DEIM2022 J21-4

複合的データ分析処理に対する拡張来歴導出手法と性能評価

山田　真也† 　北川　博之†† 　天笠　俊之††† 　的野　晃整††††

† 筑波大学院 理工情報生命学術院 システム情報工学研究群　〒 305-8573 茨城県つくば市天王台 1 丁目 1-1
†† 筑波大学 国際統合睡眠医科学研究機構　〒 305–8575 茨城県つくば市天王台 1 丁目 1-1
††† 筑波大学 計算科学研究センター　〒 305–8577 茨城県つくば市天王台 1 丁目 1-1
†††† 産業技術総合研究所 人工知能研究センター　〒 135-0064 東京都江東区青海 2-4-7
E-mail: †yamada@kde.cs.tsukuba.ac.jp, ††kitagawa@cs.tsukuba.ac.jp,
†††amagasa@cs.tsukuba.ac.jp, ††††a.matono@aist.go.jp

あらまし　データ来歴（Lineage）とは分析結果がどの入力データによって導出されたかを示す情報のことである．来歴を活用することで分析処理のトレーサビリティを高めることができるため，これまで多くの研究が行われてきた．しかしながら近年の分析処理の特徴として，コンテンツデータ処理や AI 処理のような非常に高度な処理を含むことが挙げられる．これら複合的データ解析を伴う処理（複合的データ処理）では，単に元になった入力データを示す従来の来歴だけでは示された入力データから複合的データ解析がどのようなロジックで分析結果を導出したのかを理解することはできない．そのような背景から我々の先行研究では，分析結果の元になったデータ（Lineage）に加えて複合的データ解析における判断根拠（Reason）も併せて提示する拡張来歴（Augmented Lineage）を提案した．しかしながら先行研究では，拡張来歴を求めるために DBMS のクエリオプティマイザが作成する処理木の演算子の順番を変える必要があるという制約等があった．本稿では，処理木の演算子の順番を維持した拡張来歴の導出方法を提案する．また，LFW データセットを使った性能評価実験の結果を示す．

キーワード　来歴，トレーサビリティ，複合的データ処理，User Defined Function，AI 処理

1　はじめに

データ分析処理を意思決定に活用するためには，その分析結果のトレーサビリティが保証できることが重要である．データ来歴とは分析処理において分析結果を導出する元になった入力データを導出することを指し，トレーサビリティのために不可欠な情報であることからこれまでデータベースの分野において広く研究されてきた [1].

しかしながら，単に分析結果の元になった入力データを提示する従来のデータ来歴だけでは近年の分析処理に対して十分なトレーサビリティを保証することは困難である．近年の分析処理は画像やテキストのような多様なコンテンツデータに対する処理や AI 処理や機械学習による処理（AI/ML 処理）のような非常に複雑な処理を含んでいるという特徴がある．そのような複合的データ解析を伴う処理（複合的データ処理）では，来歴情報の利用者は単に元になった入力データだけでは，なぜ分析結果が導出されたのか理解することは困難である．このため，複合的データ処理に対するトレーサビリティのためには複合的データ解析における判断根拠の情報も併せて提示することが必要になる．ここで以下のような例を考える．

例 1. 図 *1* は金融機関が顧客のローン申請を，顧客に関する情報と学習モデルを使って審査する例を表している．ここで学習モデルは顧客の収入，借入金額，ローンの申請金額の *3* つの属性を入力として受け取ると，審査結果（良，可，不可）を返すものである．このとき，なぜ一人のローン申請だけ不可と判断されたのかを調べることを考える．従来のデータ来歴に基づくと，分析結果 ⟨ 不可, 1 ⟩ の導出の元になったデータ（{{⟨0002, 土浦次郎, 土浦市, 3200⟩}, {⟨0002, 1350, 12/17⟩}, {⟨0002, 1500, 12/19⟩}}）を提示することができるが，この情報だけではどうしてその入力データから学習モデルがその顧客に対して不可と判断したのかを理解することができず，どうして一人のローン申請だけが不可と判断されたのかはわからないままである．しかしながらこの場合，元になった入力データに加えて，学習モデルはこの顧客の借入金額とローン申請額が多いため不可と判断したというような学習モデルの判断根拠も併せて提示することができれば，より説明能力の高いトレーサビリティを実現することができる．

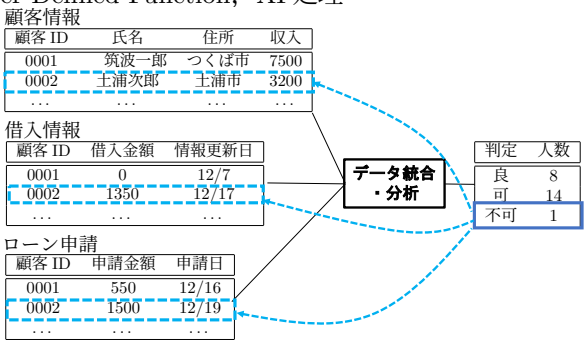

図 1　学習モデルを使ってローン審査を行う分析処理．破線で囲われているタプルは，出力タプル ⟨ 不可, 1 ⟩ の元になった入力タプルを表している．

我々の先行研究 [2] では，User Defined Function（UDF）を用いて複合的データ解析をモデル化し，従来の元になった入力データを提示するデータ来歴に加えて複合的データ解析における判断根拠（Reason）の情報も併せて提示する拡張来歴を提案することで，複合的データ処理に対するより適切なトレーサビリティの実現を可能にした．しかしながら先行研究には，拡張来歴を求めるために DBMS のクエリオプティマイザが作成する処理木の演算子の順番を変える必要があるという制約等があった．そこで本稿では処理木の演算子の順番を維持した拡張来歴の導出方法を提案する．また，LFW データセットを使った性能評価実験の結果を示す．

本稿の構成は以下の通りである．まず 2 節で関連研究を紹介する．その後，3 節で本稿で分析処理をモデル化するために使用するデータモデルを説明し，4 節で拡張来歴の定義を述べる．そして 5 節で拡張来歴の導出方法について説明し，6 節で提案手法の性能評価を行う．最後に Section 7 節で本稿をまとめる．

2 関 連 研 究

これまでデータ来歴はさまざまな分野で研究が行われてきている [3], [4]．データベースに対するクエリ結果の来歴を提示する手法として [5], [6], [7] はリレーショナルモデルに基づいている研究であり，リレーショナルクエリの結果に対して元になった入力データを提示する手法を提案している. [5] は Tracing Query と呼ばれるクエリを実行することで来歴を提示する手法で, [6], [7] は全ての入力データにあらかじめ識別子を割り当てておき分析処理を実行する際にそれらを分析結果まで継承することで来歴を求める手法である．さらにリレーショナルクエリ以外にも，XQuery や SPARQL のようなより一般的なクエリに対して来歴を提示するフレームワークも存在する [8], [9].

データベースに対するクエリ以外にも，より一般的な分析ワークフローにおける来歴研究も行われており, [10] はサイエンティフィックワークフローにおいて再現性を保証するために来歴を記録するフレームワークである．さらに近年では分散処理を対象として研究も行われており，Spark や Flink を用いた処理における来歴の導出手法が提案されている [11], [12].

より複雑な分析処理に対する来歴の提示に取り組んだ研究も存在する. [13] は UDF によってモデル化されるデータの抽出処理を伴う分析処理の来歴を導出する研究である．この研究では来歴として単に入力データを導出するだけでなく，入力データのどの部分を抽出したかという情報も併せて提示する手法の提案を行った．しかしながらこれまで説明したどのフレームワークにおいても，AI/ML 処理における判断根拠を含めた来歴についての議論は行われていない．

また一方で近年，AI/ML 処理における説明可能性に関する研究が広く行われている [14]. [15] のようなフレームワークを使用することによって AI/ML 処理の判断根拠を提示することができるようになっている．本稿の拡張来歴はこれまで AI や機械学習の分野で研究されてきた説明可能性を，データ来歴に統合することで複合的データ処理に対するトレーサビリティの実現に貢献するものである．

3 データモデル

本節では分析処理をモデル化するために使用するデータモデルを説明する．本稿では，複合的データ処理をリレーショナルモデルに基づいたオペレータによって構成されるタスクとしてモデル化する．

3.1 デ ー タ

本稿ではデータをテーブルの集合としてモデル化し，テーブル $T(A_1, \ldots, A_n)$ はタプル集合 $\{t_1, \ldots, t_p\}$ をもつ．また，テーブル T の属性集合を $\boldsymbol{T}$ と表す．$t.\boldsymbol{A}$ とはタプル t を属性集合 $\boldsymbol{A}$ ($\subseteq \boldsymbol{T}$) を持つタプルに射影することを表し，タプル集合 $\{t_1, \ldots, t_q\}$ に対して $\langle t_1, \ldots, t_q \rangle$ はそれらのタプルを結合したタプルのことを表す．ソースデータ集合 D はテーブル $(T_1, \ldots, T_m)$[1] で構成される．$\mathscr{T}$ は 3.2 節で説明するオペレータによって構成されるタスクを表し，ソースデータ集合 D を受け取るとテーブル O を出力する．つまり $O = \mathscr{T}(D) = \mathscr{T}(T_1, \ldots, T_m)$ である．例外としてタスクは 1 つもオペレータを持たないこともあり，この場合タスクはソーステーブルをそのまま出力する．つまり $\mathscr{T}$ がオペレータを持たないとき $O = T = \mathscr{T}(T)$ が成り立つ．

3.2 オペレータ

本節ではタスクを構成するオペレータを説明する．本稿では分析処理をモデル化するために，基本的な 6 つのオペレータ (Selection σ, Projection π, Join $\bowtie$, Aggregation α, Union $\cup$, Difference $-$) と，複合的データ解析をモデル化する Function オペレータ ϕ の 7 つのセットリレーショナルオペレータを用いる．タスクはこれらのオペレータから構成される木として表され，葉ノードがソーステーブルに対応する．

ここで Function オペレータを定義するために，Reason を以下のように定義する．

定義 1 (Reason)．*Reason* とは複合的データ処理の導出結果の判断根拠を表す情報のことである．

例 2. 例 *1* のような顧客の情報を用いてローン審査を行う学習モデルを考える．このとき学習モデルが審査結果を返すと同時に，例えばある閾値より大きく審査結果に貢献した属性を示すことで複合的データ処理における判断根拠を提示することができる．そのような情報が *Reason* として扱われる．

次に本稿で複合的データ解析をモデル化する UDF を定義する．ここで定義する UDF は Function オペレータの中で呼び出される．

定義 2 (UDF)．*UDF* には実行モードが *2* つあり，実行モードごとに以下に示す入出力を行う．

1：同じテーブルを複数回参照する場合には，それぞれ別のテーブルを参照しているとみなす．

（*1*） *Normal* モード

$$f_n : Domain(\boldsymbol{E}) \rightarrow Value$$

（*2*） *Reasoning* モード

$$f_r : Domain(\boldsymbol{E}) \rightarrow Value \times Reason$$

ここで $Domain(\boldsymbol{E})$ は属性 $\boldsymbol{E}$ のドメインを，$Value$ は複合的データ解析結果のドメインを，$Reason$ は *Reason* 情報のドメインを表す．つまり *UDF* は入力データを受け取ると，*Normal* モードの場合は複合的データ解析の結果のみを返し，*Reasoning* モードの場合は複合的データ解析の結果に加えて *Reason* の情報も併せて返す．なおどのような情報を *Reason* として設計するかは複合的データ解析（*UDF*）を実装する人が自由に決めることができる．

分析結果のみが必要な場合には UDF を Normal モードで実行し，分析結果に加えて判断根拠も必要な場合には UDF を Reasoning モードで実行すればよい．これは UDF の実装次第で Reason の情報を求めるための処理に非常に時間がかかることがあるためである．2 つのモードを用意することで必要なときだけ Reason を求めることを可能にしている．

次に Function オペレータを以下のように定義する．

定義 3 (Function オペレータ)**.** *Function* オペレータ $\phi_{f(\boldsymbol{E})}$ は入力タプルの属性 $\boldsymbol{E}$ に対して *UDF* f を適用し，その結果を結合するオペレータである．*Function* オペレータは *UDF* の実行モードごとに以下の出力を行う．

（*1*） *Normal* モード

$$\phi_{f_n(\boldsymbol{E})}(T) = \{\langle t, f_n(t.\boldsymbol{E}).Value\rangle \mid t \in T\}$$

（*2*） *Reasoning* モード

$$\phi_{f_r(\boldsymbol{E})}(T) = \{\langle t, f_r(t.\boldsymbol{E}).Value, f_r(t.\boldsymbol{E}).Reason\rangle \mid t \in T\}$$

本稿では説明の単純化のためにオペレータをリレーショナルオペレータに限定している．しかしながら本稿のデータモデルは，入力データと出力データの対応関係が本稿のオペレータと同じであれば，より一般的な処理や外部プログラムによる処理もモデル化することが可能である．例えば，入力タプルに対して (1) そのタプルを出力するか (2) 出力しないかのどちらかの処理を行う外部プログラムを考える．このときこの外部プログラムは Selection σ と同じ入出力関係を持つため，本稿のデータモデルの中では Selection としてモデル化することができる．同様にその他の分析処理においても本稿で取り扱うオペレータと同じ入出力関係をもつ限り，本稿のデータモデルでモデル化することが可能である．

例 3. ここまで説明したデータモデルを用いると，例 *1* の分析処理は以下のようにモデル化することができ，図 *2(a)* はその処理木を表している．

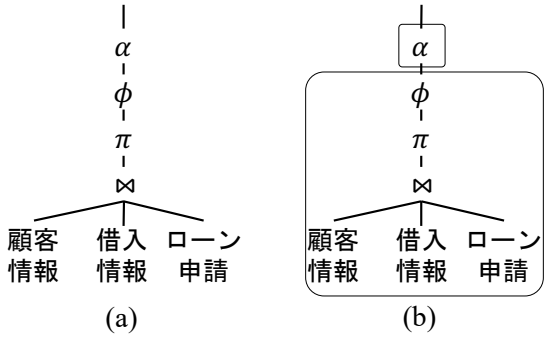

図 2　(a) 処理木．(b) セグメント化された処理木

収入	借入金額	申請金額	判定	Reason
7500	0	550	良	{ 収入, 申請金額 }
3200	1350	1500	不可	{ 借入金額, 申請金額 }
...	...	...	...	...

図 3　例 3 の Function オペレータの出力

$$O = \mathscr{T}(D) = \alpha_{\text{判定},COUNT(*)}(\phi_{f(\boldsymbol{E})}(\pi_{\boldsymbol{E}}(\text{顧客情報} \bowtie \text{借入情報} \bowtie \text{ローン申請})))$$
$$\text{s.t. } \boldsymbol{E} = \{\text{ 収入, 借入金額, 申請金額 }\}$$

ここで $\alpha_{\boldsymbol{G},g(\boldsymbol{B})}(T)$ は，入力テーブル T を属性 $\boldsymbol{G}(\subseteq \boldsymbol{T})$ でグルーピングし，それぞれのグループの属性 $\boldsymbol{B}(\subseteq \boldsymbol{T})$ に対して集約関数 g を適用することを表している．“判定” は *UDF* が出力する *Value* に対応する属性を指しており，$*$ は入力テーブルの全ての属性を指している．この例の *UDF* は *Reasoning* モードで実行すると，*Reason* としてローンの審査結果に一定以上貢献した入力データの属性名の集合を返す．*UDF* が *Reasoning* モードで実行されたときの *Function* オペレータの出力を，図 *3* に示す．

4　拡張来歴

この節で拡張来歴を定義する．拡張来歴は [16] で提案された来歴（Lineage）の拡張である．本稿ではタスク $\mathscr{T}$ の出力タプル $o(\in \mathscr{T}(D))$ の元になった入力タプルの集合を，タスク $\mathscr{T}$ におけるタプル o の ソース来歴 と呼ぶ．ここではまずはじめにオペレータ Op におけるタプル o のソース来歴を定義し，その後タスク $\mathscr{T}$ におけるタプル o のソース来歴を定義する．

定義 4 (オペレータにおけるソース来歴)**.** O をテーブル集合 $\{T_1, \ldots, T_m\}$ に対してオペレータ Op を適用したときの出力テーブルとする（$O = Op(T_1, \cdots, T_m)$）．このとき，オペレータ Op におけるタプル o のソース来歴 ($SL_{Op}(o)$) とは以下の条件を満たす $\hat{T}_i \subseteq T_i$ $(i = 1, \ldots, m)$ の集合のことを指す．

（*1*） $Op(\hat{T}_1, \ldots, \hat{T}_m) = \{o\}$

（*2*） $\forall \hat{T}_i : \forall \hat{t} \in \hat{T}_i : Op(\hat{T}_1, \ldots, \{\hat{t}\}, \ldots, \hat{T}_m) \neq \emptyset$

（*3*） $\hat{T}_i$ は *1* と *2* を満たす最大の T_i の部分集合である．

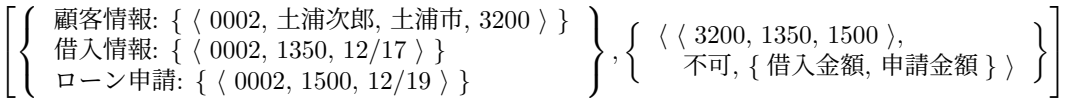

図 4　例 3 の出力タプル ⟨ 不可, 1⟩ の拡張来歴

また，オペレータ Op におけるタプル集合 $\bar{O}(\subseteq O)$ のソース来歴は次の通りである:$SL_{Op}(\bar{O}) = \hat{\bigcup}_{o\in\bar{O}} SL_{Op}(o)$．ただし $\hat{\bigcup}$ はそれぞれのテーブルの和を取ったものを出力するオペレータである．つまり，$\{\hat{T}_1^1,\ldots,\hat{T}_m^1\}\ \hat{\bigcup}\ \{\hat{T}_1^2,\ldots,\hat{T}_m^2\} = \{\hat{T}_1^1 \cup \hat{T}_1^2,\ldots,\hat{T}_m^1 \cup \hat{T}_m^2\}$ $s.t.$ $\hat{T}_i^j \subseteq T_i$ である．以下では $\hat{T}_i \in SL_{Op}(\bar{O})$ を次のように表記する: $P_{T_i}(SL_{Op}(\bar{O}))$.

定義 5 (タスクにおけるソース来歴)．O をテーブル集合 $\{T_1,\ldots,T_m\}$ に対してタスク $\mathscr{T}$ を適用したときの出力テーブルとする（$O = \mathscr{T}(T_1,\cdots,T_m)$）．このとき，タスク $\mathscr{T}$ におけるタプル o のソース来歴 ($SL_{\mathscr{T}}(o)$) とは以下の条件を満たす $\hat{T}_i \subseteq T_i$ の集合のことを指す.

（1） タスク $\mathscr{T}$ がオペレータを持たないとき（$O = T_1$）: $SL_{\mathscr{T}}(o) = o \in T_1$

（2） それ以外のとき（$\mathscr{T}(D) = Op(O_1,\ldots,O_n)$ $s.t.$ $O_i = \mathscr{T}_i(D_i), D_i \subseteq D$）: $SL_{\mathscr{T}}(o) = \bigcup_{1\leqq i\leqq n}(\hat{\bigcup}_{\hat{o_i}\in\hat{O}_i} SL_{\mathscr{T}_i}(\hat{o_i}))$ $s.t.$ $\hat{O}_i = P_{O_i}(SL_{Op}(o))$

また，タスク $\mathscr{T}$ におけるタプル集合 $\bar{O}(\subseteq O)$ のソース来歴は次の通りである: $SL_{\mathscr{T}}(\bar{O}) = \hat{\bigcup}_{o\in\bar{O}} SL_{\mathscr{T}}(o)$．以下では $\hat{T}_i \in SL_{\mathscr{T}}(\bar{O})$ を次のように表記する: $P_{T_i}(SL_{\mathscr{T}}(\bar{O}))$.

拡張来歴を定義する前に，処理の中間結果におけるタプル集合 $\bar{O}(\subseteq \mathscr{T}(D))$ の中間来歴を定義する.

定義 6 (中間来歴)．O をタスク $\mathscr{T}$ の出力テーブルとする（$O = \mathscr{T}(D) = \mathscr{T}(T_1,\cdots,T_m)$）．さらにタスク $\mathscr{T}$ を 2 分割しそれぞれを $\mathscr{T}'$, $\mathscr{T}''$ と表記する．つまりこのとき，$\mathscr{T}(D) = \mathscr{T}'(O', T_{l_{i+1}},\ldots,T_{l_m})$ $s.t.$ $O' = \mathscr{T}''(T_{l_1},\ldots,T_{l_i})$ が成り立つ．中間結果 O' におけるタプル集合 $\bar{O}$（$\subseteq \mathscr{T}(D)$）の中間来歴 $IL(\bar{O}, O')$ とは，タスク $\mathscr{T}'$ におけるタプル集合 $\bar{O}$ のソース来歴 $\hat{O'} \subseteq O'$ のことを指し，$IL(\bar{O}, O') = P_{O'}(SL_{\mathscr{T}'}(\bar{O}))$ である.

定義 7 (拡張来歴)．タスク $\mathscr{T}$ におけるタプル集合 $\bar{O}(\subseteq \mathscr{T}(D))$ の拡張来歴 $AL(\bar{O})$ とは以下に示すペアのことを指す．ただしここでは，O'_i は *Function* オペレータ（$\phi_{f_i(\boldsymbol{E}_i)}$）の出力テーブルのことを指し，それぞれの *Function* オペレータにおいて *UDF* は *Reasoning* モードで実行されているものとする.

- ソース来歴: $SL_{\mathscr{T}}(\bar{O})$
- 根拠来歴: $RL_{\mathscr{T}}(\bar{O}) = \{\langle o.\boldsymbol{E}_i, o.Value, o.Reason\rangle \mid \forall i : o \in IL(\bar{O}, O'_i)\}$

以上のことから，例 3 の分析処理における出力タプル ⟨ 不可 ,1 ⟩ の拡張来歴は，図 4 に示したものとなる.

5　拡張来歴の導出

これまでの来歴を求める手法には，分析処理の実行と同時に全ての分析結果の来歴を求める Eager アプローチと，指定された分析結果の来歴を分析処理実行の後から求める Lazy アプローチの 2 つが存在する [4]．本稿の拡張来歴導出手法は Lazy アプローチに基づいている．近年の分析処理は分析者が様々なパラメータやソースデータを試行錯誤的に試しながら開発される傾向にある．そのような場合，毎回の分析処理の実行で来歴を求めるオーバヘッドがかかる Eager アプローチよりも，来歴が必要になったときにだけ来歴を求めることができる Lazy アプローチの方が適していると考えられるため，本稿では Lazy アプローチを採用した.

本稿で説明する手法は拡張来歴を求めるタプルが指定されると，指定されたタプルが持つ属性値を手がかりに元になった入力データを求める．Projection や Selection，Join によって構成されるような非常に単純な分析処理においては，直接ソーステーブルに対して元になった入力データを求めることができる．しかしより複雑な分析処理の場合には指定された出力タプルの属性が，Aggregation オペレータにおける集約値や Function オペレータにおける Value のようなタスクの途中で導出される属性のみを持ち，ソーステーブルの属性を持たないことがある．このような場合にはタスクをいくつかのサブタスクに分割し，サブタスクごとに来歴を再帰的に求める必要がある．それを踏まえて本稿では，タスクを受け取るとまずはじめに処理木を 1 つ以上の部分木（セグメント）に分割し，その後セグメントごとに来歴を求めることによってタスクの拡張来歴を求める手法を提案する．この手法はこれまでの先行研究 [2], [5] とは異なり，処理木の演算子の順番を変えずにセグメントに分割する方法を提案している．このことは拡張来歴を求めるタスクの処理木が既にクエリオプティマイザ等によって最適化されている場合に有効である.

はじめに 5.1 節でセグメントについてより詳細に説明を行い，5.2 節でタスクをセグメントに分割する方法を述べる．5.3 節でソース来歴を求める Tracing Query を説明し， 5.4 節で拡張来歴の導出手順の概要を説明する．最後に 5.5 節で効率的な拡張来歴の導出について述べる.

5.1　セグメント

セグメントには以下に示す 2 種類が存在する.

- Non-D-segment: Difference 以外のオペレータが “ϕ^*-α^*-$(\cup|\pi|\sigma)^*$-$(\pi|\sigma|\bowtie)^*$” のパターンで並んでいるセグメント．ϕ^* と α^* はそれぞれ連続する Function オペレータと Aggregation オペレータを表しており，$(\cup|\pi|\sigma)^*$ と $(\pi|\sigma|\bowtie)^*$ はそ

れぞれの 3 つのオペレータの組み合わせで構成される処理木を示している．ただしその組み合わせには 3 つの全てのオペレータが存在するとは限らない．最も左のオペレータはその処理木のルートに位置しており，そのためセグメントのオペレータは右から左に向かってボトムアップに処理される．さらに，α^* が複数の連続する Aggregation オペレータ α_1- ... -α_m で構成されるとき，それぞれのグルーピングキー $\boldsymbol{G_i}$ は以下の条件を満たす必要がある: $\boldsymbol{G_1} \subseteq ... \subseteq \boldsymbol{G_m}$.

- D-segment: 単一の Difference オペレータで構成されるセグメント．D-segment は以下のパターンで表される:“−”.

またタスクの処理木のルートに位置するセグメントのことを以降ではルートセグメントと呼ぶ.

5.2 セグメント分割

提案手法はタスク $\mathscr{T}$ が与えられると，5.1 節で説明した各セグメントのパターンで処理木のルートからトップダウンに最長一致を適用することでタスクをセグメントに分割する．証明は紙面の都合上省略するが，この最長一致は 3.2 節で説明した 7 つのオペレータによって構成される任意の処理木を一意に分割することが可能である.

例 4. 例 *3* のタスクは上述したセグメント化の手法を適用すると，*(1)* 単一の α で構成される *Non-D-segment* と *(2)* ϕ-π-$\bowtie$ で構成される *Non-D-segment* の *2* つのセグメントに分割される．分割後の処理木を図 *2(b)* に示す.

5.3 Tracing Query

本節では Tracing Query の定義を行う．Tracing Query は（1）単一の Non-D-segment または D-segment で構成されるタスク $\mathscr{T}$（2）セグメントの入力テーブル $T_1, \ldots, T_m$（3）来歴を求めるタスク $\mathscr{T}$ の出力タプル集合 $\bar{O}$ の 3 つを入力すると，タスク $\mathscr{T}$ におけるタプル集合 $\bar{O}$ のソース来歴 $SL_{\mathscr{T}}(\bar{O})$ を導出するクエリのことである．またこの Tracing Query は関係演算子を使って記述することができる．以下では Non-D-segment に対する Tracing Query と D-segment に対する Tracing Query をそれぞれ説明する.

はじめに Tracing Query で用いる Split オペレータを定義する.

定義 8 (Split オペレータ)．テーブル T の属性集合を $\boldsymbol{T}$ と表す．このとき *Split* オペレータは，それぞれの属性集合 $\boldsymbol{T_i} \subseteq \boldsymbol{T}$ にテーブル T を射影したテーブルの集合を返すオペレータである.

$$Split_{\boldsymbol{T_1},\ldots,\boldsymbol{T_n}}(T) = \{\pi_{\boldsymbol{T_1}}(T), \ldots, \pi_{\boldsymbol{T_n}}(T)\}$$

次に Non-D-segment と D-segment に対する Tracing Query を説明する.

- Non-D-segment に対する Tracing Query: ここでは Non-D-segment に対する Tracing Query を説明するために，以下の形の Non-D-segment に対する Tracing Query を説明する.

$$\mathscr{T}(D) = \phi_{f_1(\boldsymbol{E_1})}(\ldots(\phi_{f_n(\boldsymbol{E_n})}(\alpha_{\boldsymbol{G_1},g_1(B_1)}(\ldots(\alpha_{\boldsymbol{G_k},g_k(B_k)}(\cup_i(\pi_{\boldsymbol{A_i}}(\sigma_{C_i}(T_1^i \bowtie \cdots \bowtie T_{m_i}^i)))))))))$$

任意の Non-D-segment は交換可能なオペレータの順番を入れ替えることによって上記の形に変形することが可能である．上記の Non-D-segment が与えられたとき，タスク $\mathscr{T}$ におけるタプル集合 $\bar{O} \subseteq \mathscr{T}(D)$ のソース来歴は以下の Tracing Query を実行することで求めることができる.

$$TQ_{\bar{O},\mathscr{T}}(D) = \bigcup_i Split_{\boldsymbol{T_1^i},\ldots,\boldsymbol{T_{m_i}^i}}(\sigma_{C_i}(T_1^i \bowtie \cdots \bowtie T_{m_i}^i) \ltimes \bar{O})$$

- D-segment に対する Tracing Query: D-segment $\mathscr{T}(D) = T_1 - T_2$ が与えられたとき，タスク $\mathscr{T}$ におけるタプル集合 $\bar{O} \subseteq \mathscr{T}(D)$ のソース来歴は以下の Tracing Query を実行することで求めることができる.

$$TQ_{\bar{O},\mathscr{T}}(T_1, T_2) = \{\bar{O}, T_2\}$$

5.4 拡張来歴の導出手順

本節では拡張来歴の導出手順の概要を説明する．拡張来歴を求めるためには，はじめにタスクをセグメントに分割した後，セグメントごとに来歴を求めるために Tracing Query をルートセグメントから再帰的に適用する．この Tracing Query を再帰的に実行している間に，来歴を求めるタスクの出力タプル集合に Reason の情報が含まれている場合には，根拠来歴を記録する処理を行う．すると Tracing Query の再帰的な実行が終了した時点でソース来歴と根拠来歴の両方が求められている状態になる.

例 5. ここでは例 *3* に示したタスク $\mathscr{T}$ におけるタプル集合 $\bar{O} = \{\langle$ 不可, 1 $\rangle\}$ の拡張来歴を求めることを例にとって，拡張来歴の導出手順をより詳細に説明する．拡張来歴を求めるために，まずはじめにタスクをセグメントに分割する．するとこのタスクは例 *4* で説明したように，$\mathscr{T}_1$*:* α と $\mathscr{T}_2$*:* ϕ-π-$\bowtie$ の *2* つの *Non-D-segment* に分割することができる.

次に *Tracing Query* をルートセグメントから再帰的に実行する．はじめにルートセグメントである *Non-D-segment* $\mathscr{T}_1$ に対して以下のような *Tracing Query* を実行することで，タスク $\mathscr{T}_1$ におけるタプル集合 $\bar{O}$ のソース来歴を求める.

$$\hat{INT} = INT \ltimes_{\text{判定}} \{\langle \text{不可}, 1 \rangle\}$$

ここで $\ltimes$ はセミジョインを表す記号であり，INT は図 *3* に示した *Function* オペレータの中間結果のことを指している．また $\hat{INT}$ は中間結果 INT におけるタプル集合 $\bar{O}$ の中間来歴 $IL(\bar{O}, INT)$ のことを指しており，この例では $\hat{INT} = \{\langle 3200, 1350, 1500, \text{不可}, \{ \text{借入金額}, \text{申請金額} \}\rangle\}$ となる.

次に $\hat{INT}$ テーブルには *Reason* の情報が含まれているため，根拠来歴 $\{\langle\langle 3200, 1350, 1500\rangle, \text{不可}, \{ \text{借入金額}, \text{申請金額} \}\rangle\}$ を記録する．その後，*Non-D-segment* $\mathscr{T}_2$ に対して以下に示す

Tracing Query を実行することで，タスク $\mathscr{T}_2$ におけるタプル集合 $I\hat{N}T$ のソース来歴を求める．

$$V = (\text{顧客情報} \bowtie \text{借入情報} \bowtie \text{ローン申請}) \ltimes_{\text{収入, 借入金額, 申請金額}} I\hat{N}T$$

$$\hat{\text{顧客情報}} = \pi_{\text{顧客 } ID,\ \text{氏名, 住所, 収入}}(V)$$

$$\hat{\text{借入情報}} = \pi_{\text{顧客 } ID,\ \text{借入金額, 情報更新日}}(V)$$

$$\hat{\text{ローン申請}} = \pi_{\text{顧客 } ID,\ \text{申請金額, 申請日}}(V)$$

上記の *Tracing Query* を実行することで，$\hat{\text{顧客情報}} = \{\langle 0002, \text{土浦次郎}, \text{土浦市}, 3200\rangle\}$，$\hat{\text{借入情報}} = \{\langle 0002, 1350, 12/17\rangle\}$，$\hat{\text{ローン申請}} = \{\langle 0002, 1500, 12/19\rangle\}$ を得ることができる．

これでタスクを構成する全てのセグメントに対して *Tracing Query* の実行が完了したため，拡張来歴を導出する手順は完了となる．実際，上記の手順によって図 *4* に示したソース来歴と根拠来歴のペアが導出できている．

この拡張来歴の導出手順は，3.2 節で定義したオペレータによって構成される任意のタスクにおいて同様に適用することが可能である．

本稿で説明した導出手順は複数セグメントから構成されるタスクに対して拡張来歴を求める際，Tracing Query の実行のためにセグメントの入力テーブルにあたる分析処理の中間結果が必要になり，さらに Reason 情報が含まれている Function オペレータの中間結果も併せて必要である．この中間結果をどのように用意するかを次節で説明する．

5.5 中間結果の作成戦略

拡張来歴導出に必要な分析処理の中間結果を用意する戦略として，以下の 3 つの戦略を考えることができる．

- Rerun: 中間結果が必要になった時点で分析処理を再実行して中間結果を作成するやり方．再実行時には，根拠来歴の導出に必要な Reason の情報のために UDF は必ず Reasoning モードで実行する必要がある．事前に中間結果をストアしないためストレージコストは小さいが，再実行の分，拡張来歴の導出に時間がかかる．
- Full Materialization（Full）: 最初に分析処理を実行するときに Function オペレータの UDF を Reasoning モードで実行し，拡張来歴導出に必要な全ての中間結果をあらかじめストアしておくやり方．拡張来歴の導出時に分析処理の再実行は不要であるため高速な導出が可能であるが，中間結果をストアするためストレージコストが大きい．
- Function Materialization（FM）: 最初に分析処理を実行するときに Function オペレータの UDF を Reasoning モードで実行し，Function オペレータの中間結果のみをあらかじめストアしておき，その他の中間結果は必要になった時点で再計算することで用意するやり方．Rerun と Full のトレードオフを図ることができると考えられる．

Function Materialization は，多くの場合 Function オペレータの実行コストが他のリレーショナルオペレータの実行コストより大きいことに着目し，再計算すると非常に時間がかかる中間結果だけをストアすることで再実行のコストを下げつつ，全ての中間結果を保持しないことでストレージコストも抑える戦略である．実験でこの 3 つの戦略の性能評価を行う．

表 1　ソーステーブルのタプル数

	Image	*Event*
Small	4.5×10^4	6.075×10^5
Medium	4.5×10^5	6.075×10^6
Large	4.5×10^6	6.075×10^7

6 実　　験

本節では PostgreSQL [17] 上に実装した拡張来歴導出システムを用いて，顔画像のデータセットである LFW データセット [18] を用いたタスクに対する評価実験の結果を述べる．実験ではまずはじめに 3 つの戦略における分析処理の実行時間を評価する．その目的は Full Materialization と Function Materialization の，中間結果を作成しながら分析処理を実行することで発生する分析処理実行へのオーバヘッドを測定することである．その後，3 つの戦略に基づいて拡張来歴を導出する際の実行時間とストレージコストを評価すると同時に，Function オペレータの処理コストと Function Materialization の有効性の関係についても併せて述べる．

本実験の実装には PostgreSQL 9.6 と Python 3.7.8 を用い，実験は Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz，GeForce GTX 1060 3GB，および 32GB のメモリを搭載したマシンで実行した．また全ての実験結果は 3 回測定した平均の値を示している．

6.1 実験で実行するタスク

本実験では LFW データセット [18] を用いて作成したタスクで評価を行った．ソースデータ集合は $Image(i\_imageid, i\_placeid, i\_img)$，$Event(e\_eventid, e\_placeid, e\_visitors)$ の 2 つのテーブルからなり，表 1 に示す 3 種類のサイズのソースデータ集合を用意した．*Image* テーブルの属性 *i_img* には LFW データセットに含まれる 1 枚の画像が URI の形式で保存されており，画像の URI にはその画像に写っている著名人の名前が含まれている．*Event* テーブルの属性 *e_visitors* はそのイベントの参加者の人数が保存されており，その他の属性には画像，イベントもしくはイベント会場の識別子が整数で記録されている．また *i_img* 以外の属性値は人工的に作成したものを使用している．

次に実験に使用した SQL（タスク）を図 5 に示す．この SQL は，著名人が大規模なイベント会場で何回ステージに登壇したかを調べており，そのために人物認識を行う UDF（recognition 関数）を使用している．本実験では Function オペレータの処理コストと Function Materialization の有効性の関係を調べるために，人物認識を行う UDF に 2 つの処理コストの異なる実装を用意した．次節でそれぞれの UDF の詳細を説明する．

```
SELECT i_value, COUNT(*)
FROM (
 SELECT i_imageid, i_placeid, avg, i_img,
  (recognition(i_img)).value i_value
 FROM Image, (
  SELECT e_placeid, avg(e_visitors) avg
  FROM Event
  GROUP BY e_placeid
 ) Seg2
 WHERE i_placeid = e_placeid
 AND avg >= 50000
) Seg1
GROUP BY i_value;
```

図 5　実験で使用した SQL

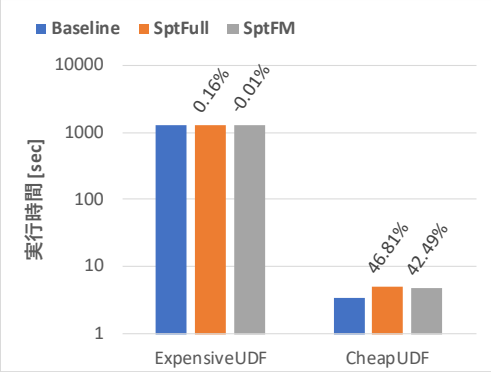

図 6　タスクの処理時間

6.2　UDF

使用する人物認識を行う UDF は，画像のパス（URI）を入力として受け取ると画像に写っている人物の名前を返す関数であり，本実験では以下に示すように処理コストの異なる 2 つの実装を用意した．

- 顔認識（Expensive UDF）

この実装で UDF は，画像の URI が与えられると学習モデルを用いて画像に誰が写っているかを判断し，その人物名を UDF の Value として出力する．さらに Reason として元の画像のどの部分にその人物が写っていたのかを示すバウンディングボックスを出力する．この実装は学習モデルによる処理を実行するため，もう一方の実装と比較して UDF の処理コストが大きい．

- 文字列処理（Cheap UDF）

この実装で UDF は，画像の URI が与えられると URI の文字列から著名人の名前を抽出し，その名前を UDF の Value として出力する．さらに Reason として URI の何文字目からその名前を抽出したかを示す領域を出力する．こちらの実装では単に文字列処理を行うだけであるため，顔認識の実装と比較すると処理コストが小さい．

また上述した 2 つの UDF の実装において，UDF が出力する Value 属性の値（人物名）はどちらも同じである．

6.3　タスク処理時間の評価

この節では 3 つの戦略（Rerun，Full Materialization（Full），Function Materialization（FM））に基づき必要に応じて中間結果を作成しながらタスクを実行したときの処理時間を比較する．Rerun に基づいてタスクを実行するときは中間結果を作る必要がないためそのままタスクを実行すれば良い．しかし一方で，Full や FM に基づいてタスクを実装するときは中間結果を作りながら実行する必要があり，さらに Full は FM より多くの中間結果を作る必要がある．

この実験には Medium サイズのソーステーブルを使用した．図 6 に実験結果を示す．図の左側が顔認識（ExpensiveUDF）の実行を伴うタスクの実行時間を表しており，右側が文字列処理（CheapUDF）の実行を伴うタスクの実行時間を表している．Baseline が中間結果を作成せずにタスクをそのまま実行するときの実行時間を表しており，これは Rerun に基づいて分析処理を実行するときに対応している．また SptFull と SptFM はそれぞれ Full と FM に基づいてタスクの中間結果を作成しながらタスクを実行したときの実行時間を表している．SptFull と SptFM の上部の数値は，Baseline と比較してどれほどタスクの実行が遅くなったのかを表している．

実験結果としては SptFull と SptFM は Baseline と比較してわずかではあるがオーバヘッドがかかる傾向があり，特に SptFull は SptFM と比較してより大きなオーバヘッドがかかる傾向があることが示された．この理由は SptFull の方が SptFM と比較してより多くの中間結果を作りながらタスクを実行する必要があるためである．またタスクの実行にかかるオーバヘッドの割合が Cheap UDF の方が大きくなっている理由は，UDF の処理コストが重い場合であっても軽い場合であっても中間結果をストアするコストは変わらないためである．

表 2　ストアする中間結果のタプル数

	Full	FM
Small	4.95×10^4	4.5×10^3
Medium	4.95×10^5	4.5×10^4
Large	4.95×10^6	4.5×10^5

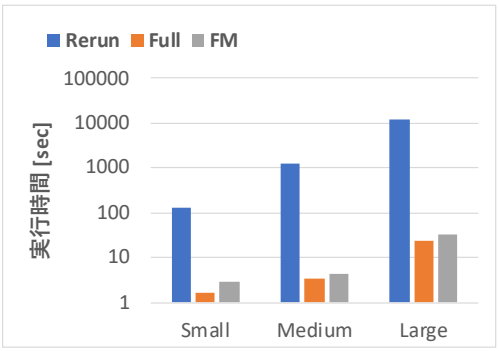

図 7　Expensive UDF の実行を伴うタスクの拡張来歴の導出時間

6.4　拡張来歴導出にかかるコスト

この節ではタスクの分析結果のうち 1 タプルに対して拡張来歴を求めるときのコストを評価する．

顔認識（Expensive UDF）の実行を伴う場合: 処理時間を図 7 に示し，ストアする中間結果のタプル数を表 2 に示す．Full Materialization（Full）と Function Materializaiton（FM）のどちらも Rerun より高速に拡張来歴を求めることができ，より詳細には，ソーステーブルが Large サイズのとき Full は 538.8 倍 Rerun より高速に，FM は 378.9 倍 Rerun よ

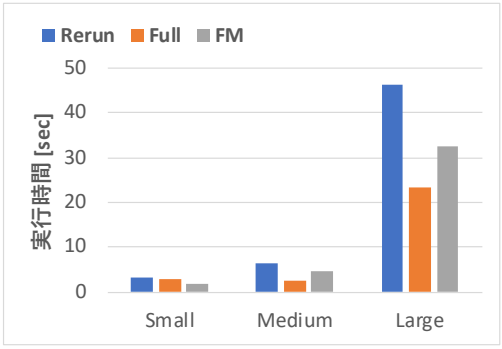

図 8　Cheap UDF の実行を伴うタスクの拡張来歴の導出時間

り高速に拡張来歴を求めることができた．またストレージコストの点において FM は Full と比較して 91%ストアするタプルの数を減らすことができるという結果が得られた．

文字列処理 (Cheap UDF) の実行を伴う場合: 処理時間を図 8 に示す．この場合においても Full と FM は処理時間の点で Rerun よりよい性能を示しており，Full はソーステーブルが Medium サイズのとき 2.5 倍，FM はソーステーブルが Small サイズのとき 1.6 倍高速に拡張来歴を求めるられることが示された．またストアする中間結果のタプル数は Expensive UDF の場合と同様であるため，Cheap UDF の場合も FM は Full と比較して少ないストレージコストで拡張来歴を求めることができている．しかしながら Cheap UDF の場合，中間結果をストアしておくことによる処理時間の高速化の割合は Expensive UDF と比較して小さくなっているが，これは UDF の処理コストが軽い場合，その UDF を実行する Function オペレータの処理コストが他のリレーショナルオペレータのコストとほとんど変わらなくなるためである．

以上の実験結果から FM は，拡張来歴を求める実行時間と中間結果のストレージコストのトレードオフを図る適切な戦略であることが示された．

7　おわりに

本稿では AI/ML 処理を伴う複合的データ処理に対するトレーサビリティを保証する拡張来歴を求める際に，処理木の演算子の順番を維持する導出方法を提案した．さらに実験では LFW データセットを用いたタスクを対象として，分析処理の実行にかかるオーバヘッドや拡張来歴を導出するためのコストについての評価を行なった．今後の予定としては今回提案した枠組みを分散処理やストリーム処理をはじめとする，より一般的なワークフローに対して適用することが挙げられる．

謝　辞

本研究の一部は，JSPS 科研費 JP19H04114，NEDO 人と共に進化する次世代人工知能に関する技術開発事業，並びに AMED ムーンショット型研究開発事業による．

文　献

[1] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. *Provenance in Databases: Why, How, and Where*. 2009.

[2] Masaya Yamada, Hiroyuki Kitagawa, Toshiyuki Amagasa, and Akiyoshi Matono. Augmented Lineage: Traceability of Data Analysis Including Complex UDFs. In *Database and Expert Systems Applications*, pages 65–77. Springer International Publishing, 2021.

[3] Rajendra Bose and James Frew. Lineage Retrieval for Scientific Data Processing: A Survey. *ACM Comput. Surv.*, 37(1):1–28, mar 2005.

[4] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, 26(6):881–906, 2017.

[5] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, June 2000.

[6] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, page 31–40, New York, NY, USA, 2007. Association for Computing Machinery.

[7] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. *The VLDB Journal*, 14(4):373–396, 2005.

[8] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated xml: Queries and provenance. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, page 271–280, New York, NY, USA, 2008. Association for Computing Machinery.

[9] Yannis Theoharis, Irini Fundulaki, Grigoris Karvounarakis, and Vassilis Christophides. On provenance of queries on semantic web data. *IEEE Internet Computing*, 15(1):31–39, 2011.

[10] Jeremy Goecks, Anton Nekrutenko, and James Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):1–13, 2010.

[11] Matteo et al. Interlandi. Titian: Data provenance support in spark. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, volume 9, page 216. NIH Public Access, 2015.

[12] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. Genealog: Fine-grained data streaming provenance in cyber-physical systems. *Parallel Computing*, 89:102552, 2019.

[13] N. Zheng, A. Alawini, and Z. G. Ives. Fine-grained provenance for matching & etl. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 184–195, 2019.

[14] Mengnan Du, Ninghao Liu, and Xia Hu. Techniques for interpretable machine learning. *Commun. ACM*, 63(1):68–77, December 2019.

[15] Selvaraju, Ramprasaath R. et al. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

[16] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *Proceedings of 16th International Conference on Data Engineering*, pages 367–378, 2000.

[17] PostgreSQL. `https://www.postgresql.org/`.

[18] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.