Department of Informatics, University of Zürich

MSc Thesis

Exact and Approximate Confidence Computation in Temporal Probabilistic Databases

Andrin Betschart

Matrikelnummer: 09-714-882

Email: andrin.betschart@uzh.ch

May 3, 2014

supervised by Prof. Dr. M. Böhlen and K. Papaioannou

University of Zurich[UZH]

Department of Informatics

Acknowledgements

First of all, I would like to thank Prof. Dr. Michael Böhlen for giving me the opportunity to write my master thesis at the Database Technology Group of the University of Zurich. Moreover, my deepest gratitude goes to Katerina Papaioannou, my supervisor who supported me throughout the entire work. Her patient guidance, enthusiastic encouragement, explanations and useful critiques contributed in numerous ways to this thesis. Finally, many thanks goes to my family and friends.

Abstract

Probabilistic databases and temporal databases are a field of interest in recent research. In this work a temporal probabilistic database schema is suggested that combines both aspects into one. To query such databases, we present a full relational algebra, which uses lineage as Boolean formulas to keep track of the origin of derived tuples. Furthermore, we investigate four algorithms to compute the confidence value of derived tuples by using lineage. We implemented all concepts of this work in the PostgreSQL database system and experiments show the performance of the lineage computation and of the four confidence computation algorithms.

Zusammenfassung

Probabilistische und temporale Datenbanken sind grosse Interessengebiete in der heutigen Forschung. In dieser Arbeit wird ein temporales probabilistisches Datenbankschema vorgeschlagen, welches die Aspekte beider Bereiche vereint. Wir präsentieren eine komplette Relationale Algebra, die es erlaubt Abfragen in solchen Datenbanken durchzuführen. Dazu verwenden wir boolesche Formeln, die sogenannte Lineage, welche die Herkunft von abgeleiteten Tupeln repräsentieren. Zudem untersuchten wir vier Algorithmen, welche es erlauben den probabilistischen Wert eines abgeleiteten Tupels mit Hilfe der Lineage zu berechnen. Alle Konzepte von dieser Arbeit wurden in das PostgreSQL Datenbanksystem integriert und Experimente zeigen die Performanz der Lineageberechnung und der vier Algorithmen zur Wahrscheinlichkeitsberechnung.

Contents

1. Introduction

In recent years a lot of research has been done in the fields of probabilistic databases and temporal databases. In this work, we now combine state of the art techniques from both fields and come up with a temporal probabilistic database schema and a full relational algebra that allows to query temporal probabilistic databases. This algebra uses lineage as Boolean formulas to keep track of the origin of derived tuples. This lineage is later used to compute the confidence value of the resulting tuples. Further, the algebra makes sure that time is perceived according to Sequenced Semantics. To do this we use the primitives suggested by Dignös et al. [2] and adapt them to our needs.

We also present four algorithms that compute the probability values of derived tuples by investigating their lineage expression. Three of those algorithms compute an exact probability value, whereas one only computes an approximation. The basic idea behind three of these algorithms is to decompose the lineage expression into parts that are easier computable and have no dependencies. For this, we use techniques which were suggested in [4, 5, 12].

Further, we implement our relational algebra operations and the confidence computation algorithms into the PostgreSQL database system. This allows us to query real temporal probabilistic data and to compute the probability value for all possibly occurring result tuples in this system.

In Section 2, we define a schema for temporal probabilistic databases and explain in detail how relational algebra operations are performed. Then, we present in Section 3 four algorithms that allow us to compute the confidence value of tuples by evaluating their lineage. Further, we explain how our concept is implemented in PostgreSQL in Section 4. Finally, we show the evaluation of our experiments in Section 5.

2. Temporal Probabilistic Databases

In this work we are dealing with Temporal Probabilistic Databases, abbreviated as TPDBs. All tuples in such databases have some standard attributes, which store any kind of information about the facts those tuples represent. Every tuple also has a time interval, called the temporal attribute T, and a probability value, called the probabilistic attribute P, attached. The time interval indicates the time period during which the tuple is valid and the probability value defines the probability to which the tuple is valid. Moreover every tuple that is stored in a TPDB must have a unique identifier x. And we further use a boolean expression, called lineage λ. This attribute is further explained in the following section.

V (Visits)

x	$Name$	$Dest$	T	P	λ
v_1	Ann	Zurich	[2014-12-04, 2014-12-14)	0.50	v_1

W (Weather)

x	Loc	$Weather$	T	P	λ
w_1	Zurich	Snow	[2014-12-05, 2014-12-10)	0.70	w_1
w_2	Zurich	Fog	[2014-12-08, 2014-12-15)	0.20	w_2

Figure 2.1.: Temporal Probabilistic Database

Figure 2.1 shows two example relations from a TPDB. The relation Visits (V) holds information about people, who visit cities and the relation Weather (W) stores information about the weather in certain cities. The fact that is stored in the tuple with the unique identifier v_1 tells us that Ann will visit Zurich from December 5, 2014 until December 13, 2014 (since December 14 is excluded) with probability 0.5. On the other hand, in tuple w_1 the information is stored, that with probability 0.7 it will snow in Zurich from December 5, 2014 until December 9, 2014 (since December 10 is excluded).

2.1. Definitions

Base & Derived Tuples: We call tuples that are stored in tables of the database, base tuples. Whereas derived tuples are tuples that were constructed from operations performed on base tuples.
Unique Identifier: Each base tuple has a random variable attached, which allows to uniquely identify the tuple. The identifier x must therefore be unique for every base tuple of the whole database.

Temporal Attribute: The time domain Ω^T is considered as a linearly ordered, finite sequence of time points. A time interval, as it is stored in the temporal attribute T of every tuple, consists of a contiguous and finite set of time points over Ω^T. We denote it by a half-open interval $[t_s, t_e)$, where $t_s, t_e \in \Omega^T$ and $t_s < t_e$. t_s is the inclusive start point and t_e the exclusive end point of the time interval that represents the tuple's valid time points.
Probabilistic Attribute: The probabilistic attribute $P \in \mathbb{R}$ has a value in the interval $(0, 1]$ and defines the probability with which the corresponding base tuple is valid. For this work, we consider a tuple-independent database. This means that the validity of one base tuple does not affect the validity of any other base tuple.
Lineage: Lineage λ is a Boolean expression that relates derived tuples to the base tuples they were derived from. The lineage is constructed from Boolean variables and the three Boolean connectives $\wedge$ (logical and), $\vee$ (logical or) and $\neg$ (logical not). The Boolean variables refer to base tuples and are therefore named with the identifier of the corresponding base tuple. We write $\phi(x)$ to refer to the lineage of the tuple with the identifier x. If x is a base tuple we have $\phi(x) = x$.

A temporal probabilistic relation schema is represented as $R^{TP} = (A_1, ..., A_m, T, P, \lambda)$, where $A_1, ..., A_m$ are the standard attributes with domain Ω_i, T is a temporal attribute, p is a probabilistic attribute and λ is the lineage. A tuple r over schema R^{TP} is a finite set that contains for every A_i a value $v_i \in \Omega_i$, for T a time interval $[t_s, t_e) \in \Omega^T \times \Omega^T$, for P a value $p \in (0, 1]$ and for λ a Boolean expression. A temporal probabilistic relation **r** over schema R^{TP} is a finite set of tuples over R^{TP}. For a tuple r and an attribute A_i, $r.A_i$ denotes the value of the attribute A_i in r. As abbreviation we use $A = A_i, ..., A_m$ and $r.A = (r.A_1, ..., r.A_m)$.

The operators of our temporal probabilistic relational algebra are selection σ^{TP}, projection π^{TP}, high aggregation ϑ^{TP}, difference $-^{TP}$, union $\cup^{TP}$, intersection $\cap^{TP}$, Cartesian product $\times^{TP}$, inner join $\bowtie^{TP}$, left outer join $⟕^{TP}$, right outer join $⟖^{TP}$, full outer join $⟗^{TP}$, normalization $\mathcal{N}^{TP}$ and alignment Φ^{TP}. For the set operators, we assume union compatible argument relations, and for $\pi_B^{TP}(\mathbf{r})$ and ${}_B\vartheta_F^{TP}(\mathbf{r})$ we require $B \subseteq A$. The probabilistic attribute P is ignored during algebra operations, since its value might not be correct for intermediate tuples. For the result tuples, lineage will be used to compute the correct probability value.

Y

	A	T	P	λ
y_1	a	[4, 14)	0.50	y_1

Z

	B	T	P	λ
z_1	a	[5, 10)	0.70	z_1
z_2	a	[8, 15)	0.20	z_2

Figure 2.2.: Example Database

2.2. Timestamp Adjustment

In order to execute queries we perceive time according to Sequenced Semantics. This means that we consider the database to be composed of a sequence of non-temporal database snapshots during execution time. A snapshot can be seen as the data that is valid at a specific point in time. In order to support Sequenced Semantics, the three properties snapshot reducibility, extended snapshot reducibility and change preservation must be satisfied. We propose two primitives, based on the ones defined by Dignös et al. [2], that transform tuples within a TPDB into a set of new ones with adjusted time intervals. The basic idea behind the primitives is to align the time intervals of the tuples in such a way that intervals that have to be compared during an algebra operation, are either identical or disjoint.

Depending on how the operation that a query consists of produces its result tuples, either the Temporal Splitter or the Temporal Aligner must be applied. For group based operators $\pi, \vartheta, -, \cup, \cap$ multiple tuples of the argument relations contribute to a single result tuple. In this case the Temporal Splitter is used to adjust the time intervals. For tuple based operators σ, $\times$,$\bowtie$, $\rtimes$,$\ltimes$, $\overline{\ltimes}$ only one tuple of each argument relation contributes to a single result tuple. In this case the Temporal Aligner is used to adjust the time intervals.

During temporal adjustments we keep lineage unmodified. This is the case, because we defined the probabilistic value of a tuple to be the probability, to which the corresponding tuple is valid. This means, that when the tuple is split into tuples with sub-time intervals, the probability of the resulting tuples is the same as of the initial tuple. For simplicity of the lineage expressions we do not care about the tuples, which caused the splitting of tuples, since they have no influence on the final confidence computation.

2.2.1. Temporal Splitter and Normalization

For group based operators Dignös et al. [2] propose a Temporal Splitter to adjust the time intervals. It is necessary that the time intervals of the tuples of the argument relations are split at every start and end point of all other tuples that satisfy the same conditions. This means that a tuple is split into tuples with identical non-temporal attributes but with disjoint adjusted time intervals, where the union of all adjusted time intervals equals the initial one.

Definition 1 (*Temporal Splitter [2]*) Let r be a tuple and $\mathbf{g}$ a set of tuples. A *temporal splitter* produces a set of tuples with the non-temporal attributes (A, P and λ) of r over the following adjusted intervals:

$$
\begin{aligned}
&T \in split(r, \mathbf{g}) \iff \\
&\quad T \subseteq r.T \wedge \forall g \in \mathbf{g}(g.T \cap T = \emptyset \vee T \subseteq g.T) \wedge \\
&\quad \forall T' \supset T(\exists g \in \mathbf{g}(T' \cap g.T \neq \emptyset \wedge T' \not\subseteq g.T) \vee T' \not\subseteq r.T).
\end{aligned}
$$

The first condition requires that each adjusted time interval is a subinterval of the initial time interval of r and that it is either contained in or disjoint from all time intervals of $\mathbf{g}$. Secondly,

each new time interval must be maximal, meaning that it cannot be enlarged without violating the first condition.

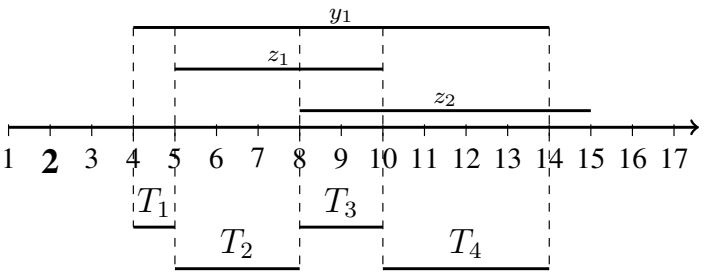


Figure 2.3.: Temporal Splitter

Example 1 *Figure 2.3 illustrates the temporal splitter applied on the tuples from Figure 2.2. We set $r = y_1$ and $\boldsymbol{g} = z_1, z_2$. The result are four disjoint time intervals T_1 to T_4, whose union builds the time interval of r. The time intervals T_2 and T_4 are both completely contained in the time interval of one of the tuples in $\boldsymbol{g}$. T_3 is completely contained in the time intervals of both tuples in $\boldsymbol{g}$, whereas T_1 is completely disjoint from the time intervals of all tuples of $\boldsymbol{g}$. In addition, all resulting time intervals are a subinterval of the time interval of tuple y_1 and cannot be enlarged without violating the first condition.*

The normalization function is a function that satisfies the properties of a Temporal Splitter. The following definition is an adaptation of the normalization function of Toman [15] cited in [2] to our TPDB.

Definition 2 (*Normalization*) Let **r** and **s** be temporal probabilistic relations. The *normalization* $\mathcal{N}_B^{TP}(\mathbf{r}; \mathbf{s})$ of **r** with respect to **s** and attributes $B \subseteq r.A$ is defined as follows:

$$\tilde{r} \in \mathcal{N}_B^{TP}(\mathbf{r}; \mathbf{s}) \iff$$
$$\exists r \in \mathbf{r}(\tilde{r}.A = r.A \wedge \tilde{r}.\lambda = r.\lambda \wedge \tilde{r}.T \in split(r, s \in \mathbf{s} | s.B = r.B)).$$

The normalization function aims to adjust the time intervals of tuples of two relations in a way that they match with all tuples with the same standard attributes. The following propositions are adapted from Dignös et al. [2] and explain this feature.

Proposition 1 *[2] Assume temporal probabilistic relations $\boldsymbol{r}$ and the temporal probabilistic normalization $\tilde{\boldsymbol{r}} = \mathcal{N}_B^{TP}(\boldsymbol{r}; \boldsymbol{r})$. All tuples $\tilde{r} \in \tilde{\boldsymbol{r}}$ with the same B-values have time intervals that are either equal or disjoint.*

Proposition 2 *[2] Assume temporal probabilistic relations $\boldsymbol{r}$ and $\boldsymbol{s}$ over schema R^{TP} and the temporal probabilistic normalizations $\tilde{\boldsymbol{r}} = \mathcal{N}_A^{TP}(\boldsymbol{r}; \boldsymbol{s})$ and $\tilde{\boldsymbol{s}} = \mathcal{N}_A^{TP}(\boldsymbol{s}; \boldsymbol{r})$. Any two tuples $\tilde{r} \in \tilde{\boldsymbol{r}}$ and $\tilde{s} \in \tilde{\boldsymbol{s}}$ with matching standard attributes A have time intervals that are either equal or disjoint.*

Example 2 *Figure 2.4 presents the result of the temporal probabilistic normalization applied on the relations from Figure 2.2.*

$\mathcal{N}_A^{TP}(\mathbf{Y};\mathbf{Z})$

A	T	P	λ
a	[4, 5)	0.50	y_1
a	[5, 8)	0.50	y_1
a	[8, 10)	0.50	y_1
a	[10, 14)	0.50	y_1

$\mathcal{N}_A^{TP}(\mathbf{Z};\mathbf{Y})$

A	T	P	λ
a	[5, 10)	0.70	z_1
a	[8, 14)	0.20	z_2
a	[14, 15)	0.20	z_2

$\mathcal{N}_A^{TP}(\mathbf{Z};\mathbf{Z})$

A	T	P	λ
a	[5, 8)	0.70	z_1
a	[8, 10)	0.70	z_1
a	[8, 10)	0.20	z_2
a	[10, 15)	0.20	z_2

Figure 2.4.: Normalization

2.2.2. Temporal Aligner and Alignment

For tuple based operators Dignös et al. [2] propose a Temporal Aligner to adjust the time intervals. It is necessary that the time intervals of the tuples of the argument relations are aligned to all other tuples that satisfy the same conditions. This means that a tuple is split into tuples with identical non-temporal attributes but with adjusted not necessarily disjoint time intervals, where the union of all adjusted time intervals equals the initial one.

Definition 3 (*Temporal Aligner [2]*) Let r be a tuple and $\mathbf{g}$ a set of tuples. A *temporal aligner* produces a set of tuples with the non-temporal attributes (A, P and λ) of r over the following adjusted intervals:

$$
\begin{aligned}
&T \in align(r, \mathbf{g}) \Longleftrightarrow \\
&\quad \exists g \in \mathbf{g}(T = r.T \cap g.T) \wedge T \neq \emptyset \vee \\
&\quad T \subseteq r.T \wedge \forall g \in \mathbf{g}(g.T \cap T = \emptyset) \wedge \\
&\quad \forall T' \supset T(\exists g \in \mathbf{g}(T' \cap g.T \neq \emptyset) \vee T' \not\subseteq r.T).
\end{aligned}
$$

The second line requires that each time interval in a tuple in $\mathbf{g}$ that intersects with the time interval of r is part of the adjusted time intervals. And the last two lines handle time intervals in r that are disjoint to the time intervals of all tuples in $\mathbf{g}$, and they are also required to be maximal.

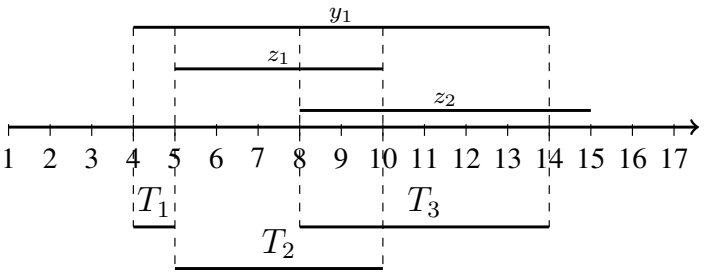


Figure 2.5.: Temporal Aligner

Example 3 *Figure 2.5 illustrates the temporal aligner applied on the tuples from Figure 2.2. We set $r = y_1$ and $\mathbf{g} = z_1, z_2$. The result are three time intervals T_1 to T_3. The time intervals*

*T_2 and T_3 are the intersections of the time interval of tuple y_1 with z_1 respectively z_2. T_1 is a subinterval of the time interval of tuple y_1, which is maximal and disjoint to all time intervals of the tuples in **g**.*

The alignment function is a function that satisfies the properties of a Temporal Aligner. The following definition is an adaptation of the Temporal Alignment function of Dignös et al. [2] to our TPDB.

Definition 4 (*Alignment*) Let **r** and **s** be temporal probabilistic relations and θ be a predicate over the standard attributes of a tuple in **r** and a tuple in **s**. The *alignment* operator $\mathbf{r}\Phi_\theta^{TP}\mathbf{s}$ of **r** with respect to **s** and condition θ is defined as follows:

$$\tilde{r} \in \mathbf{r}\Phi_\theta^{TP}\mathbf{s} \iff$$
$$\exists r \in \mathbf{r}(\tilde{r}.A = r.A \wedge \tilde{r}.\lambda = r.\lambda \wedge \tilde{r}.T \in align(r, s \in \mathbf{s}|\theta(r,s))).$$

The alignment function aims to adjust the time intervals of tuples of two relations in a way that they can be matched. The following proposition is adapted from Dignös et al. [2] and explains this feature.

Proposition 3 *[2] Assume temporal probabilistic relations **r** and **s** with temporal probabilistic alignments $\tilde{\boldsymbol{r}} = \boldsymbol{r}\Phi_\theta^{TP}\boldsymbol{s}$ and $\tilde{\boldsymbol{s}} = \boldsymbol{s}\Phi_\theta^{TP}\boldsymbol{r}$. For any two tuples $r \in \boldsymbol{r}$ and $s \in \boldsymbol{s}$ that satisfy θ and $r.T \cap s.T \neq \emptyset$, there are two tuples $\tilde{r} \in \tilde{\boldsymbol{r}}$ and $\tilde{s} \in \tilde{\boldsymbol{s}}$ with matching standard attributes for r and s, respectively, and with identical timestamps $\tilde{r}.T = \tilde{s}.T = r.T \cap s.T$.*

Example 4 *Figure 2.6 presents the result of the temporal probabilistic alignment applied on the relations from Figure 2.2.*

$\mathbf{Y}\Phi_{true}^{TP}\mathbf{Z}$

A	T	P	λ
a	[4, 5)	0.50	y_1
a	[5, 10)	0.50	y_1
a	[8, 14)	0.50	y_1

$\mathbf{Z}\Phi_{true}^{TP}\mathbf{Y}$

A	T	P	λ
a	[5, 10)	0.70	z_1
a	[8, 14)	0.20	z_2
a	[14, 15)	0.20	z_2

Figure 2.6.: Alignment

2.3. Reduction Rules

The probability value of tuples cannot simply be computed for every operation and then be propagated to the next operator. This is because dependencies might occur in operations that are applied later on. Therefore, we compute lineage during algebra operations, so that we can later elicit the base tuples, from which the result tuples were derived from. This allows us to compute the correct probability value for each result tuple.

Operator			Reduction
Selection	$\sigma_\theta^{TP}(\mathbf{r})$	$=$	$\sigma_\theta(\mathbf{r})$
Projection	$\pi_B^{TP}(\mathbf{r})$	$=$	${}_{B,T}\vartheta_{OR(\lambda)}(\mathcal{N}_B^{TP}(\mathbf{r};\mathbf{r}))$
Normalization	$\mathcal{N}_B^{TP}(\mathbf{r};\mathbf{s})$	$=$	$\pi_{r.A,r.T,r.\lambda}(\mathcal{N}_B(\mathbf{r};\mathbf{s}))$
Alignment	$\mathbf{r}\,\Phi_\theta^{TP}\mathbf{s}$	$=$	$\pi_{r.A,r.T,r.\lambda}(\mathbf{r}\Phi_\theta\mathbf{s})$
High Aggregation	${}_B\vartheta_F^{TP}(\mathbf{r})$	$=$	${}_{B,T}\vartheta_{F,AND(\lambda)}(\mathcal{N}_B^{TP}(\mathbf{r};\mathbf{r}))$
Difference	$\mathbf{r} -^{TP} \mathbf{s}$	$=$	${}_{r.A,r.T}\vartheta_{OR(AND(r.\lambda,NOT(s.\lambda)))}((\mathcal{N}_A^{TP}(\mathbf{r};\mathcal{N}_A^{TP}(\mathbf{r};\mathbf{s})))\ \overline{\bowtie}_{r.A=s.A\wedge r.T=s.T}(\mathcal{N}_A^{TP}(\mathbf{s};\mathcal{N}_A^{TP}(\mathbf{s};\mathbf{r}))))$
Union	$\mathbf{r} \cup^{TP} \mathbf{s}$	$=$	${}_{r.A,r.T}\vartheta_{OR(r.\lambda)}(\mathcal{N}_A^{TP}(\mathbf{r};\mathcal{N}_A^{TP}(\mathbf{r};\mathbf{s})) \cup \mathcal{N}_A^{TP}(\mathbf{s};\mathcal{N}_A^{TP}(\mathbf{s};\mathbf{r})))$
Intersection	$\mathbf{r} \cap^{TP} \mathbf{s}$	$=$	${}_{r.A,r.T}\vartheta_{OR(AND(r.\lambda,s.\lambda))}((\mathcal{N}_A^{TP}(\mathbf{r};\mathcal{N}_A^{TP}(\mathbf{r};\mathbf{s})))\bowtie_{r.A=s.A\wedge r.T=s.T}(\mathcal{N}_A^{TP}(\mathbf{s};\mathcal{N}_A^{TP}(\mathbf{s};\mathbf{r}))))$
Cartesian Product	$\mathbf{r} \times^{TP} \mathbf{s}$	$=$	$\pi_{r.A,r.T,s.A,s.T,AND(r.\lambda,s.\lambda)}((\mathbf{r}\Phi_{true}^{TP}\mathbf{s})\bowtie_{r.T=s.T}(\mathbf{s}\Phi_{true}^{TP}\mathbf{r}))$
Join	$\mathbf{r} \bowtie_\theta^{TP} \mathbf{s}$	$=$	$\pi_{r.A,r.T,s.A,s.T,AND(r.\lambda,s.\lambda)}((\mathbf{r}\Phi_\theta^{TP}\mathbf{s})\bowtie_{\theta\wedge r.T=s.T}(\mathbf{s}\Phi_\theta^{TP}\mathbf{r}))$

Figure 2.7.: Reduction Rules

Lineage is computed along with the query operations as also proposed by Sarma et al. [13]. During each algebra operation, the lineage of the input tuples is combined using the following connectives:

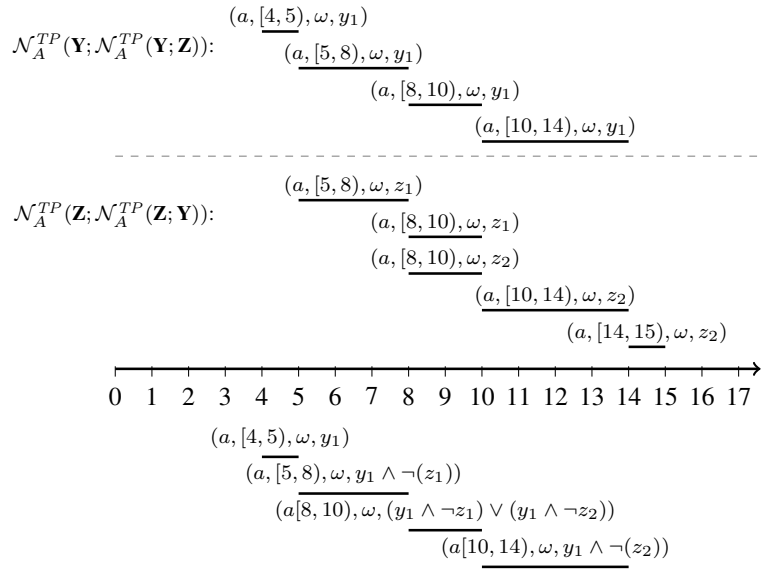
- $\wedge$**-connective:** The $\wedge$-connective is used when two tuples are combined and both need to be valid, so that the resulting tuple is valid, too.
- $\vee$**-connective:** The $\vee$-connective is used when two tuples are combined and only one of both needs to be valid, so that the resulting tuple is valid, too.
- $\neg$**-connective:** The $\neg$-connective is applied when the resulting tuple is only valid, if the input tuple is invalid.

Using those rules, we define reduction rules for a temporal probabilistic algebra with sequenced semantics, which show how temporal probabilistic algebra operations can be performed by using standard algebra operations. These rules take care of a correct lineage computation during algebra operations. Please note that we ignore the probability values during algebra operations, since the values might not be correct for intermediate tuples.

Theorem 1 *Let* $\mathbf{r}$ *and* $\mathbf{s}$ *be temporal probabilistic relations over schema* R^{TP}*,* θ *be a predicate,* F *be a set of aggregation functions over* $r.A$*,* $B \subseteq A$ *be a set of attributes and* OR*,* AND *and* NOT *functions respectively aggregation functions that combine the given lineage expressions with the defined connective. The reduction rules in Figure 2.7 define a temporal probabilistic algebra with sequenced semantics.*

The complete set of reduction rules is shown in Appendix A. This set includes the rules for outer joins and additional rules for set operations, which do not eliminate duplicates. Those are important for the implementation, since both types are available in SQL.

We now present an example of a query that uses the temporal probabilistic normalization to adjust the time intervals of tuples. Normalization is required when groups of tuples with matching arguments need to be combined. In this example we also see that the outcome of a temporal probabilistic difference operation can include more tuples than the initial input relation. This has two reasons. The first is that by applying normalization the tuples get split into multiple pieces, which all are part of the resulting tuples. The second is, that no tuples from the left input relation can be for sure removed, because the validity of the matching tuple from the right relation is not certain, since every tuple has a probability value attached. Note that we illustrate the examples on a timeline, which shows the time intervals during which the presented tuples might be valid.



$\mathcal{N}_A^{TP}(\mathbf{Y};\mathcal{N}_A^{TP}(\mathbf{Y};\mathbf{Z}))$

A	T	P	λ
a	$[4,5)$	ω	y_1
a	$[5,8)$	ω	y_1
a	$[8,10)$	ω	y_1
a	$[10,14)$	ω	y_1

$\mathcal{N}_A^{TP}(\mathbf{Z};\mathcal{N}_A^{TP}(\mathbf{Z};\mathbf{Y}))$

A	T	P	λ
a	$[5,8)$	ω	z_1
a	$[8,10)$	ω	z_1
a	$[8,10)$	ω	z_2
a	$[10,14)$	ω	z_2
a	$[14,15)$	ω	z_2

$\mathbf{Y} -^{TP} \mathbf{Z}$

A	T	P	λ
a	$[4,5)$	ω	y_1
a	$[5,8)$	ω	$y_1 \wedge \neg(z_1)$
a	$[8,10)$	ω	$(y_1 \wedge \neg z_1) \vee (y_1 \wedge \neg z_2)$
a	$[10,14)$	ω	$y_1 \wedge \neg(z_2)$

Figure 2.8.: $\mathbf{Y} -^{TP} \mathbf{Z}$

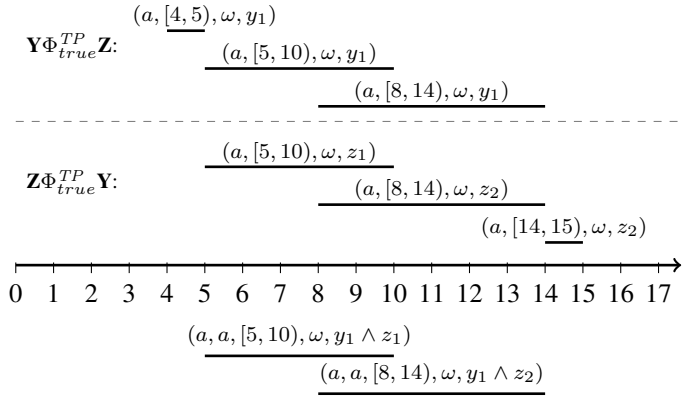
Example 5 *Figure 2.8 illustrates the reduction of the temporal probabilistic difference query* $\boldsymbol{Y} -^{TP} \boldsymbol{Z}$*. As a first step the tuples of both relations need to be split, in such a way that time intervals are either the same or disjoint. For this two normalizations are necessary. First both relations are normalized with respect to the other relation, which results in the tuples shown in Figure 2.4. And then the relations are again normalized with respect to the results from before. The resulting tuples are shown in the tables in Figure 2.8.*

As next step, a left outer join is performed. Since all tuples have the same standard attributes, all four tuples that were derived from y_1 *are joined with the tuples that were derived from relation* **Z***. This operation results in five tuples. The first tuple with the time interval* $[4, 5)$ *did not join with any tuple from* **Z***, therefore the lineage is still the same as for the initial tuple. The tuples with time intervals* $[5, 8)$ *and* $[10, 14)$ *both joined with one tuple and the tuple with the time interval* $[8, 10)$ *matched with two tuples from* **Z***. For those four resulting tuples the lineage is combined with the* $\wedge$*-connective while the lineage of the tuple from the right relation is negated first. By definition of the difference operator, a tuple is only part of the result if it occurs in the left relation but not in the right relation. But in a temporal probabilistic*

database, a tuple can also be in the result tuple if it occurs in the left and in the right relation, but only if the tuple in the left relation evaluates to valid and the tuple in the right relation evaluates to invalid.

Finally, duplicates need to be eliminated, since the difference operator in relational algebra is defined to be duplicate eliminating. Therefore, the tuples are grouped according to their standard attributes and their time interval. In this example there are two tuples with standard attribute a and time interval $[8, 10)$, those are therefore combined into one tuple and their lineages are combined with the $\vee$-connective. Because only one of the tuples needs to be valid in order for the resulting tuple to be valid too. The complete query results in four result tuples. They are shown in Figure 2.8 beyond the timeline and in the table aside.

As a second example, we show a query that uses the temporal probabilistic alignment operator to adjust the time intervals of the input tuples. Alignment is needed when a tuple of one relation needs to be combined with only one tuple from another relation.



A	B	T	P	λ
a	a	$[5, 10)$	ω	$y_1 \wedge z_1$
a	a	$[8, 14)$	ω	$y_1 \wedge z_2$

Figure 2.9.: $\mathbf{Y} \bowtie^{TP}_{true} \mathbf{Z}$

Example 6 *Figure 2.9 illustrates the reduction of the temporal probabilistic join $\boldsymbol{Y} \bowtie^{TP}_{true} \boldsymbol{Z}$. As first step, the temporal probabilistic alignment operator is applied on both relations with respect to the other relation. This results in the tuples shown in Figure 2.6.*

Afterwards the join over these new tuples is performed. One tuple from $\boldsymbol{Y}\Phi^{TP}_{true}\boldsymbol{Z}$ does not match with any tuple from $\boldsymbol{Z}\Phi^{TP}_{true}\boldsymbol{Y}$. Since a inner join is performed, this tuple is ignored. The other two tuples have matching counterparts. Therefore the standard attributes are both copied and the lineages of both sides are combined with the $\wedge$-operator. This is because for the tuple to be true both base tuples from which it was derived, need to be true as well. In Figure 2.9, the resulting tuples are shown beyond the timeline and in the table aside.

3. Confidence Computation

One might think that confidence can just be calculated by computing it within each operator and then propagating the resulting confidence to the next operator in the query. However, this does not work for all queries. Let us consider an example of two relationally equivalent queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$, applied on the relations from Figure 2.2. For simplification we ignore the temporal attributes here. First consider:

$$\mathcal{Q}_1 = \pi^{TP}_{Name}(\mathbf{V} \bowtie^{TP}_{V.Dest=W.Loc} \mathbf{W})$$

$\mathbf{V} \bowtie^{TP}_{V.Dest=W.Loc} \mathbf{W}$ produces two intermediate tuples (Ann, $Zurich$, $Zurich$, $Snow$) and (Ann, $Zurich$, $Zurich$, Fog), where the confidence value of the first tuple is $0.5 * 0.7 = 0.35$ and of the second tuple is $0.5 * 0.2 = 0.1$. When also considering lineage, we have $v_1 \wedge w_1$ for the first and $v_1 \wedge w_2$ for the second tuple. On these two intermediate tuples the projection on $Name$ is performed. This results in one tuple (Ann) and its confidence value is computed as $1 - ((1 - 0.35) * (1 - 0.1)) = 0.415$ and its lineage as $(v_1 \wedge w_1) \vee (v_1 \wedge w_2)$.

Now we consider the relationally equivalent query:

$$\mathcal{Q}_2 = \pi^{TP}_{Name}(\mathbf{V} \bowtie^{TP}_{V.Dest=W.Loc} (\pi^{TP}_{Loc}(\mathbf{W})))$$

$\pi^{TP}_{Loc}(\mathbf{W})$ produces one intermediate tuple ($Zurich$) with a confidence value of $1 - ((1 - 0.7) * (1 - 0.2)) = 0.76$ and a lineage of $w_1 \vee w_2$. Joining this tuple with the tuple from relation $\mathbf{V}$ results in the tuple (Ann, $Zurich$, $Zurich$) with lineage $v_1 \wedge (w_1 \vee w_2)$, whose confidence value can be computed as $0.5 * 0.76 = 0.38$. Finally, we project onto $Name$ to produce the result tuple (Ann) with a confidence value of 0.38 and lineage $v_1 \wedge (w_1 \vee w_2)$.

We can see that the confidence value of the resulting tuple is not the same for both queries. Since we do not want to restrict the choice of queries, we need another method to compute the confidence value. As seen above, do both queries result in a tuple with a logically equivalent lineage expression. We therefore use lineage to compute the confidence value in the approach we present here.

3.1. Preliminaries

Base variable: We call the random variables in a lineage expression, which are used to uniquely identify a base tuple, base variables. Please note that the same base variable might

appear multiple times in a lineage expression. We define $F(\lambda)$ to deliver the set of base variables that are contained in the lineage expression λ.
Negation Normal Form: A lineage expression is in negation normal form (*NNF*) if negation is only applied on base variables.
One-Occurrence Form: A lineage expression is in one-occurrence form (*1OF*) if each base variable only occurs once in the whole lineage expression.
Independence: Two lineage expressions are told to be independent of each other if none of their base variables occurs in both lineage expressions. With λ_1 and λ_2 as lineage expressions we define independence as $F(\lambda_1) \cap F(\lambda_2) = \emptyset$.
Unate and Positive Form: A lineage expression in *NNF* is unate, if each base variable either occurs only positive or only negative. And a lineage expression in *NNF* is positive, if each base variable appears positive at least once. Lineage expressions in one of those forms can trivially be converted into an equivalent expression in the other form [5].

The table in Figure 3.1 shows different lineage expressions and in which form these are.

	NNF	*1OF*	independenct to $\lambda = C$	unate	positive
$A \vee B \vee C$	$\surd$	$\surd$	$\times$	$\surd$	$\surd$
$A \vee \neg A \vee B$	$\surd$	$\times$	$\surd$	$\times$	$\surd$
$A \vee \neg B \vee C$	$\surd$	$\surd$	$\times$	$\surd$	$\times$
$\neg(A \vee B)$	$\times$	$\surd$	$\surd$	$\times$	$\times$

Figure 3.1.: Lineage expressions in different forms

All algorithms presented in the following sections compute the confidence value out of a lineage expression. In order to get the probability value of all tuples in the relation computed, one of the algorithms needs to be executed for every single result tuple. All of the algorithms take as their argument a lineage expression and they return either the exact confidence value that is represented by the given lineage or they return a small interval in which the exact confidence value must lie.

For the ease of the algorithms we represent lineage expressions in a tree structure. All leaf nodes are base variables, which are connected via the internal nodes. Those internal nodes represent the Boolean connectives $\wedge$, $\vee$ and $\neg$. Each $\wedge$-node and $\vee$-node has two children,

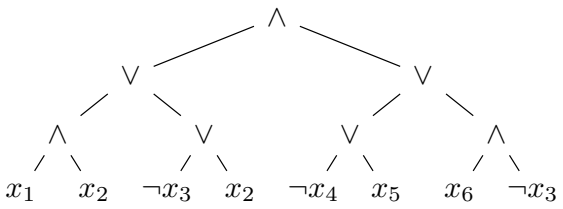


Figure 3.2.: $\mathcal{B} = (x_1 \wedge x_2 \vee \neg x_3 \vee x_2) \wedge (\neg x_4 \vee x_5 \vee x_6 \wedge \neg x_3)$

which are either other internal nodes or leaf nodes. On the other hand, the ¬-nodes only have one child, which again can be either a internal node or a leaf node.

Figure 3.2 shows an example lineage expression $\mathcal{B}$ that is in *NNF* and it will be used in the following to analyse the four algorithms. With $x_i.P$ we access the probability value that is stored in the tuple with the identifier x_i. For numerical calculations we use the following probabilities: $x_1.P = 0.1, x_2.P = 0.2, x_3.P = 0.3$, $x_4.P = 0.4$, $x_5.P = 0.5$ and $x_6.P = 0.6$. Please note that for simplification all ¬-nodes, which are followed by a leaf node, are represented as one node in all figures.

3.2. Exact Probability Computation

This section shows three algorithms for computing the exact probability of result tuples. The first algorithm that is presented takes a naive approach. It simply evaluates each possible world in which the tuple might be valid and computes the probability of those worlds being valid. The other two algorithms decompose the parts of the lineage expressions that are too complex into an equivalent and easy computable lineage expressions.

3.2.1. Brute Force Algorithm

As each base variable represents a tuple, which has a probability of being true, we can compute the final confidence by evaluating for which allocations of the base variables the lineage expression evaluates to true. An allocation assigns each base variable in a lineage expression either the value $true$ or $false$.

Example 7 *For simplicity reasons, we will illustrate this process on a part of the lineage expression in Figure 3.2. Suppose we take the left part of the lineage expression* $\mathcal{B}\rightarrow$*left* $= x_1 \wedge x_2 \vee \neg x_3 \vee x_2$*, and check if the allocation* $\{x_1 = true, x_2 = true, x_3 = false\}$ *evaluates to true. We therefore replace every occurrence of* x_i *in* $\mathcal{B}\rightarrow$*left with its allocation* $true \wedge true \vee \neg false \vee true \equiv true$*, which evaluates to true. But the allocation* $\{x_1 = true, x_2 = false, x_3 = true\}$ *evaluates to false:* $true \wedge false \vee \neg true \vee false \equiv false$*.*

x_1	x_2	x_3	$Eval(\mathcal{B}\rightarrow$*left*$)$
$false$	$false$	$false$	$true$
$false$	$false$	$true$	$false$
$false$	$true$	$false$	$true$
$false$	$true$	$true$	$true$
$true$	$false$	$false$	$true$
$true$	$false$	$true$	$false$
$true$	$true$	$false$	$true$
$true$	$true$	$true$	$true$

Figure 3.3.: Evaluated truth table for $\mathcal{B}\rightarrow$*left* $= x_1 \wedge x_2 \vee \neg x_3 \vee x_2$

Figure 3.3 shows the truth table that is formed by all possible allocations for the base variables of $\mathcal{B}\rightarrow$*left*. To compute the final confidence we only need to consider those allocations for which the lineage evaluates to true, since for all other combinations the tuple with the lineage $\mathcal{B}\rightarrow$*left* would not be valid.

For further calculations we need to be able to compute how likely it is that an allocation occurs. This can be calculated by multiplying the probability of every base variable to be either $true$ or $false$ depending on the allocation. The probability of a base variable x_i to be $true$ can be retrieved by accessing the probability value $x_i.P$ of the tuple that is identified by the given base variable. On the other hand, we have $1 - x_i.P$ indicating the probability of the base variable x_i to be $false$. For every allocation that evaluates to true we thus compute the probability of this combination happening, by multiplying the corresponding probability values.

If we sum up all those probabilities we finally get the probability of the tuple with lineage $\mathcal{B}\rightarrow$*left* to be valid. Figure 3.4 shows the table that is used to compute the final confidence value. Please note that '$-$' relates to rows for which we did not compute the confidence, as the evaluation of the corresponding allocation resulted in false.

The set-up of the truth table assures that the final confidence will be zero in case that the lineage evaluates to false for all possible allocations. On the other hand, the probability will be one if lineage evaluates to true for all possible allocations. As no outcome can exceed this boundaries, the computed confidence will always be between 0 and 1.

x_1	x_2	x_3	$P(\mathcal{B}\rightarrow$*left*$)$
0.9	0.8	0.7	0.504
$-$	$-$	$-$	$-$
0.9	0.2	0.7	0.126
0.9	0.2	0.3	0.054
0.1	0.8	0.7	0.056
$-$	$-$	$-$	$-$
0.1	0.2	0.7	0.014
0.1	0.2	0.3	0.006
		Sum:	0.76

Figure 3.4.: Confidence computation for $\mathcal{B}\rightarrow$*left* $= x_1 \wedge x_2 \vee \neg x_3 \vee x_2$

Example 8 *Since the lineage expression $\mathcal{B}\rightarrow$left from Figure 3.2 with the allocation $\{x_1 = true, x_2 = true, x_3 = false\}$ evaluates to true, we compute the probability of this allocation to be valid by: $0.1 * 0.2 * 0.7 = 0.014$. For the variable x_3, which is set to false, $1 - x_3.P$ is taken and for all other variables, which are set to true, $x_i.P$ is taken. By summing up all the probabilities of allocations that evaluate to true, we get a probability of 0.76 for the tuple with lineage expression $\mathcal{B}\rightarrow$left.*

We define f to be the number of different base variables of a lineage λ. Formally we have $f = |F(\lambda)|$. The complexity of this algorithm is $O(2^f)$, because all possible combinations, where the f distinct base variables are either set to $true$ or $false$, must be evaluated. This gives a total number of 2^f combinations. For a relation with n tuples this results in a total complexity of $O(n * 2^f)$.

3.2.2. Decomposition Algorithm

Fink et al. [4] suggest an algorithm to compute confidence without unfolding the lineage expression into disjunctive normal form. This algorithm is able to compute confidence from all lineage expressions that are possible with our temporal probabilistic algebra. The basic idea of this algorithm is to decompose complex parts of the lineage expression into easily computable but equivalent lineage expressions. We adapted their algorithm in such a way that it works with lineage expressions in tree structure as we have them in out system.

(a) Lineage expression not in *NNF*

(b) The same lineage expression in *NNF*

Figure 3.5.: Negation Normal Form (*NNF*)

As a precondition of our algorithm the lineage expression must be brought into *NNF*. The lineage expressions that are constructed by our temporal probabilistic algebra do not necessarily fulfil this condition. Therefore a algorithm was created that solves this problem. This algorithm recursively pushes negations down, until $\neg$-nodes only have a leaf nodes as their child. Figure 3.5 shows a lineage expression, which is not in *NNF* and how it is changed to be in *NNF*. Algorithm 6 in Appendix B shows an algorithm that performs this transformation.

Let $P(\lambda)$ be a function that maps the lineage expression to the probability value of the tuple that is represented by the lineage expression. Olteanu et al. [12] suggest three decompositions, which are adapted to our needs:

1. **Independent-and:** If we are at a $\wedge$-node and we have $F(\lambda\rightarrow\textit{left}) \cap F(\lambda\rightarrow\textit{right}) = \emptyset$, we can compute $P(\lambda) = P(\lambda\rightarrow\textit{left}) * P(\lambda\rightarrow\textit{right})$.

2. **Independent-or:** If we are at a $\vee$-node and we have $F(\lambda\rightarrow\textit{left}) \cap F(\lambda\rightarrow\textit{right}) = \emptyset$, we can compute $P(\lambda) = 1 - (1 - P(\lambda\rightarrow\textit{left})) * (1 - P(\lambda\rightarrow\textit{right}))$.

3. **Exclusive-or:** If none of the above decompositions can be applied, we apply so called Shannon expansion. We choose a base variable x in λ. Then, λ is equivalent to $x \wedge \lambda|_{x=true} \vee \neg x \wedge \lambda|_{x=false}$, where $\lambda|_{x=true}$ is obtained from λ by removing all parts that are redundant when x is set to $true$. We can then compute $P(\lambda) = P(x) * P(\lambda|_{x=true}) + P(\neg x) * P(\lambda|_{x=false})$.

Algorithm 1 Computing probability of the tuple with lineage expression λ

Require: lineage expression λ in *NNF*

function PROBABILITY(lineage expression λ)
  if λ is in *1OF* **then**
    return PROBABILITY1OF(λ)
  else if λ→*left* is independent from λ→*right* **then**
    if λ→*op* = $\wedge$ **then**
      return PROBABILITY(λ→*left*) $*$ PROBABILITY(λ→*right*)
    else
      return $1 - ((1 -$ PROBABILITY(λ→*left*) $) * (1 -$ PROBABILITY(λ→*right*) $))$
    end if
  else ▷ Shannon expansion
    $x \leftarrow$ base variable that occurs most in λ
    return $x.P *$ PROBABILITY($\lambda|_{x=true}$) $+(1 - x.P) *$ PROBABILITY($\lambda|_{x=false}$)
  end if
end function

With those decompositions we can incrementally decompose the initial lineage expression into parts that are in *1OF*. This is nice, because for such lineage expressions the probability can be computed in linear time. To do this we have an function P_{1OF} that computes the probability represented by the lineage expression by recursively applying those four rules:

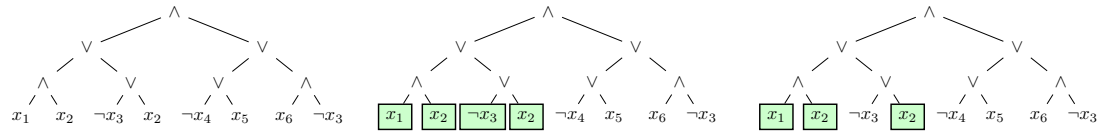
- $\wedge$**-node:** $P_{1OF}(\lambda) = P_{1OF}(\lambda\text{→}\textit{left}) * P_{1OF}(\lambda\text{→}\textit{right})$.
- $\vee$**-node:** $P_{1OF}(\lambda) = 1 - ((1 - P_{1OF}(\lambda\text{→}\textit{left})) * (1 - P_{1OF}(\lambda\text{→}\textit{right})))$.
- $\neg$**-node:** $P_{1OF}(\lambda) = 1 - P_{1OF}(\lambda\text{→}\textit{left})$.
- **leaf node:** $P_{1OF}(\lambda) = \lambda.P$.

Example 9 *Let us take the right sub-tree of $\mathcal{B}$, $\mathcal{B}$→right $= (\neg x_4 \vee x_5) \vee (x_6 \wedge \neg x_3)$, which is in 1OF, as an example and let us compute $P(\mathcal{B}$→right$)$. At the root node we have a $\vee$, we therefore have $P_{1OF}(\mathcal{B}$→right$) = 1 - ((1 - P_{1OF}(\neg x_4 \vee x_5)) * (1 - P_{1OF}(x_6 \wedge \neg x_3)))$. We further compute $P_{1OF}(\neg x_4 \vee x_5) = 1 - ((1 - P_{1OF}(\neg x_4)) * (1 - P_{1OF}(x_5)))$, since we have another $\vee$-node there and $P_{1OF}(x_6 \wedge \neg x_3) = P_{1OF}(x_6) * P_{1OF}(\neg x_3)$ because we have a $\wedge$-node there. We are now left with two calls of P_{1OF} on base variables for which we simply take the probability value P of the corresponding tuples $P_{1OF}(x_5) = x_5.P$ and $P_{1OF}(x_6) = x_6.P$ and two calls of P_{1OF} on a $\neg$-node which resolve in $P_{1OF}(\neg x_4) = 1 - P_{1OF}(x_4) = 1 - x_4.P$ and $P_{1OF}(\neg x_3) = 1 - P_{1OF}(x_3) = 1 - x_3.P$.*

The whole expression then results in $P_{1OF}(\mathcal{B}$→right$) = 1 - ((1 - (1 - ((1 - (1 - x_4.P)) * (1 - x_5.P)))) * (1 - (x_6.P * (1 - x_3.P)))) = 1 - ((1 - (1 - ((1 - (1 - 0.4)) * (1 - 0.5)))) * (1 - (0.6 * (1 - 0.3)))) = 0.884$.

An algorithm that implements function P_{1OF} is sketched in Appendix B. With the use of this algorithm and the previously defined decompositions, we come up with an algorithm that can compute the probability value for every possible lineage expression. This algorithm is

shown in Algorithm 1. Basically, we first check if the given lineage expression is in *1OF*. If this is the case, we call P_{1OF} with the lineage expression as its argument and otherwise we apply one of the decompositions. Since Shannon expansion can always be applied and leads to less base variables in the lineage expression, we never reach a dead end.



(a) $\mathcal{B} = (x_1 \wedge x_2 \vee \neg x_3 \vee x_2) \wedge (\neg x_4 \vee x_5 \vee x_6 \wedge \neg x_3)$ (b) $\mathcal{B}|_{x_2=true} = (\neg x_4 \vee x_5) \vee (x_6 \wedge \neg x_3)$ (c) $\mathcal{B}|_{x_2=false} = (\neg x_3) \wedge (\neg x_4 \vee x_5 \vee (x_6 \wedge \neg x_3))$

Figure 3.6.: Shannon cofactors

We are left with investigating Shannon expansion, which is a method to decompose the lineage expression into simpler parts by extracting a base variable. Shannon expansion is defined as $x \wedge \lambda|_{x=true} \vee \neg x \wedge \lambda|_{x=false}$, where x is the base variable according to which the Shannon cofactors $\lambda|_{x=true}$ and $\lambda|_{x=false}$ are built. As first step we need to consider, which base variable should be taken for the Shannon expansion. [5, 12] both suggest to use the base variable that occurs the most in the whole lineage expression, when considering non-tractable queries. Since we decoupled query execution and confidence computation in our work, we adapt this suggestion.

After choosing the decoupling variable x, the positive and negative Shannon cofactors $(\lambda|_{x=true}, \lambda|_{x=false})$ for the given lineage expression λ with respect to x need to be computed. The intuition behind the Shannon cofactors is that all occurrences of the base variable x in λ are set to $true$ or $false$ respectively. Because of this, more parts of the lineage expression might become redundant. To be exact, if one child of an $\wedge$-node is set to $false$ it does not matter what the other child is, the expression will always evaluate to $false$. For $\vee$-nodes we have redundancy if one child is set to $true$, in this case the whole expression will always evaluate to $true$.

Finally, if the Shannon cofactors are computed we combine them in such a way that the new expression is equal to the original one, which is the solution of the Shannon expansion $\lambda = x \wedge \lambda|_{x=true} \vee \neg x \wedge \lambda|_{x=false}$.

Algorithm 2 sketches an algorithm that creates the Shannon cofactors for a given lineage expression. The algorithm recursively investigates the nodes of the given lineage expression and creates a copy of those which are not redundant because of the setting of the base variable x. The function returns *NULL* if the underlying lineage expression for sure evaluates to $true$ or $false$. If the function returns from recursion returning *NULL*, it further checks for redundancy (lines 23 and 27). Otherwise it simply returns a copy of the current node.

Algorithm 2 Computing Shannon cofactors $\lambda|_{x=true}$ and $\lambda|_{x=false}$ of a lineage expression λ

Require: lineage expression x needs to be a leaf node in λ and λ needs to be in *NNF*

bool v ▷ global variable
$\lambda|_{x=true} \leftarrow$ SHANNONCOFACTOR($\lambda, x, true$)
$\lambda|_{x=false} \leftarrow$ SHANNONCOFACTOR($\lambda, x, false$)
function SHANNONCOFACTOR(lineage expression λ, lineage expression x, bool $v_{initial}$)
    if λ→*op* = *NONE* **then** ▷ λ is leaf node
        if $\lambda = x$ **then**
            $v \leftarrow v_{initial}$
            return *NULL*
        else
            return COPYNODE(λ)
        end if
    else if λ→*op* = $\neg$ **then**
        $l \leftarrow$ SHANNONCOFACTOR(λ→*left*, x, $v_{initial}$)
        if l **then**
            return NEWNODE($\neg, l,$ *NULL*) ▷ Return a new node with *op* = $\neg$, *left* = l and *right* = *NULL*
        else
            $v \leftarrow !v$
            return *NULL*
        end if
    end if
    $l \leftarrow$ SHANNONCOFACTOR(λ→*left*, x, $v_{initial}$)
    if $!l \wedge ((v \wedge \lambda\text{→}op = \vee) \vee (!v \wedge \lambda\text{→}op = \wedge))$ **then**
        return *NULL*
    else
        $r \leftarrow$ SHANNONCOFACTOR(λ→*right*, x, $v_{initial}$)
        if $!r \wedge (!l \vee ((v \wedge \lambda\text{→}op = \vee) \vee (!v \wedge \lambda\text{→}op = \wedge)))$ **then**
            return *NULL*
        else
            if $!l$ **then**
                return r
            else if $!r$ **then**
                return l
            else
                return NEWNODE(λ→*op*, l, r)
            end if
        end if
    end if
end function

Example 10 *Let us compute the Shannon cofactors for lineage expression $\mathcal{B}$ with respect to base variable x_2. We start with the positive Shannon cofactor $\mathcal{B}|_{x_2=true}$. See Figure 3.6b for an illustration. We set all occurrences of x_2 to $true$ and see what happens to the rest of the expression. We start at the left x_2-node and go up one level. Since we are at a $\wedge$-node and $v = true$, we cannot delete this part. In other words, the evaluation of the expression depends on the left part. Therefore, we stop there and go to the next x_2-node. There we go up one level and we come to a $\vee$-node. Since the right side of the $\vee$-node is $true$ we know that the whole expression will always be $true$, no matter what the left part is. We therefore delete this part and go up one level more. There we have a $\vee$-node again and the same happens. Then we go up again. We are now at the root node, which is a $\wedge$-node, and we have to stop there, since we do not know to what it evaluates for sure. At this point, we have no more x_2-nodes left to check. So we are left with $\mathcal{B}|_{x_2=true} = (\neg x_4 \vee x_5) \vee (x_6 \wedge \neg x_3)$. All parts in Figure 3.6b that are marked in green, get redundant because x_2 was set to $true$.*

The same procedure is executed for $\mathcal{B}|_{x_2=false}$. In this case only the leaf nodes with variable x_2 and the leaf node with variable x_1 gets redundant. We are therefore left with $\mathcal{B}|_{x_2=false} = (\neg x_3) \wedge (\neg x_4 \vee x_5 \vee (x_6 \wedge \neg x_3))$, see Figure 3.6c for an illustration.

Combined this results in $P(\mathcal{B}) = P(x_2) * P((\neg x_4 \vee x_5) \vee (x_6 \wedge \neg x_3)) + P(\neg x_2) * P((\neg x_3) \wedge (\neg x_4 \vee x_5 \vee (x_6 \wedge \neg x_3)))$.

Let us now consider en example that shows the usage of the complete probability computation algorithm:

Example 11 *We take the lineage expression $\mathcal{B}$ from Figure 3.2 and compute $P(\mathcal{B})$. As a first step we check if $\mathcal{B}$ is in 1OF. This is not the case, since the base variables x_2 and x_3 occur twice in the whole lineage expression. As a second step we check if $\mathcal{B}\rightarrow$left is independent from $\mathcal{B}\rightarrow$ right, but since x_3 occurs in both parts, this is not true either. Therefore, we will apply Shannon expansion. Since x_2 and x_3 both occur twice in $\mathcal{B}$, we are free to choose one of those variables to calculate the Shannon cofactors. If we chose x_2 this results, as we saw in the previous example, in $P(\mathcal{B}) = P(x_2) * P((\neg x_4 \vee x_5) \vee (x_6 \wedge \neg x_3)) + P(\neg x_2) * P((\neg x_3) \wedge (\neg x_4 \vee x_5 \vee (x_6 \wedge \neg x_3)))$.*

We can now recursively solve all newly occurring calls of P. In three of the four calls the given lineage expression is in 1OF, for those we therefore can simply compute their probability by calling P_{1OF}. This results in $P_{1OF}(x_2) = x_2.P = 0.2$, $P_{1OF}((\neg x_4 \vee x_5) \vee (x_6 \wedge \neg x_3)) = 0.884$ (see Example 9) and $P_{1OF}(\neg x_2) = 1 - x_2.P = 0.8$. The last expression $P((\neg x_3) \wedge (\neg x_4 \vee x_5 \vee (x_6 \wedge \neg x_3)))$ needs to be further decomposed, since it is not in 1OF. We also do not have independent parts and therefore we need to do another Shannon expansion.

The Shannon expansion with respect to x_3 results in $P(x_3) * P(false) + P(\neg x_3) * P(x_1 \wedge (\neg x_4 \vee x_5 \vee x_6))$. The positive Shannon cofactor results in $false$ since if x_3 is set to true, the whole expression can never evaluate to $true$. Now we are left only with lineage expressions which are in 1OF. We compute the probability of those as: $P(x_3) = x_3.P = 0.3$, $P(false) = 0$, $P(\neg x_3) = 1 - x_3.P = 0.7$ and $P(x_1 \wedge (\neg x_4 \vee x_5 \vee x_6)) = x_1.P * (1 - ((1 - (1 - ((1 - (1 - x_4.P)) * (1 - x_5.P)))) * (1 - x_6.P)) = 0.92$. The complete second Shannon expansion resolves in a probability of $P((\neg x_3) \wedge (\neg x_4 \vee x_5 \vee (x_6 \wedge \neg x_3))) = 0.3 * 0 + 0.7 * 0.92 = 0.644$.

Finally we can insert all values into the formula of the first Shannon expansion, which results in $P(\mathcal{B}) = 0.2 * 0.884 + .8 * 0.644 = 0.692$.

We take i as the number of nodes in lineage expression λ. As we already mentioned P_{1OF} runs in linear time, since every node needs to be visited exactly once for its execution. Therefore, its complexity is $O(i)$, Checking if the lineage expression is in *1OF* is a bit more time consuming. The algorithm that we implemented checks for every leaf node if the same base variable already occurred. Base variable that already occurred are stored in a binary search tree. Since insertion and searching in a binary search tree have complexity $O(log(n))$ in best case, we have an overall best case complexity of $O(i * log(i))$ to check if the lineage expression is in *1OF*. The same is the case for checking for independence and for finding the base variable for the Shannon expansion. In worst case the complexity is $O(i^2)$. The algorithm to compute the Shannon cofactors runs in $O(i)$ since every node needs to be visited once. Since all those computations need to be done for every node in the lineage expression, the complexity of $P(\lambda)$ is $O(i^2 * log(i))$ respectively $O(i^3)$ in the worst case.

3.2.3. Decomposition Algorithm with Theta

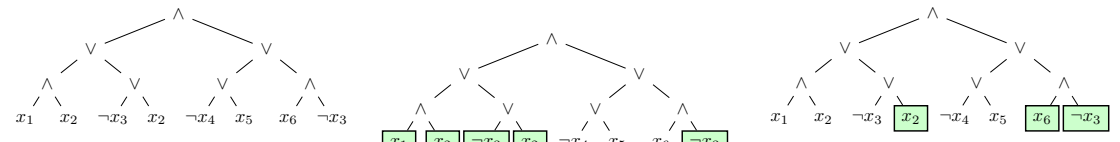
Dylla et al. [3] suggest a slightly modified version of the previously explained algorithm. They empirically found out that eagerly removing all Shannon expansions is not the best choice. Therefore a threshold θ was introduced. If the number of distinct base variables f is less or equal to θ no more Shannon expansions are performed. Instead, the Brute Force Algorithm is used to calculate the probability value for the remaining lineage expression. All other computations are completely identical to those in the previous section, this means that if $\theta = 0$ both algorithms are identical. For the evaluation we set $\theta = 4$, since Dylla et al. [3] achieved the best results with this setting. Algorithm 8 in Appendix B sketches the algorithm that uses threshold θ.

Example 12 *Let us reconsider the Example 11 from the previous section and apply this algorithm with $\theta = 4$. The first Shannon expansion is executed in the same way, since there are more than 4 variables in the initial lineage expression. But before executing the second Shannon expansion, we check again. The lineage expression there is $(\neg x_3) \wedge (\neg x_4 \vee x_5 \vee (x_6 \wedge \neg x_3))$, this expression includes exactly 4 distinct variables, this means that no further Shannon expansion is performed. The probability of this lineage expression is instead computed by applying the Brute Force Algorithm.*

3.3. Approximate Confidence Computation

This algorithm uses parts of the two Decomposition Algorithms. The main difference is that bounds are computed before any decomposition is performed. The bounds describe in what range the final probability value must be. We can restrict this range because the lineage structures delivers a certain knowledge about the composition of the probability. And after each decomposition we gain some knowledge, what further restricts the range. This range is limited by a maximal possible value, which is called the upper bound U and by a minimal possible value, which is called the lower bound L. If L and U are close enough, the evaluation of the expression is stopped and the interval between the two bounds is returned as result. To

be exact we guarantee that the approximation bounds are in a range, which is ϵ small. In the implementation we set $\epsilon = 0.05$. This means that the algorithm always returns a interval $[L, U]$, with $U - L < \epsilon = 0.05$, which includes the exact confidence value. The reason for considering this algorithm is, that decompositions are computationally expensive and often a good enough approximation can be found very fast.



(a) $\mathcal{B} = (x_1 \wedge x_2 \vee \neg x_3 \vee x_2) \wedge (\neg x_4 \vee x_5 \vee x_6 \wedge \neg x_3)$

(b) Upper bound: $\mathcal{B}_U = \neg x_4 \vee x_5 \vee x_6$

(c) Lower bound: $\mathcal{B}_L = (x_1 \wedge x_2 \vee \neg x_3) \wedge (\neg x_4 \vee x_5)$

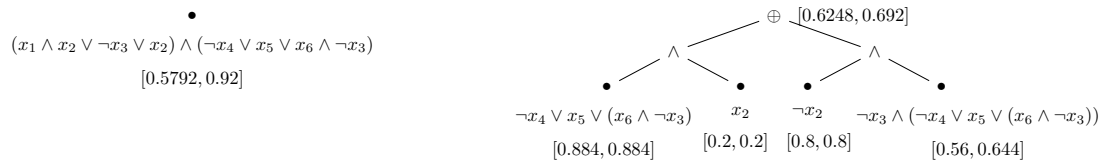
Figure 3.7.: Upper and lower bound

Fink et al. [5] suggest a method to compute lineage expressions in *1OF*, which are upper and lower bounds of the initial lineage expression. For this method to work correctly, the initial lineage expression must be positive (or equivalently, unate). Since the probability value of lineage expressions in *1OF* can be computed in linear time, this is an useful method to compute bounds U and L.

Proposition 4 *[5] Let λ be a positive (or equivalently, unate) lineage expression. By setting all but one occurrence of each base variable, including a possible negation, to true, a lineage expression λ_U, which is in 1OF and is an upper bound of λ, is obtained. By setting all but one occurrence of each base variable, including a possible negation, to false, a lineage expression λ_L, which is in 1OF and is an lower bound of λ, is obtained. Depending on which occurrences of the base variable are chosen, different bounds may be obtained.*

Example 13 *Let us compute the bounds of the tuple with lineage expression $\mathcal{B}$ from Figure 3.7a, which is unate. We start by checking which variables occur more than once. In this case these are variables x_2 and x_3. We then set all but one occurrence of those variables to $true$ or $false$ respectively. The choice of occurrences does not affect the result. For this example we will leave the first occurrences of both variables unchanged. Thus we compute the upper bound $\mathcal{B}_U$ by setting the second base variable from x_2 and x_3 to $true$. See Figure 3.7b for an illustration. When setting the second x_2 to true the whole left part of the tree gets redundant. We also set the second $\neg x_3$ to $true$. This results in the upper bound of $\mathcal{B}_U = \neg x_4 \vee x_5 \vee x_6$. The probability value of this upper bound is computed as $U = P_{1OF}(\mathcal{B}_U) = 0.92$. The lower bound is then computed by setting the second base variable from x_2 and x_3 to $false$. This results in $\mathcal{B}_L = (x_1 \wedge x_2 \vee \neg x_3) \wedge (\neg x_4 \vee x_5)$ (See Figure 3.7c), which results in a probability value of $L = P_{1OF}(\mathcal{B}_L) = 0.5792$.*

The main method of this algorithm, which is sketched in Algorithm 3, uses a *decomposition tree* T to keep track of nodes that already have been evaluated. At the beginning this tree is a simple leaf node, which includes the complete lineage expression λ. As first step, the bounds



(a) decomposition tree at beginning

(b) decomposition tree after one decomposition

Figure 3.8.: Decomposition tree

of the lineage expression λ are computed by applying the procedure described before. If the bounds satisfy the defined error guarantee ϵ, those are returned and otherwise T is decomposed by applying one of the decompositions explained in Section 3.2.2. The algorithm then recursively is called with the newly created decomposition tree as its argument. After every new decomposition, the bounds need to be computed. Since we keep track of all already computed bounds, we only need to do it for the newly created nodes. To have the bounds moving together as fast as possible, we apply the decomposition always on the node with the largest difference between the bounds.

The bounds of the complete decomposition tree T are computed by applying the following rules (Note we do not have $\neg$-nodes here, since no decomposition produces negations. But we do have $\oplus$-nodes, which represent the exclusive-or decomposition):

- $\wedge$**-node:** $U(T) = U(T{\rightarrow}$*left*$) * U(T{\rightarrow}$*right*$)$ and $L(T) = L(T{\rightarrow}$*left*$) * L(T{\rightarrow}$*right*$)$.
- $\vee$**-node:** $U(T) = 1-((1-U(T{\rightarrow}$*left*$))*(1-U(T{\rightarrow}$*right*$))$ and $L(T) = 1-((1-L(T{\rightarrow}$*left*$)) * (1 - L(T{\rightarrow}$*right*$)))$.
- $\oplus$**-node:** $U(T) = U(T{\rightarrow}$*left*$) + U(T{\rightarrow}$*right*$)$ and $L(T) = L(T{\rightarrow}$*left*$) + L(T{\rightarrow}$*right*$)$.
- **leaf node:** $U(T) = P_{1OF}(\lambda_U)$ and $L(T) = P_{1OF}(\lambda_L)$ where λ is the lineage expression at this leaf node.

Example 14 *Let us proceed this algorithm with the lineage expression $\mathcal{B}$ from Figure 3.2 with an error guarantee of $\epsilon = 0.1$. At the beginning we have a decomposition tree T, which contains the whole lineage expression $\mathcal{B}$. We start with computing the bounds of this expression, which results as we saw in Example 13 in a lower bound $L = 0.5792$ and a upper bound $U = 0.92$. This state is illustrated in Figure 3.8a. Since $U - L = 0.3408 > 0.1 = \epsilon$, we can not stop at this point and we need to decompose the tree.*

We have no independence between the left and the right part of the lineage expression $\mathcal{B}$, therefore we compute the Shannon factors as we saw in Example 10 and add those new nodes to the decomposition tree T. This results in the tree, which is shown in Figure 3.8b.

The algorithm is then again executed with the new decomposition tree. This means, we start again with computing the bounds of T. Therefore the bounds of all leaf nodes need to be evaluated. Three of the leaf nodes include lineage expressions that are in 1OF, for which the exact probability value can easily be computed. Therefore the bounds of those

Algorithm 3 Computing approximate probability of the tuple with lineage expression λ and error guarantee ϵ

Require: lineage expression λ in *NNF*

```
1: new decomposition tree T
2: T→λ ← λ, T→op ← NONE, T→bounds ← NULL
3: P ← PROBABILITYAPPROXIMATE(T)
4: function PROBABILITYAPPROXIMATE(decomposition tree T)
5:     (Bound_L, Bound_U) ← BOUND(T)
6:     if Bound_U − Bound_L <= ε then
7:         return (Bound_L, Bound_U)
8:     end if
9:     t ← lineage expression of node with largest difference between bounds in T
10:    if t→left is independent from t→right then
11:        if t→op = ∧ then
12:            Replace t in T by t→left ∧ t→right
13:        else if t→op = ∨ then
14:            Replace t in T by t→left ∨ t→right
15:        end if
16:    else                                              ▷ Shannon expansion
17:        x ← base variable that occurs most in t
18:        Replace t in T by (x ∧ t|_{x=true}) ⊕ (¬x ∧ t|_{x=false})
19:    end if
20:    return PROBABILITYAPPROXIMATE(T)
21: end function
```

nodes are set to these values. The last leaf node includes a more complex lineage expression $\lambda = \neg x_3 \wedge (\neg x_4 \vee x_5 \vee (x_6 \wedge \neg x_3))$*, for which the bounds need to be evaluated by applying the procedure that was explained in Example 13. This results in bounds of* $L = 0.56$ *and* $U = 0.644$ *for this node.*

By applying the rules for computing the bounds for decomposition trees, we get the following bounds $L(T) = 0.2 * 0.884 + 0.8 * 0.56 = 0.6248$ *and* $U(T) = 0.2 * 0.884 + 0.8 * 0.644 = 0.692$ *for* T*. Since* $U - L = 0.0672 < 0.1 = \epsilon$ *the algorithm stops here and returns those bounds as the interval, in which the exact probability must lie. The exact probability is* 0.692 *as we saw in Example 11.*

4. Implementation

We use PostgreSQL server in version 9.3.1 in which we include the concepts described in this thesis. Postgres, an more often used alternative name of PostgreSQL, is an object-relational database management system, which is distributed under an open source licence. Besides SQL and C, it allows also for other languages, so called procedural languages, like PL/pgSQL, PL/Tcl, PL/Perl and PL/Python.

4.1. PostgreSQL

Internally, a query that is executed on a Postgres server will run through different stages, before either a result or an error will be returned.

At first, the parser transforms the query according to its grammar into a parse-tree. This is done by generating corresponding nodes for each keyword in the query, e.g. *SELECT* will generate a *SelectStmt*-Node, whereas * will generate an *A_Star*-Node and so on. While generating nodes, those nodes are linked with each other, generating a tree, which will then be processed by the rewriter. By applying all applicable rewriter rules stored in the system catalogs, the rewriter transforms the parse-tree into the query-tree. In case that the query is syntactically or semantically invalid, an error will be thrown during rewriting and further execution of the query will be aborted.

Before the rewritten query-tree is executed, the optimizer will transform the query-tree. For this, the optimizer looks up all possible paths leading to the same result. By rearranging nodes and expanding the least cost path, an executable query-plan is being generated.

Finally, the executor executes the query-plan in the specified order by retrieving the necessary tuples in the database, applying operators and returning the final result to the user.

4.2. Implementation Approach

We decided to implement the lineage computation in the rewriter. More precisely in the transformation step that transforms the parse-tree into the query-tree. We transform the query, which was entered by the user, in such a way that lineage is computed as well. To do so we add calls to self defined C-functions and aggregate-functions, which take care of the correct

computation of the lineage expressions. With this approach we make use of the already existing functionality of Postgres and since we transform the queries entered by the users into other valid SQL-queries, no changes in the optimizer nor executor are required.

The computation of the confidence value is done by calling one of four C-functions and passing the computed lineage expression as an argument to them. Depending on the algorithm that should be used for the confidence computation the corresponding function is used. This function then evaluates with the algorithm described in Section 3 the confidence value for every result tuple.

4.3. Syntax of Relations

Since we are dealing with Temporal Probabilistic Databases we must make sure that all relations follow a certain syntax. In order to be able to use the concepts described in this work, each relation must have a temporal and a probabilistic attribute. This means that each relation must specify a column with the name p of type numeric, which defines the probability of the tuple's validity. Moreover there must be two columns that specify the time interval in which the tuple is valid. The first of those columns must be called ts and holds the included starting time point of the time interval and the second column must be called te and hold the excluded ending time point of the time interval. In order for the Normalization and Alignment to work, both these columns need to be at the end of the table. To be precise, ts must be the second last column and te the last one.

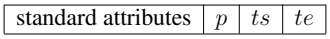


Figure 4.1.: Syntax of relations

4.4. Syntax of Queries

Regarding query execution, queries must contain keywords such that lineage and the corresponding confidence values get computed. We added the keywords *LINEAGE* and four different *CONF*-keywords, *CONF*, *CONFA*, *CONFB* and *CONFC* to the standard SQL-Syntax. If at least one of these keywords is present in the user's query the system will compute the lineage expression for every result tuple. In order to have the query result show the computed lineage expression, one must specify *LINEAGE*, while a *CONF*-keyword must be present to get the confidence value computed and displayed. Each of the four different *CONF*-keyword stands for another algorithm that is used to compute the confidence values. If none of the keywords is given in the query, lineage will not be computed. Keywords can be used in all possible combinations. The five keywords result in the following output:

- **LINEAGE:** Lineage expression

- **CONF:** Confidence computed with Brute Force Algorithm
- **CONFA:** Confidence computed with Decomposition Algorithm
- **CONFB:** Confidence computed with Decomposition Algorithm with Theta ($\theta = 4$)
- **CONFC:** Confidence computed with Approximate Algorithm ($\epsilon = 0.05$)

The keywords must be listed right before the so called SELECT-list of the outermost SELECT-statement. See the following segment of the SQL-SELECT-Statement synopsis.

```
SELECT  [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ opt_lineage ]
    * | expression [ [ AS ] output_name ] [, ...] -- SELECT-list
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
...
opt_lineage = [] | opt_lineage LINEAGE | opt_lineage CONF | opt_lineage
   CONFA | opt_lineage CONFB | opt_lineage CONFC
```

We tried to make our implementation user-friendly in a way, that the user does not need to care about the lineage computation. If one of the keywords is present in the query, our implementation automatically adds the necessary operations to compute correct lineage expressions for derived tuples. This allows the users to query the database and to display lineage and confidence, without having knowledge about its computation.

```
SELECT CONF *
    FROM
        (V ALIGN W ON V.dest=W.loc)V
    LEFT JOIN
        (W ALIGN V ON W.loc=V.dest)W
    ON V.dest=W.loc AND V.ts=W.ts AND V.te=W.te
```

name	dest	ts	te	loc	weather	ts	te	p(CONF)
Ann	Zurich	2014-12-04	2014-12-05					0.5
Ann	Zurich	2014-12-05	2014-12-10	Zurich	Snow	2014-12-05	2014-12-10	0.35
Ann	Zurich	2014-12-08	2014-12-14	Zurich	Fog	2014-12-08	2014-12-14	0.1

Figure 4.2.: Example query and its resulting table

An example of a query is presented in Figure 4.2. In this query the keyword *CONF* is used. This means that the confidence value is computed with the Brute Force Algorithm and the result is shown to the user. The figure also shows the result of the query, if it is executed on the relations from Figure 2.1. Please note that the temporal attribute is split into two columns ts and te, which is the standard syntax for relations in our implementation.

4.5. Lineage Representation

Conceptually, lineage is represented as a boolean expression, but in terms of implementation we want to represent it as a tree structure. For base tuples, lineage must be a variable that allows us to uniquely identify the tuple itself. And for derived tuples, lineage must be able to combine other lineages with the binary operators $\wedge$ and $\vee$ and the unary operator $\neg$. We therefore constructed a data-type called 'lineage' that allows us to build such constructs.

To create the lineage expressions for the base tuples, which must be unique identifiers, we make use of identifiers that are already existing in PostgreSQL. We look up the system columns called 'tableoid' and 'oid', which are unique identifiers (so called object identifiers, short OID) of the relation and the tuple respectively. Those are automatically created by the system.[1] Since the OIDs defined in PostgreSQL are 32-bit quantities and are assigned from a single database-wide counter, it is possible that the counter wraps around in large databases. To make sure that no two rows of a table are assigned with the same OID, a unique constraint on the 'oid' column should be created.[2] By combining the *tableoid* and the *oid* we are able to uniquely identify each tuple within the database. Therefore we store for the lineage of base tuples one simple node that contains the *tableoid* and the *oid* of the tuple.

For derived tuples, all leaves of the tree structure are lineages of base tuples. On the other side the internal nodes represent operators such as $\wedge$, $\vee$ and $\neg$, where all $\wedge$- and $\vee$-nodes have two children that are either leaves or other internal nodes. $\neg$-nodes only have one child, which also can be a leaf or another internal node. Figure 4.3 shows an example of a lineage expression represented in the defined tree structure. In the leaf nodes we represent the unique identifiers for simplification as *tableoid.oid*.

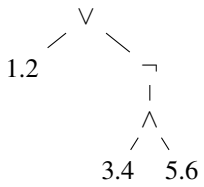


Figure 4.3.: Lineage as tree structure.

In order to be able to store this tree structure in the database, its structure must be changed. Since the values stored as lineage may vary in size, we must follow some defined standard layout, which PostgreSQL has for the representation of data types that vary in size. The rules say that the first four bytes must be a *char[4]* field, in which the size of the complete structure

[1] Till PostgreSQL version 8.1., 'OIDs were created by default unless the user specified to create tables without OIDs. But today, they are no longer created as most up to date applications do not need this attribute any more. Therefore, we modified the code such that OIDs are created by default again, as this is eminent to compute lineage.

[2] Of course, it is only possible for the table to contain fewer than 2^{32} (4 billion) rows with unique identifiers. But in practice the table size should be much less anyhow, or performance might suffer.

is stored. This first 4 bytes are never accessed directly, but they can be set and retrieved by using *SET_VARSIZE* or *VARSIZE* respectively. The rest of the structure must be stored in a *char**. We therefore need two functions. One that transforms the tree structure into this storable form and another one for the other way round.

The pack-function transforms the tree into the storable structure. To do this it traverses through the tree and arranges all nodes in a array of type char. One node consists of five parts: the *tableoid*, the *oid*, the type of the node (*AND*, *OR*, *NOT* and *NONE* for leaf nodes), a pointer to the left child and a pointer to the right child. As an example see Figure 4.4, it shows the lineage from Figure 4.3 in the storable structure. While creating the char array, the pointers between the nodes are corrected and are made relative to the pointer of the root node. This needs to be done because when the structure is retrieved the next time, the absolute pointer values are different to what they are now and the pointers would point to invalid memory. In 4.4a the structure is shown with absolute pointers and in 4.4b with relative pointers. Finally, the size of the array is analysed and it is set to the size attribute of the packed structure. See the blue part in Figure 4.4.

The other function, which transforms the storable structure into the tree structure, is called unpack-function. At first, the number of nodes of the lineage is computed by dividing the whole size of the packed lineage structure through the size of one lineage node. Afterwards the pointers in the node array are made absolute, such that they point to the correct nodes again. After this step the tree structure is restored and all manipulations can be executed.

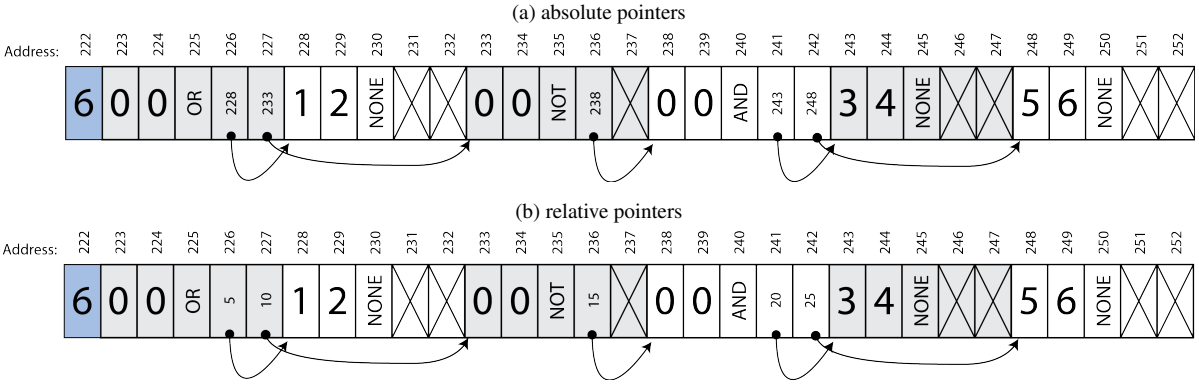


Figure 4.4.: Packed Lineage structure.

To be able to create a new user-defined data type, an input- and an output-function must be provided too. Those functions determine how the type appears as a string, for input by and output to the users. To make the data type usable for our purpose, further operators for the lineage data type were created. These allow us to combine two lineage expressions by adding a new $\wedge$-, $\vee$ or $\neg$-node. Moreover, an operator class for the new type was added. This class allows to sort and compare different values of type lineage. And since for the normalization and alignment algorithms it is necessary to sort by lineage, this operator class is required.

5. Evaluation

In this section we evaluate our implementation of a temporal probabilistic database system and we compare the four implemented confidence computation algorithms.

5.1. Setup

For these experiments, we ran our implementation on a machine with CentOS 6.4, 2 x Intel® Xeon® CPU E5-2440 @ 2.40GHz (each 12 cores), 4 x 300GB 15,000rpm, 64 GB main memory. All parameters of the PostgreSQL server were kept to default values, except that we set the creation of OIDs for every tuple as default again. For all experiments no indexes were used.

We ran the experiments with a synthetic dataset, with the same table structure as shown in Figure 2.1. The tables were filled with different numbers of tuples to evaluate the performance for different amounts of input tuples.

5.2. Lineage Computation

For this thesis, the lineage expression was implemented into the PostgreSQL database system as a tree structure. But in a previous step we represented lineage as a string. In this section those two techniques are evaluated. We evaluate different kinds of queries to see their performance in various cases.

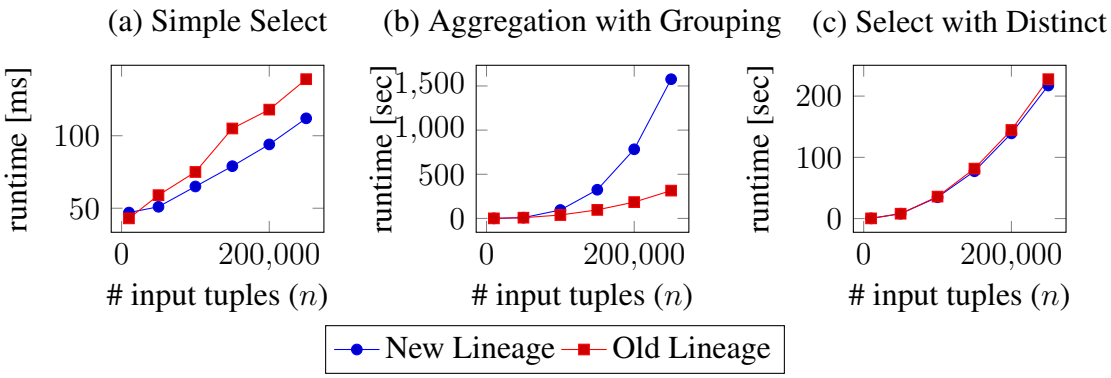


Figure 5.1.: Runtime of lineage computation for simple queries

First we evaluated the performance of simple queries. We created three types of queries that we executed on tables with a different amount of tuples. Query (a) is a simple selection of all

tuples in the table, (b) performs an temporal probabilistic aggregation with grouping and (c) is a temporal probabilistic projection on one attribute, which eliminates duplicates. The results are presented in Figure 5.1. As we see query (a) performs very fast for every amount of input tuples. The new lineage implementation performs a bit faster on average, but as we are in the range of milliseconds, this is ignorable.

The grouping with aggregation (b) results in few result tuples with large lineage expressions. In this case the implementation with strings outperforms the implementation in tree structure. We explain this by the fact that for every operation that is performed on lineage, the pack- and the unpack-function must be executed. As we have many of those operations in this query and because the lineage which must be unpacked gets larger very fast, what makes the unpacking and packing more time consuming, this query is slower with such a lineage representation. Query (c) runs with both implementations in nearly the same time. Pleas note, that we have more result tuples in this query, but the resulting lineage expressions are much shorter. For shorter lineage expressions, the packing and unpacking functions, are much less time consuming.

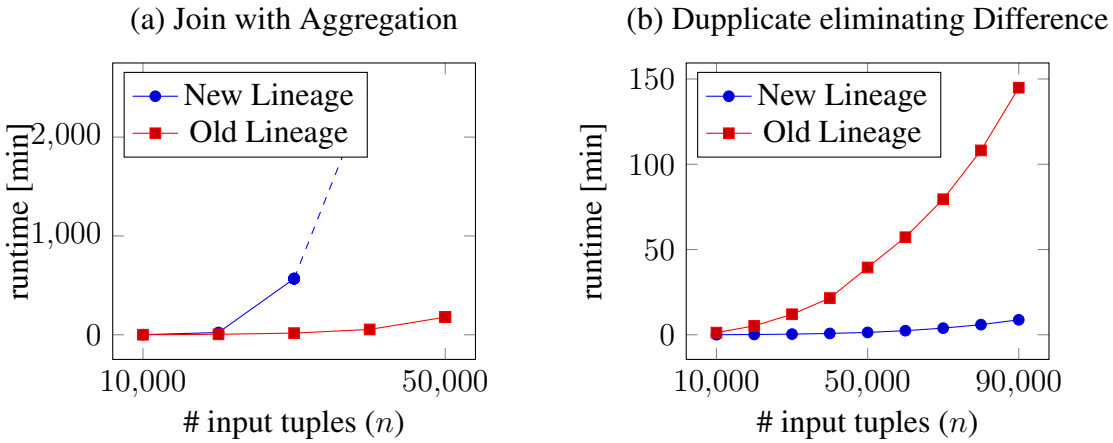


Figure 5.2.: Runtime of lineage computation for complex queries

We also run the implementations with more complex queries. The runtime of those queries is shown in Figure 5.2. The first query (a) performs an temporal probabilistic full join over two tables and groups the resulting tuples. Since the grouping is made on attributes which do not have many distinct values, we get few result tuples but those have huge lineage expressions. As we already saw in the previous examples, the new lineage representation does perform much worse in such a case.

The query (b) results in more tuples than (a), but those tuples have less complex lineage expressions. To be exact, many normalizations are performed, which do not add any complexity to the lineage expressions, and the temporal probabilistic difference is transformed into a right outer join, which also only combines the lineage of two tuples. This results in much less packing and unpacking calls, what leads to the fact that the new lineage representation is

more efficient for such queries. An explanation for why the string representation is performing worse in such queries might be, that the OIDs and the Boolean connectives must be saved as text rather than integers.

Over all, the performance of the two implementations really depend on the length of the lineage expressions that are built. The lineage represented as a string performs much better if the lineage expressions are huge, which is usually the case for aggregations over large groups of tuples. But if we have many result tuples with lineage expressions that are rather short, the new lineage representation has its advantage. For further work, we might try to find a way to evaluate during execution, which representation should be chosen. This could be done by using some heuristics when looking at the query that is executed. Or we even might think of a solution that transforms the lineage expression from one representation to the other for different kinds of operations.

5.3. Confidence Computation

In this section we evaluate the performance of the different confidence computation algorithms. For this we created four queries, each of which should evaluate a different aspect. The queries (a) and (b) produce lineage expressions which are in *1OF*, since no self joins are performed. The difference is, that query (a) results in few result tuples with huge lineage expressions and query (b) results in many result tuples with short lineage expressions. Queries (c) and (d) include self joins, which results in lineage expressions that are not in *1OF*, since one result tuple can be derived from one base tuples in multiple ways. The query (c) includes no difference operation, whereas query (d) does. This means that the lineage expressions of the result tuples from (c) include no negations but those from (d) do.

In Figure 5.3 we present the result of those experiments. As expected, does the Brute Force Algorithm perform bad for large lineage expressions. After a certain amount of input tuples is reached, its running time explodes. This is due to the fact that its complexity is $O(n*2^f)$ where f is the number of distinct base variables in a lineage expression. The other three algorithms all perform similar. Since we have lineage expressions in *1OF* here, they are all expected to run in linear time and because checking if the lineage expression is in *1OF* comes first in all three algorithms, they also should be similarly fast. For query (b) the result is different in a way that the Brute Force Algorithm performs good as well. This is because the lineage expressions of the resulting tuples are short and therefore this algorithm has no disadvantages.

The results from queries (c) and (d) are similar. The Brute Force Algorithm gets explosively slow after a certain point, the same happens to the other three algorithms but much later. For a long time, they perform quite well and nearly the same. But at a certain point their running time explodes too. For the two Decomposition Algorithms this point comes at exactly the same time. This derives from the fact that their implementation is nearly the same, too. On the other hand we have the Approximation Algorithm, which performs worse for query (c) but much better for (d). The advantages of the Approximate Algorithm in (d) can be explained

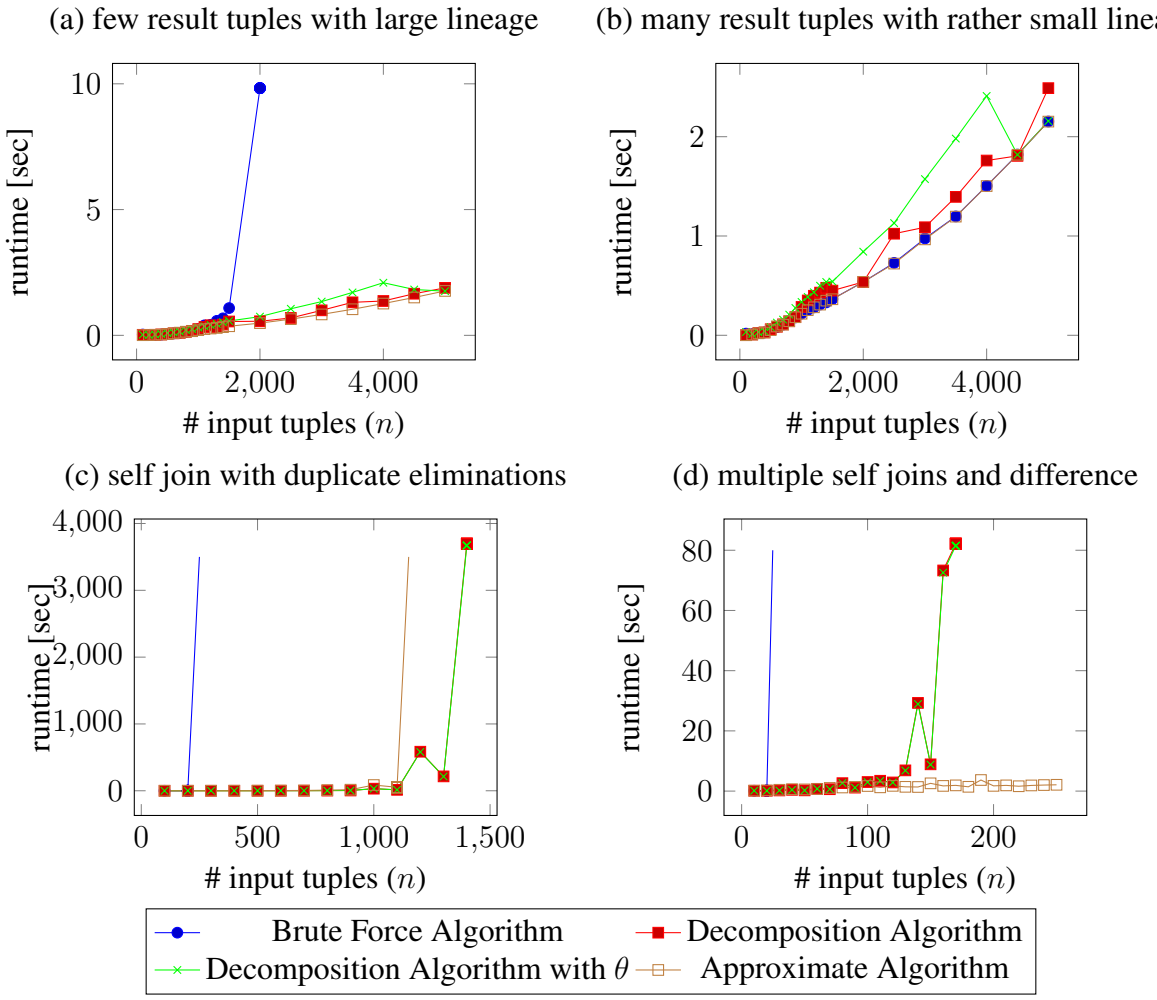


Figure 5.3.: Runtime of confidence computation

with the fact, that close enough bounds can be found fast, but to compute the exact value many further steps are needed. In (c) the algorithm might not be able to find good enough bounds and due to the overhead of computations that are needed to compute the bounds, the running time grows earlier.

As a conclusion, we might say that the Brute Force Algorithm can only be used for queries with short lineage expressions, but for those it performs very well. The two Decomposition Algorithms perform very similar in all cases. To distinguish their behaviour in detail, further experiments would be needed. And as we see, if an approximate confidence value is good enough, the Approximate Algorithm is a good choice, since it performs very well for all sorts of queries.

5.4. Real-World Application

For the implementation as it was done in this work, it is eminent, that the given data is complete. This means for the system to work properly and be able to compute correct results, every table must include time intervals and a probabilistic attribute. Without this, the data is not appropriate for our implementation. When trying to use data from the knowledge base YAGO [8] for our evaluation, the problem of incompleteness arose. Especially, in terms of time we faced obstacles. Since we were only interested in facts with time intervals, we extracted those, but many had timestamps that were not complete or multiple start or ending dates were given.

Appendices

A. Reduction Rules

We show here the complete set of reduction rules. Those include the set operations with the postfix all, where all indicates that the duplicates are not eliminated.

Let **r** and **s** be temporal probabilistic relations over schema R^{TP}, θ be a predicate, F be a set of aggregation functions over $r.A$, $B \subseteq A$ be a set of attributes and OR, AND and NOT functions respectively aggregation functions that combine the given lineage expressions with the defined connective. Then the following reduction rules define a temporal probabilistic algebra with sequenced semantics.

Operator			Reduction
Selection	$\sigma_\theta^{TP}(\mathbf{r})$	$=$	$\sigma_\theta(\mathbf{r})$
Projection	$\pi_B^{TP}(\mathbf{r})$	$=$	${}_{B,T}\vartheta_{OR(\lambda)}(\mathcal{N}_B^{TP}(\mathbf{r};\mathbf{r}))$
Normalization	$\mathcal{N}_B^{TP}(\mathbf{r};\mathbf{s})$	$=$	$\pi_{r.A,r.T,r.\lambda}(\mathcal{N}_B(\mathbf{r};\mathbf{s}))$
Alignment	$\mathbf{r}\,\Phi_\theta^{TP}\mathbf{s}$	$=$	$\pi_{r.A,r.T,r.\lambda}(\mathbf{r}\Phi_\theta\mathbf{s})$
High Aggregation	${}_B\vartheta_F^{TP}(\mathbf{r})$	$=$	${}_{B,T}\vartheta_{F,AND(\lambda)}(\mathcal{N}_B^{TP}(\mathbf{r};\mathbf{r}))$
Difference	$\mathbf{r} -^{TP} \mathbf{s}$	$=$	${}_{r.A,r.T}\vartheta_{OR(AND(r.\lambda,NOT(s.\lambda)))}((\mathcal{N}_A^{TP}(\mathbf{r};\mathcal{N}_A^{TP}(\mathbf{r};\mathbf{s}))) \mathbin{⟕}_{r.A=s.A\wedge r.T=s.T}(\mathcal{N}_A^{TP}(\mathbf{s};\mathcal{N}_A^{TP}(\mathbf{s};\mathbf{r}))))$
Difference All	$\mathbf{r} -_{ALL}^{TP} \mathbf{s}$	$=$	$\pi_{r.A,r.T,AND(r.\lambda,NOT(s.\lambda))}((\mathcal{N}_A^{TP}(\mathbf{r};\mathcal{N}_A^{TP}(\mathbf{r};\mathbf{s}))) \mathbin{⟕}_{r.A=s.A\wedge r.T=s.T}({}_{A,T}\vartheta_{OR(\lambda)}\mathcal{N}_A^{TP}(\mathbf{s};\mathcal{N}_A^{TP}(\mathbf{s};\mathbf{r}))))$
Union	$\mathbf{r} \cup^{TP} \mathbf{s}$	$=$	${}_{r.A,r.T}\vartheta_{OR(r.\lambda)}(\mathcal{N}_A^{TP}(\mathbf{r};\mathcal{N}_A^{TP}(\mathbf{r};\mathbf{s})) \cup \mathcal{N}_A^{TP}(\mathbf{s};\mathcal{N}_A^{TP}(\mathbf{s};\mathbf{r})))$
Union All	$\mathbf{r} \cup_{ALL}^{TP} \mathbf{s}$	$=$	$\mathcal{N}_A^{TP}(\mathbf{r};\mathcal{N}_A^{TP}(\mathbf{r};\mathbf{s})) \cup \mathcal{N}_A^{TP}(\mathbf{s};\mathcal{N}_A^{TP}(\mathbf{s};\mathbf{r}))$
Intersection	$\mathbf{r} \cap^{TP} \mathbf{s}$	$=$	${}_{r.A,r.T}\vartheta_{OR(AND(r.\lambda,s.\lambda))}((\mathcal{N}_A^{TP}(\mathbf{r};\mathcal{N}_A^{TP}(\mathbf{r};\mathbf{s}))) \bowtie_{r.A=s.A\wedge r.T=s.T}(\mathcal{N}_A^{TP}(\mathbf{s};\mathcal{N}_A^{TP}(\mathbf{s};\mathbf{r}))))$
Intersection All	$\mathbf{r} \cap_{ALL}^{TP} \mathbf{s}$	$=$	$\pi_{r.A,r.T,AND(r.\lambda,s.\lambda)}((\mathcal{N}_A^{TP}(\mathbf{r};\mathcal{N}_A^{TP}(\mathbf{r};\mathbf{s}))) \bowtie_{r.A=s.A\wedge r.T=s.T}({}_{A,T}\vartheta_{OR(\lambda)}\mathcal{N}_A^{TP}(\mathbf{s};\mathcal{N}_A^{TP}(\mathbf{s};\mathbf{r}))))$
Cartesian Product	$\mathbf{r} \times^{TP} \mathbf{s}$	$=$	$\pi_{r.A,r.T,s.A,s.T,AND(r.\lambda,s.\lambda)}((\mathbf{r}\Phi_{true}^{TP}\mathbf{s}) \bowtie_{r.T=s.T}(\mathbf{s}\Phi_{true}^{TP}\mathbf{r}))$
Inner Join	$\mathbf{r} \bowtie_\theta^{TP}\mathbf{s}$	$=$	$\pi_{r.A,r.T,s.A,s.T,AND(r.\lambda,s.\lambda)}((\mathbf{r}\Phi_\theta^{TP}\mathbf{s}) \bowtie_{\theta\wedge r.T=s.T}(\mathbf{s}\Phi_\theta^{TP}\mathbf{r}))$
Left Outer Join	$\mathbf{r} \mathbin{⟕}_\theta^{TP}\mathbf{s}$	$=$	$\pi_{r.A,r.T,s.A,s.T,AND(r.\lambda,s.\lambda)}((\mathbf{r}\Phi_\theta^{TP}\mathbf{s}) \mathbin{⟕}_{\theta\wedge r.T=s.T}(\mathbf{s}\Phi_\theta^{TP}\mathbf{r}))$
Right Outer Join	$\mathbf{r} \mathbin{⟖}_\theta^{TP}\mathbf{s}$	$=$	$\pi_{r.A,r.T,s.A,s.T,AND(r.\lambda,s.\lambda)}((\mathbf{r}\Phi_\theta^{TP}\mathbf{s}) \mathbin{⟖}_{\theta\wedge r.T=s.T}(\mathbf{s}\Phi_\theta^{TP}\mathbf{r}))$
Full Outer Join	$\mathbf{r} \mathbin{⟗}_\theta^{TP}\mathbf{s}$	$=$	$\pi_{r.A,r.T,s.A,s.T,AND(r.\lambda,s.\lambda)}((\mathbf{r}\Phi_\theta^{TP}\mathbf{s}) \mathbin{⟗}_{\theta\wedge r.T=s.T}(\mathbf{s}\Phi_\theta^{TP}\mathbf{r}))$

B. Algorithms

In this appendix further Algorithms are shown, which are used for the confidence computation algorithms to work. The first snippet shows the main Algorithm of the Brute Force Algorithm.

Algorithm 4 Computing Probability value of lineage expression λ with Brute Force Algorithm

Require: lineage expression λ

```
1:  function ProbabilityBruteForce(lineage expression λ)
2:      int sum
3:      array vars  ▷ Array that stores the allocation and the probability value for every base variable
4:      n ←Size(vars)
5:      while i < 2^n do
6:          int j = 0
7:          for all var in vars do
8:              if floor(i/2^(n−1−j)) %2 = 0 then        ▷ This constructs the truth table
9:                  var.allocation ← true
10:             else
11:                 var.allocation ← false
12:             end if
13:             j ← j + 1
14:         end for
15:         if Evaluate(λ, vars) then
16:             double p ← 1
17:             for all var in vars do
18:                 if var.allocation then
19:                     p ← p ∗ var.P
20:                 else
21:                     p ← p ∗ (1 − var.P)
22:                 end if
23:             end for
24:             sum ← sum + p
25:         end if
26:         i ← i + 1
27:     end while
28:     return sum
29: end function
```

Algorithm 5 shows the algorithm that evaluates a lineage expression with a given allocation for each base variable.

Algorithm 5 Evaluating if lineage expression λ is $true$ with the given allocation

Require: lineage expression λ, an allocation for every base variable in λ

```
1:  function EVALUATE(lineage expression λ, array vars)
2:      if λ→op = NONE then                                   ▷ λ is leaf node
3:          return vars[λ].allocation
4:      else if λ→op = ¬ then
5:          return !EVALUATE(λ→left, vars)
6:      else if λ→op = ∧ then
7:          return EVALUATE(λ→left, vars)∧EVALUATE(λ→right, vars)
8:      else
9:          return EVALUATE(λ→left, vars)∨EVALUATE(λ→right, vars)
10:     end if
11: end function
```

Algorithm 6 shows the algorithm that computes the *NNF* of a lineage expression. If a $\wedge$- or $\vee$-node must be negated, we change from one to the other and negate both children (lines 11-18). If two negations follow each other they can both be deleted (line 9) and otherwise nothing needs to be done.

Algorithm 6 Computes Negation Normal Form of λ

Require: lineage expression λ
Ensure: λ is in *NNF*

```
1:  λ ← NNF(λ)
2:  function NNF(lineage expression λ)
3:      if λ→op = NONE then                                   ▷ λ is leaf node
4:          return λ
5:      else if λ→op = ¬ then
6:          if λ→left is leaf node then                       ▷ λ→left is leaf node
7:              return λ
8:          else if λ→left→op = ¬ then
9:              return λ→left→left
10:         else                                              ▷ λ→left is ∧- or ∨-node
11:             λ→left ← ADDNOT(λ→left)        ▷ Add a ¬-node at the top of the left node
12:             λ→right ← ADDNOT(λ→right)
13:             if λ→op = ∧ then
14:                 λ→op ← ∨
15:             else
16:                 λ→op ← ∧
17:             end if
18:             return NNF(λ)
19:         end if
20:     else
21:         λ→left ← NNF(λ→left)
22:         λ→right ← NNF(λ→right)
23:         return λ
24:     end if
25: end function
```

Algorithm 7 shows the algorithm that computes the probability of a lineage expression in *1OF*.

Algorithm 7 Computing probability of a lineage expression λ in *1OF*

Require: lineage expression λ in *1OF*

1: **function** PROBABILITY1OF(lineage expression λ)
2: **if** $\lambda \rightarrow op = NONE$ **then** ▷ λ is leaf node
3: **return** $\lambda.P$ ▷ $\lambda.P$ is the probability value stored in the tuple with identifier λ
4: **else if** $\lambda \rightarrow op = \neg$ **then**
5: **return** $1 - \lambda \rightarrow left.P$
6: **else if** $\lambda \rightarrow op = \wedge$ **then**
7: **return** PROBABILITY1OF($\lambda \rightarrow left$) $*$ PROBABILITY1OF($\lambda \rightarrow right$)
8: **else**
9: **return** $1-((1-$ PROBABILITY1OF($\lambda \rightarrow left$) $)*(1-$ PROBABILITY1OF($\lambda \rightarrow right$) $))$
10: **end if**
11: **end function**

Algorithm 8 shows the main algorithm of the Decomposition Algorithm with Theta.

Algorithm 8 Computing probability of the tuple with lineage expression λ with threshold θ

Require: lineage expression λ in *NNF*

function PROBABILITYθ(lineage expression λ, int θ)
  if λ is in *1OF* **then**
    return PROBABILITY1OF(λ)
  else if $\lambda\rightarrow$*left* is independent from $\lambda\rightarrow$*right* **then**
    if $\lambda\rightarrow$*op* = $\wedge$ **then**
      return PROBABILITYθ($\lambda\rightarrow$*left*, θ) $*$ PROBABILITYθ($\lambda\rightarrow$*right*, θ)
    else
      return $1 - ((1-$ PROBABILITYθ($\lambda \rightarrow$ *left*, θ) $) * (1-$ PROBABILITYθ($\lambda \rightarrow$ *right*, θ) $))$
    end if
  else if $F(\lambda) <= \theta$ **then**
    return PROBABILITYBRUTEFORCE(λ)
  else ▷ Shannon expansion
    $x \leftarrow$ base variable that occurs most in λ
    return $x.P*$ PROBABILITYθ($\lambda|_{x=true}, \theta$) $+(1 - x.P)*$ PROBABILITYθ($\lambda|_{x=false}, \theta$)
  end if
end function

Algorithm 9 shows the algorithm that computes the bounds, which are in *1OF*, of a lineage expression. This algorithm is used for the Approximate Confidence Computation Algorithm.

Algorithm 9 Computing bounds of lineage expression λ

Require: λ needs to be in *NNF*

$U_{Bound} \leftarrow$ BOUNDλ(λ, NEWTREE(), $true$), $L_{Bound} \leftarrow$ BOUNDλ(λ, NEWTREE(), $false$)
function BOUNDλ(lineage expression λ, binary search tree $tree$, bool v)
  if $\lambda\rightarrow$*op* = *NONE* **then** ▷ λ is leaf node
    if SEARCH($tree$, λ) **then**
      INSERT($tree$, λ)
      return *NULL*
    else
      return COPYNODE(λ)
    end if

```
10:     else if λ→op = ¬ then
11:         l ← BOUNDλ(λ→left, tree, v)
12:         if l then
13:             return NEWNODE(¬, l, NULL)      ▷ Return a new node with op = ¬, left = l
    and right = NULL
14:         else
15:             return NULL
16:         end if
17:     end if
18:     l ← BOUNDλ(λ→left, tree, v)
19:     if !l ∧ ((v ∧ λ→op = ∨) ∨ (!v ∧ λ→op = ∧)) then
20:         return NULL
21:     else
22:         r ← BOUNDλ(λ→right, tree, v)
23:         if !r ∧ (!l ∨ ((v ∧ λ→op = ∨) ∨ (!v ∧ λ→op = ∧))) then
24:             return NULL
25:         else
26:             if !l then
27:                 return r
28:             else if !r then
29:                 return l
30:             else
31:                 return NEWNODE(λ→op, l, r)
32:             end if
33:         end if
34:     end if
35: end function
```

Algorithm 10 shows the algorithm that computes the bounds of a decomposition tree. This algorithm is also used for the Approximate Confidence Computation Algorithm.

Algorithm 10 Computing bounds of decomposition tree T

1: **function** BOUND(decomposition tree T)
2: **if** $T\rightarrow$*op* $=$ *NONE* **then**
3: **if** $T\rightarrow$*bounds* $=$ *NULL* **then** ▷ Bounds were not yet computed
4: **if** $T\rightarrow\lambda$ is positive or unate **then**
5: $T\rightarrow$*bounds* $\leftarrow$ (BOUND$\lambda(T\rightarrow\lambda,$ NEWTREE$(), false)$, BOUND$\lambda(T\rightarrow\lambda,$ NEWTREE$(), true)$)
6: **else**
7: $T\rightarrow$*bounds* $\leftarrow (0, 1)$
8: **end if**
9: **end if**
10: **return** $T\rightarrow$*bounds*
11: **else**
12: $(LeftBound_L, LeftBound_U) \leftarrow$ BOUND($T\rightarrow$*left*)
13: $(RightBound_L, RightBound_U) \leftarrow$ BOUND($T\rightarrow$*right*)
14: **if** $T\rightarrow$*op* $= \wedge$ **then**
15: **return** $(LeftBound_L * RightBound_L, LeftBound_U * RightBound_U)$
16: **else if** $T\rightarrow$*op* $= \wedge$ **then**
17: **return** $(1 - ((1 - LeftBound_L) * (1 - RightBound_L)), 1 - ((1 - LeftBound_U) * (1 - RightBound_U)))$
18: **else if** $T\rightarrow$*op* $= \oplus$ **then**
19: **return** $(LeftBound_L + RightBound_L, LeftBound_U + RightBound_U)$
20: **end if**
21: **end if**
22: **end function**

Bibliography

[1] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In Nascimento et al. [9], pages 864–875.

[2] Anton Dignös, Michael H Böhlen, and Johann Gamper. Temporal alignment. In *ACM SIGMOD 2012 international conference on Management of Data*, SIGMOD '12, pages 433–444, New York, NY, USA, MAY 2012. ACM.

[3] Maximilian Dylla, Iris Miliaraki, and Martin Theobald. A temporal-probabilistic database model for information extraction. *Proc. VLDB Endow.*, 6(14):1810–1821, September 2013.

[4] Robert Fink, Dan Olteanu, and Swaroop Rath. Providing support for full relational algebra in probabilistic databases. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 315–326, Washington, DC, USA, 2011. IEEE Computer Society.

[5] Robert Fink, Jiewen Huang, and Dan Olteanu. Anytime approximation in probabilistic databases. *The VLDB Journal*, 22(6):823–848, 2013.

[6] Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors. *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, 2009. IEEE.

[7] Christoph Koch and Dan Olteanu. Conditioning probabilistic databases. *Proc. VLDB Endow.*, 1(1):313–325, August 2008.

[8] Max-Planck-Institut für Informatik. Yago - d5: Databases and information systems, April 2014. URL `http://www.mpi-inf.mpg.de/yago-naga/yago/`.

[9] Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors. *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, 2004. Morgan Kaufmann.

[10] Dan Olteanu and Jiewen Huang. Secondary-storage confidence computation for conjunctive queries with inequalities. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 389–402, New York, NY, USA, 2009. ACM.

[11] Dan Olteanu, Jiewen Huang, and Christoph Koch. Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases. In Ioannidis et al. [6], pages 640–651.

[12] Dan Olteanu, Jiewen Huang, and Christoph Koch. Approximate Confidence Computation in Probabilistic Databases. In *Proceedings of the 26th International Conference on Data Engineering*, 2010. Long paper.

[13] Anish Das Sarma, Martin Theobald, and Jennifer Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. Technical Report 2007-15, Stanford InfoLab, March 2007.

[14] Dan Suciu, Dan Olteanu, R. Christopher, and Christoph Koch. *Probabilistic Databases*. Morgan & Claypool Publishers, 1st edition, 2011.

[15] David Toman. Point-based temporal extensions of sql and their efficient implementation. In *Temporal Databases, Dagstuhl*, pages 211–237, 1997.