pp. 436–447 (1998)
7. Chaudhuri, S., Motwani, R., Narasayya, V.R.: On random sampling over joins. In: SIGMOD, pp. 263–274 (1999)
8. Chaudhuri, S., Shim, K.: Optimization of queries with user-defined predicates. ACM Trans. Database Syst. **24**(2), 177–228 (1999)
9. Chow, C.K., Liu, C.N.: Approximating discrete probability distributions with dependence trees. IEEE Trans. Inf. Theory **14**(3), 462–V467 (1968)
10. Cowell, R.G., Dawid, P., Lauritzen, S.L., Spiegelhalter, D.J.: Probabilistic Networks and Expert Systems: Exact Computational Methods for Bayesian Networks. Springer, Berlin (1999)
11. Deshpande, A., Garofalakis, M.N., Rastogi, R.: Independence is good: dependency-based histogram synopses for high-dimensional data. In: SIGMOD, pp. 199–210 (2001)
12. Friedman, N., Goldszmidt, M.: Sequential update of bayesian network structure. In: UAI, pp. 165–174 (1997)
13. Getoor, L.: Learning statistical models from relational data. Ph.D. thesis, Stanford University (2001)
14. Getoor, L., Taskar, B., Koller, D.: Selectivity estimation using probabilistic models. In: SIGMOD, pp. 461–472 (2001)
15. Gunopulos, D., Kollios, G., Tsotras, V.J., Domeniconi, C.: Approximating multi-dimensional aggregate range queries over real attributes. In: SIGMOD, pp. 463–474 (2000)
16. Haas, P.J., Ilyas, I.F., Lohman, G.M., Markl, V.: Discovering and exploiting statistical properties for query optimization in relational databases: a survey. Stat. Anal. Data Min. **1**(4), 223–250 (2009)
17. Hellerstein, J.M.: Optimization techniques for queries with expensive methods. ACM Trans. Database Syst. **23**(2), 113–157 (1998)
18. Ilyas, I.F., Markl, V., Haas, P.J., Brown, P., Aboulnaga, A.: CORDS: Automatic discovery of correlations and soft functional dependencies. In: SIGMOD, pp. 647–658 (2004)
19. Ioannidis, Y.E.: Universality of serial histograms. In: VLDB, pp. 256–267 (1993)
20. Ioannidis, Y.E.: The history of histograms (abridged). In: VLDB, pp. 19–30 (2003)
21. Ioannidis, Y.E., Christodoulakis, S.: On the propagation of errors in the size of join results. In: SIGMOD, pp. 268–277 (1991)
22. Ioannidis, Y.E., Christodoulakis, S.: Optimal histograms for limiting worst-case error propagation in the size of join results. ACM Trans. Database Syst. **18**(4), 709–748 (1993)
23. Ioannidis, Y.E., Poosala, V.: Balancing histogram optimality and practicality for query result size estimation. In: SIGMOD, pp. 233–244 (1995)
24. Jagadish, H.V., Koudas, N., Muthukrishnan, S., Poosala, V., Sevcik, K.C., Suel, T.: Optimal histograms with quality guarantees. In: VLDB, pp. 275–286 (1998)
25. Jensen, F.V., Jensen, F.: Optimal junction trees. In: UAI, pp. 360–366 (1994)
26. Kanne, C.C., Moerkotte, G.: Histograms reloaded: The merits of bucket diversity. In: SIGMOD, pp. 663–674 (2010)
27. Markl, V., Haas, P.J., Kutsch, M., Megiddo, N., Srivastava, U., Tran, T.M.: Consistent selectivity estimation via maximum entropy. VLDB J. **16**(1), 55–76 (2007)
28. Markl, V., Raman, V., Simmen, D.E., Lohman, G.M., Pirahesh, H.: Robust query processing through progressive optimization. In: SIGMOD, pp. 659–670 (2004)
29. Matias, Y., Vitter, J.S., Wang, M.: Wavelet-based histograms for selectivity estimation. In: SIGMOD, pp. 448–459 (1998)
30. Moerkotte, G., Neumann, T., Steidl, G.: Preventing bad plans by bounding the impact of cardinality estimation errors. PVLDB **2**(1), 982–993 (2009)
31. Muralikrishna, M., DeWitt, D.J.: Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In: SIGMOD, pp. 28–36 (1988)
32. Muthukrishnan, S., Poosala, V., Suel, T.: On rectangular partitionings in two dimensions: algorithms, complexity, and applications. In: ICDT, pp. 236–256 (1999)
33. Poosala, V., Ioannidis, Y.E.: Selectivity estimation without the attribute value independence assumption. In: VLDB, pp. 486–495 (1997)
34. Ré, C., Suciu, D.: Understanding cardinality estimation using entropy maximization. In: PODS, pp. 53–64 (2010)
35. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: SIGMOD, pp. 23–34 (1979)
36. Spiegel, J., Polyzotis, N.: Graph-based synopses for relational selectivity estimation. In: SIGMOD, pp. 205–216 (2006)
37. Stillger, M., Lohman, G.M., Markl, V., Kandil, M.: Leo - db2's learning optimizer. In: VLDB, pp. 19–28 (2001)
38. Tzoumas, K., Deshpande, A., Jensen, C.S.: Lightweight graphical models for selectivity estimation without independence assumptions. PVLDB **4**(11), 852–863 (2011)