# Metacrate: Organize and Analyze Millions of Data Profiles

Sebastian Kruse[1] David Hahn[2] Marius Walter[2] Felix Naumann[1]
Hasso Plattner Institute
Prof.-Dr.-Helmert-Str. 2–3
Potsdam, Germany 14482
[1]firstname.lastname@hpi.de
[2]firstname.lastname@student.hpi.de

## ABSTRACT

Databases are one of the great success stories in IT. However, they have been continuously increasing in complexity, hampering operation, maintenance, and upgrades. To face this complexity, sophisticated methods for schema summarization, data cleaning, information integration, and many more have been devised that usually rely on data profiles, such as data statistics, signatures, and integrity constraints. Such data profiles are often extracted by automatic algorithms, which entails various problems: The profiles can be unfiltered and huge in volume; different profile types require different complex data structures; and the various profile types are not integrated with each other.

We introduce *Metacrate*, a system to store, organize, and analyze data profiles of relational databases, thereby following the proven design of databases. In particular, we (i) propose a logical and a physical data model to store all kinds of data profiles in a scalable fashion; (ii) describe an analytics layer to query, integrate, and analyze the profiles efficiently; and (iii) implement on top a library of established algorithms to serve use cases, such as schema discovery, database refactoring, and data cleaning.

## 1 THE CASE FOR DATA PROFILES

Before describing our system Metacrate, let us first make a case for the relevance of structural metadata for data management. The importance and capabilities of data in business and science have been recognized long ago. Ever since, huge amounts of data of all types and in all domains are collected. In consequence, the data and their hosting information systems have become very complex. Complex relational schemata easily comprise thousands of tables with tens of thousands of columns, rendering typical tasks, such as data cleaning or data mining extremely tedious. In fact, it is already a challenge to identify the data relevant to a certain problem in the first place [8]. It is hence necessary to automate or at least support such activities.

And indeed, such data-driven methods have been proposed in the fields of database exploration [3, 7], data cleansing [4], data anamnesis [9], schema summarization [13, 14], data discovery [8, 15], and data integration [2]. To scale to the ever-growing amounts of data, these methods operate on structural metadata rather than examining the actual data. For instance, *TF-IDF vectors* allow to detect content-wise similar schema elements [8] and schemata can be summarized using *integrity constraints* and *column statistics* [13]. We refer to such structural metadata as *data profiles*. However, all these algorithms need to manage these data profiles, which entails a host of challenges.

**Volume.** Data profiles are not necessarily small in size. The number of integrity constraints usually grows quadratically or even exponentially with the schema size. While in a single schema we might find hundreds of thousands of them [9], in a data lake it will be orders of magnitude more. In addition, one might might need to deal with a lot of different data profile types on huge or across many schemata. All this pertains to the size of the data profiles.

**Variety.** To model data profiles, a wide range of data structures is needed. For instance, a histogram of column values is modeled quite differently from a functional dependency (FD). Furthermore, there is an abundant amount of versions of each type of data profile. As an example, a recent survey reported 35 major variations of FDs [5]. We want to be able to model, store, and analyze each type. Trying to tackle this with classical data models, such as the relational one, would yield a very complex schema – and highly complex corresponding SQL queries.

**Integration.** As a third point, we emphasize the importance of integrating different data profiles. A data profile usually characterizes the underlying data w.r.t. one specific property, such as some statistic or relationship. In consequence, above mentioned data management algorithms often need to integrate different types of data profiles. Each data profile describes one or more schema elements, which in turn are interconnected in a schema graph. This naturally enables integration, but technically requires a common model for schema elements and the schema graph as well as a query interface, which is capable of expressing meaningful relationships on this common model.

Facing the challenges above, we introduce the open-source system Metacrate[1] (i) to answer the research question of how to build a scalable and flexible repository for data profiles; and (ii) to provide researchers with a tool to investigate novel data management methods based on (sophisticated combinations of) data profiles. That is, we seek to pave the way for modern data profiling methods and their complex, abundant data profiles to actual applications.

Generally speaking, Metacrate is a hub between data profiling algorithms (which produce the data profiles) and data management algorithms (which consume them). In addition to that, (i) we enable immediate storing of any kind of data profile; (ii) we expose the data

[1]https://github.com/stratosphere/metadata-ms


*CIKM'17, November 6–10, 2017, Singapore.*
 ISBN 978-1-4503-4918-5/17/11…$15.00
DOI: https://doi.org/10.1145/3132847.3133180

profiles in a well-organized, integrated fashion; (iii) we provide a query layer to explore and exploit the contents of Metacrate; (iv) and we provide a library of basic data management algorithms to quickly build workflows on top of Metacrate and also as a proof of concept for our design. In the following, we summarize the architecture of Metacrate (Section 2), exemplify its capabilities (Section 3), briefly cover related work (Section 4) and conclude (Section 5).

## 2 DESIGNING A METADATABASE

Before delving into Metacrate's design, let us briefly outline the environment it is supposed to operate in. As depicted in Figure 1, there are *data profiling tools* that produce data profiles and there are *data management tools* that need access to the data profiles to accomplish their various tasks. Metacrate acts as a hub between those two parties, thereby taking care of storing and organizing the profiles (*logical metadatabase*) and offering them via a query interface (*analytics engine*). With these main goals in mind, we describe the different components of Metacrate in the following.

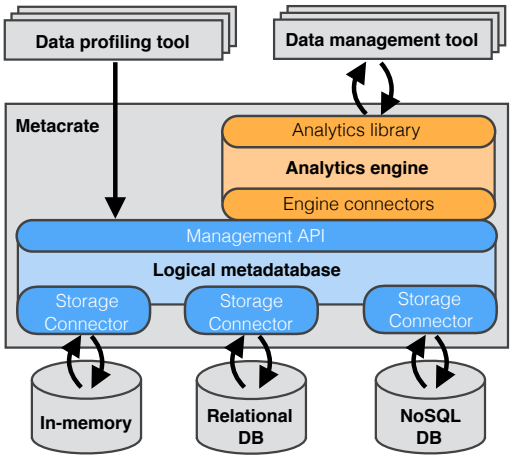



**Figure 1: Overview of Metacrate.**

### 2.1 Logical metadatabase

Metacrate provides a logical data model that is abstract enough to deal with all kinds of data profiles and at the same time is specific enough to be queried in a meaningful way. This data model can be separated into two parts, the *schema elements* and *profile collections*.

**Schema elements.** Metacrate models the elements of relational schemata, i.e., schemata, tables, and columns, in a straight-forward manner. All schema elements have a name, a description, and a location (e.g., a URI). Furthermore, they are hierarchically organized. That is, every column resides in a table, which in turn resides in a schema. As a peculiarity, Metacrate automatically assigns each schema element a *composite ID* that allows to immediately tell what kind of schema element it represents (schema, table, or column) and what the IDs of the parent and child schema elements are. Not only are the composite IDs a very compact representation of schema elements, but they also allow for basic navigation among schema elements without having to consult some physical data structure.

**Profile collection.** A profile collection is the main abstraction in Metacrate to describe data profiles. Intuitively, it is a set of data profiles of any specific type, such as histograms or functional dependencies. The only requirement is that the data profiles express their references to schema elements with the above described composite IDs. This ensures that the data profiles can be integrated with each other. In addition to the data profiles, each profile collection has a *scope* that describes for which schema elements it provides data profiles. This has two important implications: At first, it is possible to retrieve profile collections intentionally by their type of data profile and the schema element that profile is needed for (e.g., *"Retrieve the key candidates for table $T$."*). Second, it is possible to model the absence of integrity constraints (e.g., *"Does this profile collection contain a key candidate for column $T[A]$?"*). For advanced use cases, Metacrate can further keep track of custom metadata for each profile collection, such as settings of the algorithms that created the data profiles (e.g., sampling threshold or number of histogram buckets).

For external tools to interact with the above described logical model, Metacrate defines a *management API* with basic operations, in particular creating, looking up, and deleting schema elements and profile collections. It is the responsibility of *storage connectors* to execute these operations on some actual storage engine. Note that the storage connectors must be able to automatically store any kind of data profile. Otherwise, external tools would have to define how to store their custom data profiles, which contradicts our goals. As of writing this paper, Metacrate supports an in-memory storage engine, SQLite, and Apache Cassandra, leaving the choice between a light-weight or a more complex but scalable storage engine.

### 2.2 Analytics engine

Metacrate complements its storage component by an analytics engine, which can be seen as the counterpart of query engines in relational databases. However, Metacrate does not adopt a closed algebra to formulate analytics, because the various data profile types and their processing requirements are not known at design time. And even if we knew all types of data profiles in advance, we learned that formulating SQL queries, for instance, to analyze data profiles is cumbersome and error-prone. Instead, Metacrate employs a Scala-based data flow API that supports user-defined functions and whose operator set is extensible. As demonstrated in the following section, this combination allows to formulate rich analytics in a concise manner. We further provide a complementary *analytics library* with common functionality for various data management tasks (e.g., [3, 7–9, 11, 13]), so as to facilitate the development of data management tools and to allow for complex ad-hoc queries to Metacrate. Examples for this library are given in the next section.

Internally, Metacrate employs the cross-platform data processing system Rheem [1] for query execution. In detail, Metacrate feeds schema elements and profile collections to Rheem using dedicated *engine connectors*. Rheem then selects an actual data processing platform to execute the query on, e.g., Java Streams or Apache Spark. In combination with the various storage engines, Metacrate can therefore either employ light-weight components for simple scenarios or make use of distributed components to scale to complex scenarios with millions and more of data profiles.

## 3 METACRATE IN ACTION

As shown in Figure 1, Metacrate is a backend system rather than a user-facing application. That is, third-party applications can include it as a library and use it to store and query data profiles. To demonstrate our system, we use Jupyter notebooks[2] as a simple yet powerful text interface. In addition, we provide and integrate a set of specific visualizations for query results. With this tool set, we populate Metacrate with schemata and data profiles and execute various ad-hoc queries. The full notebooks for the steps outlined here can be found and re-enacted on our web page.[3]

### 3.1 Populating Metacrate

Before any analytics, Metacrate must be fed with the schemata of the datasets to be analyzed and the relevant data profiles. To load the schemata, we provide applications that can extract them from relational database management systems, SQL files, and CSV files. The creation of data profiles is not a concern of Metacrate, however. Instead, we provide integration with the data profiling tool Metanome and its many algorithms [10], such that its data profiles can be imported into Metacrate. Besides that, we provide an API for data profiling tools to write their data profiles directly into Metacrate. For the following queries, we load the schema of a dataset with 52 tables and its 10,024 basic data profiles. We have, however, also operated Metacrate with hundreds of millions of data profiles, e.g., when storing relationships among thousands of small web tables [12].

### 3.2 Querying Metacrate

Before putting a new dataset to use, it is important to first understand what data it provides, how to query it, and which potential quality problems the dataset has [9]. In the following, we demonstrate how Metacrate supports in obtaining such a data anamnesis. Assuming that Metacrate is already populated as described above, we query and visualize the data profiles in a Jupyter notebook. As a first step to approach a new dataset, we create an overview of how many columns and tuples the tables in the dataset have using Metacrate's Scala-based query API.

```
1  val schema = metacrate.getSchemaByName("schema")
2  val cols = metacrate.loadColumns(schema)
3   .map(col => (getTableId(col.id), 1))
4   .reduceByKey(_._1, (cnt1, cnt2) => (cnt1._1, cnt1._2+cnt2._2))
5  val rows = metacrate.loadProfiles[TupleCount](scope = schema)
6  val colsAndRows = cols.keyBy(_._1)
7   .keyJoin(rows.keyBy(_.getTableId))
8   .assemble { case (col, row) =>
9         (row.getTableId, col._2, row.getNumTuples) }
10  .resolveTableNames(_._1, { case (counts, table) =>
11        (table.name, counts._2, counts._3) })
```

As shown in the screenshot in Figure 2, the result can be directly visualized in the Jupyter notebook by plotting the tables on a plane using their number of columns and tuples as coordinates. This query exemplifies the most important building blocks of Metacrate queries. At first, we look up a schema by its name (Line 1). Then, we load all columns within that schema into the analytics engine (Line 2), resolve the table ID for each column (Line 3), and count how often we find each table ID (Line 4). This already gives us the number of columns per table. To obtain the numbers of tuples per table, we

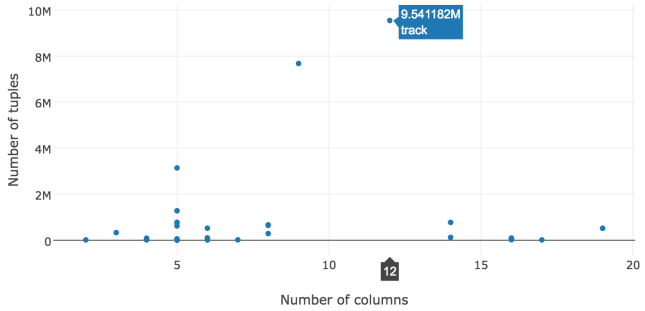



**Figure 2: Visualizations of numbers of columns and tuples for each table in a schema.**

ask Metacrate to determine and load a profile collection containing data profiles for tuple counts and with the scope encompassing our schema (Line 5). Now, we can integrate the column and tuple counts by joining on Metacrate's table IDs (Line 6–9). Eventually, we resolve the table IDs to names for presentation purposes (Line 10). To give an idea of query execution times, this and the following queries take less than a second on our example dataset, even with a light-weight storage and analytics backend.

Metacrate also allows for much more intricate analyses, in particular, when paired with the analytics library. For instance, if we do not know the foreign keys (FKs) of a schema, we can detect them from inclusion dependencies using a classifier that takes into account various different data profiles [11]. The analytics library provides this classifier and it just needs to be invoked:

```
ForeignKeyClassifier.classify(
    metacrate.loadProfiles[InclusionDependency](scope = schema),
    metacrate.loadProfiles[TupleCount](scope = schema),
    metacrate.loadProfiles[ColumnStatistics](scope = schema)
  ).store("Automatically detected FKs", scope = schema)
```

Note that in this example, we store the detected FKs in a new profile collection rather than visualizing them (Line 4). The possibility to seamlessly derive new profile collections from existing ones is very useful for complex analyses. For instance, the FKs are a valuable input to many schema summarization techniques. One such summarization technique assesses the importance of tables by analyzing their size, their join edges, and the information entropy of their columns. [13]. Metacrate's library provides this method:

```
TableImportance.assess(
  metacrate.loadProfiles[ForeignKey](scope = schema),
  metacrate.loadProfiles[TupleCount](scope = schema),
  metacrate.loadProfiles[ColumnStatistics](scope = schema)
)
```

The result is a ranking of the tables that can either be stored, visualized, or processed further. In this instance, we choose to present this ranking in a graph, where the tables are the nodes, their size corresponds to their rank, and the edges are the previously captured FKs, as shown in Figure 3.

### 3.3 More examples

Besides the above outlined data anamnesis scenario, Metacrate supports many other uses cases for that we provide further Jupyter notebooks in our demonstration. For instance, in our data discovery notebook, we employ Metacrate to discover similar columns based

[2] http://jupyter.org/
[3] https://hpi.de/naumann/projects/data-profiling-and-analytics/metacrate.html

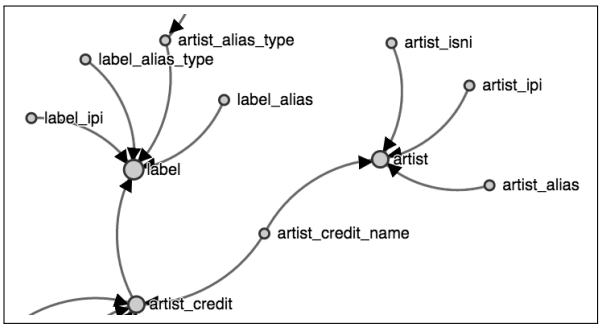
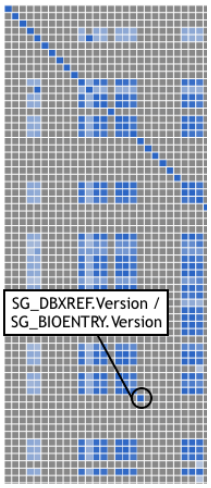



**Figure 3: Excerpt of schema visualization with automatically inferred foreign keys and importance values.**

on q-gram sketches [7] and visualize the results in a matrix plot as shown in Figure 4a. This technique can help to identify columns with data relevant to a certain task [8]. As another example, in our data cleaning notebook, we identify tables that entail suspiciously many FDs, which is an indicator for low schema quality [9]. We then sort out the FDs from those tables that are not implied by a key and explore them in a sunburst chart as shown in Figure 4b. That way, we can quickly spot opportunities for schema normalization.




**(a) Attribute similarity matrix (excerpt).**


**(b) Sunburst chart with the highlighted FD *Title, Authors, Location→CRC*.**

**Figure 4: Data discovery and data cleaning visualizations.**

## 4 RELATED WORK

Much research has been conducted on data management using data profiles and accordingly special-purpose systems have been built. Specifically, Bellman is a database exploration tool that detects, amongst others, keys and join paths [7]; Clio is a system that can manage schemata and schema mapppings [2]; and Aurum is a data discovery framework [8] to detect database tables and elements that are relevant to a concrete problem. All these tools define a fixed set of algorithms on a fixed set of data profiles. Metacrate, hence, lends itself to take care of storing the data profiles for these tools and providing a scalable execution engine for their algorithms. A further related system is Metanome, a data profiling tool [10]. While Metanome can store and visualize integrity constraints, it does not comprise further integration and analysis capabilities. However, Metanome and Metacrate are closely integrated and complement one another. Another data profiling tool proposes the SQL-like language RQL to discovery integrity constraints on datasets [6]. Again, this approach differs from Metacrate, because it is used to produce data profiles rather than integrating or analyzing them.

## 5 CONCLUSIONS

We present Metacrate, a system to store, integrate, and analyze data profiles. Built upon scalable technologies, Metacrate can handle large amounts of metadata and it is well-suited to back a broad range of data management tools as well as to execute ad-hoc analyses. Thus, Metacrate fills an open gap in the data management landscape. As future directions, we plan to improve Metacrate's integration with more data profile producers and consumers and to provide more algorithms and visualizations in the analytics library.

**Acknowledgments.** We thank Fabian Tschirschnitz, Susanne Bülow, Lawrence Benson, and Lan Chiang for the discussions on and their code contributions to Metacrate. This research was funded by the German Research Society (DFG grant no. FOR 1306).

## REFERENCES

[1] D. Agrawal, L. Ba, L. Berti-Equille, S. Chawla, A. Elmagarmid, H. Hammady, Y. Idris, Z. Kaoudi, Z. Khayyat, S. Kruse, et al. Rheem: Enabling multi-platform task execution. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 2069–2072, 2016.

[2] P. Andritsos, R. Fagin, A. Fuxman, L. M. Haas, M. A. Hernández, H. Ho, A. Kementsietsidis, P. G. Kolaitis, R. J. Miller, F. Naumann, et al. Schema management. *IEEE Data Engineering Bulletin*, 25(3):32–38, 2002.

[3] P. Andritsos, R. J. Miller, and P. Tsaparas. Information-theoretic tools for mining database structure from large data sets. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 731–742, 2004.

[4] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 746–755, 2007.

[5] L. Caruccio, V. Deufemia, and G. Polese. Relaxed functional dependencies – a survey of approaches. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(1):147–165, 2016.

[6] B. Chardin, E. Coquery, M. Pailloux, and J.-M. Petit. Rql: A query language for rule discovery in databases. *Theoretical Computer Science*, 658:357–374, 2017.

[7] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 240–251, 2002.

[8] R. C. Fernandez, Z. Abedjan, S. Madden, and M. Stonebraker. Towards large-scale data discovery: Position paper. In *International Workshop on Exploratory Search in Databases and the Web (ExploreDB)*, pages 3–5, New York, NY, USA, 2016.

[9] S. Kruse, T. Papenbrock, H. Harmouch, and F. Naumann. Data anamnesis: Admitting raw data into an organization. *IEEE Data Engineering Bulletin*, 39(2):8–20, 2016.

[10] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with Metanome. *Proc. of the VLDB Endowment*, 8(12):1860–1863, 2015.

[11] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2009.

[12] F. Tschirschnitz, T. Papenbrock, and F. Naumann. Detecting inclusion dependencies on very many tables. *ACM Transactions on Database Systems (TODS)*, 42(3):18:1–18:29, 2017.

[13] X. Yang, C. M. Procopiuc, and D. Srivastava. Summarizing relational databases. *Proc. of the VLDB Endowment*, 2(1):634–645, 2009.

[14] C. Yu and H. Jagadish. Schema summarization. In *Proc. of the VLDB Endowment*, pages 319–330, 2006.

[15] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. Automatic discovery of attributes in relational databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 109–120, 2011.