Lineage implementation in PostgreSQL

Andrin Betschart, 09-714-882
Martin Leimer, 09-728-569

3. Oktober 2013

Contents

1. Introduction

Temporal Probabilistic Databases, called TPDBs are defined as databases consisting of tuples representing an event. Each event has a time interval, called the temporal attribute, consisting of an included starting and excluded ending timestamp. Alongside there exists a probabilistic attribute, which defines the probability to which some event will occur at a specific time point within the given time interval.

Moreover, each temporal probabilistic relation consists of further attributes, called the non-temporal attributes, which store any kind of information about the event the tuple represents. This adds up to a relation having the following scheme:

	non-temporal		temporal	probabilistic
	$Name$	$Supervisor$	T	p
p_1	Ann	Kim	[1, 3)	0.95
p_2	Ann	Brad	[2, 4)	0.62

Figure 1.1: Temporal Probabilistic Relation

In order to query TPDBs, we propose the use of the reduction rules as evaluated by Anton Dignös et al. [1]. Those rewrite temporal operators using time adjustment operators normalize, align and absorb with standard database operators. The time adjustment operators adjust tuple time intervals such that any matching clause of the standard database operator can be extended with the temporal attribute.

While executing a query, the probabilistic attribute must be changed accordingly, as each result tuple is derived from multiple tuples, each having their own probability. For this, we propose the use of lineage, which will keep track from which tuples the result tuples are derived.

In Section two, we will describe how we can use lineage for probability computation. We then move on explaining how lineage is computed for each operator in detail. In Section three, we will show the evaluation algorithm we used to compute confidences of the result tuples given their lineage. Section four will give an overview about how PostgreSQL-Queries are executed in the backend. We then move on explaining how we extended PostgreSQL by lineage and the evaluation algorithm, giving abstract overviews about the implemented changes. Finally, in Section five, we present an evaluation of our implementation.

2. Lineage computation in TPDBs

In order to compute the confidence of a result tuple, we propose the use of lineage according to Da Sarma et al. [2]. Lineage allows us to keep track of which base tuples a result tuple was derived from. This will allow us to compute final confidence afterwards, as this is a requirement for this approach.

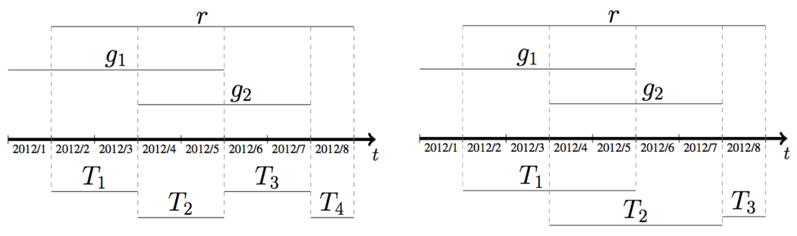
2.1. Lineage

In temporal probabilistic databases, each tuple represents an event which has a probability of being true and each event is determined by an unique identifier. On a conceptual level, all unique identifiers correspond to a boolean variable. This means that if an event tuple is true, the underlying boolean variable will evaluate to true as well and false otherwise.

A query operation corresponds to a combination of input tuples for the creation of the output ones. In order to describe the correlations between the input tuples during an operation, we use lineage. The lineage of a tuple $\lambda.t$ is a complex boolean expression, consisting of boolean variables uniquely determining other tuples in the database and it intuitively captures "how tuple t was derived".

2.2. Temporal adjustment in TPDBs

In the following, we will show how lineage is computed for different boolean operators. For this, we first have to adjust time intervals, were we propose the use of the reduction rules from Dignös et al. [1] as seen in Figure 2.2. Those reduction rules describes how a query must be rewritten using time adjustment operators in order to do correct query operations. By applying those time adjustment operators as seen in Figure 2.1, we create for each tuple a new set of tuples having identical non-temporal attributes but adjusted time intervals. These adjusted time intervals will have starting and ending points such that they will match with the corresponding tuple of the other relation. Thereafter, the matching clause of the query operator can be extended with the temporal attribute.



(a) Temporal Splitter. (b) Temporal Aligner.

Figure 2.1: Temporal adjustment operators [1]

For normalization, which is used for projection, aggregation and set operations, the tuples initial time interval is split up according to both, starting and ending time points of tuples matching on the condition being specified.

For alignment, which is used for cartesian product and join operations, the tuples initial time interval is intersected with the time intervals of the tuples matching on the condition being specified.

Operator			**Reduction**
Selection	$\sigma^T_\theta(\mathbf{r})$	=	$\sigma_\theta(\mathbf{r})$
Projection	$\pi^T_{\mathbf{B}}(\mathbf{r})$	=	$\pi_{\mathbf{B},T}(\mathcal{N}_{\mathbf{B}}(\mathbf{r};\mathbf{r}))$
Aggregation	$_{\mathbf{B}}\vartheta^T_F(\mathbf{r})$	=	$_{\mathbf{B},T}\vartheta_F(\mathcal{N}_{\mathbf{B}}(\mathbf{r};\mathbf{r}))$
Difference	$\mathbf{r} -^T \mathbf{s}$	=	$\mathcal{N}_{\mathbf{A}}(\mathbf{r};\mathbf{s}) - \mathcal{N}_{\mathbf{A}}(\mathbf{s};\mathbf{r})$
Union	$\mathbf{r} \cup^T \mathbf{s}$	=	$\mathcal{N}_{\mathbf{A}}(\mathbf{r};\mathbf{s}) \cup \mathcal{N}_{\mathbf{A}}(\mathbf{s};\mathbf{r})$
Intersection	$\mathbf{r} \cap^T \mathbf{s}$	=	$\mathcal{N}_{\mathbf{A}}(\mathbf{r};\mathbf{s}) \cap \mathcal{N}_{\mathbf{A}}(\mathbf{s};\mathbf{r})$
Cart. Prod.	$\mathbf{r} \times^T \mathbf{s}$	=	$\alpha((\mathbf{r}\Phi_{true}\mathbf{s}) \Join_{\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_{true}\mathbf{r}))$
Inner Join	$\mathbf{r} \Join^T_\theta \mathbf{s}$	=	$\alpha((\mathbf{r}\Phi_\theta\mathbf{s}) \Join_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r}))$
Left O. Join	$\mathbf{r} ⟕^T_\theta \mathbf{s}$	=	$\alpha((\mathbf{r}\Phi_\theta\mathbf{s}) ⟕_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r}))$
Right O. Join	$\mathbf{r} ⟖^T_\theta \mathbf{s}$	=	$\alpha((\mathbf{r}\Phi_\theta\mathbf{s}) ⟖_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{r}\Phi_\theta\mathbf{r}))$
Full O. Join	$\mathbf{r} ⟗^T_\theta \mathbf{s}$	=	$\alpha((\mathbf{r}\Phi_\theta\mathbf{s}) ⟗_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r}))$
Anti Join	$\mathbf{r} \triangleright^T_\theta \mathbf{s}$	=	$(\mathbf{r}\Phi_\theta\mathbf{s}) \triangleright_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r})$

Figure 2.2: Reduction Rules [1]

To illustrate the computation of lineage, we introduce the following running example. Please note that each base tuple has a probability p. However, this attribute is ignored for any intermediate operation, as it would be neither valid nor needed. Only in the result relation, when lineage shall be evaluated, do we retrieved the probabilities of the base tuples in order to compute final confidence.

P (People)

	$Name$	$Dest$	T	p	λ
p_1	Ann	Zurich	[3, 8)	0.80	p_1
p_2	Ann	Zurich	[9, 14)	0.50	p_2
p_3	Mark	Basel	[6, 12)	0.70	p_3
p_4	Jim	Luzern	[5, 10)	0.20	p_4
p_5	Tina	Bern	[10, 13)	1.00	p_5
p_6	Tina	Bern	[10, 13)	0.90	p_6

W (Weather)

	Loc	$Weather$	T	p	λ
w_1	Basel	Sun	[1, 8)	0.80	w_1
w_2	Bern	Rain	[11, 17)	0.50	w_2
w_3	Zurich	Snow	[5, 10)	0.70	w_3
w_4	Zurich	Fog	[8, 15)	0.20	w_4

Figure 2.3: Temporal Probabilistic Databases

2.3. Relational algebra operators in TPDBs

2.3.1. Selection

For any selection operator σ, we do not need to apply any reduction rules in advance. The lineage of each result tuple corresponds to the lineage of the input tuple which matters the condition θ.

As an example we want to find all the possible predictions for weather conditions in Zurich. This results in:

	Loc	$Weather$	W	λ
r_1	Zurich	Snow	[5, 10)	w_3
r_2	Zurich	Fog	[8, 15)	w_4

Figure 2.4: $\sigma_{Loc=Zurich}(W)$

Consider a select operator having selection criteria θ on relation S producing relation R, where $r_x.\lambda$ is the lineage of r_x

1. For each tuple s_i in S which fulfils θ, add s_i to R generating r_j where $r_j.\lambda = s_i.\lambda$

2.3.2. Duplicate eliminator

We explicitly allow for duplicates in TPDBs, meaning that tuples with identical non-temporal attributes and identical time interval can coexist. Such duplicates can appear both, in base relations and intermediate results, e.g. after a projection. However, even duplicates can still be distinguished from each other, as each tuple has an unique identifier used for lineage. But as relational algebra operators are duplicate eliminating by default, we must take special care of tuples having identical non-temporal and temporal attributes, but different lineage. Therefore, we apply the following algorithm to handle lineage correctly:

Consider a select operator having selection criteria θ on relation S producing relation R, where $r_x.\lambda$ is the lineage of r_x

1. For each tuple s_i in S which fulfils θ, add s_i to R generating r_j where $r_j.\lambda = s_i.\lambda$
2. For each tuple r_i in R which has a duplicate r_j in R, set lineage of r_i as $r_i.\lambda \vee r_j.\lambda$ and remove r_j afterwards

The application of this algorithm using the following example $\sigma_{Loc=Bern}(P)$ results in:

	Loc	$Weather$	T	λ
r_1	Tina	Bern	[10, 13)	$p_5 \vee p_6$

Figure 2.5: $\sigma_{Loc=Bern}(P)$

In a first step, all tuples which do not follow the selection criteria are removed, remaining tuples p_5 and p_6. Then, as the selection operator is duplicate eliminating and since p_6 is a duplicate of p_5, one of those tuples will be removed as well. But while applying this step, we must concatenate the boolean expressions of the inferred tuples, here p_5 and p_6, using $\vee$, as the result tuple r_1 is derived from p_5 and p_6.

2.3.3. Projection

For any projection operator π^T, we need to do a normalization on the relation itself first. Then, the attributes specified within the projection clause B are extended with the temporal attribute, before the projection is being executed. Regarding lineage, the boolean expression of the input tuples correspond to the result tuples except in case of duplicates, where the duplicate elimination algorithm is applied.

Assume the following example $\pi^T_{Loc}(W)$. According to the reduction rules this query is rewritten as $\pi_{Loc,T}(\mathcal{N}_{Loc}(W;W))$. By applying normalization, the intermediate result is:

	Loc	$Weather$	T	λ
x_1	Basel	Sun	[1, 8)	w_1
x_2	Bern	Rain	[11, 17)	w_2
x_3	Zurich	Snow	[5, 8)	w_3
x_4	Zurich	Fog	[8, 10)	w_3
x_5	Zurich	Fog	[8, 10)	w_4
x_6	Zurich	Fog	[10, 15)	w_4

Figure 2.6: $\mathcal{N}_{Loc}(W;W)$

Here, tuple w_3 was split up in a set of two tuples x_3 and x_4 having identical non-temporal attributes, but an adjusted temporal attribute. Moreover, lineage of x_j being created from tuple w_i corresponds to $w_i.\lambda$. Correspondingly, this is done for tuple w_4.

The application of the projection operator produces the following result:

	Loc	T	λ
r_1	Basel	[1, 8)	w_1
r_2	Bern	[11, 17)	w_2
r_3	Zurich	[5, 8)	w_3
r_4	Zurich	[8, 10)	$w_3 \vee w_4$
r_5	Zurich	[10, 15)	w_4

Figure 2.7: $\pi_{Loc,T}(\mathcal{N}_{Loc}(W;W))$

Here, tuple r_1 is derived from x_1 which has lineage $x_1.\lambda = w_1$. On the other hand, tuple r_4 is derived from x_4 and x_5 as the projection operator is duplicate eliminating. Therefore lineage is adjusted as specified in Section 2.3.2, resulting in $r_4.\lambda = x_4.\lambda \vee x_5.\lambda = w_3 \vee w_4$.

Consider a projection operator having projection criteria B on relation S producing relation R, where ${}_{x}.\lambda$ is the lineage of r_x

1. For each tuple s_i in S normalize it using s_j in S and set the lineage of each tuple x_k in the created tuple sets as $s_i.\lambda$
2. Add each x_k to R, while removing all non-temporal attributes not being specified in B
3. For each tuple r_i in R which has a duplicate r_j in R, set lineage of r_i as $r_i.\lambda \vee r_j.\lambda$ and remove r_j afterwards

2.3.4. High Aggregation

For any high aggregation operator ϑ^T, we need to do a normalization on the relation itself first, like for projection. Then, the attributes specified in the high aggregation clause B is extended with the temporal attribute, before the high aggregation is being executed. Regarding lineage, the boolean expression of the input tuples correspond to the intermediate tuples after having applied normalization. Then, for high aggregation, the boolean expressions of the tuples belonging to the same group are $\wedge$-concatenated with each other before the actual high aggregation function is being executed.

Assume the following example ${}_{Loc}\vartheta^T_{count(Loc)}(W)$. According to the reduction rules this query is rewritten as ${}_{Loc,T}\vartheta_{count(Loc)}(\mathcal{N}_{Loc}(W;W))$. Therefore, we first have to apply normalization producing same results as in Figure 2.6.

Now, the application of the high aggregation operator produces the following result:

	Loc	*count*(*Loc*)	T	λ
r_1	Basel	1	[1, 8)	w_1
r_2	Bern	1	[11, 17)	w_2
r_3	Zurich	1	[5, 8)	w_3
r_4	Zurich	2	[8, 10)	$w_3 \wedge w_4$
r_5	Zurich	1	[10, 15)	w_4

Figure 2.8: ${}_{Loc,T}\vartheta_{count(Loc)}(\mathcal{N}_{Loc}(W;W))$

Here, tuple r_1 is derived from x_1 which has lineage $\lambda.x_1 = w_1$. On the other hand, tuple r_4 is derived from x_4 and x_5 as by definition each grouping only returns one result per group. Either because duplicates are eliminated, or/and functions are applied. Here, the function $count(Loc)$ is applied, which counts the number of entries for each (Loc, T)-group. There, lineage of $r_4.\lambda$ is $x_4.\lambda \wedge x_5.\lambda = w_3 \wedge w_4$.

Consider an high aggregation operator having aggregation clause B on relation S producing relation R, where $r_x.\lambda$ is the lineage of r_x

1. For each tuple s_i in S normalize it using s_j in S and set the lineage of each tuple x_k in the created tuple sets as $\lambda.s_i$
2. For each x_i find all x_j, $i \neq j$ belonging to the same group and apply high aggregation. Add remaining tuple x_k to R producing r_l and set $r_l.\lambda = x_i.\lambda \wedge x_j.\lambda$

2.3.5. Set operations

For any set operator $\cap^T, \cup^T, -^T$ we first must apply normalization on both relations using the other relation each. Lineage and the normalization operation behaves in a same way as shown in Figure 2.6, except that we normalize on different relations. Once having applied normalization, the set operator can be applied. As lineage is being calculated differently for each set operator, we will explain this more detailed in the upcoming subsections.

Assume a relation P' being identical as P but containing only tuples p_1, p_5 and p_6. The normalizations of $\mathcal{N}_{Name,Dest}(P; P')$ and $\mathcal{N}_{Name,Dest}(P'; P)$ are as follows:

	$Name$	$Dest$	T	λ
np_1	Ann	Zurich	[3, 8)	p_1
np_2	Ann	Zurich	[9, 14)	p_2
np_3	Mark	Basel	[6, 12)	p_3
np_4	Jim	Luzern	[5, 10)	p_4
np_5	Tina	Bern	[10, 13)	p_5
np_6	Tina	Bern	[10, 13)	p_6

Figure 2.9: $\mathcal{N}_{P.Dest=P'.Dest}(P; P')$

	$Name$	$Dest$	T	λ
np'_1	Ann	Zurich	[3, 8)	p'_1
np'_2	Tina	Bern	[10, 13)	p'_2
np'_3	Tina	Bern	[10, 13)	p'_3

Figure 2.10: $\mathcal{N}_{P.Dest=P'.Dest}(P'; P)$

Obviously, normalization produced the same results for each relation np and np' as its input relation p and p' was. Although that we could have chosen a different example, for understanding purposes we try to focus here on the computation of lineage of the set operations only.

Given this normalizations, we can apply the corresponding set operator. In the following we will show how lineage is calculated given specific examples.

Union

Consider an union operator on relations S and T producing relation R, where $r_x.\lambda$ is the lineage of r_x

1. For each tuple s_i in S, add it to R producing r_j having lineage $s_i.\lambda$
2. For each tuple t_i in T, add it to T producing r_j having lineage $r_i.\lambda$
3. For each tuple r_i in R, if there exists a duplicate r_j where $i \neq j$, set the lineage of r_i to $r_i.\lambda = r_i.\lambda \vee r_j.\lambda$ and remove r_j afterwards

Consider query $P \cup^T P'$ which is rewritten as $\mathcal{N}_{Name,Dest}(P; P') \cup \mathcal{N}_{Name,Dest}(P'; P)$. Given the normalized relations np and np' the union of those equals the following.

	$Name$	$Dest$	T	λ
r_1	Ann	Zurich	[3, 8)	$p_1 \vee p'_1$
r_2	Ann	Zurich	[9, 14)	p_2
r_3	Mark	Basel	[6, 12)	p_3
r_4	Jim	Luzern	[5, 10)	p_4
r_5	Tina	Bern	[10, 13)	$p_5 \vee p_6 \vee p'_2 \vee p'_3$

Figure 2.11: $\mathcal{N}_{P.Dest=P'.Dest}(P; P') \cup \mathcal{N}_{P.Dest=P'.Dest}(P'; P)$

The lineage of a result tuple corresponds to the lineage of the input tuple, as each input tuple is added to the result relation without further dependencies. However, as the union operator is duplicate eliminating in relational algebra, r_1 derives from intermediate tuples np_1 and np'_1. On the other hand, r_5 derives from intermediate tuples np_5, np_6, np'_2 and np'_3 producing $r_6.\lambda = np_5.\lambda \vee np_6.\lambda \vee np'_2.\lambda \vee np'_3.\lambda = p_5 \vee p_6 \vee p'_2 \vee p'_3$.

Intersection

Consider an intersection operator on relations S and T producing relation R, where $\lambda.r_x$ is the lineage of r_x

1. For each tuple s_i in S, find all identical tuples t_j in T
2. Create a boolean expression b_k belonging to s_i and set b_k to true
3. For each tuple t_j matching with s_i, set $b_k = b_k \vee t_j.\lambda$
4. For each non-empty boolean expression b_k belonging to some s_i, set lineage of s_i to $s_i.\lambda = s_i.\lambda \wedge b_k$ and add s_i to R
5. For each tuple r_i in R, if there exists a duplicate r_j where $i \neq j$, set the lineage of r_i to $r_i.\lambda = r_i.\lambda \vee r_j.\lambda$ and remove r_j afterwards

Consider query $P \cap^T P'$ which is rewritten as $\mathcal{N}_{Name,Dest}(P; P') \cap \mathcal{N}_{Name,Dest}(P'; P)$. Given the normalized relations np and np' the intersection of those equals the following.

	$Name$	$Dest$	T	λ
r_1	Ann	Zurich	[3, 8)	$p_1 \wedge p'_1$
r_2	Tina	Bern	[10, 13)	$(p_5 \wedge (p'_2 \vee p'_3)) \vee (p_6 \wedge (p'_2 \vee p'_3))$

Figure 2.12: $\mathcal{N}_{P.Dest=P'.Dest}(P; P') \cap \mathcal{N}_{P.Dest=P'.Dest}(P'; P)$

Here, each tuple of the left relation np was added to R if there existed a tuple in the right relation np' having identical non-temporal and temporal attributes. Then, any duplicates in R were removed in a second step.

Regarding lineage, we searched all matches in the right relation np' for each tuple in np. For each match in the right relation np' belonging to the same tuple in the left relation, lineage was $\vee$-concatenated. This produced lineages $p'_2 \vee p'_3$ twice, as there were for both np_5 and np_6 the same matches np'_2 and np'_3 in the right relation.

Then, the produced complex boolean expression among the tuples of the right relation np', was $\wedge$-concatenated with the boolean expression of the tuple of the left relation np. This produced $p_5 \wedge (p'_2 \vee p'_3))$ and $(p_6 \wedge (p'_2 \vee p'_3))$.

However, since the union operator is duplicate eliminating and since np_5 and np_6 are duplicates, the duplicate eliminating algorithm was applied, producing $(p_5 \wedge (p'_2 \vee p'_3)) \vee (p_6 \wedge (p'_2 \vee p'_3))$.

Difference

Consider a difference operator on relations S and T producing relation R, where $r_x.\lambda$ is the lineage of r_x

1. Add each tuple s_i in S to R producing r_i, where $r_i.\lambda = s_i.\lambda$
2. Create a boolean expression b_k belonging to r_i and set b_k to true
3. For each tuple t_j in T matching with the same tuple r_i in R, set $b_k = b_k \wedge \neg t_j.\lambda$
4. For each tuple r_i in R set lineage of r_i to $r_i.\lambda = r_i.\lambda \wedge b_k$
5. For each tuple r_i in R, if there exists a duplicate r_j where $i \neq j$, set the lineage of r_i to $r_i.\lambda = r_i.\lambda \vee r_j.\lambda$ and remove r_j afterwards

Consider query $P -^T P'$ which is rewritten as $\mathcal{N}_{Name,Dest}(P; P') - \mathcal{N}_{Name,Dest}(P'; P)$. Given the normalized relations np and np' the difference of those equals the following.

	$Name$	$Dest$	T	λ
r_1	Ann	Zurich	[3, 8)	$p_1 \land \neg p'_1$
r_2	Ann	Zurich	[9, 14)	p_2
r_3	Mark	Basel	[6, 12)	p_3
r_4	Jim	Luzern	[5, 10)	p_4
r_5	Tina	Bern	[10, 13)	$(p_5 \land \neg p'_2 \land \neg p'_3) \lor (p_6 \land \neg p'_2 \land \neg p'_3)$

Figure 2.13: $\mathcal{N}_{P.Dest=P'.Dest}(P; P') \cap \mathcal{N}_{P.Dest=P'.Dest}(P'; P)$

Here, each tuple of the left relation np was added to R independent whether there existed an identical tuple in the right relation np'. Then, any duplicates in R were removed in a second step.

Regarding lineage, we searched all matches in the right relation np' for each tuple in np. For each match in the right relation np' belonging to the same tuple in the left relation, the inverse lineage was $\land$-concatenated. This produced lineages $\neg p'_2 \land \neg p'_3$ twice, as there were for both np_5 and np_6 the same matches np'_2 and np'_3 in the right relation.

Then, the produced complex boolean expression among the tuples of the right relation np', was $\land$-concatenated with the boolean expression of the tuple of the left relation np. This produced $p_5 \land \neg p'_2 \land \neg p'_3$ and $p_6 \land \neg p'_2 \land \neg p'_3$.

However, since the difference operator is duplicate eliminating and since np_5 and np_6 are duplicates, the duplicate eliminating algorithm was applied, producing $(p_5 \land \neg p'_2 \land \neg p'_3) \lor (p_6 \land \neg p'_2 \land \neg p'_3)$.

2.3.6. Join operations

For any join operator ($\bowtie, ⟖, ⟕, ⟗$) we first must apply alignment on both relations using the other relation each. Then the actual join operation can be executed, before the absorb operator will detect and remove any duplicates.

Regarding lineage, the alignment creates for each tuple a set of tuples, where each of those tuples will have the same lineage as the tuple which created it. After the join operation is processed, the lineage of the result tuple corresponds to the $\land$-concatenation of the lineages matching tuples in the join operation.

Finally, the reductions rules specify to apply the absorb operator, which eliminates tuples having same non-temporal attributes, but time intervals being a subset of some other tuple. However, as it is up to further studies to find an adequate approach computing lineage for this temporal operator, we omit it. This also in regards that in most cases such duplicates do not even exist.

Consider the following example $P \bowtie_{Loc} W$ which is rewritten as $\alpha((P\Phi_{Loc}W)⟖_{Loc}(W\Phi_{Loc}P))$. The application of alignment produces the following results

The application of the aggregation operator produces the following result:

	$Name$	$Dest$	T	λ
ap_1	Ann	Zurich	[3, 5)	p_1
ap_2	Ann	Zurich	[5, 8)	p_1
ap_3	Ann	Zurich	[9, 10)	p_2
ap_4	Ann	Zurich	[9, 14)	p_2
ap_5	Mark	Basel	[6, 8)	p_3
ap_6	Mark	Basel	[8, 12)	p_3
ap_7	Jim	Luzern	[5, 10)	p_4
ap_8	Tina	Bern	[10, 11)	p_5
ap_9	Tina	Bern	[11, 13)	p_5
ap_{10}	Tina	Bern	[10, 11)	p_6
ap_{11}	Tina	Bern	[11, 13)	p_6

Figure 2.14: $P\Phi_{Dest=Loc}W$

	Loc	$Weather$	T	λ
aw_1	Basel	Sun	[1, 6)	w_1
aw_2	Basel	Sun	[6, 8)	w_1
aw_3	Bern	Rain	[11, 13)	w_2
aw_4	Bern	Rain	[13, 17)	w_2
aw_5	Zurich	Snow	[5, 8)	w_3
aw_6	Zurich	Snow	[8, 9)	w_3
aw_7	Zurich	Snow	[9, 10)	w_3
aw_8	Zurich	Fog	[8, 9)	w_4
aw_9	Zurich	Fog	[9, 14)	w_4
aw_{10}	Zurich	Fog	[14, 15)	w_4

Figure 2.15: $W\Phi_{Dest=Loc}P$

Here tuple p_2 created tuples ap_3 and ap_4 as p_2 intersects the time interval of w_3 and w_4. Moreover, lineage of ap_3 and ap_4 were set to $ap_3.\lambda = p_2.\lambda$ and $ap_4.\lambda = p_2.\lambda$ as they were created from the same tuple. Accordingly, this was done for all other tuples as well.

The application of the join operation produces the following result:

	$Name$	$Dest$	Loc	$Weather$	T	λ
r_1	Ann	Zurich	NULL	NULL	[3, 5)	p_1
r_2	Ann	Zurich	Zurich	Snow	[5, 8)	$p_1 \wedge w_3$
r_3	Ann	Zurich	Zurich	Snow	[9, 10)	$p_2 \wedge w_3$
r_4	Ann	Zurich	Zurich	Fog	[9, 14)	$p_2 \wedge w_4$
r_5	Mark	Basel	Basel	Sun	[6, 8)	$p_3 \wedge w_1$
r_6	Mark	Basel	NULL	NULL	[8, 12)	p_3
r_7	Jim	Luzern	NULL	NULL	[5, 10)	p_4
r_8	Tina	Bern	NULL	NULL	[10, 11)	p_5
r_9	Tina	Bern	Basel	Rain	[11, 13)	$p_5 \wedge w_2$
r_{10}	Tina	Bern	NULL	NULL	[10, 11)	p_6
r_{11}	Tina	Bern	Basel	Rain	[11, 13)	$p_6 \wedge w_2$

Figure 2.16: $(P\Phi_{Dest=Loc}W)⟗_{Loc}(W\Phi_{Dest=Loc}P)$

Consider a join operation on relations S and T having join clause θ producing relation R, where $r_x.\lambda$ is the lineage of r_x

1. For each tuple s_i in S, normalize it using t_j in T and set the lineage of each tuple as_k in the created tuple sets as $s_i.\lambda$
2. For each tuple t_i in T, normalize it using s_j in S and set the lineage of each tuple at_k in the created tuple sets as $t_i.\lambda$
3. Apply join operation on relations having tuples as_k and at_k using join clause θ , producing r_i, where the lineage of r_i corresponds to $r_i.\lambda = as_k.\lambda \wedge at_k.\lambda$

3. Confidence evaluation through lineage

Lineage as a complex boolean expression consists of boolean variables and operators. As each boolean variable represents an event which has a probability of being true, final confidence can be computed while evaluating lineage. This means that we create a truth table for given lineage, which allows us to evaluate for which combinations of boolean types lineage is true. Then, by creating a probability table emerging from the result of the truth table, final confidence can be computed.

To illustrate this algorithm, recap example from Section 2.2, especially Figure 2.16, where we performed the query $(P\Phi_{Dest=Loc}W)⟗_{Loc}(W\Phi_{Dest=Loc}P)$. Assume we extend this query with a selection $\sigma_{T=[9,10)}$ on it. This produces:

	$Name$	$Dest$	Loc	$Weather$	T	λ
r_1	Ann	Zurich	Zurich	Snow	[9, 10)	$p_2 \wedge w_3$

Figure 3.1: $\sigma_{T=[9,10)}((P\Phi_{Dest=Loc}W)⟗_{Loc}(W\Phi_{Dest=Loc}P))$

Given any lineage, we can compute the corresponding confidence. For the given example above, there is only one result tuple to be evaluated, but the following algorithm could be applied for any number of result tuples, as the calculations are made for each result tuple independently.

In a first step, we create a truth table and evaluate it for given lineage. For this, boolean variables being true are represented with a 1, while boolean variables being false are represented with a 0. Then, for each row in the truth table, we take the specific boolean type each boolean variable represents and replace them in the lineage expression. If lineage evaluates to true, we set the evaluation of the corresponding row to true (1) and false (0) otherwise.

	p_2	w_3	eval
t_1	0	0	0
t_2	0	1	0
t_3	1	0	0
t_4	1	1	1

Figure 3.2: Evaluated truth table for $p_2 \wedge w_3$

This evaluation allows us to see for which combination of specific boolean types, lineage evaluates to true. Regarding our example in Figure 3.2, the only combination of specific boolean types which evaluates to true is entry t_4 where the boolean variables p_2 and w_3 are both being represented with the specific boolean type being true.

In order to compute final confidence, we only have to consider those combinations which evaluated to true, as we want to retrieve the probability of the event of the result tuple being true.

To achieve this, we have to create a probability table. For this, we look up the corresponding probability of the tuple the boolean variable belongs to. Depending whether the boolean variable was represented by a boolean type being true or false, either the probability p of the event being true, or its opposite $1 - p$ is taken. Then, for each row, those probabilities are multiplied with each other, before all probabilities are summed up over all rows. This calculation can also be seen in the following pseudo-code example:

```
evaluateP(truthTableRow) {
    rowP = 1;
    if(truthTableRow.eval == 0) {
        return 0; -- given boolean type combination did not evaluate to true
            for lineage
    }
    for each entry in truthTableRow {
        if(entry == 1) -- boolean type was set to true in the truth table
            rowP = rowP * entry.p;
        else
            rowP = rowP * (1 - entry.p);
    }
    return rowP;
}

-- main function
truthTable = {...}; -- evaluated truth table
finalP = 0; -- final confidence
for each truthTableRow in truhTable {
    rowP = evaluateP(truthTableRow);
    finalP = finalP + rowP;
}
```

Given our example, the application of this algorithm evaluates to the following result. Please note that '—' relates to rows which we did not evaluate any further, as the combination of the corresponding specific boolean types infers lineage being false.

	p_2	w_3	eval p
t_1	—	—	0.00
t_2	—	—	0.00
t_3	—	—	0.00
t_4	0.50	0.70	0.35
		sum	0.35

Figure 3.3: Evaluated probability table for $p_2 \wedge w_3$

Recap Figure 3.2 where we found out that $p_2 \wedge w_3$ evaluates to true for the combination represented by t_4. For this combination of boolean types, we took the probability of the event being represented by the boolean variable. As p_2 was set to true in entry t_4 in Figure 3.2, we take the represented probability of the event being true, $p_2 = 0.5$. Analogous, for w_3, where we take $w_3 = 0.7$ as the represented specific boolean type by w_3 for entry t_4 was 1 (true).

In a final step, the probabilities of each entry are multiplied with each other. Then, we take the sum over all entries. This will represent the final confidence of the event being true represented by the given lineage. These even holds if time intervals were adjusted, as an event is not more likely to happen if the time interval is shortened. Therefore, r_1s final confidence equals to 0.35.

The set-up of the truth table assures that the final confidence, the sum over all entries, will be zero in case that for no combination of specific boolean types lineage evaluated to true. On the opposite, the probability will be one if lineage evaluates to true for all possible combinations of specific boolean types. As no outcome can exceed this boundary, computed confidence will always be between zero and one.

4. Implementation

In this section we will first give a brief overview about how queries are executed on the PostgreSQL server. 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.

We then move on explaining how relations must be defined and how queries can be executed. Afterwards we will show how we implemented the lineage computation and the confidence computation. However, we are not going to explain each line we changed in the code, but rather giving a broad overview about the basic concept that we used.

4.1. Postgres

Once a connection to the Postgres server is established, the user is able to execute queries. Each query will go through different stages, before either an error or a result will be returned.

Firstly, 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 the 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. Database setup and usage synopsis

In order to compute lineage and confidence values, each relation must be a temporal probabilistic relation. This means that each relation must specify a column ts and te, both being of type date and specifying the time interval, from when (including) until when (excluding) the event the tuple represents holds. Moreover, there must be an attribute of type numeric that is called p, which defines the probability of the event's occurrence.

Regarding query execution, queries must contain keywords such that lineage respectively confidences get computed. We added the keywords *LINEAGE* and *CONF*. If either of both keywords is given in the user's query the system will compute lineage. In order to have the query result show the computed lineage expression, one must specify *LINEAGE*, while *CONF* must be written to get the confidence value computed and displayed. If none of the keywords is given in the query lineage will not be computed. Both keywords can be used independently and must be defined right before the so called SELECT-list. See line 2 in the following synopsis. For the complete synopsis see Appendix C.

```
1 SELECT  [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
2     [ [] | CONF | LINEAGE | CONF LINEAGE | LINEAGE CONF]
3     * | expression [ [ AS ] output_name ] [, ...] -- SELECT-list
4     [ FROM from_item [, ...] ]
5     [ WHERE condition ]
6 ...
```

4.3. 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 that was entered by the user in such a way that lineage is computed as well. To do so we add calls to self defined PL/pgSQL-functions and aggregate-function, 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 a PL/pgSQL-function and passing the computed lineage as an argument to it. This function then evaluates with the algorithm described in Section 3 the confidence value for every result tuple.

In the Appendix additional information to the implementation is given. The files that were changed for the implementation described in this paper are shown in Appendix A. And in Appendix B the functions and aggregate-functions that were created are described.

4.4. Lineage as system column

As written in Section 2.1 lineage is conceptually represented as a boolean expression, but in therms of implementation it is represented as a string. For stored relations, henceforth called base relations, lineage expressions must be an unique identifier of the tuple itself. But for derived relations the lineage expressions are concatenations of the lineage expression of base relations and boolean operators.[1] To store those complex lineage expressions we created a system column called 'lineage' of type text. Since the fields of type text are of undefined length it is optimal for this purpose, because it is likely for lineage expressions to grow fast.

To create the lineage expressions for the base relations, 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 identifers, short OID)

[1]To represent lineage expressions as text we replaced all boolean operators with mathematical ones. This means that $\wedge$ is represented with $*$, $\vee$ with $+$ and $\neg$ with $-$.

of the relation and the tuple respectively. Those are automatically created by the system.[2] 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.[3] By combining the 'tableoid' and the 'oid' we are able to uniquely identify each tuple within the database.

Lineage is defined as system column for the ease of use purposes. To be exact we do not need to care about if the relations we make manipulations on are base relations ore derived relations. We can simply access the relations 'lineage' column. This is because the column is defined for base tuples in the same way as for derived tuples. In addition, 'lineage' must not be defined when relations are created, since we make use of the already existing identifiers.

4.5. Lineage computation

As mentioned before we compute lineage by transforming the queries entered by the users to other valid SQL-queries.

In the following subsections we show in detail what transformations are done for the different kinds of SELECT-Statements. It is always shown what query the user enters and how it is transformed in order to get the correct lineage expression for every tuple. For simplification reasons we used abbreviations to express subqueries that take care of the temporal adjustment. *pNp* for example stands for *people NORMALIZE people ON true* or *pAw* for *people ALIGN weather ON dest=loc* respectively. The transformation of those statements is shown in the first subsection and it would be done similar in all other statements.

4.5.1. Temporal adjustment

Normalization

As for normalization the lineage expression of the tuple of the left relation is also the lineage expression of the resulting tuple, we simply select the original lineage expression of the left relation as the lineage expression of the resulting tuples.

```
SELECT LINEAGE *
FROM (people NORMALIZE weather ON
    dest=loc) x;
```

SELECT-Statement entered by the user

```
SELECT *, people.lineage
FROM (people NORMALIZE weather ON
    dest=loc) x;
```

Transformed SELECT-Statement that will be executed

[2]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.

[3]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.

Alignment

The transformation for alignment is similar to the one for normalization. We also select the original lineage expression of the left relation as the lineage expression of the resulting tuples.

```
SELECT LINEAGE *
FROM (people ALIGN weather ON dest=
    loc) x;
```

SELECT-Statement entered by the user

```
SELECT *, people.lineage
FROM (people ALIGN weather ON dest=
    loc) x;
```

Transformed SELECT-Statement that will be executed

4.5.2. Selection

In case of a simple selection, we only need to add the lineage column to the selection list.

```
SELECT LINEAGE *
FROM people
WHERE name = 'Ann';
```

SELECT-Statement entered by the user

```
SELECT *, lineage
FROM people
WHERE name = 'Ann';
```

Transformed SELECT-Statement that will be executed

4.5.3. Projection

Since projections are not duplicate eliminating in SQL the transformation is the same as for selections. We also add the lineage column to the selection list.

```
SELECT LINEAGE name, ts, te
FROM pNp;
```

SELECT-Statement entered by the user

```
SELECT name, ts, te, lineage
FROM pNp;
```

Transformed SELECT-Statement that will be executed

4.5.4. Duplicate eliminator

If we want duplicates to be eliminated we can use the keyword *DISTINCT* in SQL. If this is the case the lineage expressions of all tuples that result in one final tuple must be concatenated. This resembles the main idea of an aggregation. We therefore transform the SELECT-Statement such that we group by all expressions that are given in the SELECT-list and we add an aggregation-function that concatenates the lineage expressions. Like that the Grouping makes sure that the resulting tuples are distinct. We use the aggregation-function *lineage_or*, which concatenates all lineage expressions with '+' to a resulting lineage expression. See Appendix B for the definition of the aggregation-function.

```
SELECT DISTINCT LINEAGE name, ts,
    te
FROM pNp;
```

SELECT-Statement entered by the user

```
SELECT name, ts, te, lineage_or(
    lineage) AS lineage
FROM pNp
GROUP BY name, ts, te;
```

Transformed SELECT-Statement that will be executed

PostgreSQL also allows the use of *DISTINCT ON* where the resulting tuples are only distinct on the columns defined in the *ON*-Clause. By definition it keeps the value of the first row for all columns that are not defined in the *ON*-Clause. The transformation here is similar to this of a simple *DISTINCT*, the only difference is that we do not need to group by the columns, which are not in the *ON*-Clause and that we use another aggregation-function for those columns. The aggregation-function *lineage_first*, simply returns the first element of the group. See Appendix B for the definition of the aggregation-function.

```
SELECT DISTINCT ON (ts, te) LINEAGE
      name, ts, te
FROM pNp;
```

SELECT-Statement entered by the user

```
SELECT lineage_first(name), ts, te,
      lineage_or(lineage) AS lineage
FROM pNp
GROUP BY ts, te;
```

Transformed SELECT-Statement that will be executed

4.5.5. High Aggregation

For aggregations we must concatenate the lineage expressions of all tuples of the same group with '*' to get the resulting lineage expression. Therefore we add the aggregation-function *lineage_and*, which handles this concatenation, to the selection list. See Appendix B for the definition of the aggregation-function.

```
SELECT LINEAGE dest, ts, te, count
      (*)
FROM pNp
GROUP BY dest, ts, te;
```

SELECT-Statement entered by the user

```
SELECT dest, ts, te, count(*),
      lineage_and(lineage) AS lineage
FROM pNp
GROUP BY dest, ts, te;
```

Transformed SELECT-Statement that will be executed

A special case, which need to be considered separately is the useage of *DISTINCT* and *GROUP BY* in the same SELECT-Statement. By definition SQL first executes the grouping and at the end the duplicate elimination. Since we transform SELECT-Statements including *DISTINCT* to SELECT-Statements including *GROUP BY* as we saw above, we need to create a Sub-SELECT-Statement here. We take the transformation of the SELECT-Statement without *DISTINCT* as the Sub-SELECT-Statement and group by all its resulting columns to make them distinct. To concatenate the lineages of identical groups we again add the *lineage_or* aggregate-function.

```
SELECT DISTINCT LINEAGE ts , te ,
    count(*)
FROM pNp
GROUP BY ts , te;
```

SELECT-Statement entered by the user

```
SELECT stmt.ts , stmt.te , stmt.count
    , lineage_or(stmt.lineage) AS
    lineage
FROM
    (SELECT ts , te , count(*) ,
        lineage_and(lineage) AS
        lineage
    FROM pNp
    GROUP BY ts , te
    ) stmt
GROUP BY stmt.ts , stmt.te , stmt.
    count;
```

Transformed SELECT-Statement that will be executed

4.5.6. Join operations

In case of join operations the lineage expression of both tuples that contribute to a result tuple must be concatenated. We add the function *concat_lineage_and2* to the SELECT-list and pass the lineage expressions of both given relations as arguments. The *concat_lineage_and2* function concatenates the given lineages with '*' if both are defined, otherwise it simply returns the lineage that is defined, like that we can also support outer joins. See Appendix B for the definition of the function.

```
SELECT LINEAGE *
FROM pAw FULL JOIN wAp ON dest=loc;
```

SELECT-Statement entered by the user

```
SELECT *, concat_lineage_and2(pAw.
    lineage , wAp.lineage) AS
    lineage
FROM pAw FULL JOIN wAp ON dest=loc;
```

Transformed SELECT-Statement that will be executed

4.5.7. Cartesian product

The cartesian product was only implemented for reasons of completeness. It is actually not used for temporal operations since the temporal cartesian product is by the reduction rules replaced with join operations. Since our implementation also works for non-temporal queries, we came up with a solutions as well. The idea is to concatenate the lineage expressions of all relations defined in the FROM-Clause to get the lineage expression of the resulting tuples.

```
SELECT LINEAGE *
FROM people , weather , people AS p;
```

SELECT-Statement entered by the user

```
SELECT *, '(' || people.lineage ||
    ')*(' || weather.lineage || ')
    *(' || p.lineage || ')' as
    lineage
FROM people , weather , people AS p;
```

Transformed SELECT-Statement that will be executed

4.5.8. Set Operations

Set Operations are a bit more complicated, since not only must we compute lineage but also we must produce different query results in some cases. They are therefore transformed in two steps. At first a transformation using SELECT-Statements with *DISTINCT* is done. Those statements are then transformed in a second step as normal SELECT-Statements with *DISTINCT*, as we saw in Section 4.5.4. Set operations also can be executed with the keyword *ALL*, in which case duplicates are not eliminated and we simply do not add *DISTINCT* to the outer SELECT-Statement in the first transformation step.
To make the examples not too confusing we use place-holders $1 respectively $2 for the following subqueries:

```
$1 := SELECT *, lineage FROM p1Np2;
$2 := SELECT *, lineage FROM p2Np1;
```

Union

In the first transformation step we add the SELECT-Statement with *DISTINCT* and add lineage to the SELECT-list of both sides of the union. Since the lineage expression is selected at both sides duplicates will not be eliminated by the union operation as the lineage expression is not equal. And in the second step the lineage expressions can get concatenated as we have seen for SELECT-Statement with *DISTINCT*.

```
    SELECT LINEAGE *
    FROM p1Np2
UNION
    SELECT *
    FROM p2Np1;
```

(1) SELECT-Statement entered by the user

```
SELECT DISTINCT LINEAGE *
FROM
    (
        $1
    UNION
        $2
    ) stmt;
```

(2) Intermediate transformation of the SELECT-Statement

```
SELECT *, lineage_or(lineage) AS lineage
FROM
    (
        $1
    UNION
        $2
    ) stmt
GROUP BY name, dest, ts, te;
```

(3) Transformed SELECT-Statement that will be executed

Intersection

The intersection is a bit more complex since we need to first make sure that we do not have any duplicates on the right side and at the same time we ensure that we concatenate the lineage expressions of all tuples on the right side that are equal. By joining the left side with the resulting tuples from the right side we make sure that only the tuples which exist on both sides are in the result. The final transformations are done as we saw in the sections above.

```
    SELECT LINEAGE *
    FROM p1Np2
INTERSECT
    SELECT *
    FROM p2Np1;
```

(1) SELECT-Statement entered by the user

```
SELECT DISTINCT LINEAGE s1.*
FROM
        ($1) s1
    JOIN
        (SELECT DISTINCT *
        FROM ($2) s2
        ) os2
    ON s1.name=os2.name AND s1.dest
       =os2.dest AND s1.ts=os2.ts
       AND s1.te=os2.te;
```

(2) Intermediate transformation of the SELECT-Statement

```
SELECT s1.*, lineage_or(concat_lineage_and2(s1.lineage, os2.
    lineage)) AS lineage
FROM
        ($1) s1
    JOIN
        (SELECT *, lineage_or(lineage) AS lineage
        FROM ($2) s2
        GROUP BY name, dest, ts, te
        ) os2
    ON s1.name=os2.name AND s1.dest=os2.dest AND s1.ts=os2.ts AND
        s1.te=os2.te
GROUP BY s1.name, s1.dest, s1.ts, s1.te;
```

(3) Transformed SELECT-Statement that will be executed

Except

The except operation is similar to the intersection. There are two differences. The first is that we use a left join instead of a natural join. Because of that we get all tuples of the left side that do not match with any of the right side, which is the basic result set of the except operation. But in addition we also get the tuples that match with the right side. In this case the lineage expressions of all tuples on the right side that match the given tuple on the left side, must be concatenated with '*-'. We achieve this by simply using the aggregation-function *lineage_andnot* instead of the aggregation-function *lineage_or* which is normally used when a SELECT-Statement including *DISTINCT* is transformed. This is the second difference to the implementation of the intersection operation.

```
    SELECT LINEAGE *
    FROM p1Np2
EXCEPT
    SELECT *
    FROM p2Np1;
```

(1) SELECT-Statement entered by the user

```
SELECT DISTINCT LINEAGE s1.*
FROM
        ($1) s1
    LEFT JOIN
        (SELECT DISTINCT *
        FROM ($2) s2
        ) os2
    ON s1.name=os2.name AND s1.dest
        =os2.dest AND s1.ts=os2.ts
        AND s1.te=os2.te;
```

(2) Intermediate transformation of the SELECT-Statement

```
SELECT s1.*, lineage_or(concat_lineage_and2(s1.lineage, os2.
    lineage)) AS lineage
FROM
        ($1) s1
    LEFT JOIN
        (SELECT *, lineage_andnot(lineage) AS lineage
        FROM ($2) s2
        GROUP BY name, dest, ts, te
        ) os2
    ON s1.name=os2.name AND s1.dest=os2.dest AND s1.ts=os2.ts AND
        s1.te=os2.te
GROUP BY s1.name, s1.dest, s1.ts, s1.te;
```

(3) Transformed SELECT-Statement that will be executed

4.6. Confidence computation

In our implementation the computation of the confidence value is done at the very end and only for the whole result tuples. We created a PL/pgSQL-function (*lineage_conf*), which computes the confidence value with the algorithm we described in Section 3. For every result tuple we call this function and pass the computed lineage as its argument. We see the following example, where the user defined the keyword *CONF* instead of *LINEAGE*, in this case we add the call to this function.

```
SELECT DISTINCT CONF name, ts, te

FROM pNp;
```

SELECT-Statement entered by the user

```
SELECT name, ts, te, lineage_conf(
    lineage_or(lineage)) AS p
FROM pNp
GROUP BY name, ts, te;
```

Transformed SELECT-Statement that will be executed

The implemented algorithm retrieves the confidence values of the tuples of the base relations by calling our function *lineage_prob*. This function returns the confidence value that is stored in the tuple referenced by the given lineage expression.

5. Evaluation

In this section we present the evaluation we did to test our implementation. As first we will show what the limitations of our approach are, this means what are the queries which are not taken care of. And afterwards we will show how the implementation performs on a synthetically created dataset.

5.1. Lineage computation

The given implementation approach has some minor drawbacks, which are presented in the following paragraphs.

Ambiguity of columns

One problem is ambiguity, which can occur after join operations. If a join operation produced a intermediate relation having columns with similar names, any distinct, difference, intersect or duplicate eliminating union operation will fail. This is because SELECT-Statements with *DISTINCT* are transformed to SELECT-Statements with *GROUP BY*, where each column must be listed separately. No ambiguity is allowed here. The problem also occurs for difference and intersection, since those SELECT-Statements are transformed using a join operation. Here, the matching clause of the join is the problem when ambiguity occurs.

The following SQL-query is an example of a query that fails. The problem here is that the Sub-SELECT-Statement has every column twice with the same name. Therefore it is impossible to group by those columns.

```
SELECT DISTINCT LINEAGE *
FROM
    (SELECT *
    FROM people JOIN people AS p ON people.name=p.name
    ) stmt;
```

Usage of views

A further problem occurs when working with views. If the user does not specify the keyword *LINEAGE* while creating a view, no further lineage or confidence computation can be done using this view afterwards. It is recommended to only use *LINEAGE* and not also *CONF* for views, since the presence of *CONF* leads to a table with a column p at the end, which would impede the usage of temporal operators for continuing queries.

The following statement shows how a view should be defined in order to be able to use it afterwards to compute lineage or confidence value respectively and in the meantime be able to perform temporal adjustments.

```sql
CREATE VIEW pzh AS
     SELECT LINEAGE *
     FROM people
     WHERE dest = 'Zurich';
```

Probability value in Where-Clause

Finally, another drawback occurs when working with the probability attribute p in SELECT-Statements. SELECT-Statements work perfectly fine even for comparisons with p unless they are not specified on base relations. On derived relations, no comparisons using p can be done, as the probability attribute is not calculated for any intermediate results. This corresponds also to examples we have shown in Section 2.1, where we omit the probability attribute for each intermediate result. In regards that it is very unlikely to have such a query, we omitted finding a complex solution.

The following query would fail, since p is not calculated when required.

```sql
SELECT CONF *
FROM
     (SELECT dest , count(*)
     FROM people
     GROUP BY dest
     ) stmt
WHERE p > 0.5;
```

Aggregation

In case of aggregations we decided to implement so called 'High Aggregation'. This means that we only consider the case, in which all tuples of a group that contribute to an aggregate are true. Suciu et. al. [4] suggest to calculate all possible outcomes of the aggregation. In the example in Figure 5.1 this means that we only compute the tuple r_1 instead of all 3 tuples.

P1 (People)

	$Name$	$Dest$	T	p
p_1	Ann	Zurich	[4, 5)	0.8
p_2	Jim	Zurich	[4, 5)	0.3

$\sigma_{T=[4,5)}({}_{Dest,T}\vartheta_{count(Name)}(\mathcal{N}_{Dest}(P1; P1)))$

	$Dest$	$count(Name)$	T	p
r_1	Zurich	2	[4, 5)	0.24
r_2	Zurich	1	[4, 5)	0.62
r_3	Zurich	0	[4, 5)	0.14

Figure 5.1: Aggregation

Absorb

For the implementation described in this paper we ignore the keyword *ABSORB*. Nothing will be absorbed when lineage respectively confidence values should be calculated.

5.2. Confidence evaluation

In regard of confidence evaluations the limitations are due to the complexity of the evaluation algorithm. Since the algorithm has a complexity of $O(2^k)$ where k is the number of distinct identifiers. This algorithm is executed for every result tuple, therefore the overall complexity is of $O(2^k * n)$ with n as the number of result tuples.

Runtime dependent on number of distinct identifiers

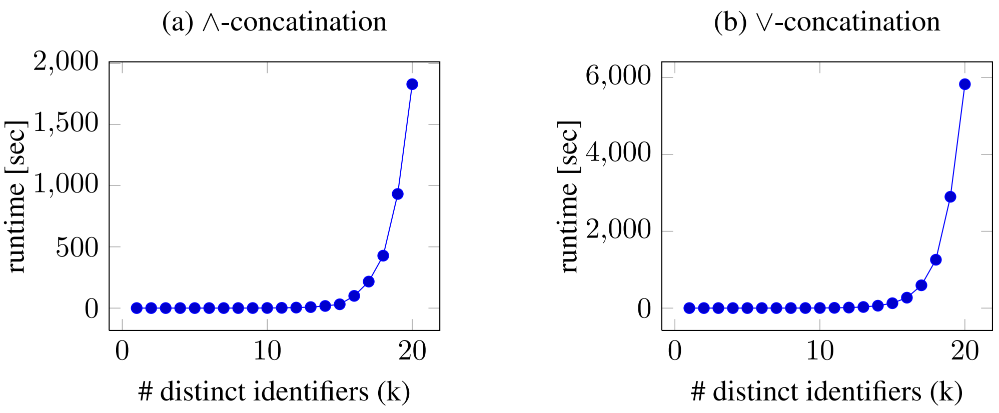


Figure 5.2: Runtime dependent on the number of distinct identifiers

As we expected, the runtime grows exponentially when the number of distinct identifiers grows. In Figure 5.2 we see that the runtime rapidly grows when the number of distinct identifiers grows about 15.

The computation of a confidence value whose lineage expression is only using ∨-operators takes about three times more time than one of equal length but using ∧-operators. This is because when ∧ is present only one entry of the truth table evaluates to true, but when ∨ is present all except one entry of the truth table evaluate to true. In this case much more calls to get the confidence value of the base tuples are required.

Runtime dependent on the number of tuples

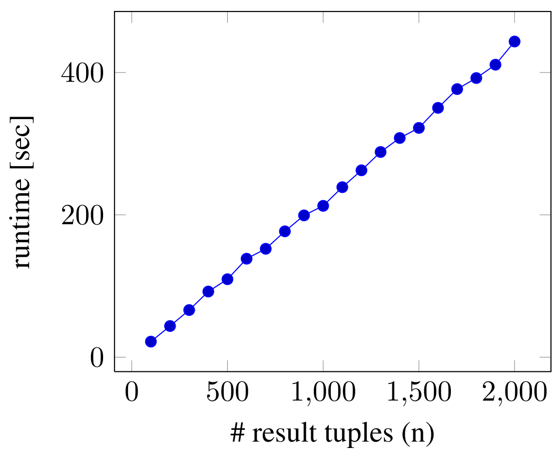


Figure 5.3: Runtime dependent on the number of tuples

In regards of the performance dependent on the number of result tuples, we can see in Figure 5.3 that the runtime grows linearly. This comes due to the fact that the confidence computation is executed for every result tuple similarly and therefore needs about the same amount of time. As we see the performance in this case is not very satisfying too. For 2000 result tuples the implementation already needs more than 7 minutes to execute.

Acknowledgements

Many thanks to Aikaterini Papaioannou, who supported us while implementing and writing the report. Moreover, we would like to thank Prof. Dr. Michael Böhlen, who has given us the opportunity to do the master project in this interesting topic.

References

[1] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Temporal alignment. In *SIGMOID Conference*, pages 433-444, 2012.

[2] Anish Das Sarma, Martin Tehobald, and Jennifer Widom. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. ???, 2007.

[3] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Query time scaling of attribute values in interval timestamped databases. In *ICDE*, 2013.

[4] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Dataabses*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[5] PostgreSQL. ”PostgreSQL 9.2.4 Documentation”. Web. 28 Aug. 2013. http://www.postgresql.org/docs/9.2/static/index.html

A. Changed Files

The stable version 9.2.4 of PostgreSQL Core Distribution was taken as basis and the following files were modified for the implementation described in this paper. To each file a brief description of the undertaken modifications is given:

- **\src\backend\access\common\heaptuple.c**

  Create and retrieve the unique tuple identifier as a boolean variable of type text (for lineage)

- **\src\backend\catalog\genbki.pl**

  Definition of lineage as a system column

- **\src\backend\catalog\heap.c**

  Definition of lineage as a system column

- **\src\backend\catalog\lineage.sql**

  Definition of functions and aggregates for lineage concatenation and confidence computation

- **\src\backend\catalog\Makefile**

  Inclusion of lineage.sql into the source directory

- **\src\backend\nodes\copyfuncs.c**

  Handle the new variables defined the SelectStmt

- **\src\backend\nodes\outfuncs.c**

  Handle the new variables defined the SelectStmt

- **\src\backend\parser\analyze.c**

  Main part of the implementation, where we rewrite the queries and add calls to self developed PL/pgSQL-functions for lineage concatenation and confidence computation

- **\src\backend\parser\gram.y**

  Definition of *CONF* and *LINEAGE* in the grammar

- **\src\backend\parser\parse_node.c**

  Definition of default values for new variables in ParseState

- **\src\backend\parser\parse_relation.c**

  Ignorance of lineage and probability column for .* expansion

- **\src\backend\parser\parse_target.c**

  Ignorance of probability column. Additional aggregation function for lineage and probability column if necessary

- **\src\backend\utils\misc\postgresql.conf.sample**

  Set the usage of OIDs as unique identifiers as default

- **\src\backend\initdb\initdb.c**

  Integration of the functions and aggregates defined in lineage.sql

- **\src\bin\pg_dump\pg_dump.c**

  Definition of lineage as a system column

- **\src\include\access\sysattr.h**

  Definition of lineage as a system column

- **\src\include\nodes\parsenodes.h**

  Definition of the LineageType which will keep track whether and how lineage shall be computed. Contains also further variables being added to the SelectStmt.

- **\src\include\parser\kwlist.h**

  Definition that *CONF* and *LINEAGE* is a preserved keyword

- **\src\include\parser\parse_node.h**

  Definition of further variables for the ParseState

B. Implemented PL/pgSQL-Functions & -Aggregates

We implemented the following functions and aggregates which are required for the lineage and confidence value computation. They are therefore automatically inserted into the database when initdb is executed.

B.1 $\vee$-concatenation

Function	concat_lineage_or(a text, b text)
Return value	text
Description	Concatenates lineage a with b using $+$
Example	$a = 1.1 * 2.1$ and $b = 3.1 * 4.1$ reveals to $1.1 * 2.1 + 3.1 * 4.1$

Aggregate	lineage_or(text)
Description	Concatenates a set of lineages belonging to the same group using $+$
Example	$1.1, 2.1 * 3.1, 4.1$ reveals to $1.1 + 2.1 * 3.1 + 4.1$

B.2 $\wedge$-concatenation

Function	concat_lineage_and(a text, b text)
Return value	text
Description	Concatenates lineage a with b using $*$, while placing b in brackets first due to the higher priority of $*$ against $+$. This is used for the lineage_and aggregation-function
Example	$a = 1.1+2.1$ and $b = 3.1+4.1$ reveals to $1.1+2.1*(3.1+4.1)$

Aggregate	lineage_and(text)
Description	Concatenates a set of lineages belonging to the same group using $*$
Example	$1.1, 2.1 + 3.1, 4.1$ reveals to $(1.1) * (2.1 + 3.1) * (4.1)$

Function	concat_lineage_and2(a text, b text)
Return value	text
Description	Concatenates lineage a with b using $*$, while placing a and b in brackets first due to the higher priority of $*$ against $+$
Example 1 Example 2 Example 3	$a = 1.1+2.1$ and $b = 3.1+4.1$ reveals to $(1.1+2.1)*(3.1+4.1)$ $a = 1.1 + 2.1$ and $b = NULL$ reveals to $1.1 + 2.1$ $a = NULL$ and $b = 3.1 + 4.1$ reveals to $3.1 + 4.1$

B.3 $\wedge\neg$-concatenation for difference operations

Function	concat_lineage_notand(a text, b text)
Return value	text
Description	Concatenates lineage a with b using $*-$, while placing a and b in brackets first due to the higher priority of $*$ against $+$
Example	$a = 1.1{+}2.1$ and $b = 3.1{+}4.1$ reveals to $(1.1{+}2.1){*}{-}(3.1{+}4.1)$

Aggregate	lineage_notand(text)
Description	Concatenates a set of lineages belonging to the same group using $*-$
Example	$1.1, 2.1 + 3.1, 4.1$ reveals to $(1.1) * -(2.1 + 3.1) * -(4.1)$

B.4 Function and aggregate to retrieve the lineage of the first element in the group

Function	lineage_first_agg (anyelement, anyelement)
Return value	anyelement
Description	Returns first left element
Example	$a = 1.1 + 2.1$ and $b = 3.1 + 4.1$ reveals to $1.1 + 2.1$

Aggregate	lineage_first(anyelement)
Description	Retreives the lineage of the first element in the group
Example	$1.1, 2.1 + 3.1, 4.1$ reveals to 1.1

B.5 Retrieve probability of a base tuple

Function	lineage_prob(lineage_expr text)
Return value	numeric
Description	Given that lineage_expr is an unique identifier, it retrieves the probability of the event of the identified tuple
Example	1.1 and tuple 1.1 = {"Ann", "Zurich", 0.80, '2013-01-01', '2013-01-03'} reveals to 0.80

B.6 Extract functions for boolean expressions

Function	lineage_vars(lineage_expr text)
Return value	text[]
Description	Given that lineage_expr is a lineage, this function retrieves all unique identifiers stored in the given expression
Example	$(1.1 + 2.1) * (1.1)$ reveals to {1.1, 2.1}

Function	lineage_tokenize(lineage_expr text)
Return value	text[]
Description	Given that lineage_expr is a lineage, this function retrieves all identifiers, operators and parenthesis stored in the given expression
Example	$(1.1 + 2.1) * (1.1)$ reveals to {(, 1.1, +, 2.1,), *, (, 1.1,)}

Function	lineage_postfix(lineage_expr text)
Return value	text[]
Description	Given that lineage_expr is a lineage, this function transforms lineage into an array in postfix transformation
Example	$(1.1 + 2.1) * (1.1)$ reveals to {1.1, 2.1, +, 1.1, *}

Function	lineage_true_or_false(vars text[], var text)
Return value	boolean
Description	Retrieves whether boolean variable var was set to true or false in the array of boolean variables
Example	{1.1 = 1, 2.1 = 0} and 2.1. reveals to 0 (false)

B.7 **Evaluation functions**

Function	lineage_evaluate(lineage_expr text, vars text[])
Return value	boolean
Description	Evaluates whether given lineage is true for the given boolean variables in the vars-array
Example	$(1.1) * (2.1) + 3.1$ and {1.1 = 1, 2.1 = 0, 3.1 = 1} reveals to 1 (true)

Function	lineage_conf(lineage_expr text)
Return value	numeric
Description	Computes the confidence value for a given lineage while using the functions defined above
Example	$(1.1) * (2.1)$ and {1.1 = {"Ann", "Zurich", 0.8, '2013-01-01', '2013-01-03'}, 2.1 = {"Marc", "Zurich", 0.6, '2013-01-01', '2013-01-03'}} reveals to 0.48

C. New SELECT-Statement synopsis

The complete SELECT-Statement synopsis with the newly added keywords, for lineage and confidence computation, *LINEAGE* and *CONF* respectively (line 3) is listed here [5].

```
1  [ WITH [ RECURSIVE ] with_query [, ...] ]
2  SELECT  [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
3      [ [] | CONF | LINEAGE | CONF LINEAGE | LINEAGE CONF]
4      * | expression [ [ AS ] output_name ] [, ...]
5      [ FROM from_item [, ...] ]
6      [ WHERE condition ]
7      [ GROUP BY expression [, ...] ]
8      [ HAVING condition [, ...] ]
9      [ WINDOW window_name AS ( window_definition ) [, ...] ]
10     [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
11     [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST |
          LAST } ] [, ...] ]
12     [ LIMIT { count | ALL } ]
13     [ OFFSET start [ ROW | ROWS ] ]
14     [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
15     [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]
16
17 where from_item can be one of:
18
19     [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
20     ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
21     with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
22     function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [,
           ...] | column_definition [, ...] ) ]
23     function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
24     from_item [ NATURAL ] join_type from_item [ ON join_condition | USING (
          join_column [, ...] ) ]
25
26 and with_query is:
27
28     with_query_name [ ( column_name [, ...] ) ] AS ( select | values | insert
           | update | delete )
29
30 TABLE [ ONLY ] table_name [ * ]
31
```