Tracer: A Machine Learning Based Data Lineage Solver with Visualized Metadata Management

by

Zhuofan Xie

S.B. in Mathematics and in Computer Science and Engineering
Massachusetts Institute of Technology, 2021

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

Author . . . .
Department of Electrical Engineering and Computer Science
December 12, 2021

Certified by . . . .
Kalyan Veeramachaneni
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Tracer: A Machine Learning Based Data Lineage Solver with Visualized Metadata Management

by

Zhuofan Xie

Submitted to the Department of Electrical Engineering and Computer Science
on December 12, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In databases, many data do not come from scratch. They are derived from some other data and what describes this is called *data lineage*. Knowing the data lineage could help us do data validation, error detection, data debugging, and privacy and access control. Unfortunately, many databases do not have well documented data lineage information, and most existing works in this area heavily relies on extra input such as metadata, source code or annotations. In this paper, we build upon Tracer, a previously purposed machine learning approach to this problem, and make it more accurate, more general, and more intuitive.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

It has been a long journey and this work would have been impossible without the invaluable help of many people. I am exceptionally fortunately to have those great individuals' company, and deeply thank my family, friends, labmates, teachers, and mentors for their support and wisdom not only during this project but throughout my life.

First and foremost, I thank my thesis supervisor Dr. Kalyan Veeramachaneni for his ideas and guidance through the entire project. I would like to thank Prof. Laure Berti-Équille and Dr. Dongyu Liu for their ideas on this project and guidence on writing. I would also like to thank Sarah Alnegheimish for her insights on the benchmark module. And I also want to thank all my DAI labmates for their support during my studies.

I deeply thank my families, Xie Chao, Liu Ying, Xie Songlin, Wang Yuying, ..., for their support. They have made it possible for me to study at MIT, and do whatever I want to do. The last but not the least, I want to thank my best friend Yiran, with whom I have grown immensely over the past years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data lineage (or *data provenance*) provides information about the data flow across databases, applications, and systems, and is useful in a number of settings. For example, knowing where the data come from helps us building more intuition of the data and grouping related fields together, and thus leads to more in-depth data analysis. Once we have found any error in the data, knowing the lineage could help us tracing to its root cause, which leads to easier data debugging. In enterprises environment, full lineage information can easily help determining which fields contain sensitive information, or are derived from sensitive information. This way, knowing the lineage can help us with privacy related regulations. We refer our readers to two in-depth survey papers [11, 5] for a more detailed discussion on the importance and applications of this problem.

The data lineage problem can be formulated as the following [11]: input relations (tables) $I_1, ..., I_k$ are fed into a series of transformations $T_1, ..., T_n$ (which might be known or unknown) and output relations $O_1, ..., O_m$ are produced. One who seeks to do data analysis, data debugging, or privacy control, using lineage information will then ask the following two questions:

(1) Given some output cell, field, or table, which inputs did the output come from? This is the *where-lineage* problem.

(2) Given some output, how were the inputs manipulated to produce the output? This is the *how-lineage* problem.

We will go over a brief example later in Section 1.1 to further illustrate what these two questions are.

Depending on the granularity of the answer, we can then classify lineage problems into two sub-categories: *schema-level* (*coarse-grained*) and *instance-level* (*fine-grained*)[11]. Using both the granularity, and how/where classifications, we are able to divide the data lineage problem into four big classes: the schema-level where-lineage problem, the schema-level how-lineage problem, the instance-level where-lineage problem, and the instance-level how-lineage problem. Later in Section 1.2 and Table 1.2, we will classify existing work into these four classes.

We can also classify lineage problems by what the focus of the problem solver is. When the applied transformations are fully known, for example when we already have the source code that generated the data, the focus typically is to efficiently represent and store the lineage information in metadata, or to augment existing big data computation platforms (e.g. Spark) so that we can record the lineage information during data processing with little overhead (e.g., see [21]) and we refer to this type of problems as lineage *tracking*. Likewise, when the transformations are not fully known, the thus the focus is to *infer* from the inputs and outputs, and we will refer to this problem as the lineage *tracing* problem. However, we want to emphasize that, depending on how much is known about the transformations, it is very common for a research to focus on both doing inference and managing metadata.

In this project, we will answer both where-lineage and how-lineage problem for the fine-grained, lineage tracing problem where the transformations are entirely unknown. The whole architecture is built upon Hofmann's previous Tracer project[9].

1.1 Motivating example

In this section, we will provide an example to illustrate the previous definition of data lineage and motivate some of our sub-problem definitions. First of all, it is beneficial to introduce the two main classes of row transformations: dispatcher and aggregator[6]. Dispatcher maps one row from the source table into zero or more rows, while aggregator maps one or more rows from the source table into one row. We will further discuss the types of aggregators[1] in Section 3.4.

In the following tables, the transformation from Table 1.1 to Table 1.2 splits the *prod-list* field by comma, extracts fields *prod* and *num* from each term, and creates a new row for each term - therefore, it is a dispatcher. The transformation from Table 1.2 to Table 1.3 sums up the *num* field by the corresponding *cust-id* field and thus is an aggregator.

order-id	cust-id	date	prod-list
0101	AAA	2/1/1999	333(10), 222(10)
0102	BBB	2/8/1999	111(10)
0379	AAA	4/9/1999	222(5), 333(5)

Table 1.1: Source table

order-id	cust-id	date	prod	num
0101	AAA	2/1/1999	333	10
0101	AAA	2/1/1999	222	10
0102	BBB	2/8/1999	111	10
0379	AAA	4/8/1999	222	5
0379	AAA	4/8/1999	333	5

Table 1.2: Output of dispatcher

cust-id	num-ordered
AAA	30
BBB	10

Table 1.3: Output of aggregator

As for column operations, we only consider many-to-one and one-to-one operations. For example, summing up several columns, or copying a column. An example of summation is illustrated in Table 1.4 and 1.5. The reason we are not considering one-to-many column operations is that it can actually be viewed as a dispatcher, which is already discussed in row operations.

[1]We will also discuss some typical dispatchers that Tracer will support, but we are currently unable to find any such example in our database.

For example, Table 1.4 is generated from Table 1.1 by mapping the *date* field into *month*, *day*, and *year*. Meanwhile, this transformation can also be viewed as a dispatcher just in the same way as we explain the transformation from Table 1.1 to 1.2. We split the *date* field by comma (it does not contain any commas, so this splitting operation creates exactly one term for each cell), extracts fields *day*, *month* and *year* from each term, and creates a new row for each term - therefore, it is a dispatcher.

We will from now on consider a database only consisted of Table 1.1 and 1.3, and try to answer the *where*-lineage of the *num-ordered* field in Table 1.3. Recall that the entries in *num-ordered* are generated using the entries in *prod-list* by first dispatching the numbers in every single entry and then suming them up by the associated *cust-id*. This means the *where*-lineage of the *num-ordered* field in Table 1.3 is the *prod-list* field in Table 1.1.

One very important note is that usually there are many different ways to generate a target field in small databases. In the above example, we may also generate the *num-ordered* field in the following way. We first dispatch the *date* field in Table 1.2 into *day*, *month*, and *year*, which gives Table 1.4, then sum up the *day* and *month* fields into a *day_plus_month* field, as in Table 1.5, and finally sum the *day_plus_month* field up by the associated *cust-id*, which gives Table 1.6.

order-id	cust-id	month	day	year	prod	num
0101	AAA	2	1	1999	333	10
0101	AAA	2	1	1999	222	10
0102	BBB	2	8	1999	111	10
0379	AAA	4	8	1999	222	5
0379	AAA	4	8	1999	333	5

Table 1.4: Dispatch the *date* field into *day*, *month* and *year*

However, this does **not** imply that there are many possible answers to this *where*-lineage problem. The answer is always "how was the target field actually generated", instead of "how can we use other fields to produce the target field". And in the above example, the lineage is always the *prod-list* field. Fortunately, when the databases get larger, the chance that multiple maps explains the same field will get

order-id	cust-id	day_plus_month	year	prod	num
0101	AAA	3	1999	333	10
0101	AAA	3	1999	222	10
0102	BBB	10	1999	111	10
0379	AAA	12	1999	222	5
0379	AAA	12	1999	333	5

Table 1.5: Sum *day* and *month* into *day_plus_month*

cust-id	**generated field**
AAA	30
BBB	10

Table 1.6: Sum *day_plus_month* by the associated *cust-id*

much smaller. For example, if we add more rows to the above database, it is unlikely that *day_plus_month* will continue to match *num-ordered* as these two fields in nature describes different things. In this case, we can say with confidence that the *prod-list* field is the only possible lineage.

We then briefly illustrate how our Tracer approaches this problem. The detailed solutions will be revisited later in Chapter 3.

1. We first identify the primary keys and then the foreign keys in this database. This allows us to link Tables 1.1 and 1.3 together, and thus know it is possible for fields in 1.3 to come from Table 1.1.

2. We then try to reproduce Table 1.2 by applying all pre-defined dispatchers to Table 1.1.

3. Next, we apply all pre-defined aggregators which aggregates any field by their corresponding *cust-id* (which is a identified foreign key) field. Note that after such aggregation, the rows will match the rows of Table 1.3, making it possible for us to detect column map operations. We then select aggregated fields that could potentially contribute to the target field (e.g. according to their cross-entropy), and join all such fields together, as illustrated in Table 1.7. Note that this step implicitly takes care of the join operation.

cust-id	sum(num)	max(num)	avg(num)	...	count(prod)	...
AAA	30	10	7.5	...	4	...
BBB	10	10	10	...	1	...

Table 1.7: Applying all pre-identified aggregators

4. Finally, we apply state-of-the-art machine learning techniques to find out which fields did the target field come from. For example, we could answer that the *num-ordered* field in Table 1.3 comes from the *sum(num)* field in Table 1.7, which back-trace to the *prod-list* field in Table 1.1. We can do such back-tracing because we carry out the transformation from Table 1.1 to 1.7 ourselves, and therefore know exactly where each field comes from. This step allows us to conclude that the lineage of the *num-ordered* field is the *prod-list* field.

1.2 Previous Work

Though there is plenty of literature on the data lineage problem, to the best of our knowledge, all of them except Hofmann's previous DataTracer [9] require some extra information on the data or transformations and focuses on solving the lineage problem for specific type(s) of databases, transformations, or data processing softwares.

We borrow this classification (Table 1.8) from [11].

	Schema	**Instance**
Where	Workflow [8]	Why and Where [2] General Warehouse [6] Warehouse View [7] Non-Answers [10] Approximate [18] Trio [22] Tracer [9]
How	Curated [3] Workflow [8]	Curated [3]

Table 1.8: A summary of previous work on the data lineage problem.

And we present a brief summary of the representative ones.

1.2.1 Lineage tracking in curated database [3]

Buneman et al. in 2006 focused on the lineage *tracking* (i.e. lineage management) problem in curated databases. In particular, the transformations supported are insert, copy, and delete, and the authors managed to store the instance-level how-lineage efficiently by (1) grouping operations into *transactions* and only record the net change for each transaction and (2) summarize multiple operations as a *hierarchical lineage*.

1.2.2 Lineage tracing in general warehouse [6]

Cui and Widom in 2001 focused on solving the lineage tracing problem in the context of general data warehouses. Specifically, the authors assumes that the transformation graph is known, and provides lineage tracking algorithms that work particularly well when properties of the transformations are known. These properties can classified into three types:

1. Transformation class. The authors classified transformations into three types: *dispatcher*, *aggregator* and *black-boxes* (examples of dispatchers and aggregators can be found in Section 1.1). The authors argued with Example 1.2.1 that, for black-box operators without any information, we can only conclude that the lineage of any output is the entire input data set. However, though this is theoretically true, *we* believe that when the data sets get huge, it is almost impossible that multiple transformations in the space (this space is eventually limited no matter how complex our solver is) we are searching for will give the same outputs. In the context of the following example, when the number of rows gets larger, it will be easy for the solver to tell if it's a filter or aggregator.

Example 1.2.1. With no restrictions, the transformation taking Table 1.9 to Table 1.10 can be either (1) filtering out rows with negative Y values, or (2) aggregate by X value, where the aggregation function is taking the sum of the Y values and then multiply it by 2.

2. Schema mapping. Which is basically equivalent to column-level where lineage.
3. Existence of inverse transformation.

X	Y
a	-1
a	2
b	0

Table 1.9: Input table

X	Y
a	2
b	0

Table 1.10: Output table

The authors also discussed when is it appropriate to trace lineage across many transformations at once.

Cui et al. in an earlier work[7] discussed a similar topic. They proposed algorithms for extracting the precise lineage when transformations are specified as relational operation or view.

1.2.3 The Tracer [9]

Unlike other solutions to the data lineage problem, the Hofmann in 2020 proposed the Tracer, which novelly solves this problem through a machine learning approach and does not rely on any additional input.

Taking in the raw database, the Tracer first predict a single-column primary key for each table. Then, it predicts the non-composite foreign key relations in the database.

Finally, to predict the where-lineage, the Tracer uses the foreign keys predicted and aggregates all other tables such that the rows now matches the target column. Just as we illustrated in Section 1.1 at the very end with Table 1.2 and 1.7. From here, the Tracer trains a random forest regressor, which takes in all columns in Table 1.7 and tries to predict the target column *num-ordered*. Once this regressor is trained on the tables we have, the Tracer will predict the columns with significant feature importance as the where-lineage of the target column.

While the previous Tracer offers this novel framework, it is not very successful in solving the data lineage problem[2] primarily due to the following reasons:

- It is designed only to predict single-column primary keys and foreign keys. If the database actually have composite primary or foreign keys, the Tracer will be incapable of detecting these keys and therefore relevant lineage.

- There is a problem in the way that the foreign key detection machine learning module is trained. The training data for this machine learning module is naturally skewed due to the fact that among all column pairs in a database, very few of them are actual foreign keys. However, the author has not recognized this issue, which resulted in a very poorly trained machine learning module.

- The regressor trained in the column map detection step suffers from over-fitting. As explained in Section 1.1, we are not looking for a possible way of generating the target column, but we are solving for how was that the target column actually generated. Unfortunately, when over-fitting happens, we usually end up getting a very complicated map which happens to generate the target column, which falls in the former case.

In this thesis, we will build upon the framework proposed in [9] and take it to a new level. More relevant details will be discussed in the next section.

1.3 Contributions of This Project

This project is built upon the framework of the previous Tracer [9], and make it more accurate, more general, and more intuitive. The previous Tracer has working primary key, foreign key and column map detection primitives that only support single keys. This means that when the primary key is composite, the previous Tracer will not be able to detect that key, and therefore all subsequent foreign keys and lineage relations referencing this key. However, many of the real-world databases[3] have composite keys,

[2]On our testing set, the previous Tracer only achieves an average F1 score of 0.275, while our new Tracer achieves an average F1 score of 0.701.

[3]Among the tables in our testing databases, which will be described in Chapter 4 and are quite

and not supporting them will lead to a huge inability of the Tracer library to recover many of the lineage information. The first contribution of this project is that we update the Tracer framework, APIs, and primitives to support composite keys. The full APIs will be introduced in Chapter 2, and the solutions will be introduced in Chapter 3.

Next, we optimized the foreign key, and "where" column map detection primitives for much higher precision and recall. And we also developed a "how" column map detection primitive, which not only answers where does the target column come from, but also how it was generated. The API and solution will be introduced in Chapter 2 and 3, respectively.

We also developed a benchmarking module, which allows the user to easily compare between different solutions on their databases, and developers to test the stability of their solutions. The details will be introduced in Chapter 4.

Finally, we developed a lineage visualization user interface, that allows the user to straightforwardly explore the database, as well as the relation between tables and lineage information. It will be introduced in Chapter 6.

1.4 Limitations

While in practice, there is no limitation on what we can do with a table and how can we generate a field from others, and we would like to support as many of them as possible, we admit that there is currently no general-purpose transformation detection algorithm. For practicability, we have to restrict our scope to the following:

- Theoretically we support all data types, but the Tracer works best with numerical, boolean, and categorical data, while we only support strings upto basic concatenation and split operations, and do not yet support operations involving both strings and numerical values.

- All transformations must be reducible to row operations and column operations.

representative of real-world big databases, 50% of them have composite primary keys.

- The lineage must be unambiguous. Equivalently, there can not be any essentially duplicate rows or columns.
- All data must be provided in tables.
- For the *how lineage* primitive to run properly, the transformation must be fully deterministic.
- Finally, while current implementation can utilize multi-processing, it is not designed to run distributedly and therefore is impractical for solving the data lineage problem in databases sizing more than 10GB. In our experiments, it on average takes the Tracer 9 hours to solve a single lineage in a 4GB database while occupying one Intel Xeon Platinum 8260 processor core (@2.40 GHz).

Chapter 2

System Design

Overall, the Tracer library solves the *where* and *how* lineage problem by breaking it down into many sub-problems, from primary key detection to the final column map detection, and executing in a relational database by executing the primitives designed for solving these sub-problems. This recovers the database's metadata step-by-step. We further assemble these primitives into pipelines so that each problem can be solved by a single execution of the corresponding pipeline.

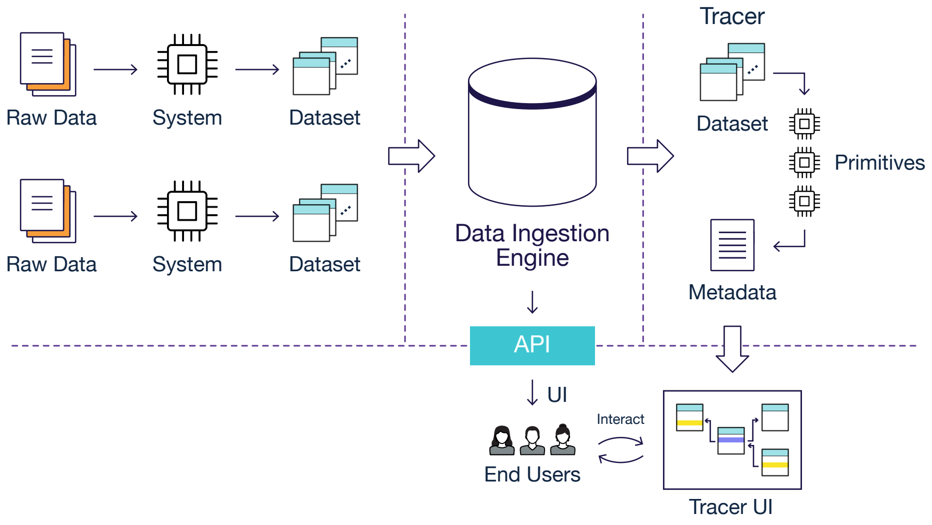

Figure 2-1: Tables with lineage but without lineage metadata are produced by various systems, and are converted into the appropriate format by data ingestion engine. The tables are then fed into the Tracer. By executing pipelines, the Tracer recovers the lineage information and presents this to the end-users through the lineage visualization UI.

At a larger scale, the Tracer also interacts with other modules in the same context [9], as shown in Figure 2-1.

Firstly, raw databases are fed into some systems where fields with lineage are produced and ingested into the tables. These systems can range from Spark computation platform, MySQL database, to even excel sheets. Unfortunately, the metadata are often poorly maintained as these tables gets updated and spread out. Especially, data lineage information often gets lost in this process. To make things even worse, we do not always know which system(s) generated these tables and therefore cannot easily infer what transformations are likely applied to the tables, which exaggerated the loss of data lineage.

These tables, which are in various formats, are then fed into the Data Ingestion Engine that helps us to transform the tables into standardized table and metadata formats. The standardized tables and metadata are then presented to end-users through various service APIs, and also passed into the Tracer, which will help recover its data lineage metadata.

The tracer then attempts answer the data lineage problem by executing various pipelines on the tables, update the metadata if necessary, and pass the metadata to the Lineage Visualization module, which is our second focus of the thesis.

Finally, the Lineage Visualization module interacts with the end-users, allowing them to straightforwardly explore the relations and data lineage in the database.

In this chapter, we will describe the primitive APIs and how are them assembled into pipelines.

2.1 Tracer Primitives and Pipelines

Tracer has two very similar end-to-end pipelines, which are responsible for answering the *where* and *how* lineage problem, respectively.[1] They both take in the database, pass it through a series of steps, called *primitives*, where each primitive takes in the output of the previous primitives and attempts to answer a sub-problem of the data

[1]The names 'Where-column-map detection' and 'How-column-map detection' are inconsistent

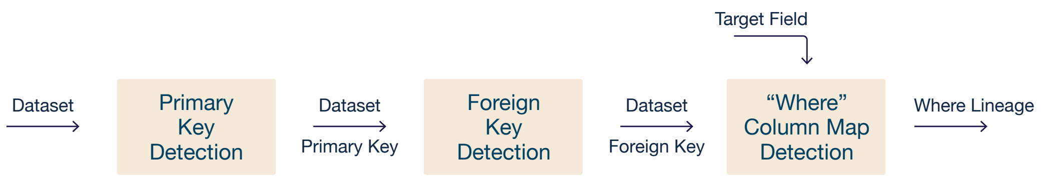

Figure 2-2: The 'Where' Column Map Detection pipeline

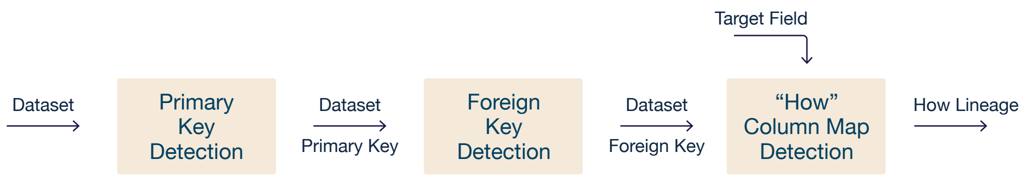

Figure 2-3: The 'How' Column Map Detection pipeline

lineage problem, and finally output the desired lineage.

As illustrated in Figure 2-2 and 2-3, both pipelines utilize the same upstream primitives, the Primary Key Detection primitive and the Foreign Key Detection primitive. The Primary Key Detection primitive takes in the database and detects the primary keys of the tables. Then the Foreign Key Detection primitive uses the database and the detected primary keys to detect the foreign key relations in the database. Finally, the "How" ("Where", resp.) Column Map Detection primitive takes in the database and the foreign keys to produce the how (where, resp.) lineage information as the final output. We will formally define the data formats and primitive APIs in the next section.

2.2 Data Formats and Primitive APIs

In this section, we will define the data formats and primitive APIs, which will be used throughout the rest of the thesis.

The data formats are our internal representation of the database and metadata-related information, as listed in Table 2.1 and 2.2.

The primitives, as briefly introduced in Section 2.1, are the Primary Key Detection

with the current tracer implementation, and the purpose of this rename is for better clarity.

Name	**Description**	**Example**

database, or tables (dictionary)	A set of tables in a relational database. Each table has a unique name, *one* or more named fields, and zero or more rows.
	It is represented using a dictionary mapping table names to the corresponding dataframes.
		`{`
		`  table1: DataFrame1,`
		`  table2: DataFrame2`
		`}`

Primary key	It indicates which column(s) are the primary key of a table, which might be single or composite. It is represented using a list of the field name(s) in the primary key.	`[‘coach_id’, ‘year’]`

Primary keys (dictionary)	The primary keys in a database.
	It is represented using a dictionary mapping table name to its primary key.
		`{`
		`  table1: [‘year’,`
		`    ‘coach_id’],`
		`  table2: [‘team’]`
		`}`

Foreign key	It indicates a foreign key relation in the database. Represented using a dictionary with fields `table`, `field`, `ref_table`, and `ref_field`. Here, `table` and `ref_table` are table names, `ref_field` is a primary key of `ref_table`, and `field` is a list of field names in `table`.	`{`
		`  ‘table’: ‘coaches’,`
		`  ‘field’: [‘year’,`
		`    ‘id’],`
		`  ‘ref_table’:`
		`    ‘roster’,`
		`  ‘ref_field’:`
		`    [‘year’,`
		`    ‘coach_id’]`
		`}`

Foreign keys (list)	It indicates all the foreign key relations in a database. In the format of a list of *foreign keys*.	`[foreignKey1,`
		` foreignKey2, ...]`

Table 2.1: Tracer Data Formats

Name	**Description**	**Example**
Lineage column	It describes one of the columns that contributes to the generation of a specified column. It is represented using a dictionary with fields `table` and `field` indicating the table and field name of this column.	`{ table: ‘roster’, field: ‘wins’ }`
Lineage columns (list)	It describes all the lineage columns of a specified column, in the form of a list of lineage columns.	`[lineageColumn1, lineageColumn2, ...]`
“How” lineage column	It describes how a lineage column is transformed to contribute towards the generation of a specified column by using a dictionary specifying (1) the lineage column, (2) according to which foreign key relation is it aggregated and (3) what aggregation function is used.	`{ ‘field’: lineageColumn, ‘foreign_key’: foreignKey, ‘aggregation’: ‘count’ }`
How lineage (dictionary)	It fully describes how a specified field is generated from other fields in the same database using a dictionary with fields `lineage_columns`, which gives a list of “how” lineage columns, and `transformation` that describes what transformation function is used to obtain the values in the target field.	`{ ‘lineage_columns’: [howLineageColumn1, ...], ‘transformation’: ‘sum’ }`

Table 2.2: Tracer Data Formats (Cont.)

Name	**Input**	**Output**
Primary Key Detection	A *database.*	A *primary keys dictionary.*
Foreign Key Detection	A *database* and a *primary keys dictionary.*	A *foreign keys list.*
"Where" Column Map Detection	A *database*, a *foreign keys list*, a target table name and target field name.	A *lineage columns list.*
"How" Column Map Detection	A *database*, a *foreign keys list*, a target table name and target field name.	A *how lineage dictionary.*

Table 2.3: Tracer primitive I/Os

primitive, the Foreign Key Detection primitive, the "Where" Column Map Detection primitive, and the "How" Column Map Detection primitive. Their corresponding I/O formats are listed out in Table 2.3. Please note that all the I/O formats are defined in Tables 2.1 and 2.2.

On top of these specified I/O formats, each primitive is also supposed to have the following properties:

- Each primitive should have a `fit` and `solve` method. In the `fit` method, it takes in databases with well-documented metadata, and trains its machine learning sub-modules, if any. In the `solve` method, it takes in input as specified in Table 2.3 and produces the corresponding output.

- It should be independent of the implementation of other primitives. This way, developers can isolate things and focus on implementing new solutions to just a small sub-problem, and the users can also freely build up pipelines that fits their databases the best.

Chapter 3

Tracer Subproblems

In this chapter we discuss how to break down the data lineage tracing problem into proper sub-problems, the nature of those sub-problems, and propose basic solutions, as well as how to create machine-learning-based solutions.

Overall, we break down the data lineage problem into two major components: detecting all possible row maps, and determining the column map.

The row map component has two sub-problems: (1) detecting the primary keys of the tables and (2) detecting the foreign key relations across the tables[1], assuming the primary keys are known. Using the foreign key relations detected, we will be able to know the possible row maps between the tables. Extensive work has been done on these two problem. We will review these existing works in details and discuss our extensions in Section 3.2 and 3.3.

The column map detection component has two independent sub-problems, one is the where-lineage problem and another one is the how-lineage problem. Both sub-problems assume knowledge of the foreign keys and asks for the where (how, resp.) lineage of a given column. Recall from Section 1.1 that the where-lineage problem of a column asks "*from which columns do this column come from*" whereas the how-lineage problem asks "*how exactly did we derived this column*". They are the problems we primarily tackle in this project, and we will discuss our solution in details in Section 3.4 and 3.5.

[1]In this paper, we also refer to a set of related tables as a *database*.

3.1 Data we use

As we discussed in the previous sections, we tackle this entire data lineage problem through a machine learning approach, which naturally requires training data. In this section, we discuss the data we use to train and test our models.

We use data from 66 real world relational databases originally maintained by Motl and Schulte [13]. These databases have primary key and foreign key metadata, and have then been widely used for testing and training various data-related algorithms [16, 14, 20]. However, none of them contains lineage metadata. Because of this, we have created many fields with lineage using the DataReactor library [23], as well as the corresponding lineage metadata, for the purpose of testing our lineage detection solutions, and the agumented databases are maintained at `tracer-data.s3.amazonaws.com`. We summarize these databases in Table 3.1, and we will revisit them later in Chapter 4 when we come to testing.

Number of databases	66
Total number of tables	572
Total number of columns	9671
Total number of primary keys	528[2]
Fraction of tables with composite primary keys[3]	44%[4]
Size of the largest primary key	4
Total number of foreign keys	750
Fraction of composite foreign keys	5%
Size of the largest foreign key	3
Number of fields with lineage	5585[5]
Total size	9.0G

Table 3.1: Summary of the training databases

[2]Some tables in the databases do not have a primary key documented in the metadata (we believe that information is either not collected by the database maintainer or lost), and few tables actually do not have any primary keys.

[3]By composite primary key, we mean a primary key with more than one columns.

[4]We care about the fraction of the tables with composite primary keys since predicting composite keys is in nature a harder problem than predicting single-column keys, as will be discussed in the following section.

[5]The original databases do not have any lineage fields, and we generate this many using the DataReactor package.

3.2 Primary Key Detection

In this section, we handle the basic of everything - figuring out the primary keys of tables. We tackle this problem using a machine learning approach, which is universal. That is, once the primary key detector is trained on tables with ground-truth primary key metadata (for our solution, we train it on the tables in databases described in Section 3.1), it can then be applied on any table to detect the primary key. Needless to say, if more training tables are provided, these primitives will be able to improve over time.

In the following sections, we will review the state-of-the-art non-machine-learning-based, and machine-learning-based solutions. Then, we will discuss how our Tracer tackles this problem.

3.2.1 Non-machine-learning solutions

Naturally, the problem of primary key detection can be broken down into two sub-problems: (1) determining which column combinations[6] could serve the indexing purpose and therefore is a primary key candidate, and (2) finding the true primary key among all candidates.

Abedjan et al. [1] have summarized a few algorithms for efficiently detecting the primary key candidates.

Specifically, a set of columns in a table can be a primary key candidate if they can serve the indexing purpose. In other words, it can be a primary key candidate if no two rows take the same values in all of these columns. Columns with this property are known as a *Unique Column Combination* (UCC).

However, the total number of column combinations in a table with m columns is 2^m, which means it is exponentially hard to search all of them. Because of this, there is a natural trade-off between the number of UCCs we search, thus higher primary key detection accuracy, and the speed of this approach. As suggested in [12], we restrict the scope of UCC search to all minimal UCCs to achieve a balance between precision

[6]We use the phrase*column combination* as a primary key can be composite, or in other words,

and speed. Here, by minimal UCC we mean an UCC which does not contain any other UCCs. For example, if we have a unique column a, and a non-unique column b, then column combinations $\{a\}$ and $\{a, b\}$ are both UCCs, but only $\{a\}$ is minimal as $\{a, b\}$ does contain $\{a\}$ as a subset so $\{a, b\}$ is not minimal. A algorithm for retrieving all minimal UCCs suggested in [1] is pretty straightforward. It is as the following:

As a little preparation, we call a column combination with k columns a *k-non-unique* if it is not a UCC, or a *k-UCC* otherwise. A brute force checking will tell us if a k-column-combination is a k-UCC in linear time. We simply go over all the tuples and keep all of them in a hash set. If in this process we ever see a tuple that is already in the hash set, then it is not an UCC, and otherwise it is an UCC.

1. We start by setting $k = 1$, and classifying all single columns as either 1-non-unique or 1-UCC. 1-UCCs are automatically minimal UCCs. Note that we cannot say any 1-UCC is the primary key at this step, as the values in a column can be unique for a number of reasons. For example, for a column that contains high-precision floating point numbers, it is very likely to be unique while it is almost surely not a primary key.

   As the 1-UCC columns cannot be contained in any other minimal UCCs, they will be set aside, and only the 1-non-uniques will remain in the game for the remaining steps.

   Then we move onto recursively finding out all k-non-uniques and minimal k-UCCs for all k.

2. After we have got the list of all minimal k-UCCs and k-non-uniques, we can move forward to investigating the $(k+1)$-column-combinations. As the ultimate goal for us is to get the minimal UCCs, we are only interested in investigating the $(k + 1)$-column-combinations which do not contain any other UCCs. Thus, instead of investigating all $(k+1)$-column-combinations, we *only* investigate all the combinations of a k-non-unique and a 1-non-unique.

have multiple columns.

3. For these $(k+1)$-column-combinations of the form k-non-unique + 1-non-unique, we first see if it already contains any other UCCs we have classified so far. If it does not include any other UCC, we classify them as either $(k+1)$-non-uniques or minimal $(k+1)$-UCCs using the above-mentioned brute-force procedure.

4. If we have identified any $(k+1)$-non-uniques, and that $k+1$ is smaller than the number of columns, we set $k = k+1$ and go back to Step 2. Otherwise, we conclude that we have found all the minimal UCCs.

We will refer to this algorithm as the **UCC detector** module later in our solution. While the worst case runtime of this algorithm is still exponential, as we can have bad examples where any column combination is non-unique and therefore the algorithm will need to go over all column combinations, it is in practice very efficient since 4 or 5 column combinations are almost surely guaranteed to be unique in real world tables.

Surprisingly, not much work has been done on the topic of finding the true primary key among all candidates.

Papenbrock and Naumann, 2017, in EDBT [15] proposed several heuristic features to distinguish true primary keys from other candidates, such as the ordinal position of the columns, the length of its fields, and if its column names contain "id" or "key". Then, they simply add up all feature scores and predict the highest scoring candidate as the true primary key.

Mainly using Papenbrock and Naumann's features, **Jiang and Naumann, 2020, in JIIS** [12] proposed an iterative algorithm for predicting the primary and foreign keys together, and their algorithm is in practice more accurate. However, the action of summing up the scores does not utilize full predictive power of these features.

3.2.2 Machine learning based solutions

Hofmann, 2020, in his masters' thesis [9] took a machine learning approach in the previous Tracer. He used a classifier, which takes in the features of the primary key candidates, and predicts if it is the actual primary key, and thus makes use of the full predictive power of these features. However, it is designed for detecting the primary key only when a table has a single-column primary key, and thus lost its power in almost 50% of the real world tables.

A summary of the previous works is provided in Table 3.2. Our solution takes advantage from all of them, and lies in the top left corner of the table.

	Machine-learning-based	**Non-machine-learning-based**
Composite Keys		Holistic PK/FK detection by Jiang and Naumann, 2020 [12] Heuristic features by Papenbrock and Naumann, 2017 [15]
Single Key	Tracer by Hofmann, 2020 [9]	

Table 3.2: Previous work on primary key detection

3.2.3 Primary Key Detection in the Tracer

To solve this primary key detection sub-problem, for each table we basically go through three steps as illustrated in Figure 3-1:

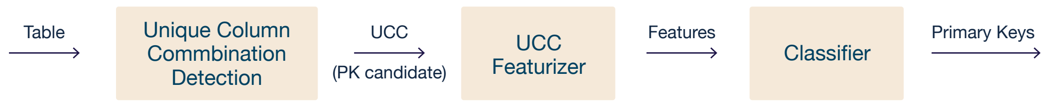

Figure 3-1: Workflow of the Primary Key Detection primitive.

1. Find out all primary key candidates using the *Unique Column Combination* (UCC) detector module. The algorithm is described in Section 3.2.1.

2. Featurize every primary key candidate.

   For this step, we build our features mainly upon the hand-crafted heuristics proposed in [15, 1].

   Specifically, for each column in a UCC, we extract these features:

- **Ordinal position.** The position of this column in this table, represented in its percentage (i.e. index / total number of columns). This is because primary keys tend to be the first few columns in a table.
- **Name**. Binary: if “key" or “id” is in the field name. This is due to the fact that these keywords often suggest the corresponding column is a key.
- **Datatype**. If the entries are integers or strings. This is not only because primary key tends to be integers or strings, but also to counter-act the fact that floating number columns tends to be unique and appears more often in the UCCs.
- **Text length**. The maximum length of the values in this column in human-readable format. We use this feature as identifiers tends to be short.

Finally, for each UCC, its feature is the average of the features of its columns, plus an additional feature:

- **UCC size**. The number of columns in this UCC. This feature is useful as usually primary key will not contain too many columns.

3. Feed the feature vectors into a classifier and predict the candidate with the highest score as the primary key.

   This is the only part that needs to be trained, and the training method is straightforward. We simply feed the labeled tables through the first two steps, and train this classifier in a fully supervised manner. Once this classifier is trained, it is universal and thus can be applied to any table.

   One thing we have not described in detail is the machine learning model we use here. Due to the small number of features we use, we believe the choice of machine learning model can only have very minimal impact on the actual performance, as long as it has enough model capacity and is fully trained. Therefore, we use the random forest classifier with default parameters specified in `scikit-learn v0.23.0`[7].

Design choices

Our solution, as well as most of the mainstream primary key detection algorithms, can be run a single table, and there is no need for *this implementation* to take in a whole database, as required in the API defined in Section 2.2. And requiring so seems to have led to a loss of flexibility in terms of the overall framework design, and the possibility of computing distributedly. For example, have different machines solve for the primary keys for different tables and then put them together no longer seems to be an option with the current API.

However, we still make this design choice observing the following advantages.

- **A unified framework**. While all downstream primitives require knowledge to the whole database, this design choice creates a more unified framework where each primitive in the pipeline is called exactly once.
- **Extendability**. While most primary key detection algorithms do not need knowledge to the whole database, some algorithms do require so. For example, [12] suggests that the foreign key detection process also provides constructive feedback to the primary key detection, as the fact that a primary key candidate is likely to be referenced somewhere else significantly increases its possibility of being the primary key. And the authors suggests that instead of scoring a single primary key candidate, scoring the whole primary keys and foreign keys choices of the entire database leads to higher precision. Currently, we have not adopted this solution as our databases are too big, and finding out all possible primary and foreign keys alone would take days on a 48-core machine. However, our current API design choice will easily allow the developer to easily adopt these solutions to the primary key detection primitive.

[7]The key parameters are: `n_estimators` = 10, `criterion` = `mse`, `max_depth` = `None`, `min_samples_split` = 2, `min_samples_leaf` = 1, and `bootstrap` = `True`. We have also experimented with the default parameters in `scikit-learn v1.0.1`, which describes a much bigger random forest, and no significant difference in performance was observed.

3.3 Foreign Key Detection

Then we move on to the final step of pin-pointing row maps. Similar to our Primary Key Detection primitive, we will build an universal machine learning solution, which can be applied to any database after it is trained.

We proceed by briefly reviewing past work on this topic.

3.3.1 Non-machine-learning Solutions

Similar to the primary key detection problem, this foreign key detection problem can be naturally broken down into determining the foreign key candidates, and predicting which are the true foreign key relations among all candidates.

A set of columns U in a table $\mathcal{A}$ is able to reference the primary key P (which is also a set of columns) of table $\mathcal{B}$ if for each row t in $\mathcal{A}$, there is also a row t' in $\mathcal{B}$ such that the values $t[U]$ and $t'[P]$ are the same. Such two sets U, P is known as a *Inclusion Dependency* (IND). Therefore, all the foreign key candidates are just all the INDs where the columns being referenced is a primary key. A brute-force yet very efficient algorithm is summarized in Abedjan et al.[1].

For a primary key P of table $\mathcal{B}$, and a target table $\mathcal{A}$, we go over the following steps. We will also be doing a running example, with $P = \{1, 2\}$ amd $\mathcal{A}$ having four columns $\{3, 4, 5, 6\}$.

1. For their to be a inclusion dependency $\mathcal{A}[U] \subset \mathcal{B}[P]$, the corresponding columns have to include one another. We write $P = \{p_1, ..., p_k\}$ for convenience. For each column $p_i \in P$, we find all columns $u \in A$ such that there is a single column dependency $\mathcal{A}[u] \subset \mathcal{B}[p]$, and call the collections of such columns U_i. This can be found via a brute-force search, which only takes linear time for each column in P.

   For example, we might end up having $U_1 = \{3, 4\}, U_2 = \{4, 5\}$. Note that these sets do not necessarily have to be disjoint.

2. Then we investigate the combinations of the single columns we got in Step

1 by considering every element in the Cartesian product $U_1 \times U_2 \times ... \times U_k$. Specifically, for every column combination $U = \{u_1, ..., u_k\}$ where $u_i \in U_i$, that is, $\mathcal{A}[u_i] \subset \mathcal{B}[p_i]$, we check with brute force if every tuple in $\mathcal{A}[U]$ appears in $\mathcal{B}[U]$. Therefore, we can know if there is inclusion dependency inclusion dependency $\mathcal{A}[U] \subset \mathcal{B}[P]$. Checking each U takes linear time, and this step will give us all the IND of columns in table $\mathcal{A}$ referencing P in $\mathcal{B}$.

Using the previous example, all set Us we will be investigating at this step are $\{3, 4\}, \{3, 5\}$ and $\{4, 5\}$.

While in worst case this algorithm runs in $|\mathcal{A}|^{|P|}$ time as we can simply have all the columns be the same and therefore the number of INDs is this many, in practice it is very efficient since there are not many single-column IND to begin with, thus very few column combinations to check. We later refer to this algorithm as the **IND detector** module.

On the attempts to predict the actual foreign keys from all candidates, a handful of heuristic features and pruning rules have been proposed.

Rostin et al., 2009, in SIGMOD [19] proposed ten feature scores to help evaluating the chance that an IND is an actual foreign key, for example, the similarity between the column names, etc. These features have been widely used by later works, including ours. However, their algorithm only works for single-column foreign keys.

On top of these features, **Zhang et al., 2010, in VLDB** [24] proposed the *randomness* feature, which measures the data distribution similarity between the two sides of the IND and the higher the similarity is the more likely it is to be a foreign key.

On top of these features, many research have been done on the topic of IND pruning, defining a set of rules that a foreign key relation, or the set of all foreign key relations, should satisfy. **Chen et al., 2014, in VLDB** [4] proposed that, for example, a one-to-one correspondence, or in other words a primary key referencing another primary key, cannot be a foreign key relation. However, for efficiency they also assumed that there can be at most one foreign key relation between any two tables, which is not true in big real-world databases.

Jiang and Naumann, 2020, in JIIS [12] further proposed pruning rules, such as there cannot be cyclic reference, and have achieved the best performance among all existing methods. However, all these pruning rules are for the purpose of eliminating false positive foreign key predictions, and as they are not 100% accurate, it will eliminate a few true positives at the same time. As will be discussed in later sections, our downstream primitives' are very sensitive to false negative foreign keys predictions, but very robust against false positives. For this reason, we will not adapt these pruning rules in our solution.

3.3.2 Machine learning based solutions

Rostin et al., 2009, in SIGMOD [19] first proposed the idea of using machine learning classifier to help predict foreign keys. After featurizing the foreign key candidates, the authors train a machine learning classifier to predict the true foreign keys.

Hofmann, 2020, in his masters thesis [9] used the same idea in previous Tracer. However, both works only support single-column foreign keys.

A summary of the previous works is provided in Table 3.3. Our solution takes advantage from all of them, and lies in the top left corner of the table.

	Machine-learning-based	**Non-machine-learning-based**
Composite Keys		Holistic PK/FK detection by Jiang and Naumann, 2020 [12] Pruning rules by Chen et al., 2014 [4] Randomness by Zhang et al., 2010 [24]
Single Key	Tracer by Hofmann, 2020[9] Heuristics-based by Rostin et al., 2009 [19]	

Table 3.3: A selection of previous work on foreign key detection

3.3.3 Foreign Key Detection in the Tracer

In order to find out the foreign key relations, we follow a very similar three-step approach as in Section 3.2.

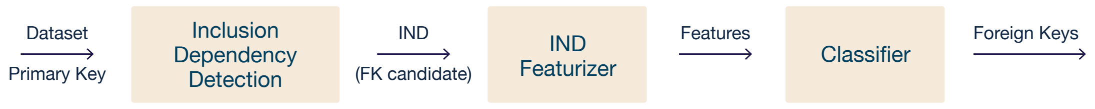

Figure 3-2: Workflow of the Foreign Key Detection primitive.

1. Find out all foreign key candidates using the *Inclusion Dependency* (IND) detector module. The algorithm has been discussed in Section **??**.

2. Featurize every foreign key candidate.

For each column pair `(column, ref_column)` in the IND, we featurize it mainly using the hand-crafted heuristics proposed in [12] and [19]:

- **Ordinal position**. The position of the child (parent, resp.) in the corresponding table, represented in its percentage. This is because keys usually have high ordinality.

- **Name similarity**. The similarities between the field names, measured in terms of the *fuzzy similarity*, as column referencing another column tends to have similar names.

- **Name**. If "key" or "id" is in the child (parent, resp.) field name. These keywords are strong indicators that the column is a key.

- **Data distribution**. The difference in the data distribution between the child and parent columns, measured in terms of the *earth-moving distance*. This is helpful as columns in a foreign key relation tends to have its entries coming from the same distribution, while columns that accidentally have an inclusion dependency do not. The reason that we use EMD as a distribution difference measure is it is empirically studied to better capture the difference between two numerical distributions in [24] and [19].

- **Data type**. Same as in the primary key detection part, if the entries in the child (parent, resp.) column are integers or strings helps us determine if it is actually a key.

Finally, the feature for an IND is calculated as the average of its column pairs' features.

3. Feed the feature vectors into a classifier and predict the foreign keys.

Similarly, this is the only part that needs to be trained, and the training method is still simply feeding the labeled tables through the first two steps and training this classifier in a supervised manner. As discussed in Section 3.2, due to the small number of features we are using, the choice of classifier does not have a significant impact on the final performance, and we use the default random forest classifier[8] in `scikit-learn v0.23.0`.

One thing to note here is that among all foreign key candidates, only a small portion of them are actual foreign keys. In other words, the training set of the classifier is significantly imbalanced, and regularization is required in order to achieve a good performance on the foreign key detection problem.

3.4 "Where" Column Map Detection

Now we can come to solving the where-lineage problem. One thing to note here is that, while our solution for the column map detection problems is machine-learning-based, the machine learning module is more of a tool to help us studying the relation within the data, and the whole solution is in fact a randomized algorithm. As a result, these approaches, and the two primitives do not need to be trained.

As briefly illustrated in Section 1.1 and Figure 3-3, we solve this sub-problem through a three-step approach.

1. The transform step. In this step, we aggregate other tables using our pre-identified aggregation functions, according to the detected foreign keys, and

[8]For a copy of the key parameters, please refer to the footnote where we first mention the random

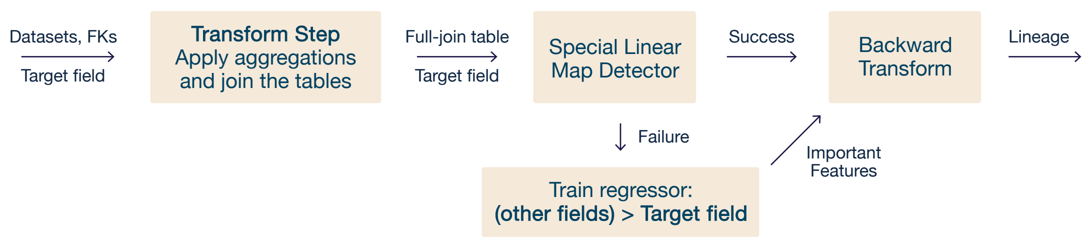

Figure 3-3: Workflow of the "Where" Column Map Detection primitive

full-join them to the target table. We will further explain this step, including what are the aggregation functions we are considering, in Section 3.4.1

2. The column map detection step. Now the rows are fully aligned, and the problem becomes detecting which other columns generate the target column. In this step we will use two modules, a special linear map detector and a general machine learning regressor, for higher accuracy. The intuition and the implementation details will be discussed in Section 3.4.2.

3. The back-transform step. After we have identified the lineage in the fully-joined table, we need to take it back to the original tables. To achieve this, we simply record where does each column in the new table come from, and use this information to recover the original lineage.

It is note-worthy that while this solution does not need many hand-crafted heuristics, it does not need any training data as well. We will discuss the implementation details of step 1 and 2 in the following sections.

3.4.1 The Transform step

It is beneficial to first discuss what aggregators (please refer to Section 1.1 for the definitions) are under our consideration. While there are potentially infinitely many aggregators, we can only try to apply very few of them due to time considerations. Therefore, we adapt the following list of basic aggregators from Peng et al.'s work in 2018 [17]: *sum*, *count*, *minimum*, *maximum*, *average*, *standard deviation*, and

forest model in Section 3.2.

variance. As the current Tracer only supports numerical data, dispatchers are unfortunately not supported for now since they are almost surely applied to strings only.

In the transform step, we produce a more useful representation of the data. Basically, for every column that is not in the target table, we consider the possible ways that it can contribute towards the target column by applying an aggregation function to itself, and store all columns after aggregations in a big table.

Specifically, we go over every foreign key relation that referenced the target table. Suppose in foreign key relation F, the columns U of table $\mathcal{A}$ reference the primary key P of our target table $\mathcal{B}$. Then for every aggregation function f, and column c of $\mathcal{A}$ that is not part of the key U, we *aggregate it according to the foreign key F using function f, into a column c'* by doing the following:

1. For every row i of table $\mathcal{B}$, we first take the value of the primary key in that row. For simplicity, we denote it as $p_i = \mathcal{B}[P][i]$, which is a tuple.

2. Then we search for all row j of table $\mathcal{A}$ such that it references row i of table $\mathcal{B}$. In other words, we find the set $J_i := \{j \mid \mathcal{A}[U][j] = p_i\}$.

3. Therefore, we can get the set of c's values in rows that references row i of table $\mathcal{B}$ as: $c_i = \{c[j]\}_{j \in J_i}$. Here c_i is a multiple set, meaning that the same value may appear many times in it.

4. Finally, we compute the aggregation, and set the corresponding value in c' as: $c'[i] = f(c_i)$.

Note that we will generate one such column for every plausible foreign key, column, and aggregation function combination. These c's are the columns that matches the target column's rows and may directly contribute to it. To avoid a full-join operation, which will make the subsequent machine learning problem computationally very costly, we will only select the aggregated columns (c' in the previous context) whose cross-entropy with the target field is high, as candidates, and join them together into a table called the *intermediate* table.

3.4.2 The Column Map Detection step

Now that the transform step has already given us an intermediate table R with rows r_i, and we shall separate the target column y from R for clarity.

The column map we are looking for is simply a function f, when applied to a row r_i, gives us $f(r_i) = y_i$. And our ultimate goal in this step is to answer the question "*which columns have an impact on f's evaluation*".

Recognizing that in most systems, the most common column maps are the very simple ones, the sum, average, or difference of some of the columns, and that they are not easy to find using general machine learning models as they tend to overfit, we implement a special linear map detection module just for this purpose. The special linear map detection module will check using a brute-foce approach to see if any of the aforementioned simple maps is the f we are looking for. Only when this module fails, or in other words that the target column cannot be expressed as the sum, difference, or average of other columns, will we use a general machine learning approach to detect the column map.

Note that while the column map in theory can be arbitrary, most of them in practice, if not already detected in the previous special linear map detector, have the following nice properties,

- Among all available (transformed) columns, only a small portion of them are involved in the column map.
- It generally takes the form of the combination of conditioning on the values in some fields, and simple arithmetic operations on some fields.

And we find that a decision tree model with a very shallow depth best fit this purpose.

Therefore, in the machine learning approach, we fit a random forest regressor, with all the default parameters[9] specified in `scikit-learn v0.23.0` but `max_depth` restricted to 10, with input being R and the prediction target being y. As this

[9]For a copy of the key parameters, please refer to the footnote where we first mention the random

approach does not give an explicit expression for the map f, we predict the lineage columns as the columns with significant feature importance.

3.5 "How" Column Map Detection

As can be seen, the solution proposed in Section 3.4 is almost ready for the purpose of answering how the target column is generated. Only two minor augmentations are needed.

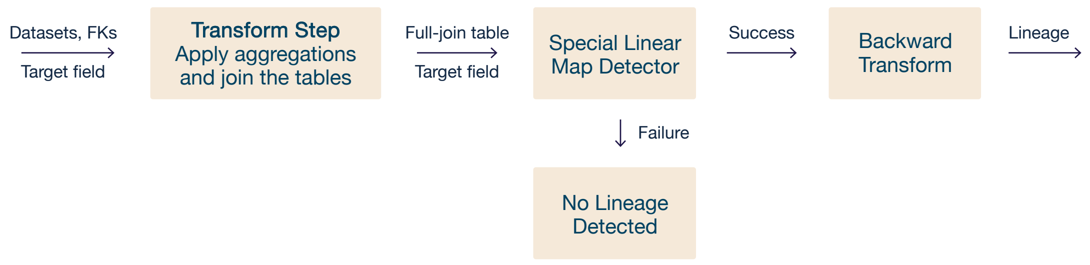

Figure 3-4: Workflow of the "How" Column Map Detection primitive

- In the transform step, we not only need to record *where* does each column come from, but also *how*. Specifically, we need to record which foreign key is used and which aggregation function is used.
- In the column map detection step, the only module that we can rely on is the special linear map detection module, but not the regressor module, as the latter one does not answer the question of what the column map is in a clean way.

forest model in Section 3.2.

Chapter 4

Benchmarking Tracer

In Tracer, we also provide a benchmarking module to enable our users to compare pipelines and primitives against each other, and our developers to test the stability and performance of their new pipelines and primitives.

In this chapter, we discuss the default benchmarking method and metrics provided. Nonetheless, our developers are always welcomed to implement new methods and introduce new metrics as long as they see a good fit.

4.1 Terminology

In the rest of this thesis, we have only discussed one possible solution for each tracer sub-problem, and introduced one primitive of each kind (Primary Key Detection, etc.). However, it is totally possible to have many solutions to the same problem, and therefore multiple primitives of the same kind, each good for a particular type of databases. For example, in the current Tracer, we have two different Primary Key Detection primitives[1]: one imposes strict uniqueness on the primary key and thus good for databases without duplicate rows, and another one imposes weaker strict uniqueness on the primary key and therefore works better for databases with duplicate rows.

[1]Not for the composite-key version, as strict uniqueness is required so that the primitive can be run in a reasonable amount of time.

Because of this, it is important to distinguish between a *sub-problem* and a *primitive* for the purpose of this chapter. A sub-problem refers to a task with a specified I/O API as specified in Section 2.2. And a primitive refers to a solver targeted at a specific sub-problem. It is possible that a sub-problem has multiple primitives.

In the benchmarking module, a *unit test* will be the evaluation of a single inference of a primitive. That is, comparing the output produced by a single invocation of a primitive to the ground truth stored in the metadata of the database.

4.2 Benchmarking Method and Metrics

4.2.1 Benchmarking Method

We will test our primitives using 66 real world databases from [13]. A summary of them are in Chapter 3 Table 3.1.

By default, we will use leave-one-out validation. Specifically, for all unit tests, we will train the corresponding primitive on all but the database we are testing on, and then feed the target database, as well as any applicable inputs, into the trained primitive for further evaluation. The performance metrics we will use will be discussed in the next section.

4.2.2 Metrics

For each unit test, if the standard output is a list of predictions of unspecified length, we will use the following metrics in Table 4.1 by default.

Metric Name	**Description**
Inference time	The time it takes to do the inference.
Precision	We take the fields in the output as predicted positives and the rest as predicted negatives. And calculate the precision, recall and F1 score of this prediction accordingly.
Recall	
F1	

Table 4.1: A list of metrics used in Tracer Benchmarking

Two special sub-problems are the Primary Key Detection problem and the "How"

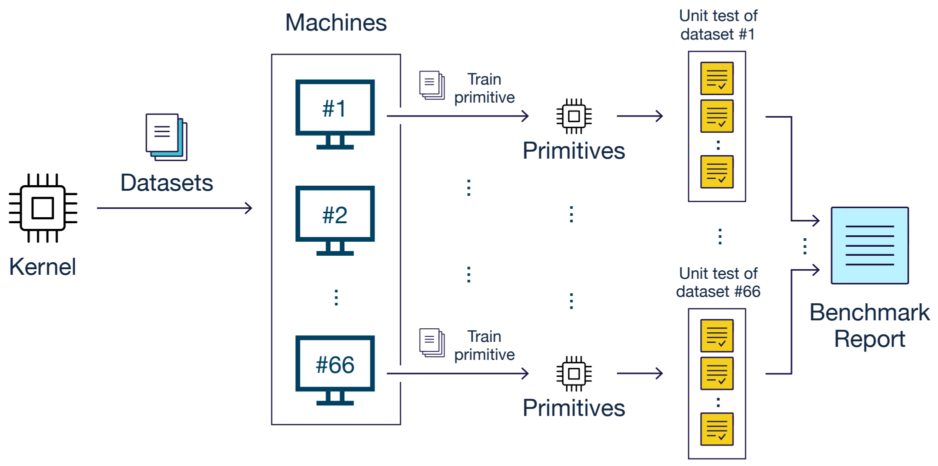

Figure 4-1: Workflow of the Tracer benchmark module

Column Map Detection problem. As a partially correct primary key prediction, say in a composite primary key we get three of the five fields correct, will not lead to any utility. And so does a partially correct how-lineage prediction, say the actual map is aggregating field A by taking the sum while we only get the "taking the sum" part correct. Hence, for these two sub-problems, we will treat every single prediction as either correct or wrong, and evaluate the corresponding primitives only based on the overall precision and inference time.

Finally, for all sub-problems, whenever applicable, we will also document the test database name, table name, field name, and iteration number for more readability.

4.3 Benchmarking Framework

For the most efficient utilization of possibly distributed computation power, we implement our benchmarking framework to shoot as many tests in parallel as possible.

As illustrated in Figure 4-1, all the databases are firstly downloaded and delivered to every machine in the network. Each machine is then responsible for running all unit tests on a specific database. And finally the results are aggregated into a final report. To simplify things, if we are having less physical machines than the total number of databases, we will run several virtual machines in a physical machine to make the numbers equal.

On a single machine, to achieve maximum concurrency while running all unit tests on a specific database, we first train the primitive on all but the tested database, and then we invoke all unit tests using this trained primitive concurrently.

However, we do recognize that there is a trade-off between simplicity in design, and precision in measuring the run-time. When we are trying to simplify the design by creating a virtual machine for each testing database and invoking all unit tests in parallel for maximum concurrency, many unit tests will be sharing the same CPU core. As the time-sharing, which is "indeterministically" decided by the system kernel, is not consistent throughout the tests, this will result in some inaccuracy in the run-time measurement of the unit tests. While one could avoid such issues by using slightly more complicated design, such as measuring the total CPU hours used, or adding some mechanism ensuring that the number of threads will be no greater than the number of CPU cores, we do not recognize an urgent need for this as the run-time is not the most important metric we are looking at, as long as it stays on the same order.

Chapter 5

Results

In this chapter, we test our primitives according to the method described in Chapter 4, and report the results.

5.1 Testing setup

We will test our primitives using 66 datasets as described in Section 4.2.1, which have 572 tables in total, and sizing 9.0GB.

The tests will be run on 8 machines, each with 2 Intel Xeon Platinum 8260 processors (2 x 24 cores @2.40 GHz) and 192GB RAM.

As most primitives have upstream primitives, we have to feed in appropriate input rather than the raw dataset in order to test them. We have two options here: one is to feed in the ground truth primary or forein keys as the input, and another one is to create pipelines ending with the target primitive and test the entire pipeline. For example, instead of testing the foreign key detection primitive alone, we build a pipeline consisting of the primary key detection primitive, and then the foreign key detection primitive, and this this pipeline instead. While the later one is more complicated, and is technically not a test of the primitive alone, we believe it is a better metric for the following reason. While we do want the primitive to perform good based on 100% correct primary/foreign keys as inputs, in application it will run on imperfect input generated by upstream primitives, and robustness against errors from upstream primitives is equally equivalent. Because of this, we believe testing the whole pipeline will better indicate the primitives' performance in real applications and therefore adopt this method.

5.2 Results

As can be seen in Tables 5.1 and 5.2, the primary key detection primitive achieves an average precision of 0.838 on all datasets, with an average precision of 0.980 on tables with single primary key and an average precision of 0.594 on tables with composite primary keys.[1] As will be discussed in Chapter 7, while this is a relatively good result, much of the errors is introduced by the fact that our *minimal* UCC assumption is too strict in some cases for our Tracer to recognize the true primary key as a candidate, and further research on this is needed.

The foreign key detection primitive, as shown in Tables 5.3 and 5.4, reaches an average f1 score of 0.654, and recall of 0.661 across the datasets.[2] We emphasize the recall score here as it has the most affect on the downstream column map detection primitive. While false positive foreign key predictions will rarely directly lead to inaccurate lineage inference (they do increase the workload of the downstream primitive and will cause timeout sometimes, which undermines the downstream task's performance), a false negative prediction will directly cause the downstream primitive's inability to detect any relevant lineage.

Finally, we report the performance of both the "where" and "how" column map detection primitives together in Tables 5.5 and 5.6, as in the current implementation, the "how" primitive succeeds if and only if the "where" primitive detects the correct linear map. The "where" primitive reaches an average f1 score of 0.701 across the datasets[3], and the "how" primitive" reaches an average precision of 0.685. While these scores are far from being perfect, and the predictions of the Tracer cannot

[1]In contrast, the previous Tracer achieves an average precision of 0.985 on tables with single primary key, but cannot predict composite primary keys.

[2]The previous Tracer achieves an average f1 of 0.562 and recall of 0.564 across the datasets.

[3]The previous Tracer only achieves an average f1 of 0.275 in contrast.

be used as ground truth, theses scores do suggest that the Tracer can detect most lineages in real world datasets, and the predictions have high reference values.

dataset	**tab**	**prec**	**sgl**	**s_prec**	**comp**	**c_prec**	**time**
PremierLeague_v1	4	1.0	3	1.0	1	1.0	26.536
Bupa_v1	9	1.0	9	1.0	0	nan	0.023
financial_v1	8	0.875	8	0.875	0	nan	129.386
Countries_v1	4	0.75	3	1.0	1	0.0	209.593
nations_v1	3	0.667	2	1.0	1	0.0	7.407
DCG_v1	2	1.0	1	1.0	1	1.0	2.031
Student_loan_v1	10	1.0	8	1.0	1	1.0	0.199
VisualGenome_v1	6	0.667	3	1.0	3	0.333	188.952
Triazine_v1	2	1.0	1	1.0	1	1.0	0.527
UTube_v1	2	0.5	1	1.0	1	0.0	0.56
MuskSmall_v1	2	1.0	2	1.0	0	nan	21.454
tpcc_v1	9	0.25	2	1.0	6	0.0	225.796
Seznam_v1	4	1.0	1	1.0	1	1.0	163.617
Accidents_v1	3	1.0	2	1.0	0	nan	20.222
CORA_v1	3	1.0	1	1.0	2	1.0	14.106
Pyrimidine_v1	2	1.0	1	1.0	1	1.0	0.354
university_v1	5	1.0	3	1.0	2	1.0	0.104
Same_gen_v1	4	1.0	1	1.0	3	1.0	0.166
legalActs_v1	5	1.0	3	1.0	2	1.0	700.24
Hockey_v1	19	0.2	0	nan	10	0.2	303.683
Airline_v1	19	0.944	18	0.944	0	nan	1247.643
lahman_2014_v1	25	0.208	3	1.0	21	0.095	3287.838
Carcinogenesis_v1	6	1.0	6	1.0	0	nan	2.506
Credit_v1	8	1.0	8	1.0	0	nan	2210.336
Facebook_v1	2	1.0	1	1.0	1	1.0	34.529
mutagenesis_v1	3	1.0	2	1.0	1	1.0	0.932
TubePricing_v1	20	1.0	7	1.0	0	nan	193.556
trains_v1	2	1.0	2	1.0	0	nan	0.064
Hepatitis_std_v1	7	0.286	4	0.25	3	0.333	1.101
SalesDB_v1	4	1.0	4	1.0	0	nan	219.323
KRK_v1	1	1.0	1	1.0	0	nan	0.061
Basketball_women_v1	8	0.333	1	1.0	2	0.0	51.911
Pima_v1	9	1.0	9	1.0	0	nan	0.039

Table 5.1: Primary Key Detection primitive benchmark results. The columns in the table are: dataset name, number of tables, overall prediction precision, number of tables with single primary key, prediction precision on tables with single primary key, number of tables with composite primary keys, prediction precision on tables with composite primary keys, and inference time, resp. from left to right.

dataset	**tab**	**prec**	**sgl**	**s_prec**	**comp**	**c_prec**	**time**
Telstra_v1	5	1.0	2	1.0	3	1.0	23.747
stats_v1	8	1.0	8	1.0	0	nan	1371.514
NCAA_v1	9	0.444	3	1.0	6	0.167	29.42
FNHK_v1	3	0.333	1	1.0	2	0.0	24.685
ftp_v1	2	1.0	1	1.0	1	1.0	25.622
classicmodels_v1	8	0.75	6	1.0	2	0.0	4.554
PTE_v1	38	0.472	15	1.0	21	0.095	1.72
UW_std_v1	4	1.0	2	1.0	2	1.0	0.244
tpcd_v1	8	0.875	6	1.0	2	0.5	3082.716
TalkingData_v1	9	0.625	5	1.0	3	0.0	2878.345
sakila_v1	15	0.933	13	1.0	2	0.5	52.568
WebKP_v1	3	1.0	1	1.0	2	1.0	7.709
imdb_ijs_v1	7	0.714	3	1.0	4	0.5	282.405
NBA_v1	4	1.0	3	1.0	1	1.0	1.351
imdb_MovieLens_v1	7	1.0	4	1.0	3	1.0	383.302
Mesh_v1	29	0.889	23	1.0	4	0.25	0.454
imdb_small_v1	7	0.833	3	1.0	3	0.667	1.454
Toxicology_v1	4	0.75	3	1.0	1	0.0	7.143
pubs_v1	11	0.778	7	0.857	2	0.5	1.224
Chess_v1	2	1.0	2	1.0	0	nan	3.742
Biodegradability_v1	5	1.0	3	1.0	2	1.0	1.379
Mooney_Family_v1	68	0.721	1	1.0	67	0.716	0.508
Elti_v1	11	0.636	1	1.0	10	0.6	0.207
Dunur_v1	17	0.294	1	1.0	16	0.25	0.199
AustralianFootball_v1	4	1.0	3	1.0	1	1.0	37.28
tpch_v1	8	0.875	6	0.833	2	1.0	3087.386
SAP_v1	4	1.0	4	1.0	0	nan	366.371
restbase_v1	3	1.0	3	1.0	0	nan	2.747
medical_v1	3	1.0	1	1.0	1	1.0	1009.238
SAT_v1	36	0.944	35	0.971	1	0.0	0.201
genes_v1	3	1.0	1	1.0	1	1.0	0.325
Atherosclerosis_v1	4	0.75	3	1.0	1	0.0	1182.025
world_v1	3	1.0	2	1.0	1	1.0	6.86

Table 5.2: Primary Key Detection primitive benchmark results (Cont.)

dataset	**num_fk**	**f1**	**prec**	**recall**	**time**
TalkingData_v1	7	0.706	0.6	0.857	42047.641
mutagenesis_v1	3	1.0	1.0	1.0	2.673
KRK_v1	0	0.0	0.0	0.0	0.061
Pima_v1	8	0.0	0.0	0.0	10.475
NCAA_v1	17	0.69	0.833	0.588	86.318
UW_std_v1	4	1.0	1.0	1.0	0.419
Seznam_v1	3	0.5	1.0	0.333	285.753
Basketball_women_v1	8	0.4	1.0	0.25	50.172
trains_v1	1	1.0	1.0	1.0	0.111
Toxicology_v1	5	0.75	1.0	0.6	15.886
Carcinogenesis_v1	13	0.0	0.0	0.0	15.315
imdb_MovieLens_v1	6	1.0	1.0	1.0	677.232
world_v1	2	1.0	1.0	1.0	7.203
FNHK_v1	2	1.0	1.0	1.0	227.93
Student_loan_v1	9	0.0	0.0	0.0	3.831
UTube_v1	1	1.0	1.0	1.0	1.075
MuskSmall_v1	1	0.0	0.0	0.0	21.108
genes_v1	3	0.8	1.0	0.667	1.064
medical_v1	2	0.667	1.0	0.5	1006.856
CORA_v1	3	0.667	0.667	0.667	24.49
SalesDB_v1	3	1.0	1.0	1.0	6825.567
Same_gen_v1	6	0.0	0.0	0.0	0.212
Elti_v1	20	0.833	0.714	1.0	1.299
Credit_v1	10	0.824	0.875	0.778	5013.326
restbase_v1	3	0.8	1.0	0.667	5.446
Dunur_v1	32	0.703	0.542	1.0	1.242
TubePricing_v1	39	0.817	0.906	0.744	254.971
Atherosclerosis_v1	3	1.0	1.0	1.0	1165.402
tpcd_v1	8	0.8	0.857	0.75	12094.78
Accidents_v1	3	1.0	1.0	1.0	302.912
ftp_v1	1	0.667	0.5	1.0	45.49
pubs_v1	10	0.632	0.667	0.6	1.528
Countries_v1	3	0.0	0.0	0.0	229.623

Table 5.3: Foreign Key Detection primitive benchmark results. The columns in the table are: dataset name, number of tables, number of foreign key relations in the dataset, prediction f1, precision, and recall, and inference time, resp. from left to right.

dataset	**num_fk**	**f1**	**prec**	**recall**	**time**
VisualGenome_v1	6	0.857	0.75	1.0	2888.612
DCG_v1	1	1.0	1.0	1.0	4.259
SAP_v1	3	0.5	1.0	0.333	2271.138
Telstra_v1	4	0.667	0.5	1.0	91.373
Facebook_v1	2	1.0	1.0	1.0	34.813
imdb_ijs_v1	6	1.0	1.0	1.0	1258.828
Bupa_v1	8	0.222	1.0	0.125	3.803
sakila_v1	22	0.462	0.529	0.409	162.178
Hepatitis_std_v1	6	0.714	0.625	0.833	11.947
Triazine_v1	1	1.0	1.0	1.0	0.778
Mesh_v1	33	0.5	0.8	0.364	2.257
legalActs_v1	5	0.75	1.0	0.6	1324.147
lahman_2014_v1	31	0.059	0.333	0.032	3433.072
Chess_v1	1	1.0	1.0	1.0	4.011
financial_v1	8	1.0	1.0	1.0	336.932
Hockey_v1	27	0.0	0.0	0.0	308.699
NBA_v1	5	0.833	0.714	1.0	2.874
tpcc_v1	10	0.111	0.125	0.1	2869.992
SAT_v1	65	0.004	0.002	0.031	141.164
stats_v1	13	0.375	1.0	0.231	1971.854
classicmodels_v1	8	0.769	1.0	0.625	5.233
AustralianFootball_v1	5	0.909	0.833	1.0	78.455
tpch_v1	8	0.857	1.0	0.75	10049.715
Pyrimidine_v1	1	1.0	1.0	1.0	0.409
PTE_v1	39	0.323	0.211	0.692	25.689
university_v1	4	1.0	1.0	1.0	0.259
Airline_v1	33	0.245	0.375	0.182	7944.249
Biodegradability_v1	5	1.0	1.0	1.0	5.581
Mooney_Family_v1	134	0.383	0.237	1.0	18.155
PremierLeague_v1	5	1.0	1.0	1.0	35.953
imdb_small_v1	6	1.0	1.0	1.0	2.411
nations_v1	3	0.4	0.5	0.333	10.866
WebKP_v1	3	1.0	1.0	1.0	16.215

Table 5.4: Foreign Key Detection primitive benchmark results (Cont.)

dataset	num	f1	prec	recall	how	time
TalkingData_v1	49	0.889	0.918	0.879	0.816	31790.779
mutagenesis_v1	11	1.0	1.0	1.0	1	29.093
KRK_v1	1	1.0	1.0	1.0	1.0	0.03
Pima_v1	8	1.0	1.0	1.0	1.0	0.866
NCAA_v1	221	0.534	0.534	0.534	0.529	4958.848
UW_std_v1	19	0.842	0.842	0.842	0.842	3.665
Seznam_v1	13	0.385	0.385	0.385	0.385	410.427
Basketball_women_v1	141	0.906	0.909	0.907	0.043	1131.755
trains_v1	3	1.0	1.0	1.0	1.0	0.344
Toxicology_v1	3	0.0	0.0	0.0	0.0	18.071
Carcinogenesis_v1	19	0.211	0.211	0.211	0.211	7.58
imdb_MovieLens_v1	31	0.989	1.0	0.984	0.968	206.728
world_v1	14	0.214	0.214	0.214	0.214	3.005
FNHK_v1	17	1.0	1.0	1.0	1.0	1365.174
Student_loan_v1	12	0.333	0.333	0.333	0.333	1.45
UTube_v1	2	1.0	1.0	1.0	1.0	0.476
MuskSmall_v1	3	1.0	1.0	1.0	1.0	1147.142
genes_v1	7	1.0	1.0	1.0	1.0	19.903
medical_v1	103	0.913	0.913	0.913	0.029	713.198
CORA_v1	11	0.364	0.364	0.364	0.636	13.378
SalesDB_v1	19	0.316	0.316	0.316	1.0	301.282
Same_gen_v1	7	0.857	0.857	0.857	0.857	3.495
Elti_v1	13	0.846	0.846	0.846	0.846	18.968
Credit_v1	71	0.866	0.873	0.862	0.845	1683.774
restbase_v1	15	0.667	0.667	0.667	0.667	21.247
Dunur_v1	19	0.789	0.789	0.789	0.789	50.342
TubePricing_v1	315	0.521	0.521	0.521	0.521	4625.901
Atherosclerosis_v1	111	0.968	0.968	0.969	0.955	210.52
tpcd_v1	43	0.965	0.979	0.979	0.93	44596.046
Accidents_v1	44	0.97	0.97	0.97	0.955	11922.903
ftp_v1	2	1.0	1.0	1.0	1.0	45.699
pubs_v1	55	0.927	0.927	0.927	0.764	22.313
Countries_v1	314	0.503	0.503	0.503	1.0	4805.837

Table 5.5: "Where" and "How" Column Key Detection primitives benchmark results. The columns in the table are: dataset name, number of fields with lineage, "where" prediction f1, precision, and recall , "how" prediction precision, and average inference time for one lineage field, resp. from left to right.

dataset	**num**	**f1**	**prec**	**recall**	**how**	**time**
VisualGenome_v1	21	0.429	0.429	0.429	0.429	119.748
DCG_v1	3	0.667	0.667	0.667	0.667	0.12
SAP_v1	29	0.92	0.914	0.931	0.897	1436.256
Telstra_v1	8	1.0	1.0	1.0	1.0	159.695
Facebook_v1	6	0.333	0.333	0.333	0.167	604.666
imdb_ijs_v1	25	0.52	0.52	0.52	0.68	1610.926
Bupa_v1	7	1.0	1.0	1.0	1.0	0.7
sakila_v1	83	0.832	0.833	0.831	0.807	232.247
Hepatitis_std_v1	15	0.867	0.867	0.867	0.933	7.2
Triazine_v1	20	1.0	1.0	1.0	1.0	2.108
Mesh_v1	28	0.25	0.25	0.25	0.25	269.031
legalActs_v1	15	0.4	0.4	0.4	0.467	159.543
lahman_2014_v1	434	0.064	0.068	0.063	0.058	4482.011
Chess_v1	12	1.0	1.0	1.0	1.0	18.152
financial_v1	29	0.759	0.759	0.759	0.759	749.406
Hockey_v1	155	0.111	0.108	0.173	0.103	6465.333
NBA_v1	20	0.35	0.35	0.35	0.35	619.357
tpcc_v1	40	0.749	0.755	0.745	0.5	1658.426
SAT_v1	56	0.786	0.786	0.786	0.786	853.727
stats_v1	89	0.18	0.18	0.18	0.18	7091.478
classicmodels_v1	49	0.633	0.633	0.633	0.776	19.622
AustralianFootball_v1	115	0.991	0.991	0.991	0.991	452.013
tpch_v1	70	0.42	0.429	0.416	0.4	3657.918
Pyrimidine_v1	24	1.0	1.0	1.0	1.0	1.945
PTE_v1	42	0.333	0.333	0.333	0.381	677.706
university_v1	12	1.0	1.0	1.0	1.0	1.395
Airline_v1	2391	0.174	0.173	0.186	0.171	5859.9
Biodegradability_v1	5	1.0	1.0	1.0	1.0	16.467
Mooney_Family_v1	12	0.083	0.083	0.083	0.083	261.057
PremierLeague_v1	32	0.964	0.99	0.956	0.938	1002.749
imdb_small_v1	16	0.938	0.938	0.938	0.562	8.928
nations_v1	4	0.75	0.75	0.75	0.75	232.935
WebKP_v1	2	1.0	1.0	1.0	1.0	8.746

Table 5.6: “Where” and “How” Column Key Detection primitives benchmark results (Cont.)

Chapter 6

Lineage Visualization

In this chapter, we justify our design choices made in visualizing the lineage information. The overarching goal is to visualize the following information,

- What are the tables in this dataset.
- How are these tables connected by the foreign key relations.
- In each table, what are the fields. Moreover, what are the primary keys, foreign keys, and for which fields we know the lineage for.
- What are the lineages in the sense of both where-lineage and how-lineage.

while prioritizing the following design principles,

- **Scalability**: When the number of tables and fields gets large, displaying all info in one figure is almost impossible. Therefore, we need to design a layered UI that allows the user to "zoom in" and "zoom out" of tables and lineages.
- **Uncertainty**: It is inevitable that some info will be just above the threshold. In other words, we will have some uncertain info that needs to be delivered. Therefore, displaying the uncertainty in an intuitive manner will be very important.

Recognizing that even for small datasets it is impossible to display all the above information in one setting, we have designed a three-stage user interface, allowing the user to start from the schema-level visualization and then explore any detailed information of their choice.

Roughly speaking, the user interface is divided into three zones: the *abstract schema graph* zone, the *schema info list* zone, and the *schema relations* zone. Due to the potentially excessive number of lineage relations, we are not able to display all of them on one page. Consequently, a three-stage interface is designed. In the first stage, the *abstract schema graph* stage, users can get a brief view of the entire dataset and the relations. By selecting any of the tables, users can proceed to the second stage, the *foreign key relations* stage, where the graph expands around the selected table with more details. Finally, the users can choose any of the fields for which we know the lineage of and proceed to the third, the *column lineage* stage, where the details of this lineage are displayed.

6.1 Abstract schema graph

In the first stage, we present our users with an overview of the entire dataset and relations.

In the first (i.e. abstract schema graph) zone, we represent the dataset using a classic node-edge graph, where the nodes are the tables and the *directed* edges are the foreign key relations pointing from the parent table into the child table. We implement it as a drawable force-directed graph so that the user can adjust its layout and focus on the tables of their interest.

If this metadata is generated by our tracer, and thus is *uncertain*, we will use solid edges to represent the foreign key relations for which we have high confidence, and dotted edges for the rest. If classification information about the tables are known, we will represent them by coloring the nodes by class. Finally, the size of the tables (in terms of number of fields) will be represented by the size of the nodes. This zone will remain roughly unchanged throughout the stages.

The second (i.e. schema info list) and the third (i.e. schema relations) zone will be left blank at this stage.

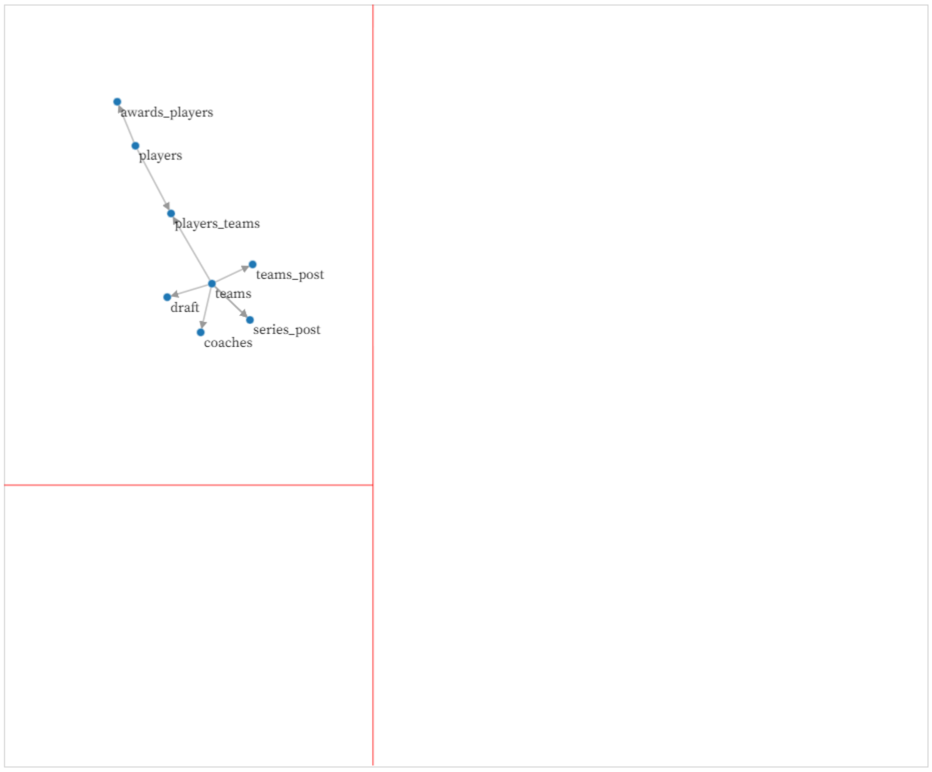

Figure 6-1: Stage 1: The relations between tables

6.2 Foreign key relations

By clicking on any of the tables in the abstract schema graph, the user can proceed to the detailed foreign key relations of this table.

In the first zone, we will make focus of this graph be our selected table, its foreign key relations, and the tables connected to it by those foreign key relations. These elements in this graph will get spotlighted.

The second zone will turn into two scrollable lists: the *child tables* list, and the *parent tables* list. The child (parent, resp.) tables list simply lists out all the child (parent, resp.) tables of this selected table. All the cells in these two lists are clickable so that the user can have the option to have that table displayed in the third zone for a further investigation of the foreign key relations.

The third zone will now be filled with the selected table (which is highlighted), and the child or parent tables that our user chose to display (by clicking them in the child or parent tables list). Each table is represented by a scrollable list of its field

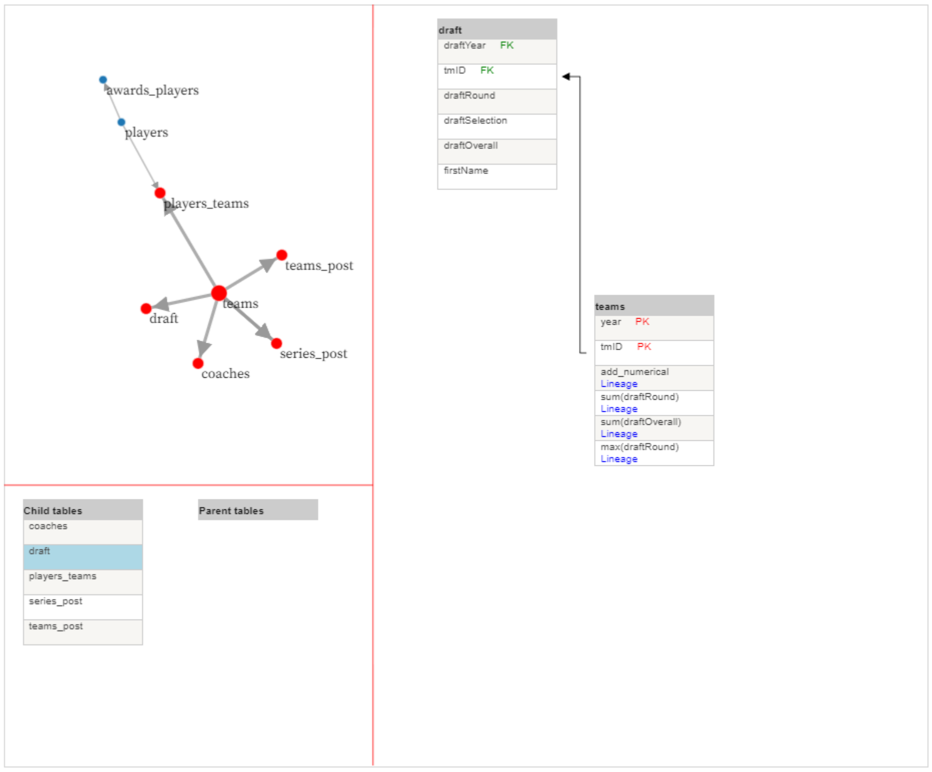

Figure 6-2: The foreign key relations related to the chosen table expands out

names. For for fields that is a primary key, foreign key, or has any lineage, there will be a sign by the field name indicating so.

6.3 Column lineage

Finally, when the user select a field to investigate its lineage, we proceed to the last stage.

The tables that contains any of the lineage columns will be added to the third zone if not already selected by the user, and the lineage fields, as well as the foreign key relations used will be highlighted. If there are too many tables in the third zone to be displayed, the UI will automatically unselect a few in the second zone to make space for the lineage tables to be added.

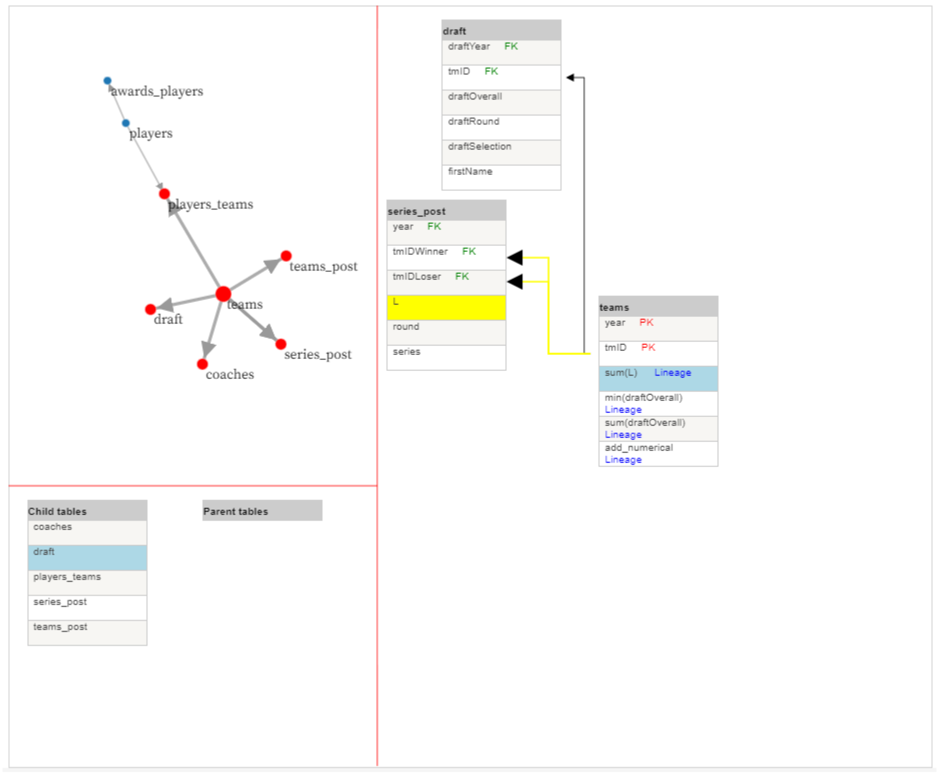

Figure 6-3: Lineage columns and foreign keys involved are highlighted

Chapter 7

Conclusion

In this paper we build upon the previous Tracer [9], a machine learning approach to the data lineage problem, and make it more precise, more general, and more intuitive.

We finish up with a complete framework with two end-to-end pipelines, that could detect both single and composite primary and foreign keys in the tables, and use them to answer both the where-lineage and how-lineage problem in relational databases without any additional input. Additionally, we develop a lineage visualization user interface that allows the user to interactively, and intuitively view the relations and lineage in the database.

With these improvements, the Tracer library now way closer to industrial level. It now has an API that is general enough to support the implementation of any solutions that solves the data lineage problem of any flavor. Its accuracy is now high enough to be a helpful reference. And due to the help of our user interface, users can now intuitively understand its results without any knowledge of the json format where our output metadata is stored in.

However, this work is far from being perfect and much can be improved. Here are a few potentially interesting lines of research:

- While the *minimal* UCC assumption we use in the primary key detection primitive achieves a relatively good balance between precision and speed, it does not work very well on small tables as the actual primary keys sometimes violate the minimality assumption. Moreover, we really can afford to explore more UCCs when the tables are small. Therefore finding a better balance between precision and speed would be interesting.

- While the current framework is pretty clean, it is isolating too many tasks. For example, the both column map detection pipelines deals with one target column at a time, and each task is completely independent. However, as some columns are generated together, their lineage will also be highly correlated, and taking this into consideration might be an interesting thing to think of.

- Following up on the previous bullet point, we also realize that the current isolated design is making the pipelines doing many repeated work when we make multiple queries to the fields in the same database. For example, the transform step is almost the same in every call. Hence, improving the API to allow efficient computation would be very nice to have.

- Though one of the Tracer's advantages is that it does not rely on any additional input, we should not turn them down if we already know something about the database. Currently, what we support is restricted to knowledge about the primary and foreign keys, for which we simply replace output the corresponding primitive with this extra knowledge. Put other knowledge about the database and our machine learning approach together would be interesting to see.

- Now that we have an user interface, which allows the user to explore the relations and lineage step-by-step, one natural question is that if we can make the Tracer compute out the lineage *on demand*. More precisely, instead having to run the Tracer in advance and put everything into a metadata file for the UI to use, we would like to connect the UI to the Tracer kernel and run the corresponding primitive/pipeline only when we have to do so. For example, we can run only the foreign key detection primitive at beginning so that we have a complete stage 1 to present. Only upon a lineage query request need we run the column map detection pipeline so that we can have that information ready for the user.

Unfortunately, the current column map detection pipeline is too slow to be run in real time. To realize this cool feature, we will need to find out a faster column map detection algorithm.

- Yet, we have only tried random forest classifier and regressor for all the machine learning tasks in the primitives. It will be interesting to see if other machine learning models will perform better.

Once the Tracer can have the above-mentioned features, we believe it will be accurate enough, fast enough, and user-friendly enough for common commercial use.

Bibliography

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: a survey. *VLDB Journal*, 24(4):557–581, 2015.

[2] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. *International Conference on Database Theory*, pages 316–330, 2001.

[3] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 539–550, 2006.

[4] Zhimin Chen, Vivek Narasayya, and Surajit Chaudhuri. Fast foreign-key detection in microsoft sql server powerpivot for excel. *Proceedings of the VLDB Endowment*, 7:1417–1428, August 2014.

[5] James Cheney, Laura Chiticariu, and Wang Chiew Tan. *Provenance in databases: Why, how, and where*, volume 1. 2007.

[6] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. *VLDB 2001 - Proceedings of 27th International Conference on Very Large Data Bases*, pages 471–480, 2001.

[7] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. *Tracing the lineage of view data in a warehousing environment*, volume 25. 2000.

[8] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflow. *SIGMOD*, 2008.

[9] Felipe Alex Hofmann. Tracer : A Machine Learning Approach to Data Lineage by. 2020.

[10] J. Huang, T. Chen, A. Doan, and J. Naughton. On the provenance of non-answers to queries over extracted data. *VLDB*, 2008.

[11] Robert Ikeda and Jennifer Widom. Data lineage: A survey. *Technical Report. Stanford InfoLab*, 2009.

[12] Lan Jiang and Felix Naumann. Holistic primary key and foreign key detection. *Journal of Intelligent Information Systems*, 54(3):439–461, 2020.

[13] Jan Motl and Oliver Schulte. The CTU Prague Relational Learning Repository. 2015. arXiv:1511.03086.

[14] Azade Nazi, Bolin Ding, Vivek Narasayya, and Surajit Chaudhuri. Efficient estimation of inclusion coefficient using hyperloglog sketches. *Proc. VLDB Endow.*, 11(10):1097–1109, jun 2018.

[15] T. Papenbrock and F. Naumann. Data-driven schema normalization. *Proceedings of the international conference on extending database technology (EDBT)*, pages 342–353, 2017.

[16] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. The synthetic data vault. pages 399–410, 2016.

[17] Jinglin Peng, Dongxiang Zhang, Jiannan Wang, and Jian Pei. AQP++: Connecting approximate query processing with aggregate precomputation for interactive analytics. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1477–1492, 2018.

[18] C. Re and D. Suciu. Approximate lineage for probabilistic databases. *VLDB*, 2008.

[19] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. *Proceedings of the ACM SIGMOD workshop on the web and databases (WebDB)*, 2009.

[20] Ljubisa Stankovic, Danilo P. Mandic, Milos Dakovic, Milos Brajovic, Bruno Scalzo, Shengxi Li, and Anthony G. Constantinides. Graph signal processing - part III: machine learning on graphs, from graph topology to applications. *CoRR*, abs/2001.00426, 2020.

[21] Mingjie Tang, Saisai Shao, Weiqing Yang, Yanbo Liang, Yongyang Yu, Bikas Saha, and Dongjoon Hyun. SAC: A system for big data lineage tracking. *Proceedings - International Conference on Data Engineering*, 2019-April:1964–1967, 2019.

[22] J. Widom. Trio: A system for data, uncertainty, and lineage. *VLDB*, 2006.

[23] Kevin Alex Zhang. Datareactor. `https://github.com/data-dev/DataReactor`, 2020.

[24] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. On Multi-Column Foreign Key Discovery. *Proceedings of the VLDB Endowment*, 3(1), 2010.