ML Based Lineage in Databases

Michael Leybovich
ML Based Lineage in Databases

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Michael Leybovich

Submitted to the Senate of the Technion — Israel Institute of Technology
Tamuz 5781 Haifa June 2021
This research was carried out under the supervision of Prof. Oded Shmueli, in the Faculty of Computer Science.

Some results in this thesis have been published in an article by the author (Michael Leybovich) and the advisor (Oded Shmueli) during the course of this master research, the most up-to-date versions of which is:


Acknowledgements

I would like to thank my advisor, Prof. Oded Shmueli, for his continuous guidance, support and availability at any time of day. It was a pleasure and an honor working with you.

I wish to thank my partner, Sharon, for being an attentive listener and providing me with emotional support throughout the whole process.

Finally, I would like to thank my mother who stressed about every step, and yet, never stopped believing in me.

The generous financial help of the Technion is gratefully acknowledged.
# Contents

List of Figures

Abstract 1

1 Introduction 3

2 Classic Provenance 5

2.1 What is Provenance? ........................................... 5
    2.1.1 Definition ........................................... 5
    2.1.2 Related Work ........................................ 5

2.2 Provenance Semirings .................................... 5
    2.2.1 Overview ........................................... 5
    2.2.2 Important cases .................................... 7

2.3 Provenance for Data-Dependent Processes ............... 7

2.4 ProvSQL - A Real World Provenance Application ....... 8
    2.4.1 Semirings with monus ................................ 8
    2.4.2 Provenance Circuits ................................ 8
    2.4.3 PostgreSQL Hooks .................................... 9
    2.4.4 Implementation ..................................... 9

3 Lineage via Embeddings 11

3.1 Word Embeddings .......................................... 11

3.2 Motivation .................................................. 11

3.3 Word Vectors Model ....................................... 12
    3.3.1 Training word vectors ............................... 12
    3.3.2 Sentence embeddings ................................. 14
    3.3.3 Similarity calculation ............................... 14

3.4 Related Work ................................................ 15

4 Per-Tuple Lineage Vectors 17

4.1 Proposed Solution ......................................... 17
    4.1.1 Lineage querying ...................................... 19
    4.1.2 Intended usage ...................................... 20
4.1.3 Comparison to exact lineage ........................................ 20
4.2 Verifying The Lineage .................................................. 21
4.3 Interim Findings and Hurdles ......................................... 22

5 Per-Column Lineage Vectors ............................................. 23
  5.1 Rationale ........................................................................ 23
  5.2 Implementation ............................................................ 23
  5.3 Discussion ...................................................................... 27

6 Improving Lineage Embeddings .......................................... 29
  6.1 Query-Dependent Column Weighting ................................ 29
    6.1.1 Rationale .................................................................. 29
    6.1.2 Implementation ........................................................ 29
    6.1.3 Note ........................................................................ 30
  6.2 Bloom-Filters of Queries ................................................ 30
    6.2.1 Rationale .................................................................. 30
    6.2.2 Implementation ........................................................ 30
    6.2.3 Notes ........................................................................ 30
  6.3 Tuple Creation Timestamp ............................................... 31
    6.3.1 Rationale .................................................................. 31
    6.3.2 Implementation ........................................................ 32
  6.4 Weighting with Query Dependency DAG .......................... 32
    6.4.1 Rationale .................................................................. 32
    6.4.2 Implementation ........................................................ 32
    6.4.3 Notes ........................................................................ 33
  6.5 Limit Provenance Analysis to Specific Base Tables .......... 34
    6.5.1 Rationale .................................................................. 34
    6.5.2 Implementation ........................................................ 34
    6.5.3 Notes ........................................................................ 35
  6.6 Where Provenance ......................................................... 35

7 Experimental Evaluation .................................................... 37
  7.1 Experimental Setup ........................................................ 37
    7.1.1 Datasets ................................................................... 37
    7.1.2 Precision Calculation ................................................ 38
    7.1.3 Distant Lineage: Per-Level Recall ............................ 39
  7.2 Examples ......................................................................... 40
  7.3 Advanced Experiments - The BFPDB Dataset .................. 43
    7.3.1 A Hierarchy of Materialized Views ............................ 44
    7.3.2 Experiments ............................................................. 45
8 Conclusions and Future Directions

8.1 Word Embedding vs. The Field ............................................ 65
8.2 Long-term scenarios and analysis ...................................... 66
8.3 Partial Explanations .......................................................... 66

A Efficient Set of Vectors Search ........................................... 69
A.1 Rationale ........................................................................... 69
A.2 A Constant Cardinality of Sets of Vectors ......................... 69
  A.2.1 Implementation ............................................................ 70
A.3 Varying Cardinalities of Sets of Vectors ......................... 72
  A.3.1 Implementation ............................................................ 73

B BFPDB Dataset ..................................................................... 77
B.1 products ........................................................................... 77
B.2 nutrients .......................................................................... 77
B.3 serving_size ...................................................................... 78
B.4 derivation_code_description ........................................... 78

C MovieLens Dataset ............................................................. 79
C.1 movies .............................................................................. 79
C.2 ratings .............................................................................. 80
C.3 tags .................................................................................. 80

D BFPDB Materialized Views .................................................. 81
D.1 prepared ........................................................................... 81
D.2 unprepared ........................................................................ 81
D.3 readytodrink ..................................................................... 81
D.4 protein ............................................................................. 82
D.5 sugars .............................................................................. 82
D.6 cholesterol ......................................................................... 82
D.7 exp₁ .................................................................................. 82
D.8 exp₂ .................................................................................. 83
D.9 exp₃ .................................................................................. 83
D.10 exp₄ ................................................................................ 83

Hebrew Abstract ..................................................................... i
List of Figures

2.1 ProvSQL system architecture ........................................ 10
4.1 Lineage vectors: Initialization Algorithm ........................ 19
4.2 Lineage vectors: + (addition) Algorithm ........................ 19
4.3 Lineage vectors: · (multiplication) Algorithm .................... 20
4.4 Incorporating lineage vectors in the ProvSQL Algorithm .......... 21
5.1 Column Lineage vectors: + (addition) Algorithm ............... 24
5.2 Column Lineage vectors: · (multiplication) Algorithm .......... 25
7.1 Materialized Views Hierarchy - The BFPDB Dataset ............... 44
Abstract

There has been extensive research on data provenance, mostly concerned with annotating the results of database (DB) queries with provenance information which is useful in explaining query results at various resolution levels. In this work, we track the lineage of tuples throughout their database lifetime. That is, we consider a scenario in which tuples (records) that are produced by a query may affect other tuple insertions into the DB, as part of a normal workflow. As time goes on, provenance explanations for such tuples become deeply nested, increasingly consuming space, and resulting in decreased clarity and readability.

We present a novel approach for approximating lineage tracking. We use Machine Learning (ML) and Natural Language Processing (NLP) techniques; mainly, word embedding. The basic idea is summarizing (and approximating) the lineage of each tuple via a small set of constant-size vectors (the number of vectors per-tuple is a hyperparameter). For explicitly (and independently of DB contents) inserted tuples - the vectors are obtained via a pre-trained word vectors model over their underlying database domain “text”. During the execution of a query, we construct the lineage vectors of the final (and intermediate) result tuples in a similar fashion to that of semiring-based exact provenance calculations. We extend the + and · operations to generate sets of lineage vectors, while emphasizing the ability to propagate information and preserve the compact representation. Therefore, our solution does not suffer from space complexity blow-up over time, and it “naturally ranks” explanations to the existence of a tuple in the DB.

We devise a genetics-inspired improvement to our basic method. The data columns of an entity (and potentially other columns) are a tuple’s basic properties, i.e., the “genes” that combine to form its genetic code. Thus, finding the lineage of a tuple in the DB is analogous to finding its predecessors via DNA examination. We design an alternative lineage tracking mechanism, that of keeping track of and querying lineage (via embeddings) at the column (“gene”) level; thereby, we manage to better distinguish between the provenance features and the textual characteristics of a tuple.

We further introduce several improvements and extensions to the basic method: column emphasis, Bloom Filters of queries, tuple creation timestamp, query dependency DAG and where provenance.

We integrate our lineage computations into the PostgreSQL system via an extension
(ProvSQL) and experimentally exhibit useful results in terms of accuracy against exact, semiring-based, justifications. In the experiments, we focus on tuples with multiple generations of tuples in their lifelong lineage and analyze them in terms of direct and distant lineage. The experiments suggest a high usefulness potential for the proposed approximate lineage methods and the further suggested enhancements. This especially holds for the column-based vectors method which exhibits high precision and per-level recall.

Finally, we outline a search algorithm for vector sets, which is based on encoding a set of vectors via a “long” single vector.
Chapter 1

Introduction

Data Lineage

The focus of this work is providing explanations (or justifications) for the existence of tuples in a Database Management System (DBMS, DB). These explanations are also known as data provenance \cite{CCT09}. Provenance in the literature \cite{IGK08,DGM15} often refers to forms of “justifying” query results. That is, the provenance context is the state of the database (DB) just before the query execution. The specific type of provenance on which we focus is lineage \cite{CWW00}, namely a collection of DB tuples whose existence led to the existence of tuple $t$ in a query result.

Distant Lineage

We take a more comprehensive look. We track lineage of tuples throughout their existence, while distinguishing between tuples that are inserted explicitly and independently of DB content (these are our “building blocks”) and tuples that are inserted via a query (or, more generally, that depend on the contents of the DB). In a real-life setting - tuples that are inserted via a query can be one of the following:

- A hierarchy of materialized views - where each view can depend both on the DB and on previously defined views.
- Tuples that are inserted via a SQL `INSERT INTO SELECT` statement.
- Tuples that are inserted via a SQL `UPDATE` statement.
- A query result with explicitly added data fields, that is added back to some table in the DB. For example, get names of customers retrieved from an `orders` table, calculate some non-database-resident “customer profile” for each customer and insert both into a `customer_profile` table for providing future recommendations.
Approximate Lineage

As time goes on, provenance information for tuples that are inserted via a query may become complex (e.g., by tracking semiring formulas, as presented in [GKT07], or circuits as presented in [DMRT14, Sen17]). Thus, the goal of this work is providing “simple to work with” and useful approximate lineage (using ML and NLP techniques), while requiring only a constant additional space per tuple. This approximate lineage is compared against state of the art “exact provenance tracking system” in terms of explainability and maintainability.

Main Contributions

The main contributions of this work are as follows:

1. The space-efficient, with constant additional space per tuple, usage of word vectors to encode lifelong lineage.

2. A family of algorithms and enhancements that render the approach practical for both direct (i.e., current DB state) and indirect (i.e., all history) lineage computations.

3. Experimentation which exhibits high usefulness potential.

Organization

The rest of this thesis is organized as follows. Previous work on provenance is surveyed in Chapter 2. In Chapter 3 we discuss relational embeddings and the motivation to use them for lineage encoding. In Chapter 4 we introduce basic algorithms for creating per-tuple lineage encoding vectors. In Chapter 5 we adapt these basic algorithms to per-column lineage encoding vectors. Chapter 6 discusses improvements to the previously presented algorithms. In Chapter 7 we present experimental results. In Chapter 8 we provide further discussion on a few topics presented in this thesis. We conclude in Chapter 9, in which we also outline directions for future research.
Chapter 2

Classic Provenance

2.1 What is Provenance?

2.1.1 Definition

Provenance - source, origin [pro20]. In computing, provenance information describes the origins and the history of data within its lifetime. When talking about database management systems, the commonly used term is data provenance [CCT+09]. The idea behind data provenance is keeping additional information (meta-data) allowing us to easily answer a large number of “meta-questions”.

Data provenance helps with providing explanations for the existence of tuples in a query result. The context of these explanations is usually the DB state prior to the query execution.

2.1.2 Related Work

Over the past 15 years, provenance research has advanced in addressing both theoretical [CCT+09, GKT07, DMRT14] and practical [IGK+08, KIT10, DGM15, DFG17, SJE18, IZHZ19] aspects. In particular, several different notions of data provenance (lineage, why, how and where) were formally defined [CWW00, BKT01, CCT+09].

A few prior works [RS08, ABD+15, LNLG17, LLG18] focus on approximate (or summarized) provenance. That is, seeking a compact representation of the provenance at the possible cost of information loss in an attempt to deal with the growing size and complexity of exact provenance information in real-life systems.

2.2 Provenance Semirings

2.2.1 Overview

Provenance semirings have been introduced by Green et al. [GKT07] as a formalism for data provenance. These semirings have been shown [KG12] to generalize previous

\footnote{Portions of this section were adapted from Green et al. [GKT07] and Karvounarakis et al. [KG12].}
works such as lineage [CWW00] and why-provenance [BKT01]. The main idea is based on annotating (tagging) every tuple in the DB with elements from some algebraic structure \((K, +, -, 0, 1)\)\(^2\) resulting in tagged-tuple relations. A notationally convenient way of working with tagged-tuple relations is to model tagging by a function on all possible tuples, with those tuples not considered to be “in” the relation tagged with a special value (usually, it is the 0 element from the set \(K\)).

Let us denote the set of all tuples over a finite set of attributes \(U\) by \(U\)-Tup. A \(K\)-relation over a finite set of attributes \(U\) is a function \(R : U\)-Tup \(\rightarrow\) \(K\) such that its support\(^3\) is finite. The following definitions view an \(\mathcal{RA}^+\) (relational algebra, excluding the difference operator) expression \(E\) as a \(K\)-relation, namely a function. The definition is recursive on the structure of \(\mathcal{RA}^+\) expression \(E\):

- **empty relation.** For any set of attributes \(U\), there is a function \(\emptyset : U\)-Tup \(\rightarrow\) \(K\) such that \(\emptyset(t) = 0\), \(\forall t \in U\)-Tup.
- **union.** If \(R_1, R_2 : U\)-Tup \(\rightarrow\) \(K\) then \(R_1 \cup R_2 : U\)-Tup \(\rightarrow\) \(K\) is defined by \((R_1 \cup R_2)(t) \triangleq R_1(t) + R_2(t), \forall t \in U\)-Tup.
- **projection.** If \(R : U\)-Tup \(\rightarrow\) \(K\) and \(V \subseteq U\) then \(\pi_V R : V\)-Tup \(\rightarrow\) \(K\) is defined by \((\pi_V R)(t) \triangleq \sum_{t = t' \text{ on } V \text{ and } R(t') \neq 0} R(t'), \forall t \in U\)-Tup.

(here \(t = t' \text{ on } V\) means \(t' \in U\)-Tup whose restriction to \(V\) is the same as \(t\); note also that the sum is finite since \(R\) has finite support).
- **selection.** If \(R : U\)-Tup \(\rightarrow\) \(K\) and the selection predicate \(P\) maps each tuple in \(U\)-Tup to either 0 or 1, then \(\sigma_P R : U\)-Tup \(\rightarrow\) \(K\) is defined by \((\sigma_P R)(t) \triangleq R(t) \cdot P(t), \forall t \in U\)-Tup.
- **natural join\(^4\).** If \(R_i : U_i\)-Tup \(\rightarrow\) \(K\), \(i = 1, 2\) then \(R_1 \bowtie R_2\) is the \(K\)-relation over \(U_1 \cup U_2\) defined by \((R_1 \bowtie R_2)(t) \triangleq R_1(t_1) \cdot R_2(t_2), \text{ where } t_1 = t \text{ on } U_1 \text{ and } t_2 = t \text{ on } U_2, \forall t \in U_1 \cup U_2\)-Tup.
- **renaming.** If \(R : U\)-Tup \(\rightarrow\) \(K\) and \(\beta : U \rightarrow U'\) is a bijection then \(\rho_\beta R\) is a \(K\)-relation over \(U'\) defined by \((\rho_\beta R)(t) \triangleq R(t \circ \beta), \forall t \in U'\)-Tup.

As defined above, the operations of \(\mathcal{RA}^+\) preserve the finiteness of supports, therefore they map \(K\)-relations to \(K\)-relations. Hence, the preceding definitions (of making \(\mathcal{RA}^+\) to operate on \(K\)-relations) define an algebra on \(K\)-relations. Green et al. [GKT07] showed that the following \(\mathcal{RA}\) properties:

- join and union are both associative and commutative;

\(^2K\) is a set, containing two distinguished elements 0, 1; and +, - are binary operators on elements from \(K\).

\(^3\text{support}(R) = \{t \mid R(t) \neq 0\}\).

\(^4\text{By renaming attributes that appear in both arguments we can use join to compute cartesian product (\(\times\)), the basic }\mathcal{RA}\text{ operator.}
• union has identity $\emptyset$;
• join is distributive over union;
• $\sigma_{\text{false}}(R) = \emptyset$ and $\sigma_{\text{true}}(R) = R$ ($\text{false}$ and $\text{true}$ are the uniformly false and true predicates, respectively).

hold for the positive algebra on $K$-relations iff $(K, +, \cdot, 0, 1)$ is a commutative semiring. I.e., $(K, +, 0)$ and $(K, \cdot, 1)$ are commutative monoids\(^5\), $\cdot$ is distributive over $+$ and $\forall a, 0 \cdot a = a \cdot 0 = 0$.

### 2.2.2 Important cases

Several important semirings that are discussed in the literature [GKT07, KG12, Sen17] are:

• $(\mathbb{B}, \lor, \land, false, true)$ - binary trust (set semantics, $\mathbb{B} = \{true, false\}$);
• $(\mathbb{N}, +, \cdot, 0, 1)$ - multiplicity (bag semantics, $\mathbb{N}$ is the set of natural numbers);
• $(\mathbb{A}, \text{min}, \text{max}, 0, P)$ - security semiring (access control), where the total order $\mathbb{A} = P < C < S < T < 0$ describes levels of security clearance: $P$ public, $C$ confidential, $S$ secret, and $T$ top-secret;
• $([0, 1], \text{max}, \cdot, 0, 1)$ - Viterbi semiring (confidence scores, probability);
• $(\mathbb{N}[X], 0, 1, +, \cdot)$ - used for a general form of provenance, the provenance polynomials (the universal semiring). $\mathbb{N}[X]$ is the set of multivariate polynomials with coefficients from $\mathbb{N}$ and variables from a set $X$.

### 2.3 Provenance for Data-Dependent Processes

A Data-Dependent Process (DDP) is a logical flow that is specified by a finite state machine (FSM) whose transitions are either governed by boolean queries over an underlying database, or otherwise correspond to external effects such as user choices, interacting applications, etc.

Deutch et al. [DMT14] suggested approaching the problem of analyzing DDPs using data provenance practices. In particular, in their study, they use provenance semirings [GKT07] to obtain a mathematical expression that describes an execution path over the FSM (such a path models a possible execution of a DDP). The transitions on such a path depend on queries over a database or on external user actions. Deutch et al. introduced a new method for generalizing provenance semirings to the context of DDP analysis. This is achieved by using two separate (and potentially, different) semirings -

\(^5\)An algebraic structure that is closed under an associative binary operation and has an identity element.
one for user actions and the other for DB tuples, and combining them by a novel tensor product construction, as opposed to completely separating the two kinds of provenance semirings by e.g., a simple cartesian product of the two semirings. The proposed method allows to construct a generalized provenance expression (using polynomials) for a possible execution of a DDP and to specialize it using homomorphisms [GKT07].

In terms of methodology, keeping track of provenance of a DDP along an execution path (as long as the path is finite) is intuitively similar to tracking provenance of a query result, where each edge on the path is a piece of data (with its own provenance, e.g., an intermediate result), and the whole path corresponds to a joint use of the pieces of data.

2.4 ProvSQL - A Real World Provenance Application

ProvSQL is an open-source project developed by Pierre Senellart et al. [SJE18]. According to the official GitHub page [Sen]: “The goal of the ProvSQL project is to add support for (m-)semiring provenance and uncertainty management to PostgreSQL databases, in the form of a PostgreSQL extension/module/plugin. It is work in progress at the moment.”

Next, we present several concepts that are incorporated in ProvSQL and briefly discuss its implementation.

2.4.1 Semirings with monus

We previously stated in section 2.2 that provenance semirings cover only $\mathcal{RA}^+$ queries. However, [GP10] identified a large class of semirings that (subject to certain restrictions) can be equipped with a monus operator $\setminus$. Thus, it is possible to generalize provenance capturing to $\mathcal{RA}$ with a difference ($\setminus$) operation (adding support for non-monotone\(^6\) queries). This class of semirings is called $m$-semirings. Furthermore, [GP10] show a universal m-semiring, i.e., it is possible to obtain a provenance evaluation in any other m-semiring by applying an appropriate semiring homomorphism.

2.4.2 Provenance Circuits

As shown previously by Green et al. [GKT07] and Karvounarakis et al. [KG12], provenance information can be expressed via semiring formulas (polynomials). These formulas may blow-up in terms of space consumption, and, thus, they are problematic for practical use. An alternative (more compact) representation for provenance annotations is circuits [DMRT14, Sen17], which are constructed per-query. A provenance circuit is an inductively built directed acyclic graph (DAG), with the following properties:

\(^6\)Non-monotone means: $I \subseteq I' \rightarrow Q(I) \subseteq Q(I')$. 
• The leaves contain annotations of tuples from the input DB.

• Inner nodes represent operators from a particular semiring (termed gates by Senellart et al.).

• The edges (termed wires by Senellart et al.) connect nodes to an operator, representing operands of an intermediate calculation.

• The sub-DAG under a given node represents the semiring formula for deriving it.

2.4.3 PostgreSQL Hooks

PostgreSQL (Postgres) hooks [Lel12] make it possible to extend/modify its behaviour without rebuilding Postgres, by interrupting the execution process at certain points. Similarly to Postgres itself, the hooks API is written in C. Every hook is accessible via a global function pointer, initially set to NULL. During an extension’s loading (following a CREATE EXTENSION command) Postgres calls the extension’s own _PG_init function (if implemented), which has access to the hooks handler pointers (at this point, a hook function can be registered). When Postgres needs to call a hook, it checks the relevant function pointer, and calls the registered function, if the pointer is set.

2.4.4 Implementation

ProvSQL [Sen] uses the planner_hook, which is called after a query has been parsed, and before it is sent to the query planner. The system architecture (as part of Postgres’s query execution pipeline) is depicted in Figure 2.1. ProvSQL currently supports a wide range of non-aggregate SQL queries (for more details see [Sen, SJE18]). The generated query result includes a provsql column of unique7 tokens, identifying gates of the produced provenance circuit.

7128-bit universally unique identifiers (UUIDs) that are generated using the uuid-ossp PostgreSQL module.

8Explicit mentions of the provsql column in the query (in any of the SELECT, FROM or WHERE clauses).
Figure 2.1: ProvSQL system architecture.
The red rectangles are a part of Postgres’s built in execution pipeline. The white rectangles are ProvSQL’s implementation of a `planner_hook`, which we use for the provenance-related calculations.
Chapter 3

Lineage via Embeddings

3.1 Word Embeddings

Classic NLP research focuses on understanding the structure of text. For example, building dependency-based parse trees [Mel88, KM04] that represent the syntactic structure of a sentence via grammatical relations between the words. These approaches did not account for the meaning of words. Word embedding aims to encode meanings of words (i.e., semantics), via low dimension (usually, 200-300) real-valued vectors, which can be used to compute the similarity of words as well as test for analogies [MYZ13]. Two of the most influential methods for computing word embeddings are the Word2Vec family of algorithms, by Mikolov et al. [MCCD13, MSC+13] and GloVe by Pennington et al. [PSM14]. Furthermore, applying neural-network (NN) techniques to NLP problems (machine translation [WSC+16], named entity recognition [GBVS16], sentiment analysis [MDP+11] etc.) naturally leads to the representation of words and text as real-valued vectors.

3.2 Motivation

A few problems become apparent when considering Distant Lineage (i.e., indirect, history long, explanations for the existence of tuples in the DB) with traditional and state of the art “exact provenance tracking” techniques:

- Formula based representations (e.g., semiring polynomials [GKT07]) may blow-up in terms of space consumption. A naive implementation scenario using semiring polynomials requires saving the full provenance polynomial for each tuple. This approach results in a massive growth in space consumption for storing those polynomials (for tuples that are produced by a query and that may depend on result tuples of previous queries).

- Inductively built representations (e.g., circuits [DMRT14, Sen17]) would become very complex over time. Thus, they result in impractical provenance querying
time. A naive implementation scenario using circuits would simply keep on constructing provenance circuits as described in section 2.4.2. During lineage querying, we may end up with very complex circuits, such that numerous leaves are derived via a circuit of their own (these leaves are tuples that were produced by a previous query and were inserted to the DB). This implies that even if a significant amount of sharing is realized across the provenance circuits - these constructions are inevitably going to blow-up in space consumption. This approach renders keeping and querying the full provenance impractical and requires limiting heavily the provenance resolution, otherwise (e.g., return a summarized explanation).

- Alternatively, if we only want lineage, rather than keeping a circuit, we could just store with each tuple a set of all the tuples it depends on. This will be cheaper than circuits but still prohibitively expensive. Here too, one could think of circuit-like techniques where tuples that have a subset in common, of tuples in their lineage, could share this subset. But again, this is complex and suffers from similar problems, as discussed above.

- Complex explanations are not very human-readable. Deutch et al. [DFG17] showed how to generate more human-readable explanations - but they are arguably still complex. A “top-N justifications” style provenance might be more useful for an analyst in a real-time interaction with the data.

### 3.3 Word Vectors Model

The system we propose is based on a word vectors model. Such a model is composed of a collection of real-valued vectors, each associated with a relevant DB term. The process of deriving vectors from DB-derived text is called embedding.

#### 3.3.1 Training word vectors

As in Bordawekar et al. [BS16], we train a WORD2VEC model [ŘS10] on a corpus that is extracted from the relevant DB. A naive transformation of a DB to unstructured text (a sequence of sentences) can be achieved by simply concatenating the textual representation of the different columns of each tuple into a separate sentence. This approach has several problems [BS16]:

- When dealing with natural language text, there is an implicit assumption that the semantic influence of a word on a nearby word is inversely proportional to the distance between them. However, not only that a sentence extracted from a tuple does not necessarily correspond to any natural language structure, but, it
can be actually thought of as “a bag of columns”; i.e., the order between different columns in a sentence has no semantic implications.

- All columns are not the same. That is, some columns-derived terms may hold more semantic importance than others in the same sentence (generated from a tuple). For instance, a primary key column, a foreign key column, or an important domain-specific column (e.g., a manufacturer column in a products table). This implies that in order to derive meaningful embeddings from the trained word vectors, we need to consider inter-column discrimination, during both training and evaluation phases. This topic is further discussed in chapter 4.

We solve these problems by properly setting hyperparameters (e.g., window size), artificially (and carefully) injecting repeated text inside a sentence, and dividing the training into multiple, different stages. A detailed outline of the “textification” and training processes follows:

1. Generate a Key column with unique values for each table in the DB.

2. Extract two corpora from the DB, one with columns (except the Key columns) as sentences, and the other with tuples as sentences.

3. Transform each numerical value in the generated corpora by concatenating the relevant column name to the number. Other methods of dealing with numerical values exist as well, and are discussed in section 3.4.

4. Transform each sentence in the generated corpora by injecting the corresponding Key value repeatedly after every $k$ (a separate hyperparameter for the columns corpus and the tuples corpus) words in the sentence. The idea is to encode more information about each tuple in its corresponding Key value vector.

5. Incrementally train a Word2Vec model on the generated corpora. First, train on the columns corpus and then on the tuples corpus. One may question why we have the same model for both the tuple-based (see chapter 4) and the column-based (see chapter 5) lineage vectors. The answer is that experimentally the models that are restricted to only the tuples corpus or only the columns corpus yield inferior results.

One more interesting caveat, of dealing with texts extracted from relations, is that missing data (empty/null cells) is sometimes automatically converted (when an external tool, e.g., Python, is used to interface with PostgreSQL) to ‘None’. Given that ‘None’ is not a stop word (words that are removed from a corpus as part of a common practice of cleanup before training), we realized that ‘None’ adds noise to the vectors (if there is a lot of missing data in the DB). Adding ‘None’ to the list of Word2Vec stop words improved drastically both the word embedding model quality and the overall
accuracy of our system.

The quality of the word vectors model is crucial to the success of our system. However, optimizing the overall performance should focus not only on the training phase, but also on the way we utilize the trained model. Next, we show a number of such optimizations.

### 3.3.2 Sentence embeddings

Extracting sentence embeddings from text has been a well-researched topic in the NLP community over the last seven years. State-of-the-art pre-trained models (e.g., Universal Sentence Encoder \([\text{CYK}^+18, \text{CYK}^+]\) and BERT [DCLT18]) are trained on natural language texts, and thus are not suitable for sentences generated from relational tuples (see discussion above). Hence we train a word embedding model and infer the sentence vectors as a function of the set of word vectors containing all the words in a sentence. We average the word vectors for each column separately, and then apply weighted average on the “column vectors” (the weight is based on the relative importance of a column, as discussed above). As will be shown, column-based vectors result in significant improvements to lineage encoding.

### 3.3.3 Similarity calculation

Given two word vectors, the similarity score is usually the cosine distance between them. In our system, we want to calculate the similarity between the lineage representations of two tuples/columns (see chapters 4 and 5, respectively); in both cases, the tuple or column, is associated with a set of lineage vectors. That is, we need to calculate the similarity between two sets of vectors\(^1\). The “logic” behind the following formula is balancing between the “best pair” of vectors (in terms of similarity) in the two sets and the similarity between their average vectors:

\[
\text{sim}(A, B) = \frac{w_{\text{max}} \cdot \text{max(ps)} + w_{\text{avg}} \cdot \text{avg(ps)}}{w_{\text{max}} + w_{\text{avg}}}
\]

where \(ps\) is the set of pair-wise similarities between a pair of vectors, one taken from set \(A\) and one taken from set \(B\). \(w_{\text{max}}\) and \(w_{\text{avg}}\) are (user-specified) hyperparameters. \(\text{max}\) and \(\text{avg}\) are functions that return the maximum and average values of a collection of numbers, respectively\(^2\).

\(^1\)We note that [BS16, BBS17, BS17] have also used various similar methods for measuring similarity between two sets of vectors.

\(^2\)This logic holds for both tuple-based vectors and column-based vectors (i.e., for each column separately).
3.4 Related Work

**Relational embedding** is a very active area of research [TTO18, Gro20, Shm20]. When converting a relational DB to unstructured text (see section 3.3) special care is required to support numerical values correctly. Bordawekar and Shmueli [BS16] take the route of tokenizing numerical values from the DB by preceding each number with a “range designator” (e.g., 1-10, 50-100, SMALL, BIG, etc.) in the generated corpus. In [BBS17] Bordawekar, Bandyopadhyay and Shmueli use clustering techniques to represent numerical values textually. In contrast, in this work we deal with numerical values by concatenating the relevant column name to each number in the generated corpus. This way, we ensure our model separates between numerical values of different fields, as they are conceptually different, semantically speaking (for example, we want the value 300 for a `nutrient_code` field in a `nutrients` table to have a different learned vector than the value 300 for a `household_serving_size` field in a `serving_size` table, from the USDA BFPDB [KMSR17] dataset). In the specific database example with which we experimented, this simple technique proved sufficient.
Chapter 4

Per-Tuple Lineage Vectors

4.1 Proposed Solution

We devise a novel approach to lineage tracking, which is based on ML and NLP techniques. The main idea is summarizing, and thereby approximating, the lineage of every tuple with a set of up to $\text{max\_vectors\_num}$ (a hyperparameter) vectors. For tuples that are inserted explicitly into the DB, the vectors are obtained using a pre-trained word embeddings model $M$ over the underlying "text" of the tuple (see Figure 4.1). During a query execution process we form the lineage of query result tuples in a similar fashion to that of provenance semirings [GKT07]. We extend the $+$ and $\cdot$ operations (see Figures 4.2 and 4.3, respectively) to generate lineage embeddings, while emphasizing the ability to propagate information and preserve the representation of lineage via a set of up to $\text{max\_vectors\_num}$, constant-size vectors. We obtain lineage embeddings for query output tuples by using this process. These new tuples (and their lineage) may be later inserted into the DB (depending on the specific application).

A real world provenance system like ProvSQL [Sen] can make use of our lineage vectors during the construction of provenance circuits. Recall that each node in a provenance circuit is associated with some annotation/token, either representing a tuple from the input DB or an intermediate calculation (see section 2.4). Lineage vectors can be incorporated into such a system by calling the Algorithm in Figure 4.4, for instance. This is an “online” approach, meaning the lineage vectors are generated during the query execution process. Another approach to incorporating lineage vectors in a real world provenance system might be using such a system as a black box, generating a provenance how-formula for each query output tuple. Later, this formula can be analyzed “offline”, parsed, and converted to lineage vectors by evaluating the parsed expression in a hierarchical manner. Each intermediate evaluation step would call either Figure 4.2 or 4.3, based on the parsing of the formula, mimicking the online nature of algorithm 4.4.

Running Example 1. Next, we show examples of the addition and multiplication lineage vectors constructions (Figures 4.2 and 4.3, respectively). Let $\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{v}_4 \in \mathbb{R}^2$ be
vectors, such that:
\[ \vec{v}_1 = \begin{pmatrix} -1 \\ 0.5 \end{pmatrix}, \vec{v}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \vec{v}_3 = \begin{pmatrix} -0.5 \\ 1 \end{pmatrix}, \vec{v}_4 = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \]

Suppose we have two tuples \( t_1, t_2 \) with respective sets of lineage vectors \( LV_1, LV_2 \), such that:
\[ LV_1 = \{ \vec{v}_1, \vec{v}_2, \vec{v}_3 \}, LV_2 = \{ \vec{v}_4 \} \]

Finally, the hyperparameters are:
\[ max\_vectors\_num = 3 \]

**Example 1.1.** Let us follow the construction of \( LV_3 \), which represents the lineage embeddings of \( t_1 + t_2 \) using the addition Algorithm in Figure 4.2 (corresponds to alternative use of data, i.e., OR in the query):

1. \( LV_1 \cup LV_2 = \{ \vec{v}_1, \vec{v}_2, \vec{v}_3 \} \cup \{ \vec{v}_4 \} = \{ \vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{v}_4 \} \)
2. \( |\{ \vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{v}_4 \}| = 4 > 3 = max\_vectors\_num \)
3. \( LV_3 = \text{ClusterVectorsUsingKMeans}(\{ \vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{v}_4 \}) = \{ \vec{c}_1, \vec{c}_2, \vec{c}_3 \} \)
   \[ \vec{c}_1 = \begin{pmatrix} -0.75 \\ 0.75 \end{pmatrix}, \vec{c}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \vec{c}_3 = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \]
   are the centroids of the three clusters.

**Example 1.2.** Let us follow the construction of \( LV_3 \), that represents the lineage embeddings of \( t_1 \cdot t_2 \) using the multiplication Algorithm in Figure 4.3 (corresponds to joint use of data, i.e., AND in the query):

1. \( \text{CartesianProduct}(LV_1, LV_2) = \{ (\vec{v}_1, \vec{v}_4), (\vec{v}_2, \vec{v}_4), (\vec{v}_3, \vec{v}_4) \} \)
2. \( \{ \text{Avg}(\vec{v}_1, \vec{v}_4), \text{Avg}(\vec{v}_2, \vec{v}_4), \text{Avg}(\vec{v}_3, \vec{v}_4) \} = \{ \vec{a}_1, \vec{a}_2, \vec{a}_3 \} \)
   such that:
   \[ \vec{a}_1 = \begin{pmatrix} -0.5 \\ -0.25 \end{pmatrix}, \vec{a}_2 = \begin{pmatrix} 0.5 \\ 0 \end{pmatrix}, \vec{a}_3 = \begin{pmatrix} -0.25 \\ 0 \end{pmatrix} \]
   are the average vectors, of each pair, in the cartesian product.
3. \( |\{ \vec{a}_1, \vec{a}_2, \vec{a}_3 \}| = 3 \leq 3 = max\_vectors\_num \)
4. \( LV_3 = \{ \vec{a}_1, \vec{a}_2, \vec{a}_3 \} \)
Input: a pre-trained word embeddings model \( M \), an input database \( D \)

Result: calculate an initial lineage vector \( v_t \) for every tuple \( t \) in \( D \)

1. \( \text{for } t \in D \text{ tuples do} \)
2. \hfill /* \( M \) is a pre-trained word embeddings model and \( t \) is a DB tuple */
3. \hfill \( \text{word}_\text{vectors}_t = \{ M(w) \mid w \in t \}; \)
4. \hfill /* Perform a "smart" averaging over the word vectors of every word in \( t \). That is, average the word vectors for each column separately, and then apply weighted average on the "column vectors". */
5. \hfill \( v_t = \text{WeightedAverage}(\text{word}_\text{vectors}_t); \)
6. \( \text{end} \)

Figure 4.1: Lineage vectors: Initialization Algorithm

Hyper-Parameters: \( \text{max}_\text{vectors}_\text{num} \) - maximum number of vectors per-tuple

Input: two sets of lineage vectors \( LV_1, LV_2 \) that represent the lineage of two tuples \( t_1 \) and \( t_2 \), respectively

Result: a new set of lineage vectors \( LV_3 \), that represents \( t_1 + t_2 \)

1. \hfill /* Call this algorithm via \( +(LV_1, LV_2) \) or using the infix notation \( LV_1 + LV_2 \) */
2. \hfill \( LV_3 = LV_1 \cup LV_2; \)
3. \hfill \text{if } |LV_3| > \text{max}_\text{vectors}_\text{num} \text{ then}
4. \hfill /* Perform K-Means clustering over the vectors into \( \text{max}_\text{vectors}_\text{num} \) groups and return the centers (i.e., each center is a vector) of each group */
5. \hfill \( LV_3 = \text{ClusterVectorsUsingKMeans}(LV_3); \)
6. \( \text{end} \)

Figure 4.2: Lineage vectors: \( + \) (addition) Algorithm

4.1.1 Lineage querying

As was mentioned in section 3.1, word embeddings (i.e., vectors) provide insights into the meaning of words via a word-word similarity score. The similarity measure is the cosine distance between the word vectors. Our goal is to construct and maintain lineage embeddings, which can provide insights to the reasons for the existence of tuples via a tuple-tuple similarity score, which is analogous to word-word similarity score; the tuples of interest are the ones generated by a query, which in turn use previously query-generated tuples as well as explicitly inserted ones.

Consequently, given a tuple and its lineage embedding vectors, we can calculate the pair-wise similarity against every other tuple in the DB (or a subset, e.g., in a specific table) and return the top \( N \) (a parameter) most lineage-similar tuples (these resemble
Hyper-Parameters: 

- max_vectors_num - maximum number of vectors per-tuple

Input: two sets of lineage vectors $LV_1, LV_2$ that represent the lineage of two tuples $t_1$ and $t_2$, respectively

Result: a new set of lineage vectors $LV_3$, that represents $t_1 \cdot t_2$

```plaintext
/* Call this algorithm via $(LV_1, LV_2)$ or using the infix notation $LV_1 \cdot LV_2$ */

1. $LV_3 = \{ \text{Avg}(v_1, v_2) \mid v_1, v_2 \in \text{CartesianProduct}(LV_1, LV_2) \}$;
2. if $|LV_3| > \text{max\_vectors\_num}$ then
   /* Perform K-Means clustering over the vectors into max_vectors_num groups and return the centers (i.e., each center is a vector) of each group */
   3. $LV_3 = \text{ClusterVectorsUsingKMeans}(LV_3)$;

end
```

Figure 4.3: Lineage vectors: $(\cdot$ (multiplication)) Algorithm

a subset of the lineage [CWW00]). There are many algorithms for approximate vector search, e.g., based on LSH [GIM99]. Approximate vector search is a very active area of research and we can utilize known algorithms (see, e.g., [SKI16]); however, these algorithms are not directly applicable to our method, since we operate on sets of vectors instead of single vectors. In appendix A we provide the first step towards solving this problem. Due to the reliance of lineage embeddings on an underlying statistical ML model, we expect our produced lineage to approximate the exact lineage.

### 4.1.2 Intended usage

The intended usage of lineage vectors is as follows:

- Each manually inserted tuple - has a set consisting of a single tuple vector.
- Each tuple in the result of a query - has a set of up to max_vectors_num tuple vectors.
- When calculating similarity - we always compare between two sets of lineage vectors, by using the formula we presented in section 3.3.3.

### 4.1.3 Comparison to exact lineage

In section 3.2 we listed some problems that arise when approaching distant (i.e., over the whole DB history) lineage computation with traditional techniques. Our proposed solution addresses these problems as follows:

- Lineage embeddings require only a constant space per-tuple.
Input: a pair of tokens $token_1, token_2$ and their associated sets of lineage vectors $LV_1, LV_2$

an operation $op \in \{+.,\}$

Result: a new token $token_3$, that is associated with the new set of lineage vectors $LV_3 = LV_1 \ op \ LV_2$

1 switch $op$ do
2    case $+$ do
3       $LV_3 = LV_1 + LV_2$ ;
4       // see Algorithm 4.2
5    end
6    case $.$ do
7       $LV_3 = LV_1 \cdot LV_2$ ;
8       // see Algorithm 4.3
9    end
10 end
11 tag the newly generated $LV_3$ as $token_3$;

Figure 4.4: Incorporating lineage vectors in the ProvSQL Algorithm

- Lineage embeddings are immutable once computed\(^1\), and do not become more complex over time (querying time depends only on the number of tuples, with which we compare the target tuple, i.e., the one to be explained via lineage).

- Returned explanations (produced lineage) are simply the top-N justifications, and may assist in further analysis.

We approximate lineage using embedding techniques. As noted, lineage is a type of provenance, and it can be expressed using semirings. Now, semirings can also express how-provenance, which is more general and informative than lineage, but, it becomes extremely complex to track as histories develop. Thus, lineage seems to be a more practical tool for analysts. A major advantage of our approach is realized for distant provenance. A process that is equivalent functionally to recursive drill through is done automatically (i.e., if we envision the distant provenance of a tuple as a recursive tree-like structure).

4.2 Verifying The Lineage

Given a method that approximates the top-N justifications for the existence of a query result tuple, as the one presented above, we propose a simple technique to verify the collection of lineage candidates. The idea is applying the query $q$ that generated the tuple to be explained, $t$ to the collection of lineage tuples obtained by our system. If $t$ is output in the result of applying $q$ on these lineage candidates, then we found

\(^1\)This is the case when the text characteristics of the DB-extracted text are stable. The case of a changing text characteristics is left for future research.
a sufficiently small (though, non-precise) explanation for \( t \). Furthermore, one could utilize an exact provenance tracking application, e.g., ProvSQL \([\text{Sen}]\) by applying \( q \) to the collection of tuples obtained by our system, to explain \( t \) (given that \( t \) is “verified”). This technique relies on the fact that executing \( q \) on a small collection of tuples (even with exact provenance tracking) is significantly cheaper than applying the query on the whole, potentially large DB.

4.3 Interim Findings and Hurdles

After developing the concept of lineage embeddings in isolation, it is essential to test it as part of a real-world implementation. We developed a Python module that adds “lineage via embeddings” capabilities to a Postgres DBMS, by integrating our module with the ProvSQL \([\text{SJE18}]\) extension. The construction of the lineage vectors is based on the algorithms presented above.

We conducted a series of experiments (see chapter 7) and came to the following interim conclusions:

- **Our explanations** (returned tuples comprising our lineage approximation) encode the DB-resident text and not the tuples. Thus, relations with many similar tuples (text-wise) result in our system performing poorly. We conjecture that most of the problems listed below are symptoms of the same root cause.

- **Analysis of direct lineage** (i.e., *direct* explanations for the existence of tuples in a query result in terms of existing DB tuples, regardless of their historical raison d'être) on simple, complex (ones that might involve multiple tables with non-trivial joins and non-trivial “where” conditions) and composite (that contain sub-queries) queries gives good results (> 90% accuracy for “top 50%” of the tuples in the lineage) generally. However, results suffer from inconsistencies sometimes, identifying tuples incorrectly as part of the lineage due to their textual similarity to actual lineage tuples.

- **Distant lineage** provides useful results. However, it suffers from the same problem (as identified above) of textual closeness.

In the next chapters we introduce methods that are designed to overcome these deficiencies.
Chapter 5

Per-Column Lineage Vectors

5.1 Rationale

As noted, text obtained from database relations does not behave like natural language text, and a sequence of columns values in a tuple does not usually construct a natural language sentence. Thus, building lineage vectors at the tuple level and comparing tuples on this basis is noisy and potentially lossy in terms of information. Instead, we devise a genetics-inspired approach. Suppose a tuple represents the genetic code of an entity (a living organism in the DB world). The columns of an entity are its basic properties, i.e., the genes that combine to construct its genetic code. In this setting, querying the direct lineage of a result tuple is analogous to finding its predecessors through DNA testing. Following the approach presented above, we design an alternative lineage tracking mechanism, that of keeping track of and querying lineage (via embeddings) at the column (gene) level. This way, we manage to better distinguish between the provenance features and the textual characteristics of a tuple.

5.2 Implementation

1. Each tuple has a set of (up to max_vectors_num) column lineage vectors per-column, instead of a single set of tuple vectors (per-tuple). Let us denote the set of columns (features, characteristics) for which a tuple t has lineage embeddings as t.lineage_columns.

2. From now on, we formally denote a column name, using the full name notation, as T.Attr, such that T is a relation name and Attr is a column name. This is meant to achieve better distinction at the lineage vectors level between tuples that originate from different relations with the same column names.
3. Given a tuple \( t \), we say that \( t\text{.lineage\_columns} = t\text{.native\_columns} \cup t\text{.distant\_columns} \), such that \( t\text{.native\_columns} \) is the set of columns that construct \( t \) and \( t\text{.distant\_columns} \) is the set of columns (features, characteristics) that \( t \) “inherited” from its predecessors, but are not reflected as data columns in \( t \). Note that the same column name may be in \( t \) and inherited from its predecessors as well. This issue will be dealt with shortly.

In the following examples we use \( A, B, C, D \) to represent the full name of a column for brevity. For example, suppose a tuple \( t \in T_{AB} \) (a table with only two columns - \( A \) and \( B \)) has the column lineage vectors \( CV_t = \{A: LV_A, B: LV_B, C: LV_C, D: LV_D\} \) (a map of: \( \text{columns} \rightarrow \text{sets of lineage vectors} \)). We say that \( t\text{.native\_columns} = \{A, B\} \), \( t\text{.distant\_columns} = \{C, D\} \) and \( t\text{.lineage\_columns} = \{A, B, C, D\} \).

**Note.** From here on, we use the notation \( CV_t\text{.columns} \) to denote the set of columns in the domain of the map \( CV_t \).

4. When combining lineage embeddings (see Algorithms 5.1, 5.2), all calculations are done at the column level.

```
Input: two maps of: \( \text{columns} \rightarrow \text{sets of lineage vectors} \) \( CV_1, CV_2 \) that represent the lineage of two tuples \( t_1 \) and \( t_2 \), respectively
Result: a new map of: \( \text{columns} \rightarrow \text{sets of lineage vectors} \) \( CV_3 \), that represents \( t_1 + t_2 \)

1  \( CV_3 = \{\} \rightarrow \{\}; \quad \text{// an empty map of: \( \text{columns} \rightarrow \text{sets of lineage vectors} \)}
2  \text{foreach column } c \in (CV_1\text{.columns} \cup CV_2\text{.columns}) \text{ do}
3    /* \( CV_t\text{.columns} \) is the domain and \( CV_t[c] \) is the corresponding set of lineage vectors of tuple \( t_t \) for the column \( c \in CV_t\text{.columns} \) */
4    if \( c \in (CV_1\text{.columns} \cap CV_2\text{.columns}) \) then
5      \( CV_3[c] = CV_1[c] + CV_2[c]; \quad \text{// see Algorithm 4.2} \)
6    else if \( c \in CV_1\text{.columns} \) then
7      \( CV_3[c] = CV_1[c]; \)
8    else
9      \( CV_3[c] = CV_2[c]; \)
10  end
11 end
```

Figure 5.1: Column Lineage vectors: + (addition) Algorithm

5. After constructing a new tuple \( t \in T \) via a query \( q \) and its per-column lineage vectors \( CV_t \) (via a series of + and · operations, see Algorithms 5.1 and 5.2,
Input: two maps of: columns → sets of lineage vectors $CV_1, CV_2$ that represent the lineage of two tuples $t_1$ and $t_2$, respectively

Result: a new map of: columns → sets of lineage vectors $CV_3$, that represents $t_1 \cdot t_2$

1 $CV_3 = \{\} \rightarrow \{\}$; // an empty map of: columns → sets of lineage vectors

2 foreach column $c \in (CV_1.columns \cup CV_2.columns)$ do
   3 /* $CV_i.columns$ is the domain and $CV_i[c]$ is the corresponding set of lineage vectors of tuple $t_i$ for the column $c \in CV_i.columns$ */
   4 if $c \in (CV_1.columns \cap CV_2.columns)$ then
      5 $CV_3[c] = CV_1[c] \cdot CV_2[c]$; // see Algorithm 4.3
   6 else if $c \in CV_1.columns$ then
      7 $CV_3[c] = CV_1[c]$;
   8 else
      9 $CV_3[c] = CV_2[c]$;
   10 end

Figure 5.2: Column Lineage vectors: $\cdot$ (multiplication) Algorithm

respectively) - special care must be taken in dealing with the native_columns of $t$. That is, for every column $A \in t.native_columns$ ($A$ represents the full name of a column for brevity):

(a) If $A \notin CV_1.columns$ and $t.A$ is set to an existing value from some column in the DB, e.g., $q = (INSERT INTO T SELECT A' FROM ...)$ then we set $CV_i[A] = CV_i[A']$.

(b) If $A \notin CV_1.columns$ and $t.A$ is set to some constant value, e.g., $q = (INSERT INTO T SELECT const FROM ...)$ then we set $CV_i[A] = \{initial\_vector(const)\}$, such that initial\_vector(const) is calculated according to Algorithm 4.1 with the “textified” (see section 3.3) form of const as input data.

(c) If $A \in CV_1.columns$ and $t.A$ is set to some constant value, e.g., $q = (INSERT INTO T SELECT const FROM ...)$ then we set $CV_i[A] = CV_i[A] \cdot \{initial\_vector(const)\}$. This way, we incorporate both the existing lineage data and the newly calculated one for the column $A$.

6. When comparing a tuple $t$ (and its lineage embeddings) to a set of other tuples $T'$:

(a) If $t'.lineage\_columns \notin t.lineage\_columns$ ($t' \in T'$) then $t'$ is definitely not a part of the lineage of $t$. Otherwise, all of the “genes” (columns) of $t'$ would be reflected in $t$. 

25
The similarity between \( t \) and \( t' \in T' \) is averaged over the pair-wise similarities of their respective mutual genes (lineage columns). For example, say a tuple \( t \) that has lineage vectors for columns \( A, B, C \) is compared to another tuple \( t' \) that has lineage vectors for columns \( B, C, D \) (here, \( t' \) is an arbitrary tuple, which is not necessarily in the lineage of \( t \)):

\[
sim(t, t') = \text{avg}\left(\{\sim(t.B, t'B), \sim(t.C, t'C)\}\right)
\]

where \( \text{avg} \) is a function that returns the average of a collection of numbers. \( \sim(t.B, t'B) \) is the calculated similarity between the lineage vectors for column \( B \) of \( t \) and \( t' \) (similarly for column \( C \)). Observe that \( A \) and \( D \) are not mutual “genes” of the tuples \( t \) and \( t' \), and, thus they do not hold lineage information that is valuable to the similarity calculation.

**Running Example 2.** Next, we show examples of the addition and multiplication column lineage vectors constructions (Algorithms 4 and 5, respectively). Suppose we have two tuples \( t_1, t_2 \) with respective maps of columns \( \rightarrow \) sets of lineage vectors \( CV_1, CV_2 \), such that:

\[
CV_1 = \{A: LV_A, B: LV_{B_1}\}, \; CV_2 = \{B: LV_{B_2}, C: LV_C\}
\]

\( A, B, C \) are full column names and \( LV_A, LV_{B_1}, LV_{B_2}, LV_C \) are sets of lineage vectors.

**Example 2.1.** Let us follow the construction of \( CV_3 \), which represents the column lineage embeddings of \( t_1 + t_2 \) using Algorithm 5.1 (corresponds to alternative use of data, i.e., OR in the query):

1. \( CV_3 = \{\} \rightarrow \{\} \)
2. \( CV_1.columns \cup CV_2.columns = \{A, B\} \cup \{B, C\} = \{A, B, C\} \)
3. \( A \in CV_1.columns \land A \notin CV_2.columns \)
   \( \Rightarrow CV_3[A] = CV_1[A] = LV_A \)
4. \( B \in (CV_1.columns \cap CV_2.columns) \)
   \( \Rightarrow CV_3[B] = CV_1[B] + CV_2[B] = LV_{B_1} + LV_{B_2} \)
5. \( C \notin CV_1.columns \land C \in CV_2.columns \)
   \( \Rightarrow CV_3[C] = CV_2[C] = LV_C \)
6. \( CV_3 = \{A: LV_A, B: LV_{B_1} + LV_{B_2}, C: LV_C\} \)

**Example 2.2.** Let us follow the construction of \( CV_3 \), that represents the column lineage embeddings of \( t_1 \cdot t_2 \) using Algorithm 5.2 (corresponds to joint use of data, i.e., AND in the query):

1. \( CV_3 = \{\} \rightarrow \{\} \)
2. \(CV_1.columns \cup CV_2.columns = \{A, B\} \cup \{B, C\} = \{A, B, C\}\)

3. \(A \in CV_1.columns \land A \notin CV_2.columns\)
\[
\Rightarrow CV_3[A] = CV_1[A] = LV_A
\]

4. \(B \in (CV_1.columns \cap CV_2.columns)\)
\[
\Rightarrow CV_3[B] = CV_1[B] \cdot CV_2[B] = LV_{B_1} \cdot LV_{B_2}
\]

5. \(C \notin CV_1.columns \land C \in CV_2.columns\)
\[
\Rightarrow CV_3[C] = CV_2[C] = LV_C
\]

6. \(CV_3 = \{A: LV_A, B: LV_{B_1} \cdot LV_{B_2}, C: LV_C\}\)

**Example 2.3.** Let us follow the construction of \(CV_3\), that represents the column lineage embeddings of \(t_3 = t_1 + t_2\), such that the result is a new tuple \(t_3 \in T_{AB}\) (a table with only two columns - \(A\) and \(B\)). In addition, we use a bound of \(b = 2\) on the number of retained lineage columns and implement native priority:

1. (see example 2.1)
\[
CV_3 = \{A: LV_A, B: LV_{B_1} + LV_{B_2}, C: LV_C\}
\Rightarrow t_3.lineage_columns = \{A, B, C\}
\]

2. \(t_3 \in T_{AB} \Rightarrow t_3.native_columns = \{A, B\}, t_3.distant_columns = \{C\}\)

3. \(|t_3.lineage_columns| = 3 > 2 = b\)

4. \(CV_3 = \{A: LV_A, B: LV_{B_1} + LV_{B_2}\}\)
   (the distant column \(C\) was eliminated)

**5.3 Discussion**

- The rationale behind this method is partially inspired by the construction of “tuple vectors” by means of averaging the “sentence vectors” of the columns (as discussed in section 3.3.2).

- \(t.lineage_columns\) (for a tuple \(t\)) might get large and include (almost) all the columns in the DB. This can render this solution impractical, since large modern systems may operate with hundreds and even more columns across the DB. There are various practical techniques that can serve to limit the number of lineage columns per tuple:

  1. Set a bound (hyperparameter) on the number of lineage columns.
  2. Give native priority. That is, prefer keeping native columns and cutting-off distant columns, whose influence on this tuple is more remote, when the bound is reached (see example 2.3 below).
3. Remove the relation name prefix from a column name, i.e., consider all $T.Attr$ as simply $Attr$. This might result in a loss of information, but hopefully not too harmful to the overall performance of the method.
Chapter 6

Improving Lineage Embeddings

6.1 Query-Dependent Column Weighting

6.1.1 Rationale

When analyzing lineage embeddings of query-result tuples against other tuples in the DB, we would like to emphasize certain columns of interest. These columns can be derived from the structure of the query. For example, if a query asks for distinct manufacturer names of products that contain soda, then intuitively, we want the manufacturer and ingredients columns (of the products table) to have more influence on the respective lineage embeddings than other columns that are not mentioned in the query.

6.1.2 Implementation

1. Given a query $q$, we parse $q$ to retrieve the columns of interest and save them as additional meta information for every tuple in the result of $q$.

2. When comparing a tuple from the result of $q$ with another tuple from the DB, we compare between respective column vectors and calculate a weighted average of the similarities while prioritizing the columns of interest. For example, say a tuple $t$ that has lineage vectors for columns $A,B$ was created by a query $q$, such that $q.cols\_of\_interest = \{B\}$; and we compare $t$ with another tuple $t'$ that has lineage vectors for columns $A,B$:

$$\text{sim}(t, t') = \frac{w_A * \text{sim}(t.A, t'.A) + w_B * \text{sim}(t.B, t'.B)}{w_A + w_B}$$

where $w_A$ and $w_B$ are respective column weights such that $w_B > w_A$ as column $B$ is mentioned in the query. $\text{sim}(t.A, t'.A)$ is the calculated similarity between the lineage vectors for column $A$ of $t$ and $t'$ (similarly for column $B$). Observe that although column $A$ is not a column of interest in the example, it is a mutual “gene” of tuples $t$ and $t'$, and, thus it still holds lineage information that is valuable to
the similarity calculation.

6.1.3 Note

This technique is applicable mainly for direct provenance. In the context of distant provenance we can use it to get “immediate contributors” but also take into account lineage tuples identified without this technique (they will naturally tend to be ranked lower).

6.2 Bloom-Filters of Queries

6.2.1 Rationale

We would like to distinguish between very similar/nearly identical tuples (text-wise). The main idea here is keeping additional information per-tuple, that encodes the queries that had the tuple in their lineage (i.e., for every tuple \( t \) - keep track of every query \( q \), such that \( t \) was involved meaningfully in the evaluation of \( q \)). Also, for each query-inserted tuple \( t \) record the query \( q \) that inserted it. This approach enables us to filter out (in a probabilistic manner) tuples that were not involved meaningfully in the evaluation of a query.

6.2.2 Implementation

1. Initialize a Scalable Bloom Filter [ABPH07] of size \( B \) (a hyperparameter) for every tuple in the DB.

2. For every tuple \( t \) that is involved meaningfully in the evaluation of a query \( q \): insert \( q \) (hashed) to \( t \)'s Bloom filter.

3. When comparing a tuple \( t \) (and its lineage embeddings) that was created by a query \( q \) (as recorded with this tuple) to a set of other tuples \( T' \), we can precede similarity calculations with the following step:

   (a) If \( q \notin t'.\text{bloom\_filter} \) then \( t' \) was definitely not involved in the evaluation of \( q \). Thus, it cannot be a part of \( t \)'s lineage.

6.2.3 Notes

- This basic filtering technique is not applicable to the querying of distant provenance, as it (obviously) looks for tuples that were not directly involved in the evaluation of a query. Extending this basic method to handle indirect provenance is the subject of current research. A possible direction is a recursive algorithm that takes into account the query that produced each already discovered (presumed lineage) tuple.
• Bloom Filters are probabilistic, but they do not produce false negatives. Thus, they can be reliably used to filter out non-lineage tuples (in our case, a false negative is a tuple \( t \) that was involved in the evaluation of a query \( q \), but \( q \notin t.bloom\_filter \)).

• In this work, we emphasize a size-bounded provenance representation. Therefore, it seems that using a Scalable Bloom filter (per-tuple) contradicts this notion, as its size is ultimately bounded by the number of queries served by the system. Therefore, we devise a “switch & reset policy” for these bloom filters:

1. Instead of only one Bloom filter - maintain two Bloom filters per-tuple (we fill them both).
2. One filter is “big and old” and one is “small and young”.
3. Lineage “filtering” is done against the old one.
4. Once the young one has seen sufficiently many queries - we switch to the young one and recycle the old filter. We then call the young old and create a new (empty) young filter.
5. This way we lose information but only about relatively “old” queries.
6. In order to maintain the invariant of “no false negatives” when querying the old Bloom filter of a tuple \( t \) - it is important to keep track of the oldest query id that is tracked by the old filter. That is, say the current old filter of \( t \) encodes information only for queries \( q \) with \( q.id \geq t.oldest\_query\_id \); then given a tuple \( t' \) which was created by a query \( q' \) with \( q'.id < t.oldest\_query\_id \) - we do not test for the existence of \( q' \) in \( t.bloom\_filter \) (otherwise, we risk returning a false negative).

• Bloom Filters are a universal enhancement that can be applied to any lineage-approximation technique. They complement beautifully the word embedding based lineage vectors, but are far from being a “silver bullet” on their own. In particular, in the extreme case where most (or even all) the tuples of a table \( A \) participate in the evaluation of a query \( q \), the Bloom Filters (almost) will not help when comparing a tuple from the result of \( q \) to tuples from table \( A \).

6.3 Tuple Creation Timestamp

6.3.1 Rationale

Filtering out non-lineage tuples by a tuple’s creation timestamp is a good practice, and can help with filtering out very similar but non-lineage tuples, which might not be detected by the other methods. It is especially important for querying distant lineage

---

1We view query id’s as timestamps.
(when the DB has existed for a sufficiently long time, so that a tuple’s creation timestamp becomes a strong distinguishing factor).

6.3.2 Implementation

1. Keep track of a tuple’s creation timestamp:
   - If a tuple $t$ was inserted explicitly to the DB then $t.timestamp$ is its insertion time.
   - If $t$ was calculated by a query, then $t.timestamp$ is the query’s execution time.

2. When comparing a tuple $t$ (and its lineage embeddings) to a set of other tuples $T'$, we can precede similarity calculations with the following step:
   - If $t'.timestamp > t.timestamp$ ($t' \in T'$) then $t'$ is newer than $t$ and therefore cannot be a part of its lineage.

6.4 Weighting with Query Dependency DAG

6.4.1 Rationale

We would like to distinguish between very similar/nearly identical tuples (text-wise) during both direct and distant lineage querying (Bloom filters are one way to achieve this for direct lineage). Additionally, we want to enhance the natural ranking of lineage tuples by amplifying the “generational gaps”. The idea is that tuples from earlier generations in the distant lineage tree structure (of a query-inserted tuple) are to be assigned a lower similarity score during distant lineage querying. The technique is keeping track of dependencies between queries in a DAG structure and weighting the similarity scores of query-inserted tuples (during lineage querying) inversely proportional to the distance, between their inserting queries to the query that computed the explained tuple, in the query dependency DAG. We say that a query $q$ depends on a query $p$ if a tuple $t_p$ that was created by $p$ was meaningfully involved in the evaluation of $q$ (i.e., $t_p$ is in the distant lineage of some result tuple $t_q$ in the result of $q$). This approach enables us to give lower similarity scores to tuples that were not involved meaningfully in the evaluation of a query (directly or remotely) and amplify the natural ranking of our results in terms of query creation dependencies.

6.4.2 Implementation

1. If there is a tuple $t_p$ that is involved meaningfully in the evaluation of a query $q$: we say that $q$ depends on the query $p$ that inserted $t_p$ into the DB.
2. Initialize an empty query dependency DAG $G = (V, E)$ with a maximum number of nodes $S$ and a maximum height $H$ (hyperparameters), such that $V$ is a set of queries and $E = \{(q, p) \in V \times V \mid q \text{ depends on } p\}$.

3. When comparing a tuple $t_q$ (and its lineage embeddings) that was created by a query $q$ (as recorded with this tuple) to another tuple $t_p$ in the DB, we can replace similarity calculations with the following steps:

   (a) Denote the sub-tree of $q$ in $G$ (i.e., rooted at $q$) as $G_q = (V_q, E_q)$.

   (b) Denote the query that created the tuple $t_p$ (as recorded with this tuple) as $p$.

   (c) If $p \in V_q$ then multiply the similarity for $t_p$ by $w_d \leq 1$, which is inversely proportional to the distance $d$ from $q$ to $p$ in $G_q$. One possible implementation for the distance-dependent weighting is $w_d = max\{\frac{1}{2}, 1 - \frac{(d-1)}{10}\}$, such that $w_1 = 1$, $w_2 = \frac{9}{10}$, etc., and $\forall 1 \leq d \leq H : w_d \geq \frac{1}{2}$. Other implementations exist as well.

   (d) If $p \notin V_q$ then multiply the similarity for $t_p$ by $0 < w_{\text{outsider}} < w_{d=H}$ (a hyperparameter). Note that since we defined limits on the maximum number of nodes $S$ and maximum height $H$, some nodes will need to be removed when those limits are reached (see details below). Thus, this method might start producing false negatives at some point; and to not lose those tuples completely, we choose to multiply their similarity by some small, non-zero constant, instead of plain filtering them.

6.4.3 Notes

- With each node in the graph $G$ we keep additional information about its height. This information is maintained and updated upon new edge additions to $G$.

- Maintain a hash table of queries, for a quick look-up of nodes in the query dependency graph.

- Maintain a doubly linked list of queries that is sorted by query execution order (oldest to newest). This list helps us to quickly find the “oldest” queries that need to be removed from the graph, when the cardinality limit $S$ is reached.

- When the height limit $H$ is reached, we eliminate “deep” leaves (those are the oldest queries in a given branch).

- In this work we emphasize a lineage tracking system with constant additional space per tuple. The query dependency DAG adheres to the same philosophy and is limited in size by the hyperparameters $S$ and $H$.
6.5 Limit Provenance Analysis to Specific Base Tables

6.5.1 Rationale

We consider those tables that do not depend on the contents of the DB when these tables are created, and thereafter, as base tables. Note that tuples can be manually inserted to and deleted from base tables, but, not in a programmatic manner over the DB, e.g., a SQL \texttt{INSERT} statement that is based on the DB contents. At times, an analyst might be interested only in a subset of base tables for lineage analysis of tuples that were created by a query. That is, an analyst’s workflow may be comprised of the following steps:

1. Choose a tuple of interest $t$ that was created by a query - for analysis.

2. Compare the tuple $t$ (via lineage vectors) to tuples from a pre-defined subset of the base tables (e.g., $\{A, B\} \subset \{A, B, C, D, E\}$, where $A, B, C, D, E$ are base relations in the DB).

3. Find the Top-$K$ ($K$ is a parameter) source tuples (by similarity) that were meaningfully involved in the creation of $t$.

This workflow seems useful and worthy of consideration. Note that it entails a relaxed assumption about the lineage analysis requirements in our system, and we would like to exploit it to produce better (i.e., more precise) answers.

A modern DB may include a large number of base tables, and new tuples that are created from base tables (by a query). As time goes on and the queries become more complex and involve (directly or indirectly) more base tables - we anticipate a growing amount of “noise” in the lineage vectors that we maintain. Given an assumption that an analyst is only interested in a (relatively small) subset of the base tables for lineage analysis, we hypothesize that it is possible to diminish the amount of “noise” that we propagate over time (via the lineage vectors) by limiting lineage tracking to tuples from base tables that are of interest to the analyst, and tuples that are created from them (by queries).

6.5.2 Implementation

This technique can be implemented as a wrapper to the addition and multiplication operations on lineage vectors, as presented previously in Algorithms 4.2 and 4.3:

1. Denote the subset of base tables that is of interest to the analyst as $S$.

2. For each tuple $t$, define $t\.related\_tables = \{T' \mid \exists \t' \in T' : \t' \text{ is in the distant lineage of } t\}$. That is, all the tables that were involved (directly or indirectly) in the evaluation of $t$.
3. Suppose we are given two tuples \( t_1 \) and \( t_2 \) with corresponding sets of lineage vectors \( LV_1 \) and \( LV_2 \) (either tuple vectors, or column vectors for a specific column) and an operation \( op \in \{ +, \cdot \} \), such that we are interested in calculating \( LV_1 \ op \ LV_2 \).

4. If \( t_1.related\_tables \cap S \neq \emptyset \) and \( t_2.related\_tables \cap S \neq \emptyset \), then we proceed as usual, namely, return \( LV_1 \ op \ LV_2 \).

5. Otherwise, if only one of \( t_1, t_2 \) has related tables in \( S \), i.e., suppose without loss of generality that \( t_1.related\_tables \cap S \neq \emptyset \) and \( t_2.related\_tables \cap S = \emptyset \) then we return \( LV_1 \). That is, we propagate forward only the relevant lineage data.

6. Otherwise, if both \( t_1 \) and \( t_2 \) are of no interest to an analyst, then the output value is of no relevance either, and no vector calculation is performed.

### 6.5.3 Notes

- Borrowing from semiring theory, this technique is analogous to tagging each “irrelevant” base tuple as additive and multiplicative identity in terms of lineage vectors operations.

- Propagating the “related tables” information (during lineage vectors computation) is essential for the correctness of this method.

- Suppose there are analysts \( A_1, \ldots, A_m \) each with their own set of base tables of interest \( B_1, \ldots, B_m \). Then, we may conceivably apply this technique separately to each \( A_i \) and \( B_i \) (of course, with the implied storage overhead).

### 6.6 Where Provenance

Recording lineage vectors per fully named columns (see chapter 5) enables implementing a form of where provenance (we are only interested in non-distant provenance here). By where provenance we mean the following. Given a tuple \( t \) in relation \( T \) of the DB, a native column \( A \) in \( t \) and a value \( v_A \) for \( t \) in column \( T.A \), determine the likely tuples \( t \), the where tuples in the DB, from which the value \( v_A \) originated. If the \texttt{SELECT} clause of \( q \), the query that inserted \( t \) into the DB, indicates an assigned constant, then no such where tuples exist.

Otherwise, the technique for locating likely where tuples explaining \( v_A \) is as follows. Let \( CV_t \) be the mapping associated with the tuple \( t \):

1. Find the where tuples \textit{candidates} by analyzing the structure of \( q \), the query that inserted \( t \) into the DB. That is, if \( q \) indicates in the \texttt{SELECT} clause that \( T'.A' \) is assigned to \( T.A \), then the candidates are the tuples \( t' \in T' \) that agree with \( t \) on the value of \( v_A \) (i.e., \( v'_A = v_A \), where \( v'_A \) is the value for \( t' \) in column \( T'.A' \)). Note that:
• If the SELECT clause of \( q \) contains other columns from \( T' \), e.g., the column \( T'.B' \) is assigned to \( T.B \), then additional filtering of candidate where tuples is possible, as these tuples must agree with \( t \) on the respective value \( v_B \) of tuple \( t \) for column \( T.B \) as well.

• If \( q \) indicates in the SELECT clause that \( T.A \) is assigned to by combining multiple columns, e.g., \( T'_1.A'_1 \ast T'_2.A'_2 \) is assigned to \( T.A \), then the candidates are the tuples \( t'_1 \in T'_1 \) and \( t'_2 \in T'_2 \) that (when combined) agree with \( t \) on the value of \( v_A \) (i.e., \( v'_{A_1} \ast v'_{A_2} = v_A \), where \( v'_{A_i} \) is the value for \( t'_i \) in column \( T'_i.A'_i \)).

2. A where tuple needs be directly involved in the computation of \( t \), namely used by the query \( q \) that inserted \( t \) into the DB. This information is encoded both in \( t' \)'s Bloom Filter of queries and the query dependency DAG (some redundancy, though the Bloom Filter is more precise in this regard). So, if \( q \notin t'.\text{bloom\_filter} \) or if \( t \) was inserted by a query \( p \) and \((q, p)\) is not an edge in the query dependency DAG, \( t \) cannot be a where tuple explaining \( v_A \) in \( T.A \).

3. To test closeness between \( t \) and a candidate where tuple \( t' \), we simply calculate \( \text{sim}(t, t') \), the similarity between these two tuples, using the mappings \( CV_t \) and \( CV_{t'} \) associated with \( t \) and \( t' \), respectively (as explained in chapter 5). Intuitively, it is more beneficial to consider the full mappings \( CV_t \) and \( CV_{t'} \), when testing closeness between \( t \) and \( t' \), instead of just comparing \( CV_t[T.A] \) and \( CV_{t'}[T'.A'] \). The reason is that the full mappings usually contain additional useful information, based on which the candidates can be compared and better differentiated.
Chapter 7

Experimental Evaluation

In this chapter we first describe the datasets on which we conducted the experiments, and establish an objective explanation quality measure. We then present and analyze results of our approximate lineage system experimentation.

7.1 Experimental Setup

We used two datasets from different domains (food and movies) to compare the performance of our approximate lineage system against ProvSQL. The goal of these experiments is evaluating the precision of our approximate explanations against an “exact provenance tracking system”.

Note. max_vectors_num (see section 4.1) was chosen manually to be 4. The choice of hyperparameters, and, specifically, max_vectors_num - is the subject of ongoing research.

7.1.1 Datasets

1. The USDA Branded Food Products Database (BFPDB) [KMSR17] is the result of a Public-Private Partnership, whose goal is to enhance public health and the sharing of open data by complementing USDA Food Composition Databases with nutrient composition of branded foods and private label data, provided by the food industry. Among others, the dataset includes three tables:

   - **products** - contains basic information on branded products (manufacturer, name, ingredients).
   - **nutrients** - each tuple contains all nutrient value information about a product.
   - **serving_size** - each tuple contains information about a product’s serving size (as well as unit of measurement) and food preparation state.
• **derivation_code_description** - contains all nutrient derivation codes and their detailed descriptions.

Each product in the BFPDB dataset has a unique identifier, **ndb_no**. For a detailed description of the BFPDB dataset see appendix B.

2. The MovieLens dataset [HK15] describes people’s preferences for movies. These preferences are entered by way of the MovieLens website [mov] — a recommender system that asks its users to provide movie ratings in order to receive personalized movie recommendations. Among others, the dataset includes three tables:

- **movies** - each tuple contains basic information about a movie (title, genres).
- **ratings** - each tuple contains information about a user’s expressed preference of a movie (a 0-5 star rating).
- **tags** - each tuple contains information about a user’s classification tag of a movie (silly, funny, etc.).

Each movie in the MovieLens dataset has a unique identifier, **movieId**. For a detailed description of the MovieLens dataset see appendix C.

### 7.1.2 Precision Calculation

In section 4.1 we stated that we expect our explanations to approximate the exact lineage of query result tuples. Thus, in order to test the aforementioned algorithms and implementation, we devised an explanation quality measure for explaining a single tuple $t$, where $n$ is a parameter:

$$\text{Precision}(t, n) = \frac{|\text{ApproxLineage}(t, n) \cap \text{ExactLineage}(t)|}{|\text{ApproxLineage}(t, n)|}$$

where $\text{ApproxLineage}(t, n)$ is the set of the top $n$ (by lineage similarity) tuples, returned as explanations (i.e., approximate lineage) for $t$ by our system. $\text{ExactLineage}(t)$ is the set of tuples that comprise the exact **distant** lineage of $t$, it can be calculated recursively from the semiring polynomial that is returned by the ProvSQL system for $t$. For example, if the direct lineage of $t_4$ and $t_5$ are the sets $\{t_1, t_2, t_3\}$ and $\{t_1\}$, respectively, and the direct lineage of a tuple $t_6$ is the set $\{t_4, t_5\}$ then the total distant lineage for the tuple $t_6$ is $\{t_1, t_2, t_3, t_4, t_5\}$. The parameter $n$ is set by us in each experiment, that is, it is a parameter of the experiment. Precision$(t, n)$ is tunable via the parameter $n$, i.e., when $n$ is small we test for precision of the “top” explanations we found.

Classically, we are usually interested in precision and recall (e.g., in traditional statistics and classification problems). However, here the situation is slightly different, what we really are interested in is to assess the quality of our explanations by measuring “how many of the top $n$ (by lineage similarity) tuples are actually part of the exact
lineage?”. By contrast, traditional recall does not seem to be a meaningful metric in our case, as many query-result tuples might have long histories. Hence, a top-N justifications result is preferred over “returning all the correct lineage tuples”, and this is what we measure.

7.1.3 Distant Lineage: Per-Level Recall

In order to assess the quality of a distant lineage answer in a more insightful way, we save the “exact distant lineage” in a hierarchical structure, and analyze it per-level. An implementation is a list $L$ of sets per-tuple, s.t. $L[i]$ is the set of all lineage tuples at the $i^{th}$ derivation depth level. Note that $i$ starts at 0, s.t. the $0^{th}$ lineage level is a set containing the target tuple only. For example, if the direct lineage of $t_4$ and $t_5$ are the sets $\{t_1, t_2, t_3\}$ and $\{t_1\}$, respectively, and the direct lineage of a tuple $t_6$ is the set $\{t_3, t_4, t_5\}$ (note that $t_3$ appears in both the $1^{st}$ and the $2^{nd}$ lineage levels of $t_6$, in this example) then the hierarchical lineage DAG for the tuple $t_6$, and the hierarchical list structure that represents it, look as follows:

Here, each rectangle is an entry in the list $L$.

We expect our explanations to have a “natural ranking” property in terms of lineage levels. That is, we expect the similarity between a target tuple and tuples in its distant lineage to be inversely proportional to the distance between them in the hierarchical lineage DAG structure. Thus, we devise an explanation quality measure for the explanation of a single tuple $t$ and its lineage level $i$:

$$Recall(t, i) = \frac{|ApproxLineage(t, D(t, i)) \cap L_t[i]|}{|L_t[i]|}$$

where $ApproxLineage(t, D(t, i))$ is the set of the top $D(t, i)$ (by lineage similarity) tuples, returned as explanations (i.e., approximate lineage) for $t$ by our system. $D(t, i)$ is the number of unique returned tuples in the exact lineage of $t$ up until the $i^{th}$ level (including). Formally, it is defined as follows: $D(t, i) = |\bigcup_{j=1}^{i} L_t[j]|$. $L_t$ is the hierarchical list structure for $t$, as defined above, i.e., $L_t[i]$ is the set of all lineage tuples at the $i^{th}$ derivation depth level of $t$. 

39
Table 7.1: userId indicates the unique user’s identifier returned by the query. Lin. size is the size of the exact lineage for a result tuple (containing only tuples from the tags table). Precision$^{tuples}$ and Precision$^{cols}$ display our system’s precision for each tuple in the result, for both “lineage tuple vectors” and “lineage column vectors” (with query-specific column weighting), respectively. Precision computations are done for the top $n = |Lineage_{tags}(t)|$ tuples in the approximate lineage for each result tuple $t$.

Table 7.2: manufacturer is the name of the manufacturer returned from the query. Lin. size is the size of the exact lineage for a result tuple (containing only tuples from the nutrients table). Precision$^{tuples}$, Precision$^{cols}$ and Precision$^{cols+weighting}$ display our system’s precision for each tuple in the result, for “lineage tuple vectors”, “lineage column vectors” with simple averaging, and “lineage column vectors” with query-induced column weighting, respectively. Precision computations are done for the top $n = |Lineage_{nutrients}(t)^2|$ tuples in the approximate lineage for each result tuple $t$.

7.2 Examples

Having established an explanation quality measure, we conducted a series of experiments. In particular, we tested the precision of direct provenance of different queries, varying from “simple” (examples 3.1 and 4.1) to “complex” (example 4.2) queries. We show some of the experimental results and present a qualitative analysis.

Note. In all the examples, the lineage (and its size - ‘Lin. size’ column) of a tuple, is calculated as a subset of the exact lineage. That is, if we compare a query result tuple with products.tuples, i.e., all the tuples in the products table (in order to find the top $N$ most similar tuples) - we use only product tuples in the exact lineage for precision calculations ($Lineage_{products}(t) = ExactLineage(t) \cap products.tuples$).

Running Example 3. We test queries on a small subset of the MovieLens dataset. Each table (i.e., movies, ratings, tags) consists of about 10,000 tuples.
Table 7.3: A comparison of different approximate lineage computation methods on a single query-result tuple (*general mills sales*). Lin. size is the size of the exact lineage for a result tuple (containing only tuples from the *serving_size* table). \( p^{\text{Top}} \) shows the precision of various methods for the top \( p \cdot |\text{Lineage}| \) tuples in the approximate lineage. For example, \( 0.50^{\text{Top}} \) is the precision for the top 983 (\( = 0.5 \cdot 1996 \)) tuples in the approximate lineage.

**Example 3.1.** We ask for distinct users that tagged a Comedy/Romance genre movie as one of funny, superhero, family, music. The results are shown in Table 7.1. Precision calculation is discussed in section 7.1. The approximate lineage is calculated from comparing result tuples to tuples from the *tags* table (see preceding Note). Right away, it is evident that provenance column vectors produce better approximate lineages than tuple vectors.

In this example tuples No.0 and No.6 display superior precision for column vectors. Tuple No.7 is interesting as it is a case of lineage size > 1 for which both methods (tuple and column vectors) produce an accuracy of 1.0. It is interesting to examine tuple No.7 more closely to gain a better grasp of the differences between tuple and column vectors. Now, let us analyze the top 3 (by similarity) tuples from the *tags* table, which are returned by our system for tuple No.7 in Table 7.1 (userId=282405, Lin. size=2) for both the tuple-based and column vectors-based methods:

### Tuple Vectors:

<table>
<thead>
<tr>
<th>userId</th>
<th>movieId</th>
<th>tag</th>
<th>Similarity</th>
<th>Lineage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>282405</td>
<td>152081 (Zootopia)</td>
<td>funny</td>
<td>0.9</td>
</tr>
<tr>
<td>1</td>
<td>282405</td>
<td>164909 (La La Land)</td>
<td>music</td>
<td>0.88</td>
</tr>
<tr>
<td>2</td>
<td>282405</td>
<td>152081 (Zootopia)</td>
<td>acceptance</td>
<td>0.86</td>
</tr>
</tbody>
</table>

### Column Vectors:

<table>
<thead>
<tr>
<th>userId</th>
<th>movieId</th>
<th>tag</th>
<th>Similarity</th>
<th>Lineage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>282405</td>
<td>164909 (La La Land)</td>
<td>music</td>
<td>0.95</td>
</tr>
<tr>
<td>1</td>
<td>282405</td>
<td>152081 (Zootopia)</td>
<td>funny</td>
<td>0.94</td>
</tr>
<tr>
<td>2</td>
<td>282405</td>
<td>177765 (Coco)</td>
<td>music</td>
<td>0.89</td>
</tr>
</tbody>
</table>
As shown in the two tables above, both methods give maximal similarity (top 2) for the correct tuples in the exact lineage. However, the “column vectors” method produces better separation ($0.94 - 0.89 = 0.05$ vs. $0.88 - 0.86 = 0.02$) in terms of similarity between correct and incorrect lineage tuples. Another interesting point of distinction between the two methods is the $3^{rd}$ most similar tuple returned:

- The “tuple vectors” method returns a tuple which describes a tag of a correct movie (Zootopia) by the correct user (282405) with an incorrect label - acceptance.
- The “column vectors” method returns a tuple which describes a tag of an incorrect movie (Coco - not a Romance/Comedy movie) by the correct user (282405) with a correct label - music.

The reason for this dissimilarity in behavior might be the result of giving the tag column more weight when comparing lineage column vectors (as discussed in section 6.1).

**Running Example 4.** We test queries on a small subset of the BFPDB dataset. Each table (products, nutrients, serving_size) consists of about 10,000 tuples.

**Example 4.1.** We ask for distinct manufacturers of products that have cholesterol value information in the nutrients table. Let us first look at the query:

```sql
SELECT DISTINCT manufacturer
FROM products p, nutrients n
WHERE p.ndb_no = n.ndb_no
AND n.nutrient_name = 'cholesterol'
```

Now, by parsing the query we conclude that the columns manufacturer, ndb_no, nutrient_name are the “columns of interest” (discussed in section 6.1). Thus, when comparing lineage column vectors of a query result tuple with another tuple from the DB - we give the columns of interest more weight. The results (and comparisons between the different techniques) are shown in Table 7.2. The approximate lineage is calculated from comparing result tuples to tuples from the nutrients table. As evident from the presented results, incorporating query-based column weighting can significantly improve the precision of our system.

**Note.** This example displays lineage querying for result tuples with more than just a few tuples in their exact lineage (i.e., up to two hundreds).

**Example 4.2.** We ask for distinct manufacturers of products that have reeses in their name and contain protein nutrient information. Also, these manufacturers must produce unprepared products (this information is stored in the serving_size table). The only manufacturer in the BFPDB dataset that satisfies the (albeit a complex one) query is general mills sales. Moreover, its exact lineage size (obtained by ProvSQL) is a relatively big number - 1966; hence, it makes an interesting case study for the
existence of drift errors in the construction of lineage vectors. That is, there is a potential for noise accumulation when combining a large number of lineage vectors with $+$, $\cdot$ and clustering operations (as discussed in section 4.1). The results are shown in Table 7.3. The approximate lineage is calculated from comparing result tuples to tuples from the serving_size table. As evident from the results, our solution (with query-based weighted column vectors and Bloom filters of queries) does not suffer from drift errors (limiting drift errors for distant provenance and direct provenance at scale are the subjects of ongoing research).

**Notes.**

- The query-result tuple for *general mills sales* has a lineage of size 1966, which is $\sim20\%$ of the tuples in the serving_size table in our experiments (in a production setting, it can be orders of magnitude larger). Instead, we approximate the lineage with a small (and bounded) number of vectors. We argue that if lineage is huge our method makes more sense for direct provenance calculation than exact lineage calculations.

- In this example the query accesses three tables in a non-trivial way, which suggests the way more complex queries would behave.

### 7.3 Advanced Experiments - The BFPDB Dataset

In this section we present results of our lineage tracking system over the USDA BFPDB dataset (see section 7.1.1) on more elaborate and complex test scenarios. In particular, we focus on tuples with multiple generations in their lifelong lineage history and analyze them in terms of direct and distant lineage. We assess the performance of our system quantitatively using precision and per-level recall (see sections 7.1.2 and 7.1.3, respectively) and qualitatively by observing the “top-$K$” returned lineage tuples (by similarity).

**Notes.**

- Bloom-Filters of queries are not applicable to the querying of distant lineage, as they look for tuples that were not directly involved in the evaluation of a query. Hence, they are not used in the following experiments. Extending this basic method to handle indirect lineage is the subject of current research.

- Suggested improvements (chapter 6) such as query-dependent column weighting, tuple creation timestamp and weighting with query dependency DAG are implicitly implemented (in the following experiments) for both the Tuple Vectors and the Column Vectors approximate lineage computation methods.

\[ \text{Lineage}_{\text{nutrients}}(t) = \text{ExactLineage}(t) \cap \text{nutrients.tuples} \]
7.3.1 A Hierarchy of Materialized Views

We want to simulate a DBMS that contains a significant portion of tuples that depend on the contents of the DB, as a platform for testing and analyzing distant lineage. Thus, we built a hierarchy of “materialized views”, such that each “view” is a table, constructed from tuples that were generated by a single query. See Figure 7.1 for a diagram, depicting the hierarchy of our materialized views.

![Materialized Views Hierarchy Diagram](image)

Figure 7.1: Materialized Views Hierarchy - The BFPDB Dataset.

An edge connecting two tables $T_2 \rightarrow T_1$ means that there are tuples in $T_2$ that were directly created from tuples in $T_1$, by a query. For example, the materialized view $\text{exp}_4$ was directly created from tuples from the tables $\text{exp}_2$ and $\text{readytodrink}$. The text colors are intended as a visual aid, such that each lineage generation has a brighter color, the farther it is from the base relations.

As stated earlier, we consider those tables that do not depend on the contents of the DB when these tables are created, and thereafter, as base tables. Again, note that tuples can be manually inserted to and deleted from base tables, but, not in a programmatic manner over the DB, e.g., a SQL `INSERT` statement that is based on the DB contents. In our case, the base tables are: $\text{products}$, $\text{nutrients}$ and $\text{serving_size}$ (see section 7.1.1 for more details). Note that each materialized view depends directly on tuples from the base tables (as defined above) or on tuples from other previously constructed materialized views.
7.3.2 Experiments

**Running Experiment 1.** We test queries on a subset of the BFPDB dataset, that consists of all the tables and materialized views depicted in Figure 7.1. In the experiments we mimic an analyst’s interaction with the data more precisely by comparing the approximate lineage vectors of a target tuple with a “heterogeneous” group of tuples (e.g., all related base tables or all related materialized views) and ranking all the tuples among the group according to similarity scores, as opposed to dealing with one table at a time (as was previously shown during the initial experimentation of section 7.2).

**Experiment 1.1.** We ask for distinct manufacturers of products that have water, sugar and salt as ingredients, and contain protein value information in the nutrients table; also, these manufacturers must produce prepared and unprepared products (this information is stored in the serving_size table):³

```sql
SELECT p.manufacturer
FROM exp3, products p
WHERE exp3.manufacturer = p.manufacturer
AND POSITION('salt' IN p.ingredients) > 0
GROUP BY p.manufacturer
```

Table 7.4 presents lineage related statistics, collected when computing the approximate lineage of the single result tuple in the output of this query (red gold) against all the tuples from the related base tables: products, nutrients and serving_size, which contain 29,322 tuples in total. Tables 7.5 and 7.6 present the top 20 tuples from the related base tables, ranked by similarity, using the CV and TV methods, respectively. We make some observations by analyzing the results:

1. The lineage sizes of the different levels (1-3) sum up to more than the total distant lineage size (63 + 0 + 160 > 160). In this case, it means that all 63 tuples from the 1st lineage level (which contains tuples from the products relation) are also a part of the 3rd lineage level (which contains tuples from the products, nutrients and serving_size relations). From analyzing the query we may conclude that these are the 63 tuples from the products table that affect the query result directly (via the FROM clause) and indirectly (via tuples from exp3, see a path from exp3 to products in Figure 7.1).

2. The Column Vectors (CV) method demonstrates overall superiority, compared to the Tuple Vectors (TV) method. Moreover, the CV method has a precision of 0.9 for the top 0.75 \cdot 160 = 120 tuples in the approximate lineage, and 0.74 for the top 160 tuples. That is, most of the errors are produced for the lower ranked tuples (ranked 121, ..., 160).

³For a detailed description of exp3 see section D.9 in the appendix.
<table>
<thead>
<tr>
<th>Provenance Method</th>
<th>Tuple Vectors</th>
<th>Column Vectors</th>
<th>Target Tuple</th>
<th>Lineage Size</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>160 63 0 160</td>
<td>160 63 0 160</td>
<td>0.75 Top</td>
<td>0.75 L[3]</td>
<td>L[1] 0.31</td>
<td>0.6</td>
</tr>
<tr>
<td>1</td>
<td>160 63 0 160</td>
<td>160 63 0 160</td>
<td>0.75 Top</td>
<td>0.75 L[3]</td>
<td>L[1] 0.78</td>
<td>0.74</td>
</tr>
</tbody>
</table>

Table 7.4: Experiment 1.1 red gold vs. related base tables (products, nutrients and serving size). A comparison of different approximate lineage computation methods on a single query-result tuple (red gold). Total Lineage Size is the size of the exact distant lineage for a result tuple (containing only tuples from the related base tables). Similarly, L[i] Lineage Size is the size of the i-th lineage level. \( p^{Top} \) Precision shows the precision of various methods for the top \( p \cdot |TotalDistantLineage| \) tuples in the approximate lineage. For example, 0.50 \( p^{Top} \) is the precision for the top 80 (\( = 0.5 \cdot 160 \)) tuples in the approximate lineage. L[i] Recall shows the i-th lineage level recall as described in section 7.1.3.

3. The CV method exhibits relatively high L[1] and L[3] recall scores (see Table 7.4), evidenced by observing the Lineage Level(s) column in Table 7.5. That is, not only that all the top-20 tuples by similarity are really a part of the distant lineage, but also, the ranking preserves a level-based bias (most of the top-20 are in the 1-st lineage level). By contrast, the TV method is less impressive on this front, as is evidenced by the No results in Table 7.6.

4. Note that a random choice of the top 160 lineage tuples would have yielded a \( \frac{160}{29,322} \approx 0.006 \) precision score, which is several orders of magnitude worse than the scores of the CV and TV methods.

Table 7.7 presents lineage related statistics, collected when computing the approximate lineage of the single result tuple in the output of this query (red gold) against all the tuples from the related materialized views: exp3, protein, prepared and unprepared, which contain 10,295 tuples in total. Tables 7.8 and 7.9 present subsets of the top 90 and 100 (respectively) tuples from the related materialized views, ranked by similarity, using the CV and TV methods, respectively. We make some observations by analyzing the results:

1. Both the CV and TV methods exhibit impressive outcomes, in terms of total precision and per-level recall, although the CV method performs better. That is, out of the top 85 tuples, only two are mistakenly ranked as lineage tuples (with a total of 83 tuples actually in the exact distant lineage), as can be seen in Table 7.8.

2. The top rated tuple, in both methods, is the only tuple in the 1-st lineage level (from exp3). This somewhat supports our claim of level-based bias.

3. The CV method showcases a significant separation between the similarity scores of the lowest ranked “correct” lineage tuples (up to the 85-th tuple, including) and the next non-lineage one (0.78 > 0.67), see Table 7.8. On the other hand, the
An Analyst’s View

<table>
<thead>
<tr>
<th>Tuple ID</th>
<th>Origin Table</th>
<th>Similarity</th>
<th>Lineage Level(s)</th>
<th>Lineage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id1</td>
<td>nutrients</td>
<td>0.86</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>id2</td>
<td>products</td>
<td>0.86</td>
<td>1, 3</td>
</tr>
<tr>
<td>2</td>
<td>id3</td>
<td>products</td>
<td>0.86</td>
<td>1, 3</td>
</tr>
<tr>
<td>3</td>
<td>id4</td>
<td>products</td>
<td>0.86</td>
<td>1, 3</td>
</tr>
<tr>
<td>4</td>
<td>id5</td>
<td>products</td>
<td>0.86</td>
<td>1, 3</td>
</tr>
<tr>
<td>5</td>
<td>id6</td>
<td>products</td>
<td>0.86</td>
<td>1, 3</td>
</tr>
<tr>
<td>6</td>
<td>id7</td>
<td>products</td>
<td>0.86</td>
<td>1, 3</td>
</tr>
<tr>
<td>7</td>
<td>id8</td>
<td>products</td>
<td>0.86</td>
<td>1, 3</td>
</tr>
<tr>
<td>8</td>
<td>id9</td>
<td>products</td>
<td>0.85</td>
<td>1, 3</td>
</tr>
<tr>
<td>9</td>
<td>id10</td>
<td>products</td>
<td>0.85</td>
<td>1, 3</td>
</tr>
<tr>
<td>10</td>
<td>id11</td>
<td>products</td>
<td>0.85</td>
<td>1, 3</td>
</tr>
<tr>
<td>11</td>
<td>id12</td>
<td>nutrients</td>
<td>0.85</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>id13</td>
<td>products</td>
<td>0.85</td>
<td>1, 3</td>
</tr>
<tr>
<td>13</td>
<td>id14</td>
<td>products</td>
<td>0.85</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>id15</td>
<td>products</td>
<td>0.85</td>
<td>1, 3</td>
</tr>
<tr>
<td>15</td>
<td>id16</td>
<td>products</td>
<td>0.85</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>id17</td>
<td>products</td>
<td>0.85</td>
<td>1, 3</td>
</tr>
<tr>
<td>17</td>
<td>id18</td>
<td>products</td>
<td>0.85</td>
<td>1, 3</td>
</tr>
<tr>
<td>18</td>
<td>id19</td>
<td>products</td>
<td>0.85</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>id20</td>
<td>products</td>
<td>0.85</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 7.5: Experiment 1.1 red gold vs. related base tables with Column Vectors - an analyst’s view. Top-20 tuples from the related base tables, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Column Vectors method.

TV method has no evident separation at all, as can be observed in Table 7.9 (see tuples 81-100).

Experiment 1.2. We ask for distinct pairs of (manufacturer, name) for products that are listed in the exp4 materialized view:

```sql
SELECT p.manufacturer, exp4.name
FROM exp4, products p
WHERE exp4.name = p.name
GROUP BY p.manufacturer, exp4.name
```

Table 7.10 presents lineage related statistics, collected when computing the approximate lineage of the (two) result tuples in the output of this query ((campbell soup company, v8 beverage carrot mango), (campbell soup company, v8 vfusion beverage peach)

---

4 For a detailed description of exp4 see section D.10 in the appendix.
Table 7.6: Experiment 1.1 red gold vs. related base tables with Tuple Vectors - an analyst’s view. Top-20 tuples from the related base tables, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Tuple Vectors method.

<table>
<thead>
<tr>
<th>Tuple ID</th>
<th>Origin Table</th>
<th>Similarity</th>
<th>Lineage Level(s)</th>
<th>Lineage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id(_1)</td>
<td>products</td>
<td>0.78</td>
<td>1, 3</td>
</tr>
<tr>
<td>1</td>
<td>id(_2)</td>
<td>products</td>
<td>0.78</td>
<td>1, 3</td>
</tr>
<tr>
<td>2</td>
<td>id(_3)</td>
<td>products</td>
<td>0.78</td>
<td>1, 3</td>
</tr>
<tr>
<td>3</td>
<td>id(_4)</td>
<td>products</td>
<td>0.77</td>
<td>1, 3</td>
</tr>
<tr>
<td>4</td>
<td>id(_5)</td>
<td>products</td>
<td>0.77</td>
<td>1, 3</td>
</tr>
<tr>
<td>5</td>
<td>id(_6)</td>
<td>serving_size</td>
<td>0.74</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>id(_7)</td>
<td>products</td>
<td>0.74</td>
<td>1, 3</td>
</tr>
<tr>
<td>7</td>
<td>id(_8)</td>
<td>products</td>
<td>0.74</td>
<td>1, 3</td>
</tr>
<tr>
<td>8</td>
<td>id(_9)</td>
<td>products</td>
<td>0.74</td>
<td>1, 3</td>
</tr>
<tr>
<td>9</td>
<td>id(_10)</td>
<td>serving_size</td>
<td>0.74</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>id(_11)</td>
<td>nutrients</td>
<td>0.73</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>id(_12)</td>
<td>serving_size</td>
<td>0.73</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>id(_13)</td>
<td>serving_size</td>
<td>0.73</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>id(_14)</td>
<td>nutrients</td>
<td>0.73</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>id(_15)</td>
<td>serving_size</td>
<td>0.73</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>id(_16)</td>
<td>products</td>
<td>0.73</td>
<td>1, 3</td>
</tr>
<tr>
<td>16</td>
<td>id(_17)</td>
<td>nutrients</td>
<td>0.72</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>id(_18)</td>
<td>nutrients</td>
<td>0.72</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>id(_19)</td>
<td>serving_size</td>
<td>0.72</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>id(_20)</td>
<td>nutrients</td>
<td>0.72</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.7: Experiment 1.1 red gold vs. materialized views (exp\(_3\), protein, prepared and unprepared). A comparison of different approximate lineage computation methods on a single query-result tuple (red gold). Total Lineage Size is the size of the exact distant lineage for a result tuple (containing only tuples from the related materialized views). Similarly, L\([i]\) is the size of the \(i\)th lineage level. \(p^{Top}\) Precision shows the precision of various methods for the top \(p \cdot|TotalDistantLineage|\) tuples in the approximate lineage. For example, 0.50\^{Top} is the precision for the top 41 (\(= \lfloor 0.5 \times 83 \rfloor\)) tuples in the approximate lineage. L\([i]\) Recall shows the \(i\)th lineage level recall as described in section 7.1.3.

<table>
<thead>
<tr>
<th>Provenance Method</th>
<th>Lineage Size</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple Vectors</td>
<td>Total L[1]</td>
<td>1.00^{Top}</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td>L[2]</td>
<td>0.75^{Top}</td>
<td>0.87</td>
</tr>
<tr>
<td></td>
<td>L[3]</td>
<td>0.50^{Top}</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>L[4]</td>
<td>0.25^{Top}</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L[1]</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Technion - Computer Science Department - M.Sc. Thesis MSC-2021-31 - 2021
Table 7.8: Experiment 1.1 red gold vs. related materialized views with Column Vectors - an analyst’s view. A subset of the top-90 tuples from the related materialized views, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Column Vectors method.
Table 7.9: Experiment 1.1 red gold vs. related materialized views with Tuple Vectors - an analyst’s view. A subset of the top-100 tuples from the related materialized views, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Tuple Vectors method.
mango)) against all the tuples from the related base tables: products, nutrients and serving_size, which contain 29,322 tuples in total. Tables 7.11 and 7.12 present the top 20 tuples from the related base tables, ranked by similarity to result tuple No.0, using the CV and TV methods, respectively. We make some observations by analyzing the results:

1. The lineage sizes (of both result tuples) of the different levels (1-4) sum up to more than the total distant lineage sizes. E.g., looking at result tuple No.0: 2 + 0 + 4 + 80 > 82. In this case, it means that at least two of the four tuples from the 3rd lineage level (which contains tuples from the products and serving_size relations, via exp_4 → readytodrink, see Figure 7.1) are also a part of either the 1st lineage level (which contains tuples from the products relation, via the FROM clause), or the 4th lineage level (which contains tuples from the products, nutrients and serving_size relations).

2. The Column Vectors (CV) method demonstrates overall superiority, compared to the Tuple Vectors (TV) method, topping at a precision score of 0.81 for the top 82 (total lineage size) approximate lineage tuples, for both result tuples. Moreover, the TV method demonstrates unusually low (≈ 0.2) precision scores for this query.

3. The TV method exhibits perfect L[1] and L[3] recall scores, and a low L[4] recall score, for both result tuples (see Table 7.10). We conclude that the TV method is highly effective in finding the 1st and 3rd lineage-level tuples, but performs poorly in finding the 4th lineage-level tuples, as is evidenced by the No results in Table 7.12.

4. The CV method exhibits relatively high L[1] and L[4] recall scores and a somewhat mediocre L[3] recall score for both result tuples (see Table 7.10), evidenced by observing the Lineage Level(s) column in Table 7.11. That is, not only that almost all the top-20 tuples by similarity are really a part of the distant lineage, but also, the ranking preserves a level-based bias (the top-2 tuples are the only ones that appear both in the 1st, 3rd and 4th lineage levels).

5. Analyzing the Tables 7.11 and 7.12, we see that the two 3rd lineage-level tuples from the serving_size relation are discovered significantly earlier in the ranking by the TV method (3rd and 4th tuples, by similarity) in comparison with the CV method (80th and 83rd tuples, by similarity). This means that the TV method does a better job on the 3rd lineage level, in terms of recall. This observation is backed up by the L[3] recall results we see in Table 7.10.

6. Notice the 4th ranked tuple (by similarity) in Table 7.11, which is a No lineage tuple, with a relatively high similarity score. A closer look reveals that this tuple is a product named v8 splash beverage mango peach, which is also produced by campbell soup company. Interestingly, it also appears as the first No tuple when
Tables 7.10: Experiment 1.2 query results vs. related base tables (products, nutrients and serving size). A comparison of different approximate lineage computation methods on the query-result tuples. Total Lineage Size is the size of the exact distant lineage for a result tuple (containing only tuples from the related base tables). Similarly, L[$i$] Lineage Size is the size of the $i^{th}$ lineage level. $p^{Top}$ Precision shows the precision of various methods for the top $p \cdot |TotalDistantLineage|$ tuples in the approximate lineage. For example, $0.50^{Top}$ is the precision for the top 41 ($= 0.5 \cdot 82$) tuples in the approximate lineage. L[$i$] Recall shows the $i^{th}$ lineage level recall as described in section 7.1.3.

analyzing result tuple No.1 vs. related base tables with the CV method (we do not show this explicitly).

7. Note that a random choice of the top 82 lineage tuples would have yielded a $\frac{82}{29322} \approx 0.003$ precision score, which is several orders of magnitude worse than the scores of the CV (and even the TV) methods.

Table 7.13 presents lineage related statistics, collected when computing the approximate lineage of the (two) result tuples in the output of this query ((campbell soup company, v8 beverage carrot mango), (campbell soup company, v8 vfusion beverage peach mango)) against all the tuples from the related materialized views: exp$_4$, exp$_2$, protein, unprepared and readytodrink, which contain 9,658 tuples in total. Tables 7.14 and 7.15 present the top 20 tuples from the related materialized views, ranked by similarity to result tuple No.0, using the CV and TV methods, respectively. We make some observations by analyzing the results:

1. The CV method demonstrates overall superiority, compared to the TV method, in terms of total precision and per-level recall. For example, for result tuple No.0, out of the top 44 tuples, only four are mistakenly ranked as lineage tuples (with a total of 40 tuples actually in the exact distant lineage), as can be partially seen in Table 7.14.

2. The top rated tuple in the CV method (Table 7.14), is the only tuple in the 1$^{st}$ lineage level (from exp$_4$). This somewhat supports our claim of level-based bias (the L[2] and L[3] recall scores are relatively high as well).

3. By contrast, for result tuple No.0, the TV method ranks the only 1$^{st}$ lineage-level tuple from exp$_4$ in 3$^{rd}$ place, which is evidenced by the $0^5$ L[1] score in Table 7.13 and can be seen in detail in Table 7.15.

$^{5}$Following the definitions in section 7.1.3:

Recall($t_0, 1$) = $\frac{|ApproxLineage_tD(t_0, 1) \cap L_{t_0}[1]|}{|L_{t_0}[1]|}$ = $\frac{|ApproxLineage_tD(t_0, 1) \cap L_{t_0}[1]|}{|L_{t_0}[1]|}$ =
4. Both the TV and CV methods rank the other, non-lineage tuple from \( \text{exp}_4 \), in the top 20, without a significant similarity score separation between the correct and the incorrect tuples from \( \text{exp}_4 \) (0.99 vs. 0.94 for the CV method, and 0.96 vs. 0.9 for the TV method). This suggests a relatively high similarity in terms of lineage, between the two tuples in the \( \text{exp}_4 \) materialized view (from transitivity).

**Experiment 1.3.** We ask for distinct pairs of \((\text{manufacturer}, \text{name})\) for products that contain *rice* in their name, and are produced by manufacturers of products that have *water*, *sugar* and *salt* as ingredients, and contain *protein* value information in the *nutrients* table; also, these manufacturers must produce *prepared* and *unprepared* products (this information is stored in the *serving_size* table):

```sql
SELECT p.manufacturer, t.name
FROM (SELECT DISTINCT sgr.name
      FROM exp2, sugars sgr
      WHERE exp2.manufacturer = sgr.manufacturer
      AND POSITION('rice' IN sgr.name) > 0) t, products p
WHERE t.name = p.name
GROUP BY p.manufacturer, t.name
```

Table 7.16 presents lineage related statistics, collected when computing the approximate lineage of four (out of nine, for brevity) of the result tuples in the output of this query against all the tuples from the related base tables: *products*, *nutrients* and *serving_size*, which contain 29,322 tuples in total. Tables 7.17 and 7.18 present subsets of the top 100 and 20 (respectively) tuples from the related base tables, ranked by similarity to result tuple No.2, using the CV and TV methods, respectively. We make some observations by analyzing the results:

1. This query presents a case where the size of the total distant lineage of result tuples is significant in terms of the input DB size. E.g., the total distant lineage size of result tuple No.2 is 4,000, while the related base tables contain a total of 29,322 tuples \((\approx 1 : 10)\).

2. The Column Vectors (CV) method demonstrates overall superiority, compared to the Tuple Vectors (TV) method. Moreover, analyzing result tuple No.2, the CV method has a precision of 0.97 for the top \(0.5 \cdot 4,000 = 2,000\) tuples in the approximate lineage, and 0.67 for the top 4,000 tuples. That is, most of the errors are produced for the lower ranked tuples (ranked 2001, ..., 4000). These numbers are shown explicitly in Table 7.17, that shows the 52\(^{nd}\) ranked tuple as the only non-lineage tuple in the top 100.

3. The TV method exhibits perfect \(L[1]\) and \(L[2]\) recall scores, and a low \(L[3]\) recall score, for all four result tuples (see Table 7.16). We conclude that the TV method

\[
\frac{|\text{ApproxLineage}(t_0,1) \cap L[t_0][1]|}{|\text{ApproxLineage}(t_0,1)|} = |\{t_{\text{protein}}\} \cap \{t_{\text{exp}}\}| = |\emptyset| = 0
\]

\(^6\)For a detailed description of \( \text{exp}_2 \) see section D.8 in the appendix.
### An Analyst’s View

<table>
<thead>
<tr>
<th>Tuple ID</th>
<th>Origin Table</th>
<th>Similarity</th>
<th>Lineage Level(s)</th>
<th>Lineage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id1 products</td>
<td>0.99</td>
<td>1, 3, 4</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>id2 products</td>
<td>0.99</td>
<td>1, 3, 4</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>id3 products</td>
<td>0.96</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>id4 products</td>
<td>0.95</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>id5 products</td>
<td>0.95</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>id6 products</td>
<td>0.95</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>id7 products</td>
<td>0.95</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>id8 nutrients</td>
<td>0.95</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>id9 products</td>
<td>0.95</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>id10 products</td>
<td>0.95</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>id11 products</td>
<td>0.95</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>id12 products</td>
<td>0.95</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>id13 products</td>
<td>0.95</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>13</td>
<td>id14 products</td>
<td>0.94</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>14</td>
<td>id15 products</td>
<td>0.94</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>15</td>
<td>id16 products</td>
<td>0.94</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>id17 products</td>
<td>0.94</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>17</td>
<td>id18 products</td>
<td>0.94</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>18</td>
<td>id19 products</td>
<td>0.94</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>19</td>
<td>id20 products</td>
<td>0.94</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>79</td>
<td>id80 serving_size</td>
<td>0.91</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>82</td>
<td>id83 serving_size</td>
<td>0.89</td>
<td>3</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 7.11: Experiment 1.2 result tuple No.0 vs. related **base tables** with **Column Vectors** - an analyst’s view. Top-20 tuples from the related base tables, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Column Vectors method.
Table 7.12: Experiment 1.2 result tuple No.0 vs. related base tables with Tuple Vectors - an analyst’s view. Top-20 tuples from the related base tables, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Tuple Vectors method.

Table 7.13: Experiment 1.2 query results vs. related materialized views (exp2, protein, unprepared and readytodrink). A comparison of different approximate lineage computation methods on the query-result tuples. Total Lineage Size is the size of the exact distant lineage for a result tuple (containing only tuples from the related materialized views). Similarly, L[i] Lineage Size is the size of the i\(^{th}\) lineage level. \(p^{Top}\) Precision shows the precision of various methods for the top \(p\cdot|TotalDistantLineage|\) tuples in the approximate lineage. For example, 0.50\(^{Top}\) is the precision for the top 21 ( = 0.5 \cdot 43) tuples in the approximate lineage. L[i] Recall shows the i\(^{th}\) lineage level recall as described in section 7.1.3.
<table>
<thead>
<tr>
<th>Tuple ID</th>
<th>Origin Table</th>
<th>Similarity</th>
<th>Lineage Level(s)</th>
<th>Lineage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(id_1)</td>
<td>(exp_4)</td>
<td>0.99</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>(id_2)</td>
<td>readytodrink</td>
<td>0.98</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>(id_3)</td>
<td>readytodrink</td>
<td>0.98</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>(id_4)</td>
<td>protein</td>
<td>0.97</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>(id_5)</td>
<td>protein</td>
<td>0.97</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>(id_6)</td>
<td>(exp_2)</td>
<td>0.97</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>(id_7)</td>
<td>protein</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>(id_8)</td>
<td>unprepared</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>(id_9)</td>
<td>protein</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>(id_{10})</td>
<td>protein</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>(id_{11})</td>
<td>protein</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>(id_{12})</td>
<td>protein</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>(id_{13})</td>
<td>protein</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>(id_{14})</td>
<td>protein</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>(id_{15})</td>
<td>protein</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>(id_{16})</td>
<td>protein</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>(id_{17})</td>
<td>(exp_4)</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>(id_{18})</td>
<td>protein</td>
<td>0.94</td>
<td>3</td>
</tr>
<tr>
<td>18</td>
<td>(id_{19})</td>
<td>protein</td>
<td>0.93</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>(id_{20})</td>
<td>protein</td>
<td>0.93</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 7.14: Experiment 1.2 result tuple No.0 vs. related **materialized views** with **Column Vectors** - an analyst’s view. Top-20 tuples from the related base tables, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Column Vectors method.
<table>
<thead>
<tr>
<th>Tuple ID</th>
<th>Origin Table</th>
<th>Similarity</th>
<th>Lineage Level(s)</th>
<th>Lineage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id₁</td>
<td>0.97</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>id₂</td>
<td>0.96</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>id₃</td>
<td>0.96</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>id₄</td>
<td>0.96</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>id₅</td>
<td>0.96</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>id₆</td>
<td>0.91</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>id₇</td>
<td>0.9</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>id₈</td>
<td>0.77</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>id₉</td>
<td>0.75</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>id₁₀</td>
<td>0.75</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>id₁₁</td>
<td>0.75</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>id₁₂</td>
<td>0.75</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>12</td>
<td>id₁₃</td>
<td>0.74</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>13</td>
<td>id₁₄</td>
<td>0.74</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>14</td>
<td>id₁₅</td>
<td>0.74</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>15</td>
<td>id₁₆</td>
<td>0.73</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>16</td>
<td>id₁₇</td>
<td>0.72</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>17</td>
<td>id₁₈</td>
<td>0.72</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>18</td>
<td>id₁₉</td>
<td>0.72</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>19</td>
<td>id₂₀</td>
<td>0.72</td>
<td>3</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 7.15: Experiment 1.2 result tuple No.0 vs. related materialized views with **Tuple Vectors** - an analyst’s view. Top-20 tuples from the related base tables, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Tuple Vectors method.
is highly effective in finding the 1st and 2nd lineage-level tuples, but performs poorly in finding the 3rd lineage-level tuples, as is evidenced by the No results in Table 7.18.

4. The CV method exhibits perfect L[1] recall scores and relatively high L[2] and L[3] recall scores (see Table 7.16). Analyzing result tuple No.2 and Table 7.17, the 0.5 L[2] recall score seems a little misleading, since the 2nd L[2] lineage-level tuple is not “lost”, but is ranked 3rd (instead of 2nd). This somewhat supports our claim of level-based bias.

5. Note that a random choice of the top 4,000 lineage tuples would have yielded a \( \frac{4,000}{322} \approx 0.136 \) precision score, which is comparable with the scores of the TV method, but, is significantly worse than the CV method scores.

Table 7.19 presents lineage related statistics, collected when computing the approximate lineage of four (out of nine, for brevity) of the result tuples in the output of this query against all the tuples from the related materialized views: exp2, protein, sugars and unprepared, which contain 10,298 tuples in total. Tables 7.20 and 7.21 present the top 20 tuples from the related materialized views, ranked by similarity to result tuple No.2, using the CV and TV methods, respectively. We make some observations by analyzing the results:

1. Once again, the CV method demonstrates overall superiority, compared to the TV method, in terms of total precision and per-level recall.

2. Analyzing result tuple No.2, the top rated tuple in both the CV and TV methods (see Tables 7.20 and 7.21, respectively) is the only sugars tuple in the 1st lineage level. The 2nd tuple in the 1st (L[1]) lineage level (from exp2) is “discovered” earlier (ranked 3rd vs. 5th) in the CV method, with no similarity score separation from the higher ranked tuples. By contrast, the TV method exhibits a significant
similarity score drop all across the top 20 ranked tuples (0.97 for 1st ranked vs. 0.81 for 5th ranked vs. 0.64 for 20th ranked). We view a more stable similarity score progression as indicating a better overall performance.

3. Analyzing Tables 7.19 and 7.20, it seems the numbers for result No.2 are not as good (compared to previous experiments and the result tuples No.0 and No.1 in this current experiment). A closer look reveals that out of the top 100 ranked tuples (by similarity) - all the mistakes are made on tuples from the sugars and protein materialized views. The No tuples from sugars and protein are mostly products that appear also in the unprepared materialized view, and are actually in the lineage (i.e., Yes tuples) of result tuple No.2. It seems that these No tuples from sugars and protein have the same lineage vectors per the product related columns as the respective Yes tuples from unprepared. Now, the Bloom Filters are not applicable here (since we are looking for distant lineage), and the Query Dependency DAG has no real filtering capabilities in this case (since the queries that created sugars and protein are relevant), hence we are not able to effectively get rid of these mistakes.
### Table 7.17: Experiment 1.3 result tuple No.2 vs. related base tables with Column Vectors - an analyst’s view

A subset of the Top-100 tuples from the related base tables, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Column Vectors method.

<table>
<thead>
<tr>
<th>Tuple ID</th>
<th>Origin Table</th>
<th>Similarity</th>
<th>Lineage Level(s)</th>
<th>Lineage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id₁</td>
<td>0.99</td>
<td>1, 2, 3</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>id₂</td>
<td>0.99</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>id₃</td>
<td>0.96</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>id₄</td>
<td>0.95</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>id₅</td>
<td>0.95</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>id₆</td>
<td>0.95</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>id₇</td>
<td>0.95</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>id₈</td>
<td>0.95</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>id₉</td>
<td>0.95</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>id₁₀</td>
<td>0.95</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>id₁₁</td>
<td>0.95</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>id₁₂</td>
<td>0.95</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>id₁₃</td>
<td>0.95</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>13</td>
<td>id₁₄</td>
<td>0.94</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>14</td>
<td>id₁₅</td>
<td>0.94</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>15</td>
<td>id₁₆</td>
<td>0.94</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>16</td>
<td>id₁₇</td>
<td>0.94</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>17</td>
<td>id₁₈</td>
<td>0.94</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>18</td>
<td>id₁₉</td>
<td>0.94</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>19</td>
<td>id₁₀₀</td>
<td>0.94</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>52</td>
<td>id₅₃</td>
<td>0.89</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>97</td>
<td>id₉₈</td>
<td>0.88</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>98</td>
<td>id₉₉</td>
<td>0.88</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>99</td>
<td>id₁₀₀</td>
<td>0.88</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
### An Analyst’s View

<table>
<thead>
<tr>
<th>Tuple ID</th>
<th>Origin Table</th>
<th>Similarity</th>
<th>Lineage Level(s)</th>
<th>Lineage</th>
</tr>
</thead>
</table>
| 0        | id
| 1        | nutrients    | 0.9        | 1, 2, 3          | Yes     |
| 2        | nutrients    | 0.87       | 2               | Yes     |
| 3        | nutrients    | 0.85       | No              |         |
| 4        | nutrients    | 0.85       | No              |         |
| 5        | nutrients    | 0.83       | No              |         |
| 6        | nutrients    | 0.83       | No              |         |
| 7        | nutrients    | 0.83       | No              |         |
| 8        | nutrients    | 0.83       | No              |         |
| 9        | nutrients    | 0.82       | No              |         |
| 10       | nutrients    | 0.82       | No              |         |
| 11       | nutrients    | 0.81       | No              |         |
| 12       | nutrients    | 0.81       | No              |         |
| 13       | serving_size | 0.81       | 3               | Yes     |
| 14       | nutrients    | 0.8        | No              |         |
| 15       | nutrients    | 0.79       | No              |         |
| 16       | nutrients    | 0.76       | No              |         |
| 17       | nutrients    | 0.75       | No              |         |
| 18       | nutrients    | 0.75       | No              |         |
| 19       | nutrients    | 0.74       | No              |         |

Table 7.18: Experiment 1.3 result tuple No.2 vs. related base tables with Tuple Vectors - an analyst’s view. Top-20 tuples from the related base tables, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Tuple Vectors method.

### Hindsight

<table>
<thead>
<tr>
<th>Target Tuple</th>
<th>Lineage Size</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>conagra brands healthy choice microwaveable chicken and rice soup</td>
<td>2.03</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>general mills sales inc annies homegrown organic vegan mac elbow rice pasta sauce</td>
<td>2.03</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>general mills sales inc marie callenders cheesy chicken and rice dinners</td>
<td>2.03</td>
<td>0.42</td>
<td>0.42</td>
</tr>
</tbody>
</table>

Table 7.19: Experiment 1.3 query results vs. related materialized views (exp2, protein, sugars and unprepared). A comparison of different approximate lineage computation methods on the query-result tuples. Total Lineage Size is the size of the exact distance lineage for a result tuple (containing only tuples from the related materialized views). Similarly, $L[i]$ Lineage Size is the size of the $i^{th}$ lineage level. $p^{Top}$ Precision shows the precision of various methods for the top $p \cdot |TotalDistantLineage|$ tuples in the approximate lineage. For example, $0.50^{Top}$ is the precision for the top 918 ($= 0.5 \times 1836$) tuples in the approximate lineage for the first result tuple. $L[i]$ Recall shows the $i^{th}$ lineage level recall as described in section 7.1.3.
<table>
<thead>
<tr>
<th>Tuple ID</th>
<th>Origin Table</th>
<th>Similarity</th>
<th>Lineage Level(s)</th>
<th>Lineage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id₁ sugars</td>
<td>0.98</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>id₂ protein</td>
<td>0.94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>id₃ exp₂</td>
<td>0.94</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>id₄ sugars</td>
<td>0.93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>id₅ unprepared</td>
<td>0.91</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>id₆ protein</td>
<td>0.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>id₇ sugars</td>
<td>0.89</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>id₈ protein</td>
<td>0.89</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>id₉ sugars</td>
<td>0.89</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>id₁₀ unprepared</td>
<td>0.89</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>id₁₁ sugars</td>
<td>0.89</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>id₁₂ protein</td>
<td>0.89</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>id₁₃ protein</td>
<td>0.88</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>13</td>
<td>id₁₄ protein</td>
<td>0.88</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>14</td>
<td>id₁₅ protein</td>
<td>0.88</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>15</td>
<td>id₁₆ protein</td>
<td>0.88</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>16</td>
<td>id₁₇ protein</td>
<td>0.88</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>17</td>
<td>id₁₈ protein</td>
<td>0.88</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>18</td>
<td>id₁₉ protein</td>
<td>0.88</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>19</td>
<td>id₂₀ protein</td>
<td>0.88</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.20: Experiment 1.3 result tuple No.2 vs. related materialized views with Column Vectors - an analyst’s view. Top-20 tuples from the related base tables, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Column Vectors method.
Table 7.21: Experiment 1.3 result tuple No.2 vs. related **materialized views** with **Tuple Vectors** - an analyst’s view. Top-20 tuples from the related base tables, ranked by similarity. Tuple ID contains unique ids of the tuples in question, so that an analyst can quickly retrieve them from the DB. Origin Table is the table name of the tuple’s origin. Lineage Level(s) and Lineage show additional meta-data, only known in hindsight (not available to an analyst during lineage querying) to showcase the effectiveness of the Tuple Vectors method.
Chapter 8

Conclusions and Future Directions

We briefly surveyed classic approaches to provenance implementation in DBMSs, and described a state-of-the-art “exact provenance tracking” system (ProvSQL). We then presented a novel approach, in which we approximate lineage (a specific type of provenance) using embedding techniques; drawing inspiration from word vectors in the NLP domain. Finally, we showed preliminary and advanced experimental results of our approximate lineage computation, and provided insights as to its performance when compared to the exact lineage obtained from the ProvSQL system.

The examples we presented suggest a high usefulness potential for our proposed approximate lineage methods and the further suggested enhancements. This especially holds for the Column Vectors method which exhibits high precision and per-level recall.

Our method implicitly produces a “natural ranking” of explanations. It is unclear how to get this kind of ranking from semiring provenance polynomials (it might require significant additional manual work). To this end, there is a work by Deutch et al. [DGM15] that ranks derivation trees for Datalog programs; the ranking is based on weights assigned to the deriving Datalog rules and tuples from the underlying database. In contrast, we deal with simpler SQL queries and we look at a much simpler notion: lineage, providing a succinct good-enough and useful one.

8.1 Word Embedding vs. The Field

Comparisons to other, possibly simple methods, is needed; yet, obvious ideas seem inferior to word embedding. For example, one could think of achieving the same functionality using simple data-type dependent featurization like numerical scaling and one-hot encoding. This “simpler approach” makes assumptions about the problem domain, e.g., the vocabulary. Thus, rendering the update of such vocabulary impractical, as opposed to word embeddings via Word2Vec that can be modified to support incremental training [BS]. Most importantly, word embeddings support similarity queries...
by grouping closely “semantically” similar words in a low dimensional space. This property lies at the heart of our lineage-approximation work, as it enables encoding lineage in a low-dimensional space with similar “grouping” properties. One-hot encoding can not help achieve any of these characteristics.

Next, we describe promising research directions which we plan to explore in the area of approximate provenance computation.

8.2 Long-term scenarios and analysis

This thesis presents a novel approach to approximating lineage and initial experimentation results. Thorough distant lineage experimentation and results are at the heart of ongoing research. Some interesting research issues that need be addressed and tested are:

- **DELETE** and **UPDATE** SQL statements. Clearly, deleted tuples may still be part of the lineage of DB tuples. Thus, deletion should be a “marking as deleted” rather than actual elimination. Update may be viewed as deletion followed by insertion. This may be a necessary implementation given that lineage vectors may need be associated with both pre- and post-update states of the DB.

- How well does the Bloom Filter extension work in scenarios where the overhead of exact lineage tracking approaches makes their usage impractical?

- How fast does the accuracy degrade as tuple history become more complex?

- Show a scenario where the performance of the exact approach makes it completely unusable but the approximate solution is able to produce useful outputs.

- Analyze the relationship between the complexity of tuple history and the hyper-parameter `max_vectors_num`.

- How does the overhead of increasing `max_vectors_num` compare to the overhead of Bloom Filter information with regard to final precision?

8.3 Partial Explanations

In this work, we aspire to approximate the exact lineage explanations by giving a top-\(N\) lineage answer, when \(N\) is a user (analyst) defined parameter. Another way to deal with the growing complexity of lineage answers is giving partial explanations. There are ways to generate **small partial explanations**:

- **Using ProvSQL** and keeping, say, just two (hopefully, typical) explanations. This implies that if we want to “dig deeper”, we need to work recursively.
• Doing the same with **vector-based methods**; the main advantage over ProvSQL: recursive drill through is done automatically.

• **NN Learned Explanations** - this is a different approach for generating a *single* explanation. In this approach, we try to *learn* a small explanation using a neural network. The input is a tuple \( t \) (fields and values) and the query (represented smartly [KKR+19, HS20]) and the output are tuple(s) (say, maximum one per relation), explaining \( t \) being part of the query result. This approach may be adapted to generate a small constant number of such explanations.
Appendix A

Efficient Set of Vectors Search

A.1 Rationale

In this work, we obtain a lineage via an embedding based method, such that each tuple/column (see chapters 4 and 5) is associated with a set of vectors. We want to adapt known approximate vector search algorithms (e.g., based on LSH [GIM99]) to efficiently implement our method (we define a “similarity between two sets of vectors” formula in section 3.3.3). That is, we need an efficient approximate search algorithm of nearest sets of lineage vectors. We then can utilize this algorithm in lineage querying for locating contributors, instead of comparing a target tuple explicitly with all of its lineage candidates and sorting them by similarity.

First, we present a basic method under the assumption that all the sets of lineage vectors in the DB are of a constant cardinality $N$. Then, we introduce a refined version that supports sets of vectors of varying cardinalities, such that each set of vectors $V$, which is associated with a tuple or a tuple column in the DB, has a cardinality $|V| \in [1, max\_vectors\_num]$ ($max\_vectors\_num$ is a hyperparameter of our system).

A.2 A Constant Cardinality of Sets of Vectors

Here, we assume that all the sets of lineage vectors in the DB are of a constant cardinality $N$. The idea is transforming each set of vectors into a single long vector, so that the dot product of two such long vectors computes all pair-wise interactions, and is equivalent to the similarity calculation (between sets of vectors) we devised in section 3.3.3. Note that all the lineage vectors in the DB have the same constant dimension $D$ (a hyperparameter of our system).
A.2.1 Implementation

Long Vectors Construction

1. Suppose a target tuple \( t_A \) is associated with a set \( A \) of lineage vectors of cardinality \( N \).

2. Let \( t_V \) be a candidate tuple for \( t_A \), that is associated with a set \( V \) of lineage vectors of cardinality \( N \). Construct a long candidate vector \( \bar{L}_V \in \mathbb{R}^{\left| A \right| \times \left| V \right| \times D} \) from \( V \) by concatenating \( \left| A \right| \) copies of each (normalized) \( \bar{v}_i \in V \), in order \( \bar{v}_1, \ldots, \bar{v}_{\left| V \right|} \): 

\[
\bar{L}_V = \begin{pmatrix}
\frac{\bar{v}_1}{\left\| \bar{v}_1 \right\|} \\
\vdots \\
\frac{\bar{v}_{\left| A \right|}}{\left\| \bar{v}_{\left| A \right|} \right\|} \\
\vdots \\
\frac{\bar{v}_{\left| V \right| 1}}{\left\| \bar{v}_{\left| V \right| 1} \right\|} \\
\vdots \\
\frac{\bar{v}_{\left| V \right| \left| A \right|}}{\left\| \bar{v}_{\left| V \right| \left| A \right|} \right\|}
\end{pmatrix}
\]

such that \( \bar{v}_{i,j} \) is the \( j^{th} \) copy of \( \bar{v}_i \).

3. Now, we build a long vector \( \bar{L}_A \in \mathbb{R}^{\left| A \right| \times \left| V \right| \times D} \) by concatenating all (normalized) \( \bar{a}_i \in A \), in order \( \bar{a}_1, \ldots, \bar{a}_{\left| A \right|} \), and duplicating the result \( \left| V \right| \) times: 

\[
\bar{L}_A = \begin{pmatrix}
\frac{\bar{a}_1}{\left\| \bar{a}_1 \right\|} \\
\vdots \\
\frac{\bar{a}_{\left| A \right|}}{\left\| \bar{a}_{\left| A \right|} \right\|} \\
\vdots \\
\frac{\bar{a}_{\left| V \right| 1}}{\left\| \bar{a}_{\left| V \right| 1} \right\|} \\
\vdots \\
\frac{\bar{a}_{\left| V \right| \left| A \right|}}{\left\| \bar{a}_{\left| V \right| \left| A \right|} \right\|}
\end{pmatrix}
\]

such that \( \bar{a}_{i,j} \) is the \( j^{th} \) copy of \( \bar{a}_i \).

4. \( \bar{L}_A \) is used to capture the average of all pair-wise cosine similarities with vectors from a candidate set \( V \) (note that \( a_{i,j} = a_i \) and \( v_{j,i} = v_j \)):

\[
\bar{L}_A \cdot \bar{L}_V = \sum_{i=1}^{\left| A \right|} \sum_{j=1}^{\left| V \right|} \frac{\bar{a}_{i,j}}{\left\| \bar{a}_i \right\|} \cdot \frac{\bar{v}_{j,i}}{\left\| \bar{v}_j \right\|} = \text{sum}(ps) = \text{avg}(ps) \times \left| A \right| \times \left| V \right| \times N^2
\]

where \( ps \) is the multi-set of pair-wise similarities between a pair of vectors, one taken from set \( V \) and one taken from set \( A \).

5. To capture the maximum of the pair-wise similarities (denoted \( \text{max}(ps) \)) we build
$|A| \times |V| = N^2$ long “selector” vectors $\vec{\sigma}_{1,1}, \ldots, \vec{\sigma}_{|A|,1}, \ldots, \vec{\sigma}_{1,|V|}, \ldots, \vec{\sigma}_{|A|,|V|}$, each $\vec{\sigma}_{i,j} \in \mathbb{R}^{|A| \times |V| \times D}$ “assumes” which of the $|A| \times |V| = N^2$ pair-wise interactions is maximal:

$$
\vec{\sigma}_{i,j} = \begin{pmatrix}
\vec{0}_{1,1} \\
\vdots \\
\vec{0}_{|A|,1} \\
\vdots \\
\vec{1}_{i,j} \\
\vdots \\
\vec{0}_{1,|V|} \\
\vdots \\
\vec{0}_{|A|,|V|}
\end{pmatrix}
$$

i.e., $\vec{\sigma}_{i,j}$ is a concatenation of $O_s = (j - 1) \times |A| + (i - 1)$ $\vec{0}$ vectors, followed by one $\vec{1}$ vector, and ending with $|A| \times |V| - (O_s + 1)$ $\vec{0}$ vectors, where $\vec{0} \in \mathbb{R}^D$ is the “all zeros” vector and $\vec{1} \in \mathbb{R}^D$ is the “all ones” vector. $\vec{\sigma}_{i,j}$ “assumes” the maximum occurs in the cosine similarity product between $\vec{a}_i$ and $\vec{v}_j$. Consequently, we get:

$$
\vec{\sigma}_{i,j} \odot \vec{L}_A = \begin{pmatrix}
\vec{0}_{1,1} \\
\vdots \\
\vec{0}_{|A|,1} \\
\vdots \\
\vec{a}_{i,j} \\
\vdots \\
\vec{0}_{1,|V|} \\
\vdots \\
\vec{0}_{|A|,|V|}
\end{pmatrix}
$$

such that $\vec{a}_{i,j}$ is the $j^{th}$ copy of $\vec{a}_i$ and $\odot$ is the Hadamard (i.e., element-wise) product. This results in:

$$
(\vec{\sigma}_{i,j} \odot \vec{L}_A) \cdot \vec{L}_V = \frac{\vec{a}_{i,j}}{\|\vec{a}_i\|} \cdot \frac{\vec{v}_{j,i}}{\|\vec{v}_j\|} = \frac{\vec{a}_i}{\|\vec{a}_i\|} \cdot \frac{\vec{v}_j}{\|\vec{v}_j\|}
$$

6. Next, we construct $|A| \times |V| = N^2$ long target vectors $\vec{\tau}_{1,1}, \ldots, \vec{\tau}_{|A|,1}, \ldots, \vec{\tau}_{1,|V|}, \ldots, \vec{\tau}_{|A|,|V|}$ ($\vec{\tau}_{i,j} \in \mathbb{R}^{|A| \times |V| \times D}$):

$$
\vec{\tau}_{i,j} = \frac{1}{w_{\text{max}} + w_{\text{avg}}} \cdot \left( w_{\text{max}} \cdot (\vec{\sigma}_{i,j} \odot \vec{L}_A) + w_{\text{avg}} \cdot \frac{|A| \times |V|}{|A| \times |V|} \cdot \vec{L}_A \right)
$$

where $w_{\text{max}}$ and $w_{\text{avg}}$ are (user-specified) hyperparameters, as specified in section 71.
3.3.3.

7. Each long target vector $\vec{\tau}_{i,j}$ computes the desired similarity calculation via a dot product with a long candidate vector $\vec{L}_V$, under the assumption that $\vec{a}_i$ and $\vec{v}_j$ have the maximal pair-wise similarity:

$$\vec{\tau}_{i,j} \cdot \vec{L}_V = \frac{w_{\text{max}} \cdot (\vec{a}_{i,j} \odot \vec{L}_A) \cdot \vec{L}_V + w_{\text{avg}}}{w_{\text{max}} + w_{\text{avg}}} =$$

$$= \frac{w_{\text{max}} \cdot \frac{\vec{a}_{i,j}}{||\vec{a}_i||} \cdot \frac{\vec{v}_j}{||\vec{v}_j||} + w_{\text{avg}} \cdot \text{avg}(ps)}{w_{\text{max}} + w_{\text{avg}}}$$

Search

1. Insert all the long candidate vectors (of a constant dimension $N^2 \times D$), associated with tuples or tuple columns in the DB, into a vector search structure $S$. Note that known approximate vector search techniques, such as LSH [GIM99], can be utilized here for efficiency.

2. Recall that for a tuple $t_V$ (or a column of a tuple $t_V$), that is associated with a set of vectors $V$, the long candidate vector is $\vec{L}_V$.

3. Now, suppose you are given a target tuple $t_A$, that is associated with a set $A$ of lineage vectors of cardinality $N$. Construct $N^2$ long target vectors $\vec{\tau}_{1,1}, \ldots, \vec{\tau}_{N,N}$.

4. We look separately for the closest (dot-product wise) long candidate vector $\vec{L}_{i,j}$ (and its respective candidate tuple $t_{i,j}$) in $S$, to each $\vec{\tau}_{i,j}$, respectively.

5. We compute the similarity scores $\vec{\tau}_{1,1} \cdot \vec{L}_{1,1}, \ldots, \vec{\tau}_{N,1} \cdot \vec{L}_{N,1}, \ldots, \vec{\tau}_{1,N} \cdot \vec{L}_{1,N}, \ldots, \vec{\tau}_{N,N} \cdot \vec{L}_{N,N}$. The one yielding the highest score, say $\vec{L}_{i,j}$, identifies the desired candidate tuple $t_{i,j}$, which is associated with a set $V_{i,j}$ of lineage vectors, according to our set-oriented similarity formula.

A.3 Varying Cardinalities of Sets of Vectors

In section A.2.1, we presented the construction of a long candidate vector $\vec{L}_V$ from a set of lineage vectors $V$, that is associated with a candidate tuple $t_V$, and a collection of long target vectors $\vec{\tau}_{i,j}$ (where $i \in [1, |A| = N]$ and $j \in [1, |V| = N]$) from a set of lineage vectors $A$, that is associated with a target tuple $t_A$, under the assumption that all the sets of lineage vectors in the DB are of a constant cardinality $N$. It is evident that these constructions are tightly coupled with the cardinalities of the sets of lineage vectors of the target and candidate tuples, namely $|A|$ and $|V|$. Hence, we introduce a refined version that supports sets of vectors of varying cardinalities, such that each set of vectors $V$, which is associated with a tuple or a tuple column in the DB, has a cardinality $|V| \in [1, \text{max\_vectors\_num}]$ ($\text{max\_vectors\_num}$ is hereafter denoted as...
$M$, for brevity). Recall that all the lineage vectors in the DB have the same constant dimension $D$.

The general idea is pre-computing $M$ long candidate vectors $\vec{L}_V^1, ..., \vec{L}_V^M$ instead of a single $\vec{L}_V$, and a collection of long target vectors $\vec{\tau}_{i,j}^k$, for each $k \in [1, M]$ (where $i \in [1, |A|]$ and $j \in [1, k]$). Each $\vec{L}_V^n \in \mathbb{R}^{n \times |V| \times D}$, where $n \in [1, M]$, “assumes” in its construction that the cardinality of the set of lineage vectors of a target tuple is $n$. Each $\vec{\tau}_{i,j}^k \in \mathbb{R}^{|A| \times k \times D}$, where $k \in [1, M]$, “assumes” in its construction that the cardinality of the set of lineage vectors of a candidate tuple is $k$. Consequently, instead of a single form of long candidate and target vectors (of dimension $N^2 \times D$, as is the case in section A.2.1), we potentially get $M \times M$ different such forms (in terms of dimension and $(n, k)$ construction parameters), such that each long vector of dimension $n \times k \times D$, where $n, k \in [1, M]$, is associated with a separate search structure $S_{1,1}, ..., S_{M,M}$, depending on its $(n, k)$ form (there are additional options, e.g., combining such search structures of equal dimensions).

A.3.1 Implementation

Long Vectors Construction

1. Let $t_V$ be a candidate tuple, that is associated with a set $V$ of lineage vectors of cardinality $|V| \in [1, M]$. Construct $M$ long candidate vectors $\vec{L}_V^1, ..., \vec{L}_V^M$, such that each $\vec{L}_V^n \in \mathbb{R}^{n \times |V| \times D}$, where $n \in [1, M]$ is constructed from $V$ by concatenating $n$ copies of each (normalized) $\vec{v}_i \in V$, in order $\vec{v}_1, ..., \vec{v}_{|V|}$:

$$\vec{L}_V^n = \begin{pmatrix} \vec{v}_{1,1} / \|\vec{v}_1\| \\ \vdots \\ \vec{v}_{1,n} / \|\vec{v}_1\| \\ \vec{v}_{2,1} / \|\vec{v}_2\| \\ \vdots \\ \vec{v}_{|V|,1} / \|\vec{v}_{|V|}\| \\ \vdots \\ \vec{v}_{|V|,n} / \|\vec{v}_{|V|}\| \end{pmatrix}$$

such that $\vec{v}_{i,j}$ is the $j^{th}$ copy of $\vec{v}_i$. $\vec{L}_V^n$ “assumes” the cardinality of the set of lineage vectors of the target tuple is $n$.

2. Suppose a target tuple $t_A$ is associated with a set $A$ of lineage vectors of cardinality $|A| \in [1, M]$.

3. Now, we build $M$ long vectors $\vec{L}_A^1, ..., \vec{L}_A^M$, such that $\vec{L}_A^k \in \mathbb{R}^{|A| \times k \times D}$, where $k \in [1, M]$, is built by concatenating all (normalized) $\vec{a}_i \in A$, in order $\vec{a}_1, ..., \vec{a}_{|A|}$, and duplicating the result $k$ times:
\[
\vec{L}_A^k = \begin{pmatrix}
\vec{a}_{i,1}/\|\vec{a}_i\| \\
\vdots \\
\vec{a}_{|A|,1}/\|\vec{a}_{|A|}\| \\
\vdots \\
\vec{a}_{i,k}/\|\vec{a}_i\| \\
\vdots \\
\vec{a}_{|A|,k}/\|\vec{a}_{|A|}\|
\end{pmatrix}
\]

such that \(\vec{a}_{i,j}\) is the \(j^{th}\) copy of \(\vec{a}_i\). \(\vec{L}_A^k\) “assumes” the cardinality of the set of lineage vectors of the candidate tuple is \(k\).

4. Let \(t_V\) be a candidate tuple for \(t_A\), that is associated with a set \(V\) of lineage vectors of cardinality \(|V| \in [1, M]\). Long vector \(\vec{L}_V^{k=|V|} \in \mathbb{R}^{|A| \times |V| \times D}\) is used to capture the average of all pair-wise cosine similarities with vectors from \(V\) (\(\vec{L}_V^{n=|A|} \in \mathbb{R}^{|A| \times |V| \times D}\)):

\[
\vec{E}_A^{k=|V|} \cdot \vec{L}_V^{n=|A|} = \vec{E}_A^{|V|} \cdot \vec{L}_V^{|A|} = \sum_{i=1}^{|A|} \sum_{j=1}^{|V|} \frac{\vec{a}_{i,j}}{\|\vec{a}_i\|} \cdot \frac{\vec{v}_{j,i}}{\|\vec{v}_j\|} = sum(ps) = \text{avg}(ps) \times |A| \times |V|
\]

where \(ps\) is the multi-set of pair-wise similarities between a pair of vectors, one taken from set \(V\) and one taken from set \(A\).

5. To capture the maximum of the pair-wise similarities (denoted \(\text{max}(ps)\)) we build \(|A| \times k\) long “selector” vectors \(\vec{\sigma}_{i,j}^k = \cdots, \vec{\sigma}_{|A|,1}^k, \cdots, \vec{\sigma}_{i,k}^k, \cdots, \vec{\sigma}_{|A|,k}^k\) for each \(k \in [1, M]\).

That is, a total of \(\sum_{k=1}^M |A| \times k = |A| \times \sum_{k=1}^M k = |A| \times \frac{M(M+1)}{2}\) long “selector” vectors. Each \(\vec{\sigma}_{i,j}^k \in \mathbb{R}^{|A| \times k \times D}\) “assumes” which of the \(|A| \times k\) pair-wise interactions is maximal:

\[
\vec{\sigma}_{i,j}^k = \begin{pmatrix}
\vec{0}_{i,1} \\
\vdots \\
\vec{0}_{|A|,1} \\
\vdots \\
\vec{1}_{i,j} \\
\vdots \\
\vec{0}_{i,k} \\
\vdots \\
\vec{0}_{|A|,k}
\end{pmatrix}
\]

i.e., \(\vec{\sigma}_{i,j}^k\) is a concatenation of \(O_s = (j - 1) \times |A| + (i - 1)\) \(\vec{0}\) vectors, followed by one \(\vec{1}\) vector, and ending with \(|A| \times k - (O_s + 1)\) \(\vec{0}\) vectors, where \(\vec{0} \in \mathbb{R}^D\) is the “all zeros” vector and \(\vec{1} \in \mathbb{R}^D\) is the “all ones” vector. \(\vec{\sigma}_{i,j}^k\) “assumes” the
Next, we construct each long target vector $\vec{w}$ where $\vec{a}$ is $\vec{a}$ under the assumption that $\vec{a}$ is the cardinality of the set of lineage vectors of the candidate tuple is $k$. Consequently, we get:

$$\vec{\sigma}^k_{i,j} \odot \vec{L}_A^k = \begin{pmatrix} \vec{\sigma}_{1,1} \\ \vdots \\ \vec{\sigma}_{|A|,1} \\ \vdots \\ \vec{\sigma}_{i,k} \\ \vdots \\ \vec{\sigma}_{|A|,k} \end{pmatrix}$$

such that $\vec{a}_{i,j}$ is the $j$th copy of $\vec{a}_i$ and $\odot$ is the Hadamard (i.e., element-wise) product. This results in:

$$(\vec{\sigma}^{|V|}_i \odot \vec{L}^{|V|}_A) \cdot \vec{L}_V^{|A|} = (\vec{\sigma}^{|V|}_i \odot \vec{L}^{|V|}_A) \cdot \vec{L}_V^{|A|} = \frac{\vec{a}_{i,j}}{||\vec{a}_i||} \cdot \vec{v}_{j,i} = \frac{\vec{a}_i}{||\vec{a}_i||} \cdot \vec{v}_j$$

6. Next, we construct $|A| \times k$ long target vectors $\vec{\tau}^k_{i,j}$, $\vec{\tau}_{i,j}$, $\vec{\tau}_{i,j}$, $\vec{\tau}_{i,j}$, $\vec{\tau}_{i,j}$, $\vec{\tau}_{i,j}$, $\vec{\tau}_{i,j}$ for each $k \in [1, M]$. That is, a total of $\sum_{k=1}^{M} |A| \times k = |A| \times \sum_{k=1}^{M} k = |A| \times \frac{M(1+M)}{2}$ long target vectors $\vec{\tau}^k_{i,j} \in \mathbb{R}^{|A| \times k \times D}$:

$$\vec{\tau}^k_{i,j} = \frac{1}{w_{\text{max}} + w_{\text{avg}}} \cdot (w_{\text{max}} \cdot (\vec{\sigma}^k_{i,j} \odot \vec{L}_A^k) + \frac{w_{\text{avg}}}{|A| \times k} \cdot \vec{L}_A^k)$$

where $w_{\text{max}}$ and $w_{\text{avg}}$ are (user-specified) hyperparameters, as specified in section 3.3.3. $\vec{\tau}^k_{i,j}$ “assumes” the cardinality of the set of lineage vectors of the candidate tuple is $k$.

7. Each long target vector $\vec{\tau}^{|V|}_{i,j} \in \mathbb{R}^{|A| \times |V| \times D}$ computes the desired similarity calculation via a dot product with a long candidate vector $\vec{L}_V^{|A|} \in \mathbb{R}^{|A| \times |V| \times D}$, under the assumption that $\vec{a}$ and $\vec{v}$ have the maximal pair-wise similarity:

$$\vec{\tau}^{|V|}_{i,j} \cdot \vec{L}_V^{|A|} = \frac{w_{\text{max}} \cdot (\vec{\sigma}^{|V|}_{i,j} \odot \vec{L}_A^{|V|}) \cdot \vec{L}_V^{|A|} + \frac{w_{\text{avg}}}{|A| \times |V|} \cdot \vec{L}_A^{|V|} \cdot \vec{L}_V^{|A|}}{w_{\text{max}} + w_{\text{avg}}} = \frac{w_{\text{max}} \cdot \vec{a}_{i,j}}{||\vec{a}_i||} \cdot \vec{v}_{j,i} + w_{\text{avg}} \cdot \text{avg}(ps) = \frac{w_{\text{max}} \cdot \vec{a}_{i,j} \cdot \vec{v}_{j,i} + w_{\text{avg}} \cdot \text{avg}(ps)}{w_{\text{max}} + w_{\text{avg}}}$$

75
Search

1. Initialize $M \times M$ vector search structures $S_{1,1}, \ldots, S_{M,M}$, such that $S_{n,k}$ holds long candidate vectors of dimension $n \times k \times D$ (i.e., long vectors of candidate sets of cardinality $k$, assuming target sets of cardinality $n$). Note that known approximate vector search techniques, such as LSH [GIM99], can be utilized here for efficiency.

2. For each candidate tuple $t_V$, that is associated with a set $V$ of lineage vectors of cardinality $|V| \in [1, M]$, and its respective construction of $M$ long candidate vectors $\tilde{L}_V^i \in \mathbb{R}^{n \times |V| \times D}$, for each $n \in [1, M]$, insert $\tilde{L}_V^i$ into the vector search structure $S_{n,|V|}$.

3. Now, suppose you are given a target tuple $t_A$, that is associated with a set $A$ of lineage vectors of cardinality $|A| \in [1, M]$. Construct $|A| \times k$ long target vectors $\tau_{1,1}^k, \tau_{1,2}^k, \ldots, \tau_{|A|,k}^k$, for each $k \in [1, M]$.

4. Hereafter, we denote a set $V$ of lineage vectors with cardinality $k$ as $V_k$.

5. For each $k \in [1, M]$, we look separately for the closest (dot-product wise) candidate long vector $\tilde{L}_{V_k_{i,j}}^{|A|} \in \mathbb{R}^{|A| \times k \times D}$ (and its respective candidate tuple $t_{V_k_{i,j}}$) in $S_{|A|,k}$, to each $\tau_{i,j}^k \in \mathbb{R}^{|A| \times k \times D}$, respectively. Intuitively, this focuses on candidate tuples associated with sets of vectors of cardinality $k$.

6. Next, intuitively, we need to choose the “best” one among the “winners” of different $k$ values. We compute the similarity scores $\tau_{1,1}^k \cdot \tilde{L}_{V_k_{i,j}}^{|A|}$, $\tau_{1,2}^k \cdot \tilde{L}_{V_k_{i,j}}^{|A|}$, $\ldots$, $\tau_{|A|,1}^k \cdot \tilde{L}_{V_k_{i,j}}^{|A|}$, $\tau_{|A|,2}^k \cdot \tilde{L}_{V_k_{i,j}}^{|A|}$, $\ldots$, $\tau_{|A|,k}^k \cdot \tilde{L}_{V_k_{i,j}}^{|A|}$, for each $k \in [1, M]$ (a total of $\sum_{k=1}^M |A| \times k = |A| \times \sum_{k=1}^M k = |A| \times \frac{M(1+M)}{2}$ computations). The one yielding the highest score, say $\tilde{L}_{V_k_{i,j}}^{|A|}$, identifies the desired candidate tuple $t_{V_k_{i,j}}$, which is associated with the set $V_k_{i,j}$ of lineage vectors, according to our set-oriented similarity formula.
Appendix B

BFPDB Dataset

Following is a detailed description of the data fields, of each table in the BFPDB dataset (used in the experiments in chapter 7).

B.1 products

- \textit{ndb\_no} - 8-digit Nutrient Database (NDB) number that uniquely identifies a food item. Links to the \textit{nutrients} table and the \textit{serving\_size} table.

- \textit{name} - 200-character description of a food item.

- \textit{manufacturer} - the company that manufactured the product.

- \textit{ingredients} - ingredients of the product.

B.2 nutrients

- \textit{ndb\_no} - 8-digit Nutrient Database (NDB) number that uniquely identifies a food item. Links to the \textit{products} table and the \textit{serving\_size} table.

- \textit{nutrient\_code} - unique 3-digit identifier code for a nutrient.

- \textit{nutrient\_name} - name of nutrient/food component.

- \textit{derivation\_code} - a code indicating how the \textit{output\_value} was determined. The codes used are defined in the \textit{derivation\_code\_description} table.

- \textit{output\_value} - amount in 100 g, edible portion. This value is calculated from the amount per serving value on the Nutrition Facts Panel supplied by the data provider.

- \textit{output\_uom} - units of measure for the \textit{output\_value}. 

77
B.3 serving_size

- *ndb_no* - 8-digit Nutrient Database (NDB) number that uniquely identifies a food item. Links to the *products* table and the *nutrients* table.

- *household_serving* - the amount of the *household_serving_size_uom*, i.e., the number of cups, tablespoons, teaspoons in a serving. May be a fraction, such as 0.25 or 0.50.

- *household_serving_size_uom* - the Units of Measure for the *household_serving*, i.e., cup, tablespoon, teaspoon.

- *preparation_state* - indicates if the information from the Nutrition Facts Panel is for the prepared or unprepared food. Only included if supplied.

B.4 derivation_code_description

- *derivation_code* - a code indicating how the *output_value* was determined. Links to the *nutrients* table.

- *derivation_code_description* - description of the *derivation_code*. 
Appendix C

MovieLens Dataset

Following is a detailed description of the data fields, of each table in the MovieLens dataset (used in the experiments in chapter 7).

C.1 movies

- **movieId** - a number that uniquely identifies a movie item. Links to the ratings table and the tags table. These movie ids are consistent with those used on the MovieLens web site (e.g., id 1 corresponds to the URL https://movielens.org/movies/1).

- **title** - movie titles, which are entered manually or imported from https://www.themoviedb.org/, and include the year of release in parentheses.

- **genres** - genres are a pipe-separated list, and are selected from the following:
  - Action
  - Adventure
  - Animation
  - Children’s
  - Comedy
  - Crime
  - Documentary
  - Drama
  - Fantasy
  - Film-Noir
  - Horror
  - Musical
  - Mystery
- Romance
- Sci-Fi
- Thriller
- War
- Western
- (no genres listed)

C.2 ratings

- userId - a number that uniquely identifies a user. Links to the tags table.
- movieId - a number that uniquely identifies a movie item. Links to the movies table and the tags table. These movie ids are consistent with those used on the MovieLens web site (e.g., id 1 corresponds to the URL https://movielens.org/movies/1).
- rating - a rating of the movie identified by movieId, rated by the user identified by userId. Ratings are made on a 5-star scale, with half-star increments (0.5 stars - 5.0 stars).
- timestamp - a timestamp of the rating action. Timestamps represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

C.3 tags

- userId - a number that uniquely identifies a user. Links to the ratings table.
- movieId - a number that uniquely identifies a movie item. Links to the movies table and the ratings table. These movie ids are consistent with those used on the MovieLens web site (e.g., id 1 corresponds to the URL https://movielens.org/movies/1).
- tag - tags are user-generated metadata about movies. Each tag is typically a single word or short phrase. The meaning, value, and purpose of a particular tag is determined by each user.
- timestamp - a timestamp of the tagging action. Timestamps represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.
Appendix D

BFPDB Materialized Views

Following is a detailed description of the materialized views (used in the experiments in section 7.3) and the queries that created them.

D.1 prepared

Contains all the products-related information from the products table, for products that have a prepared preparation state in the serving_size table:

```sql
SELECT p.ndb_no, p.manufacturer, p.name, p.ingredients
FROM products p, serving_size ss
WHERE p.ndb_no = ss.ndb_no
AND ss.preparation_state = 'prepared'
```

D.2 unprepared

Contains all the products-related information from the products table, for products that have a unprepared preparation state in the serving_size table:

```sql
SELECT p.ndb_no, p.manufacturer, p.name, p.ingredients
FROM products p, serving_size ss
WHERE p.ndb_no = ss.ndb_no
AND ss.preparation_state = 'unprepared'
```

D.3 readytodrink

Contains all the products-related information from the products table, for products that have a readytodrink preparation state in the serving_size table:
D.4 protein

Contains all the products-related information from the products table, for products that have a protein nutrient-related information in the nutrients table:

```sql
SELECT p.ndb_no, p.manufacturer, p.name, p.ingredients
FROM products p, nutrients n
WHERE p.ndb_no = n.ndb_no
AND n.nutrient_name = 'protein'
```

D.5 sugars

Contains all the products-related information from the products table, for products that have a sugars nutrient-related information in the nutrients table:

```sql
SELECT p.ndb_no, p.manufacturer, p.name, p.ingredients
FROM products p, nutrients n
WHERE p.ndb_no = n.ndb_no
AND n.nutrient_name = 'sugars'
```

D.6 cholesterol

Contains all the products-related information from the products table, for products that have a cholesterol nutrient-related information in the nutrients table:

```sql
SELECT p.ndb_no, p.manufacturer, p.name, p.ingredients
FROM products p, nutrients n
WHERE p.ndb_no = n.ndb_no
AND n.nutrient_name = 'cholesterol'
```

D.7 exp_1

Contains distinct manufacturers of products that have sugar as an ingredient, and contain protein nutrient information. Also, these manufacturers produce prepared products:
SELECT prt.manufacturer
FROM protein prt, (SELECT DISTINCT manufacturer FROM prepared) t
WHERE POSITION('sugar' IN prt.ingredients) > 0
AND prt.manufacturer = t.manufacturer
GROUP BY prt.manufacturer

D.8 \text{exp}_2

Contains distinct manufacturers of products that have \textit{water} as an ingredient, and contain \textit{protein} nutrient information. Also, these manufacturers produce \textit{unprepared} products:

SELECT prt.manufacturer
FROM protein prt, (SELECT DISTINCT manufacturer FROM unprepared) t
WHERE POSITION('water' IN prt.ingredients) > 0
AND prt.manufacturer = t.manufacturer
GROUP BY prt.manufacturer

D.9 \text{exp}_3

Contains distinct manufacturers of products that have \textit{water} and \textit{sugar} as an ingredient, and contain \textit{protein} nutrient information. Also, these manufacturers produce \textit{prepared} and \textit{unprepared} products:

SELECT t2.manufacturer FROM
(SELECT prt.manufacturer
 FROM protein prt, (SELECT DISTINCT manufacturer FROM prepared) t
 WHERE POSITION('sugar' IN prt.ingredients) > 0
 AND prt.manufacturer = t.manufacturer
 GROUP BY prt.manufacturer) t1,
(SELECT prt.manufacturer
 FROM protein prt, (SELECT DISTINCT manufacturer FROM unprepared) t
 WHERE POSITION('water' IN prt.ingredients) > 0
 AND prt.manufacturer = t.manufacturer
 GROUP BY prt.manufacturer) t2,
WHERE t1.manufacturer = t2.manufacturer

D.10 \text{exp}_4

Contains distinct names of products from the \textit{readytodrink} materialized view, that contain \textit{mango} in their name, and are produced by manufacturers from the \text{exp}_2 materialized view:

83
SELECT DISTINCT rtd.name
FROM exp_2, readytodrink rtd
WHERE exp_2.manufacturer = rtd.manufacturer
AND POSITION('mango' IN rtd.name) > 0
Bibliography


[KMSR17] Alison Kretser, Delia Murphy, and Pamela Starke-Reed. A partnership for public health: Usda branded food products database. *Journal of Food


Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In Lucy Vanderwende, Hal Daumé


After the description of the central idea, we present improvements and extensions to the method intended to make it more practical and useful for analysts. These analysts, who wish to use vectorial historical records to find the results of a specific record in the database, can do this more efficiently by comparing vector historical records of the group of interest in the database (each group of records) and rank them according to "closeness"—in a manner similar to the method of searching words with semantically similar vectors.

PostgreSQL integrates the calculation of historical record results into the database management system and presents the results with useful accuracy and reliability. ProvSQL extends this method based on "natural" results, which are not affected by the specific application of the method, and presents them in a manner similar to the method of comparing vector historical records of a specific group in the database. This method, which we call "natural," is useful in finding results based on historical records.

In conclusion, we examine the benefits of combining the search for a group of historical records with the search for the "closest" group in another database. When searching for the closest group, we use the method of "vector vector search," which is efficient and useful in finding results based on historical records.

Bibliography
תקציר

יתוסי של נתונים והז口コミ רבד多少钱 האחסנות. עבור קודמות עסק בהיות של תוצאת
שאילתה על לב מוסרי נתונים על ידי "מעטרנטון" המעיין על החוסני של התוצאת. מצא נתונים
של הנקה מת晊ים בבוסר של תוצאת שיאילתה ברמה פורח שמה.

בנ班车カラ והאנוסием של התמקדות ארח המגעוחת של רישום המשני נתונים לארד היית, תזכ
הบทנה בין רישום שעונים באום יTextLabel טילית תכלות מסד הנתונים (אלא "بثני קל" שלנה)
לבני רישום הענה בת ער הסוליצים (אלא מילוי טילית תכלות המסד הנתונים). לכל, לכל
אנתו מניין סרס סבי רישום המגלה על ידי ושאילתה עונה על תבנית היחסית למסדה של ישומת
שאילתה למסד הנתונים, אאות הכליה מת年之 עבורה ל데athroom במערבות. בראשית מתוית - ישומת
(1): הבחרת של מסדה של יוזר אחות מודרני בبحث: שאלות בתכונות מסד הנתונים וירוג בוחנה
- כל של שול חלונות טילית תכלות מסד הנתונים ורב עיטיס נוחותו בבלוב בלם קדואד ז"ח
(2): ישומת אוש מנסנת למסד הנתונים אנפת פקודת (3: INSERT INTO SELECT
(רשומות 3) ; UPDATE
אשר מנסנת למסד הנתונים אנפת פקודת (4) ; UPDATE
אשר מנסנת למסד הנתונים אנפת פקודת
(1) ; INSERT INTO SELECT
שבא לשון אשתת על שודת השופט
באופי ידני, אשר מנסנת בחרה לטבל אחות המסד הנתונים.

עם חלוק ה��, אילנות החוסני (החרסיבים הנובעים منه) של ישומת אል התוכנים המקוניגוס ומטבע
מאמץ מחזיק רכיכות יקרה, אם וישוב עבורה פורטיאת ישומת, חברות קגיית. אם מנהיגים ייש
שדיה השם מקורב של כן הוצאת יישומת יודה שמוש וביבתוכות לחמיט מוכנותуйודכ שופט
(בעיבוד, שיקוך יילום ברמת אוקלידי).

הרעיון המרכזי הוא עם מה שסבל אם סבל (באופי מיקוד) את החוסני של כל ישומת על יד קובץ
תוכני (מספוע רחון מילוי פ-רשום ושאילתו הפירפורסרית המיחנה). עבורה רישום המוכמות
למסד הנתונים באומן י seri רחון מחשיבות באומן מיאיר מיאיר של רחון מילוי על
гиб הפרסוס" של ישומת. הם חללי יזוע של ישומת על בים מסד הנתונים, ואר מחשיבות את
очно כי ישומת של רישום החוסני בתצוגה של ישומת במאגרי והנאה שונים דומין מתודיה התוכנים
( novitàו משלובים: " seri לובב"), אם מכללים את פועלות היישוב המוכמות לעבורה על בטוח
ישומת,_undo שים גבר על היכולים לפער את מידע התוכנים ושימו על יוזי הירוקהמקסימי של מסדר
סיפי הוספה של יוקורי יוספי פ-רשום.
המחקר בוצע בהנחיית של פרופסור עודד שמואלי, בפקולטה למדעי המחשב.

תחל היוצר בחתונא הדברים אמר פורסמו בכתביון מאת המחבר (מיכאל לייבוביץ') והמנחת (עודד שמואל) במחלקה המחקר שלטמתברר, אשרراسלאותיהם העדכניות ביותר הינו:


תודות

אני רוצה לודות למנהלה של עודד שמואל על ההכון, התוכנה והעריכה בשיעות ולא שיעות.

לעביד אתידת התחלתו והסיפת דוגלה.

אני רוצה לודות לתוך מתים שללו שירים על ההכבה וההכנתה העדכנית לאורך בזירה בעלים.

הנסכים.

ארור לילה, אני רוצה לודות לאומיה שלחלצת יוזר מכלה במכלシェיב, עם זאת, לא הפיסקה

לה.ResumeLayout בלה.

אני מודה לטכניק על התוכנה הספגת נגיבת בחשוש我爱你.
למידועת מכونة של מزانות רシュומות במדעי
היסטוריה

היבור על מחקר

לשימ שלולי חלקי של הדירישות לכתבת התואר
מונסרת להדיעו במעידי המשבע

מיכאל לייבוביץ’

הנהל לתנועת הסוכנויות – מרכז טכנולוגיה לישראל
�名 התכנית “א" חפוך ווינ 2021
למידת מכונה של מוצואות רשהות בניות

מיכאל לייבוביץ'