SPROUT: Lazy vs. Eager Query Plans for Tuple-Independent Probabilistic Databases

Dan Olteanu[1], Jiewen Huang[1], and Christoph Koch[2]

[1]*Oxford University Computing Laboratory, Oxford, OX1 3QD, UK*
[2]*Department of Computer Science, Cornell University, Ithaca, NY 14853, USA*

***Abstract*— A paramount challenge in probabilistic databases is the scalable computation of confidences of tuples in query results. This paper introduces an efficient secondary-storage operator for exact computation of queries on tuple-independent probabilistic databases. We consider the conjunctive queries without self-joins that are known to be tractable on any tuple-independent database, and queries that are not tractable in general but become tractable on probabilistic databases restricted by functional dependencies.**

Our operator is semantically equivalent to a sequence of aggregations and can be naturally integrated into existing relational query plans. As a proof of concept, we developed an extension of the PostgreSQL 8.3.3 query engine called SPROUT. We study optimizations that push or pull our operator or parts thereof past joins. The operator employs static information, such as the query structure and functional dependencies, to decide which constituent aggregations can be evaluated together in one scan and how many scans are needed for the overall confidence computation task. A case study on the TPC-H benchmark reveals that most TPC-H queries obtained by removing aggregations can be evaluated efficiently using our operator. Experimental evaluation on probabilistic TPC-H data shows substantial efficiency improvements when compared to the state of the art.

I. Introduction

Applications in data cleaning, data integration, and scientific databases call for systems for managing probabilistic data [3], [2], [5], [14], [1], [10], [15], [9], [16].

In this paper we study the following problem: Given a query Q and a probabilistic database D, compute the distinct possible tuples in the result of Q on D together with their exact confidences. Conceptually, a probabilistic database represents a set of possible worlds, each containing a relational database, and the query is evaluated in each world individually. The confidence in a tuple t is the sum of the probabilities of all those possible worlds in which t is part of the query result.

Dalvi and Suciu's work on the evaluation of conjunctive queries on tuple-independent probabilistic databases [7] shows that the class of conjunctive queries can be partitioned into "easy" queries (with polynomial-time data complexity) and "hard" (#P-complete) queries. Their method of processing the easy queries without self-joins uses so-called *safe plans* to compute both the answer tuples and their confidences [5]. Safe plans are a restricted class of query plans in that their join orders adhere to the hierarchical structure of the query and employ excessive duplicate elimination in temporary results.

This paper shows that the restrictions imposed by safe plans are not necessary and any query plan can be used to compute the answer tuples. For computing the confidences, we define a new operator that can blend in any relational query plan and is governed by optimizations specific to sequences of standard aggregations. As we will see later, the safe plans correspond to particular *eager* plans in our framework, where confidence computation is performed after each join and projection. We however enable the search space for query plans beyond the eager ones and allow, for instance, *lazy* plans that compute confidences only at the very end in the plan. The decision on which plans to use solely depends on the optimizer. As a proof of concept, we developed SPROUT (Scalable PROcessing of Uncertain Tables), an extension of the PostgreSQL 8.3.3 query engine with our new operator for confidence computation (`http://www.comlab.ox.ac.uk/projects/SPROUT/`).

We illustrate our operator using a tuple-independent probabilistic version of TPC-H, where we associate each tuple with a distinct Boolean random variable and accept any probability distribution over these variables. Fig. 1 gives a TPC-H-like database, where the random variables associated with tuples are given in the V-columns and their probabilities (for the "true" assignment) in the P-columns. Let Q be a query asking for the dates of discounted orders shipped to customer 'Joe',

$$\pi_{odate}(\sigma_{cname='Joe',discount>0}(\text{Cust} \bowtie_{ckey} \text{Ord} \bowtie_{okey,ckey} \text{Item})).$$

The answer to a query on a probabilistic database can be represented by a relation pairing possible result tuples with propositional formulas over the random variables. For a given tuple, its formula captures the set of worlds in which that tuple occurs in the query result. For conjunctive queries, it is known that these formulas, in the form of a DNF, can be efficiently computed (just like the overall representation). This technique is directly employed or at least implicit in a number of approaches [5], [3], [1].

In our example, the answer consists of one distinct tuple produced by combining the first tuple of Cust and the first tuple of Ord with either the first or the second tuple of Item (Fig. 1). These combinations lead to the formula $x_1y_1z_1 \vee x_1y_1z_2$, which admits the simpler factored form $x_1y_1(z_1 \vee z_2)$. We call this factored form one-occurrence form (1OF), because each input variable occurs at most once. We can efficiently compute the probability of 1OF formulas by mapping AND into product and OR into probability computation of independent events. For any tuple-independent probabilistic database, the formulas associated with distinct answer tuples of any easy query without self-joins can be brought into 1OF [11].

Cust

ckey	cname	V	P
1	Joe	x_1	0.1
2	Dan	x_2	0.2
3	Li	x_3	0.3
4	Mo	x_4	0.4

Ord

okey	ckey	odate	V	P
1	1	1995-01-10	y_1	0.1
2	1	1996-01-09	y_2	0.2
3	2	1994-11-11	y_3	0.3
4	2	1993-01-08	y_4	0.4
5	3	1995-08-15	y_5	0.5
6	3	1996-12-25	y_6	0.6

Item

okey	discount	ckey	V	P
1	0.1	1	z_1	0.1
1	0.2	1	z_2	0.2
3	0.4	2	z_3	0.3
3	0.1	2	z_4	0.4
4	0.4	2	z_5	0.5
5	0.1	3	z_6	0.6

Q

odate	V_c	V_o	V_i
1995-01-10	x_1	y_1	z_1
1995-01-10	x_1	y_1	z_2

Fig. 1. Tuple-independent probabilistic TPC-H-like database and answer to query Q (P_c, P_o, P_i omitted).

Given a relational encoding of such DNF formulas, our operator needs a few scans to turn them into 1OF and simultaneously compute their probabilities. It essentially makes use of the structure of the query which coincides with the nesting structure of the 1OF formula. Such structures are called *query signatures* throughout the paper. The query signature in our example is $(\text{Cust}^*(\text{Ord}^*\text{Item}^*)^*)^*$ and states the many-to-many relationships between Ord and Item, and between Ord $\bowtie$ Item and Cust. These relationships are directly derived from the join conditions of the query. They hold between the tuples of the input tables and thus between their associated variables.

Using this query signature, we first sort the answer tuples by (odate,V_c,V_o,V_i) and then use one scan to compute the probabilities of the pairs OL=(Ord,Item) and subsequently one scan for the pairs (Cust,OL).

The query Q can also be evaluated using safe plans [5]. Similarly to query signatures, safe plans capture the structure of the queries. Fig. 2 gives the safe plan for Q as produced by MystiQ [5]. The standard operators are extended to compute both the answer tuples and their probabilities. A join of two tuples also multiplies their probabilities. The projection requires that all duplicate tuples are pairwise independent in order to compute the probability of distinct tuples. This constraint can be guaranteed for easy queries (without self-joins) by careful yet restrictive join orderings, which may lead to suboptimal query plans. For instance, the safe plan for Q requires an unselective join of the two largest tables Ord and Item first, and also five rather expensive projections. The projection done after the aforementioned unselective join, which is needed for duplicate removal, accounts in general for most of the processing time. If Item would be joined first with the few selective tuples of Cust, then we would obtain two non-independent duplicate answer tuples (as shown in Fig. 1) and the plan fails. In contrast, SPROUT chooses a better (lazy) query plan, which first joins Cust with Ord, then with Item, and computes the tuple confidences at the end. We experimentally observed that for queries similar to Q, lazy plans perform orders of magnitude faster that the safe plans.

Consider now a slightly changed scenario where the table Item has no ckey attribute (as it is the case in real TPC-H), and the query Q loses the join condition $o.ckey = l.ckey$. This new query Q' has now the pattern of the prototypical hard query: Table Ord joins with two tables Cust and Item on different attributes ckey and okey and in general Ord can arbitrarily pair tuples from Cust and Item. If both attributes ckey and okey of Ord would be involved in a join with one

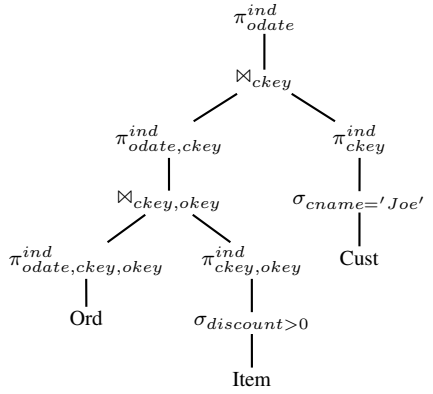


Fig. 2. Safe plan for query Q.

of the two tables, then the query would become easy. More precisely, the necessary and sufficient condition for a query (without self-joins) to be easy is that for any two join attributes that occur in the same table and are not in the outermost projection list, one of them must participate in all joins of the other. Because of this property, such queries are also called *hierarchical* [6]. We can check that Q is hierarchical: ckey participates in both joins, whereas okey participates only in one join. In contrast, Q' is non-hierarchical, because each of the two join attributes of Ord participates in a different join.

The non-hierarchical query Q' can be, however, computed efficiently on restricted databases, such as databases where the functional dependency (FD) $okey \rightarrow ckey$ holds. This restriction is in fact natural and already holds on TPC-H data, because okey is a key in Ord. Moreover, under this FD, the two queries Q and Q' have the same answer.

Functional dependencies are very effective in taming non-hierarchical queries and even in improving the efficiency of hierarchical queries. We develop a new method that rewrites (possibly non-hierarchical) queries into Boolean hierarchical queries under given FDs. The operator for confidence computation can then use the signature of the latter to efficiently process the answer of the former. Our experiments on TPC-H data show that the existing TPC-H FDs enable efficient evaluation of five (out of 9) non-hierarchical queries, and reduce the number of necessary scans for confidence computation in case of hierarchical queries. For instance, if okey and ckey are keys in Ord and Cust, respectively, then the query Q' has the signature $(\text{Cust}(\text{Ord Item}^*)^*)^*$ and the distinct tuples and their confidences can be computed in only one scan after sorting the answer tuples.

The main contributions of this paper are as follows.

- We introduce a new approach to exact confidence computation for conjunctive queries without self-joins on tuple-independent probabilistic databases. This approach is based on a close connection between query signatures and a particular factored form (1OF) of formulas over input random variables that are associated with distinct answer tuples. The probability of 1OF formulas can be computed in time linear in the number of their variables.
- We extend the frontier of queries computable in polynomial time by exploiting semantic knowledge about the databases in the form of functional dependencies. Such schema information is natural and ubiquituous in real database scenarios. To accommodate functional dependencies in a principled way, we define a method for rewriting queries into Boolean hierarchical ones. The signature of the latter can then be used to process the former. As evidenced by the set of TPC-H queries, or their conjunctive subqueries, this rewriting technique substantially extends the class of practical queries for which efficient exact evaluation techniques are known.
- We extend the query engine of PostgreSQL 8.3.3 with a new low-level query operator for probability computation. This operator turns DNF formulas into 1OF and computes their probabilities on the fly.
- We experimentally show the efficiency of our query engine on 18 out of the 22 TPC-H queries. (The remaining queries remain outside the thus extended class of tractable queries.) In particular, our plans can outperform the safe plans of MystiQ by up to two orders of magnitude.

The paper is structured as follows. We first define tuple-independent probabilistic databases, hierarchical queries, and query signatures. Then, we discuss query rewritings and how to compute good signatures given functional dependencies. The semantics, optimizations, and evaluation of our secondary storage operator for confidence computation are then introduced, followed by a discussion on the structure of the TPC-H benchmark queries in the light of our query processing techniques, and by our experimental findings.

II. Preliminaries

A. Tuple-independent Probabilistic Databases

Let $\mathbf{X}$ be a finite set of (independent) Boolean random variables. *A tuple-independent probabilistic table* R^{rep} is a relation of schema $(\overline{A}, V, P)$ with functional dependency $\overline{A} \rightarrow VP$, and two distinguished columns V and P such that the values in V are from $\mathbf{X}$ and the values in P are numbers in $(0, 1]$. A probabilistic database D is a set of probabilistic tables and represents a set of possible worlds. Each possible world is identified by a truth assignment of all variables from $\mathbf{X}$. There is a one-to-one correspondence between possible worlds and database instances. To obtain one instance, fix a truth assignment f. Under f, the instance of each probabilistic table R^{rep} is the set of tuples $\vec{a}$ such that $(\vec{a}, x, p) \in R^{rep}$ and $f(x)$ is true. The probability of that world is

$$\Pr[f] = \prod_{(\vec{a},x,p)\in R^{rep}, R^{rep}\in D} \begin{cases} p & \dots \quad f(x) \text{ true} \\ 1-p & \dots \quad f(x) \text{ false} \end{cases}$$

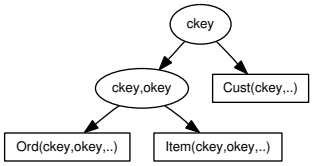


Fig. 3. Tree representation of the hierarchical query Q.

We will use probabilities of Boolean formulas over the variables from $\mathbf{X}$. The probability of a formula ϕ is

$$\Pr[\phi] = \sum_{f: f \text{ implies } \phi} \Pr[f]$$

where f can be thought of as the formula

$$\Big(\bigwedge_{x\in\mathbf{X}: f(x) \text{ true}} x \Big) \wedge \bigwedge_{x\in\mathbf{X}: f(x) \text{ false}} \neg x.$$

Clearly, if $(\vec{a}, x, p) \in R^{rep}$ then the probability that tuple $\vec{a}$ is in R is $\Pr[x] = p$. Summing up the probabilities of possible worlds satisfying such formulas is impractical because there are exponentially many worlds. Computing $\Pr[\phi]$ is #P-complete in general and the essential goal of this paper is to develop and study large classes of queries for which the computation of $\Pr[\phi]$ can be done efficiently for the formulas ϕ constructed during query evaluation.

B. Hierarchical Conjunctive Queries without Self-joins

We consider conjunctive queries without self-joins (i.e., any relation name appears at most once) written in form $\pi_{\overline{A}}\sigma_{\phi}q$, where $\overline{A}$ is the projection list, ϕ is a conjunction of atomic formulas comparing attributes with constants, and q is a join query of the form $R_1 \bowtie \dots \bowtie R_n$. For simplicity, we assume that the join attributes have the same name in the joined tables.

This class of queries can be partitioned into *hierarchical* queries, which admit polynomial-time evaluation on tuple-independent probabilistic databases, and non-hierarchical queries, which are #P-complete in general [5].

Definition II.1. *A Boolean conjunctive query is hierarchical if for any two join attributes that occur in the same table, one of them participates in all joins of the other.*

In case of non-Boolean queries, the attributes that occur in joins and in the projection list $(\overline{A})$ are not used for deciding the hierarchical property. Section IV develops a rewriting framework that can transform non-Boolean (and possibly non-hierarchical) queries into equivalent Boolean hierarchical ones.

Hierarchical queries admit tree representations, where the leaves are tables and the inner nodes are join attributes occurring in all their descendant nodes. In case of a relational product of hierarchical subqueries, the root is the empty set.

signature (query tree q)	= $\tau(q, \emptyset)$
τ (inner node $\bar{A}(q_1, \ldots, q_n)$, L)	= **ite**$(L = \bar{A},$ $\tau(q_1, \bar{A}) \circ \ldots \circ \tau(q_n, \bar{A}))$
τ (leaf node $R(\bar{A})$, L)	= **ite**$(L = \bar{A}, R)$
ite (Cond, t)	= **if** Cond **then** t **else** $(t)^*$

Fig. 4. Deriving query signatures from hierarchical queries.

Example II.2. Fig. 3 gives the tree representation of the Boolean version of the query from the Introduction. The attribute ckey is the root. If we remove $ckey$ from either Ord or Item, we obtain a non-hierarchical query. The safe plan of Fig. 2 has the same overall structure as the tree of Fig. 3. □

C. Query Evaluation on Probabilistic Databases

Conceptually, queries are evaluated in each world individually. Given a query Q and a probabilistic database D, the confidence in an answer tuple t is the probability of t being in the result of Q on the worlds of D, or equivalently,

$$\Pr[t \in Q(D)] = \sum_{f:\, t \text{ in world } f \text{ of } Q(D)} \Pr[f].$$

The evaluation of query operators on a probabilistic database follows the standard semantics [8], where the columns for variables and probabilities are copied along in the answer tuples. These columns store relationally a DNF formula over Boolean random variables: In the answer to our query from the Introduction, the columns for variables store the formula $x_1y_1z_1 \vee x_1y_1z_2$. We denote the expression associated with t by $\phi_{t,Q,D}$ (or $\phi_{t,Q}$ if D is clear from the context). If Q is Boolean, we write ϕ_Q as a shorthand for $\phi_{(true),Q}$. The following result is folklore.

Proposition II.3. *For any query Q, probabilistic database D, and a distinct tuple t in $Q(D)$, $\Pr[t \in Q(D)] = \Pr[\phi_{t,Q,D}]$.*

III. Query Signatures and Factored Normal Form

We capture the tree structure and the one/many-to-one/many relationships between the tables of hierarchical queries in a so-called *query signature* (called variable order type in [11]).

Definition III.1. *Syntactically, a query signature is*

- *a table name R or*
- *of the form α^*, where α is a signature, or*
- *a concatenation $\alpha\beta$ of two signatures α and β.*

The equivalence between signatures $(\alpha^*)^*$ and α^* is trivial and is considered implicit in the remainder of the paper.

The function **signature** of Fig.4 computes the signature of a hierarchical Boolean query using pattern matching on its tree structure. While traversing the tree top-down, we keep in L the join attributes of the parent node (which includes the attributes of its ancestors); initially, $L = \emptyset$. For a table $R(\bar{A})$, we create a signature R or R^*. The former case occurs when $\bar{A}$ represents the parent variables, and thus there is one tuple per distinct $\bar{A}$-value. Otherwise, there may be several tuples per distinct $\bar{A}$-value, and hence the exponent (*). In case of an inner node, we recursively compute the signatures for the children and then concatenate them. Note that $\bar{A} = \emptyset$ covers the case of unconnected hierarchical subqueries.

Example III.2. The signature of our query from the Introduction is $(\text{Cust}^*(\text{Ord}^*\text{Item}^*)^*)^*$, which follows the nesting of its tree representation (see Fig.3) and specifies many-to-many relationships using (*): The tables Ord and Item join on a subset of their attributes and hence in general there is a many-to-many relationship between them, and, by similar arguments, there is a many-to-many relationship between Cust and the join result of Ord and Item. Because in general there may be several pairings of Cust-tuples and (Ord $\bowtie$ Item)-tuples, we add (*) to the previous signature. Of course, in case ckey and okey are keys, then the above signature can be simplified by turning many-to-many into one-to-many relationships, and our signature becomes $(\text{Cust}(\text{Ord Item}^*)^*)^*$. □

Definition III.3. *Let a signature s and a query tree t for a hierarchical query Q, and let a set of table names $T = \{\alpha_1, \ldots, \alpha_n\}$ that occur in Q. A minimal cover of T in s is the signature of the minimal subtree in t that contains all tables of T.*

Example III.4. Let the signature $s = (\text{Cust}^*(\text{Ord}^*\text{Item}^*)^*)^*$ for our query Q. The minimal cover of $\{\text{Ord}, \text{Item}\}$ in s is $(\text{Ord}^*\text{Item}^*)^*$, and of $\{\text{Cust}, \text{Ord}\}$ in s is s itself. □

The signature of a hierarchical query q is particularly useful for turning the formula ϕ_Q into a factored form, called *one-occurrence normal form* (1OF), where each variable of ϕ_Q occurs exactly once.

Proposition III.5 ([11])**.** *A DNF expression ϕ can be turned into 1OF according to a signature*

- $\mathbf{X}$ *if ϕ is in one variable that occurs in the table* $\mathbf{X}$*;*
- α^* *if there exist DNF expressions $\phi_1, \ldots, \phi_n$ that can be factored according to α, partition the clauses of ϕ, and use disjoint sets of variables;*
- $\alpha\beta$ *if there exist DNF expressions ϕ_1 and ϕ_2 that can be factored according to α and β, respectively, use disjoint sets of variables, and $\phi = (\phi_1) \wedge (\phi_2)$.*

Example III.6. The expression $\phi_{Q,D}$ associated with Q and D of Fig.1 can be factored according to signature $(\text{Cust}(\text{Ord Item}^*)^*)^*$: $x_1y_1z_1 \vee x_1y_1z_2 = x_1(y_1(z_1 \vee z_2))$, where x_1 is a variable of Cust, y_1 is a variable of Ord, and z_1 and z_2 are variables of Item. The signature states that one tuple (and thus variable) in Cust is paired with several pairs of one tuple in Ord and several tuples in Item. For our database D, we have the special case of one Cust-tuple paired with on pair of one Ord-tuple and two Item-tuples. □

Proposition III.7 ([11])**.** *Given a hierarchical Boolean query Q and a probabilistic database D, the expression $\phi_{Q,D}$ can be factored into 1OF according to the signature of Q in PTIME.*

We later present in this paper an efficient secondary-storage algorithm for bringing $\phi_{Q,D}$ into 1OF.

IV. Rewriting under Functional Dependencies

An important property of tuple-independent probabilistic databases is that a functional dependency holds in the database if and only if it holds in each of the represented worlds. We can thus consider the notion of functional dependencies (fds) with the usual meaning. The closure $\mathit{CLOSURE}_\Sigma(\overline{A})$ of a set of attributes $\overline{A}$ under a set of fds Σ is defined in the normal way. For example, $\mathit{CLOSURE}_{\{A\to D; BD\to E\}}(ABC) = ABCDE$.

We will consider the following simplification of queries.

Definition IV.1 (FD-reduct)**.** *Given a set of fds Σ and a conjunctive query of the form*

$$Q = \pi_{\overline{A_0}}(\sigma_\phi(R_1(\overline{A_1}) \bowtie \ldots \bowtie R_n(\overline{A_n}))$$

where ϕ is a conjunction of unary predicates. Then, the Boolean query

$$Q_{fd} = \pi_\emptyset(\sigma_\phi(R_1(\mathit{CLOSURE}_\Sigma(\overline{A_1}) - \mathit{CLOSURE}_\Sigma(\overline{A_0})) \bowtie \ldots \bowtie R_n(\mathit{CLOSURE}_\Sigma(\overline{A_n}) - \mathit{CLOSURE}_\Sigma(\overline{A_0}))))$$

is called the FD-reduct of Q under Σ.

The importance of FD-reducts is twofold. First, non-hierarchical queries can admit hierarchical FD-reducts and we can use the latter to answer the former. Second, we accommodate non-Boolean (possibly non-hierarchical) queries by rewriting them into hierarchical FD-reducts such that the signature of the latter can be used to factor the DNF formulas associated with distinct tuples in the answers to the former.

Remark IV.2. Functional dependencies that hold in tuple-independent probabilistic databases can be used to *decide* on whether a given query admits safe plans [5]. We go further and use functional dependencies to statically *compute* or *refine* query signatures that capture the structure of the formulas in the query answer and are effectively used in planning the number of scans necessary to turn such formulas into 1OF. □

Example IV.3. Consider a slight modification of the guiding query from the Introduction (ckey is not an attribute of Item)

$$\pi_{cname}(\text{Item(okey, discount)} \bowtie \text{Ord(okey, ckey, odate)} \bowtie \text{Cust(ckey, cname)}).$$

The FD-reduct under the fd Ord: okey→ ckey odate is

$$\pi_\emptyset(\text{Item(okey, discount, ckey, odate)} \bowtie \text{Ord(okey, ckey, odate)} \bowtie \text{Cust(ckey)}).$$

Whereas the latter is a Boolean hierarchical query, the former is a non-Boolean and non-hierarchical query.

The signature of the latter query is Cust(Ord Item*)* and can be used to factor the DNF formulas associated with each bag of duplicates of the former query. □

The answers of a query and of its FD-reduct are intimately related, as explained next. Let t be an arbitrary tuple possible in R_i and let $sch'(R_i) = \mathit{CLOSURE}_\Sigma(sch(R_i))$ be the extended schema of R_i obtained by the construction of the FD-reduct under Σ. Consider two possible worlds $\mathcal{A}$ and $\mathcal{B}$ of a tuple-independent probabilistic database. If their query results each contain tuples that involve t, i.e., $t \in \pi_{sch(R_i)}Q^\mathcal{A}$ and $t \in \pi_{sch(R_i)}Q^\mathcal{B}$, then

$$\sigma_{sch(R_i)=t}(Q^\mathcal{A}) = \sigma_{sch(R_i)=t}(Q^\mathcal{B}) = \{t'\},$$

i.e. a singleton that extends t by functionally determined additional values that must be the same in all worlds. This is due to the tuple-independence property and ensures that if we extend our representations of the input relations by joining in the columns that are added by the closure computation, the modified query on the altered database will return the same result in each possible world as the original query on the original database. We do not actually need to carry out this rewriting of query and database to be able to apply our efficient query evaluation technique. If the FD-reduct is hierarchical, then the operator that will be presented in Section V uses its signature to efficiently and correctly evaluate the original query on the original database.

The reason for discarding $\mathit{CLOSURE}(\overline{A_0})$ is to obtain FD-reducts with more precise signatures. Duplicate tuples in the answer to non-Boolean queries agree on the $\overline{A_0}$-values (and on values that are functionally implied by them). Fixing such values would correspond to dropping these attributes from the query and lead to a simplified signature that can be used to factor the DNF formula associated with each bag of duplicates.

Example IV.4. Consider the following hierarchical query

$$\pi_{okey}(\text{Item(ckey, okey, discount)} \bowtie \text{Ord(okey, ckey, odate)} \bowtie \text{Cust(ckey, cname)})$$

with signature (Cust*(Ord*Item*)*)*.

The FD-reduct under the fd Ord: okey→ ckey odate is

$$\pi_\emptyset(\text{Item(discount)} \bowtie \text{Ord()} \bowtie \text{Cust()})$$

with signature Cust Ord Item*. □

It has remained open whether we have an effective way of finding hierarchical queries using functional dependencies if such hierarchical rewritings exist. The answer is in the affirmative: By always computing the full attribute closure of the functional dependencies we never miss out on hierarchical rewritings. The argument is made precise by the chase of functional dependencies, which stepwise, given an attribute set $\overline{A}$ and an fd $\overline{B} \to \overline{C}$, adds $\overline{C}$ to $\overline{A}$ if $\overline{B} \subseteq \overline{A}$.

Proposition IV.5. *Given a query Q and a set of fds Σ over the relations of Q. If there is a sequence of chase steps that turns Q into a hierarchical query, then the fixpoint of the chase, i.e., the FD-reduct, is hierarchical.*

Proof Sketch. This works because a single application of the chase rule can never turn a hierarchical query nonhierarchical: Applying fd $R_i : \overline{A} \to \overline{B}$ to R_j with $\overline{A} \in sch(R_i) \cap sch(R_j)$ adds $\overline{B}$ to $sch(R_i) \cap sch(R_j)$; thus the attributes of $\overline{B}$ may move up the hierarchy but can never invalidate it. The overall result follows from this argument by induction. □

$$
\begin{aligned}
[\![R]\!](Q,(V,P)) &= (Q,(R.V,R.P))\\
[\![\alpha\beta]\!](Q,(V,P)) &= \text{let } (Q_2,(V_2,P_2)) = [\![\beta]\!](Q,(V,P)),\\
&\qquad (Q_1,(V_1,P_1)) = [\![\alpha]\!](Q_2,(V_2,P_2)),\\
&\qquad P_1' = (P_1 \cdot P_2 \text{ as } P_1),\\
&\qquad a = attrs(Q) - \{P_1,V_2,P_2\} \cup \{P_1'\}\\
&\quad \text{in } (\pi_a(Q_1),(V_1,P_1))\\
[\![\alpha^*]\!](Q,(V,P)) &= \text{let } (Q_1,(V_1,P_1)) = [\![\alpha]\!](Q,(V,P)),\\
&\qquad V' = (\min(V_1) \text{ as } V_1),\\
&\qquad P' = (\text{prob}(P_1) \text{ as } P_1),\\
&\qquad a = attrs(Q) - \{P_1,V_1\}\\
&\quad \text{in } (\mathsf{GRP}[a;V',P'](Q_1),(V',P'))\\
\mathsf{GRP}[a;b](Q) &= \text{select distinct } a,b \text{ from } Q \text{ group by } a
\end{aligned}
$$

Fig. 5. Semantics of our operator given by translation to SQL.

We also consider rewritings applicable to queries with self-joins. If the query is of the form

$$\sigma_\phi(R(\overline{A},\overline{C})) \bowtie \sigma_\psi(\rho_{\overline{A}\to\overline{B}}(R(\overline{A},\overline{C}))) \bowtie Q_0$$

such that $\phi(\overline{A},\overline{C})$ and $\psi(\overline{B},\overline{C})$ are mutually exclusive, then we can think of the two partitions of R as different relations and the query can be treated like one without a self-join.

V. An Operator for Probability Computation

We introduce a new query plan operator for efficient probability computation in case of hierarchical queries on tuple-independent probabilistic databases. As discussed in Section IV, a large class of non-hierarchical queries can directly benefit from the results of this section.

Given a probabilistic table representing the answer to a (hierarchical) query on tuple-independent databases, this operator computes the set of distinct data tuples together with their exact probabilities. This aggregate function is semantically equivalent to a sequence of standard distinct and group-by operators that work on the variable and probability columns of probabilistic tables. Its salient characteristic is that, although it is more complex than a simple group-by, it can blend into *any* relational query plan and benefit from existing optimizations concerning aggregations [4], in particular group-by push down and pull up [17]. The key aspect of this characteristic is surprisingly simple: In addition to columns for storing probabilities, our tuple-independent data model also defines columns for storing the Boolean random variables associated with the input tuples. This characteristic is not shared by the state-of-the-art approach of MystiQ [5], which works on probabilistic tables without variable columns and where only restricted ("safe") query plans can be used for correct probability computation. Preserving the variables during query evaluation is sufficient to understand the relationships between tuples in the query answer, and it can be exploited for probability computation [1], [13], [10]. It is also known from incomplete information databases that variables and their propagation through queries can ensure the closure of the data model under various query languages [8], [12]. These important benefits come in exchange for additional little storage of the variables, which can be represented as integers.

We next introduce our operator, signature-based optimizations, and sketch an efficient low-level implementation.

A. Semantics

Fig.5 gives the semantics of our operator by translation to SQL. For a hierarchical query Q with signature s, the query returned by the statement

$$\begin{aligned}&\text{let } (Q',(V,P)) = [\![s]\!](Q,(\_,\_))\\ &\text{in select } attrs(Q')\text{-}\{V\} \text{ from } Q'\end{aligned}$$

computes the distinct answer tuples and their probabilities.

For simplicity, we assume the existence of an aggregate function prob that computes the joint probability of independent variables of a column (this can be simulated using an aggregate function product as 1-product(1-P)). For a given query, the function $attrs$ returns its selection attributes. The pair (V,P) stands for the variable and probability columns of the table encountered last in the bottom-up traversal of the signature's tree structure; its initial value is irrelevant.

Our operator allows for any query plan to compute the answer to Q. All that it needs is the query signature. A signature R corresponds to queries that are identity on table R. There are no duplicates and the probabilities are those already in the tuples. Signatures $\alpha\beta$ and α^* are processed by *propagation* and *aggregation* steps, respectively. A signature $\alpha\beta$ corresponds to a join between tables without duplicates: Each tuple in Q represents a distinct pair of independent tuples from α and β. The values in the probability column of α are multiplied with the values in the probability column of β, and the variable and probability columns of β are dropped. A signature α^* corresponds to queries with projections. Duplicates can naturally arise in such cases. The signature α can be either a table name or a composite signature. In both cases, variables of (input) tables in α may occur several times within a (result) column, and we aggregate them using the GRP statement. The special factored form of expressions over the variables encoded in variable columns ensure that, by aggregating such a variable column, we partition its variables into disjoint sets. Because any two distinct variables are independent, we can compute the probability of each disjoint set as mentioned above. For each set, we also choose a representative variable: Provided variables are represented by integers, the representative variable is simply the one with minimal id. We do not drop the aggregated variable column because of many-to-many relationships. We explain using the signature R^*S^*, which states a many-to-many relationship between tables R and S. We first aggregate the variable and probability columns of S and compute the representative variables together with the probability of each partition. By this, we reduce the many-to-many relationship to a many-to-one relationship. The representative variables are needed to further reduce this many-to-one relationship to a one-to-one relationship by next aggregating the variable and probability columns of R. After the relationship is reduced to one-to-one, we apply the case of signature $\alpha\beta$.

Q_1= GRP[odate, Cust.V, Cust.P, Ord.V, Ord.P; min(Item.V) as Item.V, prob(Item.P) as Item.P](Q)
Q_2= GRP[odate, Cust.V, Cust.P, Item.V, Item.P; min(Ord.V) as Ord.V, prob(Ord.P) as Ord.P](Q_1)
Q_3= $\pi_\phi(Q_2)$, where ϕ = odate, Cust.V, Cust.P, Ord.V, Ord.P·Item.P as Ord.P
Q_4= GRP[odate, Cust.V, Cust.P; min(Ord.V) as Ord.V, prob(Ord.P) as Ord.P](Q_3)
Q_5= GRP[odate, Ord.V, Ord.P; min(Cust.V) as Cust.V, prob(Cust.P) as Cust.P](Q_4)
Q_6= $\pi_\phi(Q_5)$, where ϕ = odate, Cust.V, Cust.P·Ord.P as Cust.P
Q_7= GRP[odate; min(Cust.V) as Cust.V, prob(Cust.P) as Cust.P](Q_6)

$$(\text{Cust}^*(\text{Ord}^*\text{Item}^*)^*)^* \xrightarrow{Q_1} (\text{Cust}^*(\text{Ord}^*\text{Item})^*)^* \xrightarrow{Q_2} (\text{Cust}^*(\text{Ord Item})^*)^* \xrightarrow{Q_3} (\text{Cust}^*\text{Ord}^*)^* \xrightarrow{Q_4} (\text{Cust}^*\text{Ord})^* \xrightarrow{Q_5} (\text{Cust Ord})^* \xrightarrow{Q_6} \text{Cust}^* \xrightarrow{Q_7} \text{Cust}$$

Signatures of constituent aggregation and propagation steps: Q_1[Item*]; Q_2, Q_4[Ord*]; Q_3[Ord Item]; Q_5, Q_7[Cust*]; Q_6[Cust Ord]

Fig. 6. Probability computation and signature transformations through successive aggregations and propagations (Examples V.1 and V.2).

Example V.1. Fig.6 gives the sequence of group-by statements that define our operator in case of query Q from the Introduction. This query has the signature $(\text{Cust}^*(\text{Ord}^*\text{Item}^*)^*)^*$.

We recurse into the signature until we reach Item, and return $(Q, (\text{Item}.V, \text{Item}.P))$. We aggregate Item* using the GRP statement Q_1 that removes the duplicate Item-tuples paired with the same Cust and Ord-tuples. We choose as a representative of each set of duplicates the one with the minimal variable (any representative would do). We proceed similarly for Ord and remove the duplicate Ord-tuples paired with the same Cust and Item-tuples (Q_2). The two aggregation steps reduce $(\text{Ord}^*\text{Item}^*)^*$ to $(\text{Ord Item})^*$, where the Ord-tuples and Item-tuples are independent. We now reduce (Ord Item) to Ord in a propagation step (Q_3) that computes the joint probabilities of Ord-tuples and Item-tuples, stores them in the probability column of Ord, and drops the variable and probability columns of Item. We are now left with signature $(\text{Cust}^*\text{Ord}^*)^*$. We proceed similarly to the previous three steps and reduce this signature first to $(\text{Cust}^*\text{Ord})^*$ (Q_4), then to $(\text{Cust Ord})^*$ (Q_5), and finally to Cust* (Q_6). As a last step (Q_7), we select the distinct odate-tuples of Q_6.

We now check that the GRP statements Q_1 to Q_7 correctly compute the probabilities of distinct tuples in the answer to Q. Fig.1 gives this answer for our toy database (V_c is Cust.V, V_o is Ord.V, and V_i is Item.V; the probability columns are omitted in the figure). Query Q_1 aggregates on Item, and reduces the answer to $(1995\text{-}01\text{-}01,\ x_1, 0.1, y_1, 0.1, z_1, 0.28)$ over schema $(odate,\ \text{Cust}.V, \text{Cust}.P, \text{Ord}.V, \text{Ord}.P, \text{Item}.V, \text{Item}.P)$, where z_1 was chosen as representative variable for $\{z_1, z_2\}$, and 0.28 is the probability $\mathsf{prob}(\{z_1, z_2\}) = 1 - (1 - 0.1) \cdot (1 - 0.2)$. Query Q_3 is the next to change the answer by dropping Item.V and Item.P and updating Ord.P to Ord.P·Item.P. We hence obtain the tuple $(1995\text{-}01\text{-}01, x_1, 0.1, y_1, 0.028)$. Similar to Q_3, Q_6 changes this tuple into $(1995\text{-}01\text{-}01, x_1, 0.0028)$. Finally, we drop x_1 and return $(1995\text{-}01\text{-}01, 0.0028)$.

For the more precise signature $(\text{Cust}(\text{Ord Item}^*)^*)^*$, we only need to generate three instead of five GRP statements (corresponding to the *'s in the signature). Section IV shows that under natural assumptions concerning functional dependencies that hold on TPC-H databases, both signatures are correct for our query and we can thus use the more precise one. As we show later, a low-level implementation of our operator can process this complex signature in only one scan. □

Sequences of aggregation and propagation steps, which define probability computation operators, can be stated as transformations of signatures. Given a sequence of steps with signatures $s_1, \ldots, s_n$, we denote the probability computation operator defined by this sequence as $[s_1, \ldots, s_n]$.

Example V.2. Fig.6 gives the overall signature of query Q, and its decomposition into signatures of each aggregation and propagation step of our probability computation operator. □

Theorem V.3. *Given any query plan P of a hierarchical query Q with signature s, the plan $[s](P)$ computes the probability of each distinct tuple in the answer to Q.*

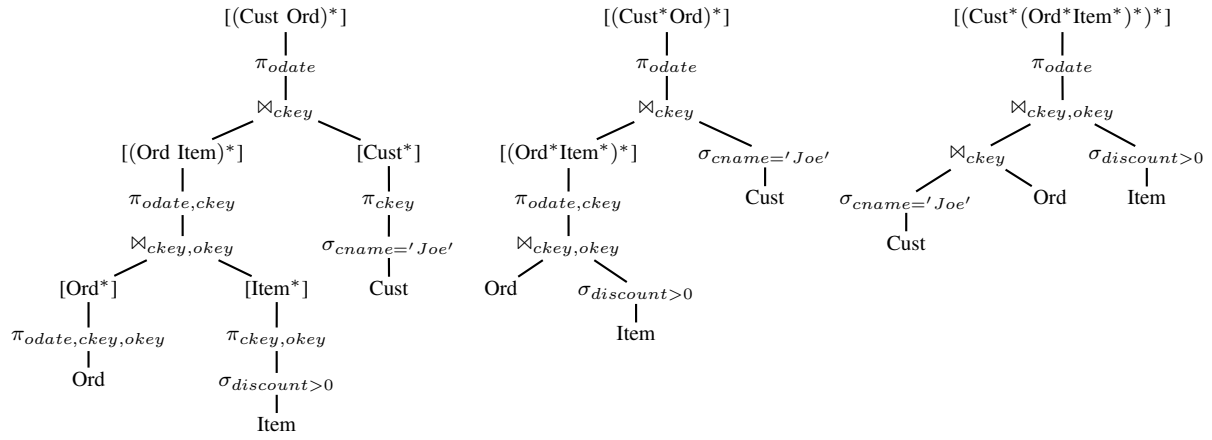
B. Optimizations

The semantics of our operator already gives an efficient implementation as a sequence of aggregations (group-by) and propagation (projection) statements, and suggests a *lazy* approach to probability computation, whereby we first compute the answer tuples and then the probability of the distinct ones. This approach contrasts with the state-of-the-art method that can only use eager plans under restrictive join orders [5].

Our framework allows for pulling up or pushing down probability computation operators in the query plan, thus adjusting the plan between the two extremes of eager and lazy plans. These optimizations are governed by rules that exploit the query signature. Pushing our operator past a join pays off when it dramatically reduces the size of a join input table and the join is rather unselective. Pulling it up past a join is beneficial in case of selective joins. The latter case has also the advantage that the order of tuples after most joins favours grouping and thus our operator.

It is, however, not always possible to completely push or pull a large sequence of aggregations, but rather small groups of them. To allow for query optimization, we may treat each simple aggregation of a probability computation operator as a standalone operator. The signatures offer the right mechanism to this effect. An operator can be split into several (possibly overlapping) operators of precise signatures, and, reversely, several operators can be merged into a single one.

Example V.4. Fig.6 gives a decomposition of the probability computation operator for query Q into smaller non-



(a) Eager plan (join order as in Fig. 2) (b) Hybrid plan (join order as in Fig. 2) (c) Lazy plan (better join order)

Fig. 7. Query plans for query Q from the Introduction using signature $(\text{Cust}^*(\text{Ord}^*\text{Item}^*)^*)^*$.

decomposable operators. Further decompositions are given in the query plans of Fig. 7. These plans are explained later.

The first two (aggregation) operators Q_1 and Q_2 can be composed into one operator $[\text{Item}^*, \text{Ord}^*]$. If we add the third (propagation) operator Q_3, we obtain $[\text{Item}^*, \text{Ord}^*, (\text{Item Ord})]$, or simpler $[\text{Item}^*\text{Ord}^*]$. □

Our optimizations are enabled by the following result.

Proposition V.5. *Any subquery of a hierarchical query is hierarchical.*

We can place a probability computation operator on top of any node n of a plan P for a given query. This operator requires to be preceeded by a projection on the query's selection attributes and all the join attributes needed for the joins that are not underneath. This ensures that duplicate elimination and probability computation can be performed after dropping irrelevant (data) columns.

The signature s of our operator is computed based on the query signature and the probability computation operators in the subplan P' rooted at n. We start with the query signature and drop all tables that do not occur in the subplan P'. In case P' contains probability computation operators, we replace in s each of their signatures t by the leftmost table name in t (this is our convention also used in defining the semantics in Fig. 5). In this process, we only need to look at the operators nearest to n in each branch of P'. We are now left with a signature that captures all aggregations steps (tables with star), but may specify propagation steps that are not captured by other operators in P'. We need to check which of these steps are valid at node n. A propagation step $\alpha\beta$ is valid at n, if P' contains all the tables in the minimal cover of the tables of α and β in the query signature. In case a propagation step is not valid at n, we need to split the signature s such that it does not specify that step anymore, i.e., $[\alpha\beta]$ is split into $[\alpha, \beta]$.

After inserting an operator $[\alpha_1, \ldots, \alpha_n]$ at node n, we need to update all probability computation operators that are its ancestors in the plan P. The update procedure is the same as for the operators in the subplan P' and their ancestor we have just inserted: For each ancestor operator, we replace in its signature each α_i by the leftmost table name in α_i.

Example V.6. We analyze the plans of Fig.7 for our query Q from the Introduction. The query signature is that of the probability computation operator in Plan (c). The key constraints are ignored here, otherwise the signature would be refined to $[(\text{Cust}(\text{Ord Item}^*)^*)^*]$.

At the node n with $[(\text{Ord}^*\text{Item}^*)^*]$ in Plan (b), the signature is computed by dropping from the query signature the table Cust. The propagation step $\text{Ord}^*\text{Item}^*$ is also kept because the subplan rooted at node n contains all the tables in the minimal cover of $\{\text{Ord}, \text{Item}\}$ (which are Ord and Item).

Assume we drop the operator $[(\text{Ord Item})^*]$ in Plan (a); call this node m. The new signature of the top operator is obtained from the query signature by replacing Ord^* by Ord, Cust^* by Cust, and Item^* by Item: $(\text{Cust}(\text{Ord Item})^*)^*$. Let us now re-insert an operator at node m. Its signature is the query signature where Cust is dropped: $(\text{Ord}^*\text{Item}^*)^*$. We then perform the substitutions given by the signatures of the underneath operators and replace Ord^* by its leftmost table name (Ord), and replace similarly Item^* by Item: $(\text{Ord Item})^*$. We finally update its ancestor operators (in this case the top operator) and obtain $[(\text{Cust Ord})^*]$ from $[(\text{Cust}(\text{Ord Item})^*)^*]$.

Let us add a new probability computation operator at the node p immediately after the join on $ckey$ in Plan (c). The projection to be added on top of this join is $\pi_{odate,ckey}$. We first eliminate Item from the query signature, because it is not contained in the subplan rooted at p, and obtain $(\text{Cust}^*(\text{Ord}^*)^*)^*$. The propagation step $\alpha\beta$, where $\alpha = \text{Cust}^*$ and $\beta = \text{Ord}^*$ is not valid at node p because the table Item, which is in the minimal cover of $\{\text{Cust}, \text{Ord}\}$, is not in the subplan rooted at p. We thus split the signature into Cust^* and Ord^* and obtain the operator $[\text{Cust}^*, \text{Ord}^*]$. Finally, we update the probability computation operator at the top of Plan (c) by replacing Cust^* by Cust and Ord^* by Ord in its signature: $[(\text{Cust}(\text{Ord Item}^*)^*)^*]$. □

Just as for traditional algebraic optimization rules, some

placements do not necessarily lead to better query plans. Cost-based decisions can be made using the host relational database engine. We let the engine find a good query plan for Q, where a distinct construct is added to the outermost select clause of Q and the variable columns are dropped. The `Unique` operators in the plan (for Postgres), which implement the distinct construct, denote places where probability computation operators can be added.

Example V.7. Fig.7 gives three equivalent plans for our query Q from the Introduction. Plan (a) is eager and structurally similar to the safe plan in Fig. 2. The aggregation operators are pushed down towards the leaves of the plan. Plan (c) is lazy and follows the translation of Fig.5. Here, the entire probability computation is done after computing the answer tuples of Q. Plan (b) is hybrid: It only pushes a part of the aggregations down to a temporary table, which is expected to have many duplicate (odate,ckey)-tuples. The probability computation operators on top of the input tables are dropped, given the high cost associated with their sizes and the low selectivity of the joins. In TPC-H scenarios, Plan (c) outperforms the other two due to its different join order: It first joins the very few tuples of Cust with name 'Joe' with Ord on the key of Cust-table. Plans (a) and (b) first perform a join on two large tables. □

Although we did not explicitly address the improvements brought by functional dependencies to our optimizations, the results of Section IV directly apply here: We show there how to use functional dependencies for inferring simpler Boolean hierarchical queries and hence more precise signatures. For instance, under TPC-H functional dependencies, the query signature of Q becomes $(\text{Cust}(\text{Ord Item}^*)^*)^*$ and some of the operators in the plans of Fig.7 have fewer or no aggregation or propagation steps; their signatures become trivial (a simple table name) or simpler, e.g., $[\text{Cust}^*]$ becomes [Cust], and $[(\text{Ord}^*\text{Item}^*)^*]$ becomes $[(\text{Ord Item}^*)^*]$.

C. Execution

The implementation of our probability computation operator by a sequence of aggregation and propagation steps, as suggested by the semantics in Fig.5, is not optimal: Each aggregation and propagation step is done independently of the others. Our key observation is that some steps can be grouped and computed in a few sequential scans following one sorting of the answer tuples. Before accessing the input table, an operator computes the number of scans required by the steps, which aggregations and propagations to do in which scan, and the sort order of the columns of the input table. We next discuss these issues.

Definition V.8. *Given a query signature α.*

$$\mathit{1scan}(\alpha) = (\forall \beta^* \in \alpha(\exists\ \mathit{table}\ R \in \beta \wedge \mathit{1scan}(\beta))).$$

A signature has the *1scan* property if each of its composite expressions is made up by concatenating signatures with the *1scan* property and at least a table without (*). For signature α, we denote by $\#scans(\alpha)$ one plus the number of its subexpressions β^*, including itself, without the *1scan* property.

```
1scan_compute_prob(1scanTree t){
  prevSlot = NULL; fetch the first tuple into crtSlot;
  foreach node n in t do {
      enable n, n.crtP = 0, n.allP = 0;
      n.index = index of its column in crtSlot; }

  {Compare prevSlot and crtSlot, find leftmost unmatched column i;
   propagate_prob(t, i, crtSlot);
   prevSlot = crtSlot; fetch the next tuple into crtSlot;
  } do while (prevSlot ≠ NULL)

  return t.root.allP;
}
propagate_prob(1scanTree n, int i, Slot crtSlot){
   foreach child c of n do propagate_prob(c, i, crtSlot);
   if (n is enabled && n.index ≥ i)
      if (n is a leaf node && n.index = i)
         n.crtP = 1 - (1 - n.crtP) · (1 - crtSlot[n.index].prob);
      else {
         foreach child c of n do n.crtP = n.crtP · c.allP;
         n.allP = 1 - (1 - n.crtP) · (1 - n.allP);
         if (n.index = i) {
            foreach descendant d of n do
               enable d; d.allP = 0; d.crtP = crtSlot[d.index].prob;
            n.crtP = crtSlot[n.index].prob; }
         else
            disable n and all its descendants; }
}
```

Fig. 8. Probability computation for queries with *1scan* signature (sketch).

Example V.9. The signature $(\text{Cust}(\text{Ord Item}^*)^*)^*$ has the 1scan property: It contains the table Cust, and all nested signatures of the form β^* contain tables, for instance (Ord Item^*) contains Ord. On the other hand, the signature $(\text{Cust}^*(\text{Ord}^*\text{Item}^*)^*)^*$ does not have the 1scan property, because it does not contain a table (but only closures on tables or on composite signatures).

Further examples of 1scan signatures: R^*S^* (relational product) and $\text{Nation}_1\text{Supp}(\text{Nation}_2(\text{Cust}(\text{Ord Item}^*)^*)^*)^*$ (the signature of the conjunctive subquery of TPC-H query 7). □

This property captures signatures of queries with foreign key joins, which are natural in cases with one-to-many relationships between tables (like in TPC-H). It turns out that this property is sufficient for computing aggregations in one scan.

Proposition V.10. *An operator $[\alpha]$ needs $\#scans(\alpha)$ scans.*

This result extends to sequences of operators, whose signatures cannot be captured into a single signature, by summing up the number of scans necessary for each of them.

Example V.11. For $[(\text{Cust}^*(\text{Ord}^*\text{Item}^*)^*)^*]$ we need two scans to turn $(\text{Ord}^*\text{Item}^*)^*$ and $(\text{Cust}^*(\text{Ord}^*\text{Item}^*)^*)^*$ into 1scan signatures. This corresponds to computing $[\text{Ord}^*]$ in the first scan and $[\text{Cust}^*]$ in the second scan. A third scan is used by $[(\text{Cust}(\text{Ord Item}^*)^*)^*]$, which has the 1scan property. In contrast, the semantics of our operator suggests five aggregations and two projections (Fig.6). This can require to sort and aggregate the answer tuples five times. □

As it has become evident in the previous example, we first schedule aggregations so as to obtain a 1scan signature. We then allocate scans such that the innermost subexpressions β^*

without the 1scan property turn into 1scan signatures.

Our efficient probability computation for hierarchical queries with 1scan signatures uses a tree representation of signatures, called *1scanTree*, where each node corresponds to one variable column occuring in the query answer. A 1scanTree is constructed from the hierarchical representation of the query. Recall that, for a hierarchical query, the nesting structure of its signatures coincides with its hierarchical representation. We traverse bottom-up the hierarchical representation and replace each inner node with one of its children that is a table name. This is always possible because the signature has the 1scan property and hence each nested subexpression (which corresponds to a subtree in the hierarchical representation) has the 1scan property and one table name without star (*).

The sort order of the input to our operator is given by the columns that hold input data followed by the variable columns corresponding to the table names in any preorder traversal of the 1scanTree of its input signature.

Example V.12. The signature $(\text{Cust}(\text{Ord Item}^*)^*)^*$ has as 1scanTree the path $(\text{Cust}, \text{Ord}, \text{Item})$, where Cust is the root. The sort order for query Q from the Introduction is $(odate, V(\text{Cust}), V(\text{Ord}), V(\text{Item}))$.

The 1scanTree of the signature $(R_1(R_2R_3^*)^*(R_4R_5^*)^*)^*$ can be serialized as $R_1(R_2(R_3), R_4(R_5))$ with root R_1 and two paths (R_1, R_2, R_3) and (R_1, R_4, R_5). □

Fig.8 sketches our probability computation algorithm for 1scan signatures, which uses the 1scanTree representation of the signature and performs one scan over its input table. The algorithm assumes wlog that the sorting order of the variable columns is the same as the order in which they appear in the tuples, and annotates each node in the 1scanTree with the index of its corresponding variable column in this order. We run this algorithm independently for each bag of duplicates (i.e., that have the same values in the data columns).

The core of the algorithm is the procedure propagate_prob, which incrementally updates running probabilities at the 1scanTree nodes in postorder. A partition is defined by a set of variables that occur in that column and are paired with the same variables from other columns. Our algorithm keeps track of current and completed partitions for each variable column at the corresponding 1scanTree node. On reading an input tuple, we update three properties of the 1scanTree nodes: The running probability crtP of the current partition, the running probability allP of all finished partitions, and an enabling flag. While within a partition, we incrementally compute its probability and keep it in crtP. When the end of a partition in a variable column is reached, the allP value at the corresponding node is updated with the probabilities of the children and of the finished partition. The postorder traversal ensures that the allP values of the descendants of a node are updated before the allP value at that node.

A partition can re-occur in a variable column, for instance in case of many-to-many relations between variables in several columns. We avoid redundant computation by disabling nodes during the time when old partitions re-occur in their corresponding variable columns. Such nodes are re-enabled when new partitions of their ancestors are encountered in the input.

When the bag of tuples is finished, the value allP of the 1scanTree root node is the desired exact probability.

Example V.13. Consider the operator $[(\text{Cust}(\text{Ord Item}^*)^*)^*]$ on top of a plan for our query Q. The answer tuples are sorted on $(odate, V(\text{Cust}), V(\text{Ord}), V(\text{Item}))$. We have one-to-many relations between $V(\text{Ord})$-variables and $V(\text{Item})$-variables, and between $V(\text{Cust})$-variables and pairs of $(V(\text{Ord}), V(\text{Item}))$-variables. All allP and crtP values are initially set to 0. While scanning the tuples in the given order, we encounter distinct $V(\text{Item})$-variables for the same $V(\text{Ord})$-variable (and $V(\text{Cust})$-variable). With every new $V(\text{Item})$-variable, we update node(Item).crtP using the probability computation formula for independent variables. When a new $V(\text{Ord})$-variable is read, we assign crtP to allP in node(Item), set node(Ord).crtP to node(Ord).crtP×node(Item).allP, and update node(Ord).allP with node(Ord).crtP like for node(Item). We continue until a new $V(\text{Cust})$-variable is met and proceed similar to the case of a new $V(\text{Ord})$-variable. At the end, node(Cust).allP contains the desired probability. □

D. Implementation

The SPROUT prototype has been implemented in PostgreSQL8.3.3 and has about 2500 lines of code. It is currently used by the query engine of the probabilistic database management system MayBMS and is freely available. The major changes to PostgreSQL we have done so far are the addition to the SQL query language of an aggregation construct conf() for probability computation and of a construct for creating tuple-independent tables from standard tables, a module that creates query signatures and refines them using the key constraints existing in the database, and our efficient operator for probability computation. More information about SPROUT (and a link to the software repository) is available on the SPROUT webpage (see the Introduction).

VI. Case Study: TPC-H

We analyzed which of the 22 TPC-H queries can benefit from our efficient evaluation technique. A complete report of our findings is available at the SPROUT webpage.

For each TPC-H query, we considered its largest subquery without aggregations and inequality joins but with the special conf() aggregation for specifying exact probability computation for distinct tuples in query answers. We consider two flavours of each of these queries: A version with original selection attributes (again, without aggregations), and a version where we drop keys from the selection attributes. The reason for the latter version is that the presence of keys in the selection attributes usually implies the hierarchical property. Many practical queries, however, do not have keys among the selection attributes. For such cases, we also investigated whether we can derive hierarchical queries. This latter version includes the case of Boolean queries, of course.

Many of these queries are hierarchical in the absence of the TPC-H key constraints: 13 out of the 22 queries with

the original selection attributes and 8 out of 22 queries with selection non-key attributes (four queries occur in both sets). In the presence of the TPC-H key constraints, our approach can cope with four more queries in each of the two classes.

TPC-H has interesting cases for our techniques to discover hierarchical FD-reducts. For the queries 2, 11, and 18 we use the existing TPC-H keys to derive hierarchical FD-reducts. Query 18 is very similar to our query from the Introduction. Query 7 is a fairly complex join on six tables, where two of them are copies of Nation. The self-join causes no problems because each table copy has distinct tuples. Query 19 has a disjunction of three hierarchical conjunctions that are mutually exclusive, which thus select disjoint sets of independent tuples.

Five queries do not admit hierarchical FD-reducts. Queries 5, 8, and 9 involve at least two joins of table Item with different non-key attributes that do not occur as selection attributes. Query 13 is a left outer join on customer and orders. Query 22 has subqueries that involve aggregations and inequality joins, and by removing them, it becomes a simple selection.

VII. Experiments

The experiments were conducted on an Athlon-X2(4600+) 64bit/1.8GB/Linux2.6.20/gcc4.1.2/PostgreSQL8.3.3 machine.

TPC-H Data and Queries. Our data set consists of tuple-independent probabilistic databases obtained from deterministic databases produced by TPC-H 2.7.0 by associating each tuple with a Boolean random variable and by choosing at random a probability distribution over these variables. We report on experiments with TPC-H scale factor 1 (1GB database size) and without indices. We evaluated the TPC-H-like queries mentioned in Section VI: 17 TPC-H queries and the Boolean variants of 9 of them, without aggregations and inequality joins but with the conf() aggregation.

Query Engines. We compare our SPROUT engine with MystiQ [5], which implements the state-of-the-art exact evaluation technique especially tuned for hierarchical queries. MystiQ is a middleware that rewrites hierarchical queries into SQL queries that use aggregations to compute probabilities (as of June 2008). MystiQ was configured to work on tuple-independent databases (by appropriately setting database configuration files and dropping the columns for variables). Some of the queries (1, 4, 12, and the Boolean queries B1, B4, B6, B12, B14, B15, B16) could not be computed by MystiQ due to a minor technical problem: Given n events with probabilities $p_1, \ldots, p_n$ stored in a column P, the probability $1-\Pi_i(1-p_i)$ of their disjunction is computed as 1 − POWER(10.000, SUM(log(1.001 - P))). In case of large n, the latter formula requires the computation of logarithms of very small numbers and leads to runtime errors.

For all experiments, we report wall-clock execution times of queries run in the PostgreSQL8.3.3 psql shell with a warm cache obtained by running a query once and then reporting the average runtime over ten subsequent, identical executions.

1. Lazy plans, eager plans, MystiQ plans. We compared experimentally three different types of query plans for TPC-H queries: The eager plans that aggregate after each table, including the temporary tables, MystiQ plans, which are also eager, but with possibly suboptimal join orders, and lazy plans that use our probability computation operator at the very end.

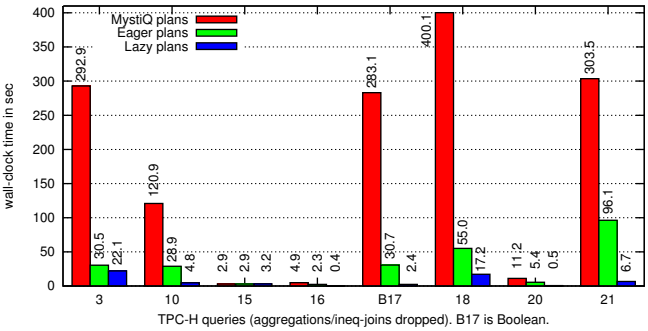


Fig. 9. Comparison: Lazy, eager, and MystiQ plans (scale factor 1).

Fig.9 shows that for the 8 TPC-H-like queries, lazy plans perform up to two orders of magnitude better than MystiQ's plans and about one order of magnitude better than our eager plans. For each query, the time needed by the lazy plans to compute the probability of the distinct answer tuples is insignificant when compared to the time needed to compute and store on disk the answer tuples without duplicate elimination but ordered as required by our operator. This is because the functional dependencies make even large query signatures very precise, and our operator computes the probability of each distinct tuple in only one scan of the answer tuples.

We experimentally confirmed the following differences between lazy, eager, and MystiQ plans. Eagerness, while fundamental for safe plans and sometimes quite beneficial in removing duplicates early on, turns out to be inappropriate in the TPC-H context, where small relations produced by selective conditions (like Cust) are joined on keys with large relations (Item). For instance, query B17 is a join of Item and a rather small subset of Part on the key pkey of Part. Any eager plan first computes the probability of each distinct pkey-value in the very large table Item, although most of these values do not occur in the selective join partner. A further important difference concerns the restrictive join orderings in MystiQ plans for queries with at least three tables (queries 10, 18, 20, and 21). Query 18, similar to our running query from the Introduction, is a join of Cust, Ord, and Item on ckey and okey, respectively; and Cust has a very selective condition. MystiQ plans compute the unselective join on Ord, and Item on okey, then the probabilities of each distinct ckey-value in the result of this join, and finally they join the temporary table of the previous join with a few (precisely one in case of query 18) Cust-tuples. Query 3 has the same joins as query 18, except that the key okey is in the projection list, which drops the restriction on join ordering.

Fig. 10 shows the performance of query evaluation for the remaining 18 queries that do not admit MystiQ plans or could not be computed by MystiQ due to runtime errors. The time needed to compute and store the answer tuples is about two orders of magnitude larger than the time taken to compute the distinct tuples and their probabilities using a lazy plan.

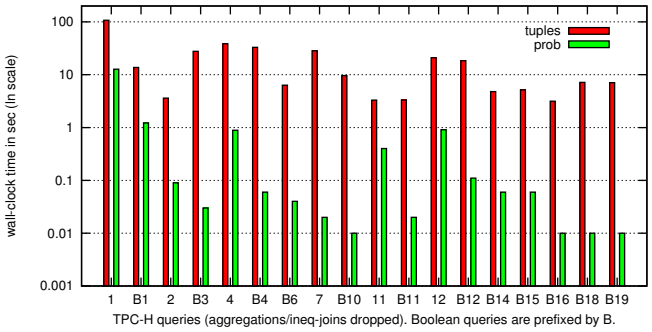
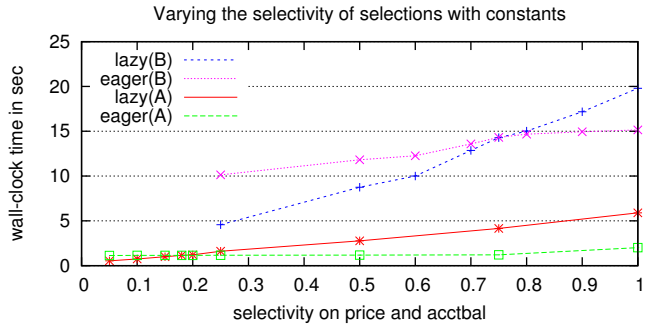


Fig. 10. Lazy plans for the remaining 18 queries (scale factor 1).



$A = \pi_{name}(\text{Nation} \bowtie_{nkey} \sigma_{acctbal<\texttt{ct}}(\text{Supp}) \bowtie_{skey} \text{Psupp})$

$B = \pi_{ckey,name}(\text{Cust} \bowtie_{ckey} \sigma_{odate<\texttt{'1996-09-01'},price<\texttt{ct}}(\text{Ord}))$

Fig. 11. Rendez-vous for eager and lazy plans (scale factor 1).

We investigated the behaviour of lazy and eager plans when the selectivity of selections with constants is varied for the two queries of Fig.11 by changing the constant `ct`. Given n tuples, a selectivity of p means here that $p \times n$ tuples satisfy the selection condition. We observed that lazy plans are preferred in case p is small. This is because the selections already filter out most of the tuples and few duplicates are produced by the last projection. In case of large p, the selections become irrelevant and the projection operators on top of the joins can create many duplicates. Removing duplicates early on, before they are multiplied by the joins, pays off for large p.

2. Lazy plans, eager plans, hybrid plans. We further conducted a limited analysis of cases where hybrid plans, i.e., plans that combine eager and lazy probability computation, outperform the extremes. The hybrid plans can avoid expensive and sometimes useless eager aggregation on large tables and save time by reducing the size of temporary tables. Fig.12 gives the timing for two queries. Both queries have plans that first avoid eager aggregation on large tables (Item and Psupp), and then push down aggregations between unselective joins.

3. The effect of functional dependencies (FDs). We experimentally compared the sequential scan and our operator with and without FDs. Fig.13 suggests that on TPC-H data, our operator's performance is very close to that of a sequential scan if FDs are used, as it only needs one scan and its computation is mainly I/O bound. In contrast, in the absence of FDs, it needs considerably more time (from 2 to 100 times more), which can be explained by the larger number of scans.

Query	Eager	Lazy	Hybrid	Eager/Hybrid	Lazy/Hybrid
C	71.10s	5.22s	4.02s	17.69	1.3
D	1.16s	0.78s	0.52s	2.23	1.5

$C = \pi_{ckey,name}(\text{Cust} \bowtie_{ckey} \sigma_{odate<\texttt{'1992-01-31'}}(\text{Ord}) \bowtie_{okey} \text{Item})$

$D = \pi_{nkey}(\text{Nation} \bowtie_{nkey} \sigma_{acctbal<600}(\text{Supp}) \bowtie_{skey} \text{Psupp})$

Fig. 12. Hybrid versus eager and lazy plans (scale factor 1).

Query	2	7	11	B3
Time for seqscan	0.02s	0.02s	0.09s	0.01s
Time for sorting	0.03s	0.07s	0.12s	0.03s
Our operator (no FDs)	0.20s	0.66s	4.23s	0.05s
Our operator (with FDs)	0.09s	0.02s	0.40s	0.03s
#answer tuples	642	5924	31680	4488
#distinct answer tuples	642	796	29818	1

Fig. 13. Influence of FDs on performance (scale factor 1).

VIII. Future Work

We see many exciting research directions that can be taken from here, mostly centered around novel secondary-storage algorithms for exact and approximate probability computation for various classes of queries and probabilistic database models. To date, the algorithms for probability computation in most available systems are main-memory and deal suboptimally with the (few) known tractable classes of queries. Additionally, little is known on which queries admit polynomial-time exact evaluation on the various proposed data models [6].

References

[1] L. Antova, T. Jansen, C. Koch, and D. Olteanu. "Fast and Simple Relational Processing of Uncertain Data". In *Proc. ICDE*, 2008.
[2] L. Antova, C. Koch, and D. Olteanu. "From Complete to Incomplete Information and Back". In *Proc. SIGMOD*, 2007.
[3] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. "ULDBs: Databases with Uncertainty and Lineage". In *Proc. VLDB*, 2006.
[4] S. Chaudhuri and K. Shim. "An Overview of Cost-based Optimization of Queries with Aggregates". *IEEE Data Engineering Bulletin*, 1995.
[5] N. Dalvi and D. Suciu. "Efficient Query Evaluation on Probabilistic Databases". *VLDB Journal*, **16**(4):523–544, 2007.
[6] N. Dalvi and D. Suciu. "Management of Probabilistic Data: Foundations and Challenges". In *Proc. PODS*, 2007.
[7] N. Dalvi and D. Suciu. "The Dichotomy of Conjunctive Queries on Probabilistic Structures". In *Proc. PODS*, 2007.
[8] T. Imielinski and W. Lipski. "Incomplete information in relational databases". *Journal of ACM*, **31**(4):761–791, 1984.
[9] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. "MCDB: a Monte Carlo Approach to Managing Uncertain Data". In *Proc. SIGMOD*, pages 687–700, 2008.
[10] C. Koch and D. Olteanu. "Conditioning Probabilistic Databases". *PVLDB*, 1(1), 2008.
[11] D. Olteanu and J. Huang. "Using OBDDs for Efficient Query Evaluation on Probabilistic Databases". In *Proc. SUM*, 2008.
[12] D. Olteanu, C. Koch, and L. Antova. "World-set Decompositions: Expressiveness and Efficient Algorithms". *TCS*, 403(2-3), 2008.
[13] A. D. Sarma, M. Theobald, and J. Widom. "Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases". In *Proc. ICDE*, 2008.
[14] P. Sen and A. Deshpande. "Representing and Querying Correlated Tuples in Probabilistic Databases". In *Proc. ICDE*, 2007.
[15] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. Hambrusch, and R. Shah. "Orion 2.0: Native Support for Uncertain Data (Demo)". In *Proc. SIGMOD*, Vancouver, Canada, 2008.
[16] M. Soliman, I. Ilyas, and K. Chang. "Probabilistic Top-k and Ranking-Aggregate Queries". In *ACM TODS*, volume 33:3, 2008.
[17] Y. Yan and P. Larson. "Eager Aggregation and Lazy Aggregation". In *Proc. VLDB*, 1995.