Aurum: A Data Discovery System

Raul Castro Fernandez, Ziawasch Abedjan[#], Famien Koko, Gina Yuan, Sam Madden, Michael Stonebraker

MIT <raulcf, fakoko, gyuan, madden, stonebraker>@csail.mit.edu [#] *TU Berlin* abedjan@tu-berlin.de

***Abstract*—Organizations face a *data discovery problem* when their analysts spend more time looking for relevant data than analyzing it. This problem has become commonplace in modern organizations as: i) data is stored across multiple storage systems, from databases to data lakes, to the cloud; ii) data scientists do not operate within the limits of well-defined schemas or a small number of data sources—instead, to answer complex questions they must access data spread across thousands of data sources. To address this problem, we capture relationships between datasets in an *enterprise knowledge graph* (EKG), which helps users to navigate among disparate sources. The contribution of this paper is AURUM, a system to *build*, *maintain* and *query* the EKG. To *build* the EKG, we introduce a *Two-step process* which scales to large datasets and requires only one-pass over the data, avoiding overloading the source systems. To *maintain* the EKG without re-reading all data every time, we introduce a *resource-efficient sampling signature* (RESS) method which works by only using a small sample of the data. Finally, to *query* the EKG, we introduce a collection of composable primitives, thus allowing users to define many different types of discovery queries. We describe our experience using AURUM in three corporate scenarios and do a performance evaluation of each component.**

I. INTRODUCTION

With a myriad of data sources spread across multiple heterogeneous databases, modern organizations face a *data discovery* problem. Analysts spend more time finding relevant data to answer the questions at hand than analyzing it.

For example, consider an analyst at a large drug company who is assigned the task of predicting the change in stock price of the company after a presidential election. To build her model, she decides she needs i) the company stock variations in recent elections; ii) the mentions of the company in social media channels; iii) the number of drugs about to be approved by the FDA; and iv) the current productivity of the research department. She needs to find tables with all of these attributes, as well as additional attributes that can be used to link the different drugs, years, and mentions together. The discovery challenge is to find data sources containing this information among the many thousands of tables in RDBMS, data lakes and warehouses in the organization. Doing this manually is extremely labor intensive, as the analyst has to browse a large number of files and tables to find those that might match, and then try to find key attributes that can be used to join these tables together. Our work with data analysts at a number of organizations, including Merck, British Telecom, the City of New York, and our university suggests that they all struggle with such problems on a daily basis.

Although some companies have built in-house solutions to help their analysts find files in their data lakes and databases, these systems are engineered around a customized index and search algorithms to solve a predetermined set of use cases; they do not solve the more general discovery problem presented here. For example, systems such as Goods [1], Infogather [2] and Octopus [3] do not provide a flexible way to support different discovery queries than those they were designed to solve. Analysts want to find datasets according to different criteria, e.g., datasets with a specific schema, similar datasets of one of reference, joinable datasets, etc. To avoid the productivity bottleneck of discovering new data sources, we need a more general structure that organizes the data sources, represents their relationships, and allows flexible querying; otherwise data discovery becomes a bottleneck that hampers productivity within organizations.

This paper is concerned with the design and implementation of a discovery system that permits people to flexibly find relevant data through properties of the datasets or syntactic relationships amongst them, such as similarity of content or schemas, or the existence of primary key/foreign key (PK/FK) links. The key requirements of such a system are: 1) it must work with large volumes of heterogeneous data; 2) it must incrementally adapt to continuously evolving data; and 3) it must provide querying flexibility to support the varied discovery needs of users.

We represent relationships between the datasets in a data structure we call the **enterprise knowledge graph (EKG)**, which analysts use to solve their discovery problems. The contribution of this paper is AURUM, a system to *build*, *maintain* and *query* the EKG:

• **Build:** Building the EKG requires performing an all-pairs comparison ($O(n^2)$ time) between all datasets for each of the relationships we wish to find. With a large number of sources this quadratic cost is infeasible because of its time and because of the need to read external sources multiple times, thus incurring IO overhead in the external systems. We introduce a **two-step process** that consists of a *profiler* that summarizes all data from sources into space-efficient signatures by reading data only once, and a *graph builder*, which finds syntactic relationships such as similarities, PK/FK candidates in $O(n)$ using sketching techniques.

• **Maintain:** Because data is always changing (req. 2), we must *maintain* the EKG in a scalable manner: we introduce a **resource-efficient signature sampling (RESS)** method that avoids repeatedly re-reading source data, while still keeping the EKG up-to-date.

• **Query:** Finally, because discovery needs are varied we cannot pre-define queries (req. 3). Instead, we introduce a **source retrieval query language (SRQL)** based on a set of discovery primitives that compose arbitrarily, allowing users to

express complex discovery queries. We implement SRQL on a RDBMS-based execution engine, augmenting it with a graph index, **G-Index**, which helps speeding up expensive discovery path queries. In addition, the query engine allows users to express different ranking criteria for results.

We have used AURUM to solve use cases within companies. We report the results of surveys conducted by our users, as well as performance benchmarks.

II. AURUM OVERVIEW

Here we describe the data discovery problem (II-A), our approach to solve it based on the enterprise knowledge graph (EKG) (II-B), and finish with an overview of AURUM in II-C.

A. The Data Discovery Problem

Consider the set of structured data sources T within an organization. A data source is structured when it is comprised of a set of columns, which may or may not have a label or attribute name, i.e., tabular data such as CSV files or tables in databases, file systems, data lakes, and other sources for which a schema can be obtained. Each data source has a set of attributes or properties, P, and has different relationships, R, with other data sources. Note that R may not be explicitly defined in the data; AURUM discovers these relationships while building the EKG. The sources, T, have different degrees of quality, from highly curated and *clean* to poorly designed schemas with ambiguous names and many missing values. Data sources can be human- or machine-generated, and may be intended for human or machine consumption, with schema names ranging from highly descriptive to completely obscure.

Finding relevant data. Data discovery is the process of finding a subset S of *relevant* data sources among the many sources in the organization, T, which are relevant to a user-supplied discovery query. A data source is relevant to a query when it satisfies a constraint or selection criteria C, of which we consider two types.

Property constraints select $s \in S$ based on P. For example, selecting columns with unique values, or columns with a string in the schema name, which are all properties of the data. For example, the analyst who is building the stock change prediction model may start with a search for tables that include metrics of relevance, such as stock prices and mentions in social media (*schema similarity*).

Relationship constraint select s based on R and another set of sources, S_{ref}, which is an input parameter to the selection criteria. In other words, a source, s, is relevant to the user because it is related to another source in S_{ref}. For example, the analyst may be interested in finding similar datasets to the ones found so far to make sure no information is missing (*content similarity*). Or, having already found a handful of relevant datasets, the analyst may want to find a join path to join them together (*a primary-key/foreign-key (PK/FK) candidate*).

The challenge. One key challenge of data discovery is that users' needs vary and change over time, meaning that a single discovery workflow is insufficient to address the breadth of discovery tasks, i.e., each user defines relevance according to a very wide set of selection criteria, C. Hence, rather than trying to prescribe a particular set of discovery or relevance queries, we aim to make it possible to answer a wide range of discovery queries by capturing a variety of relationships between datasets, and allowing users to flexibly query those relationships, which we represent in the enterprise knowledge graph (EKG):

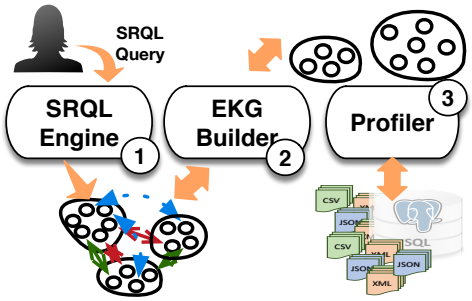

Fig. 1. Aurum architecture overview

B. Enterprise Knowledge Graph (EKG)

The EKG is a hypergraph where nodes represent columns of the data sources, edges represent relationships between two nodes, and hyperedges connect any number of nodes that are hierarchically related, such as columns of the same table, or tables of the same source (RDBMS, lake, etc.). In addition, each edge has a weight to indicate the strength of the relationship, e.g., how similar two nodes are, or how confident we are of them being a PK/FK. The nodes represent columns and not individual values because it is more scalable to build and does not compromise the discovery needs we have identified: although finer granularity (e.g., columns values) is supported by AURUM, our experience suggests that the additional storage costs are not offset by the gain in expressiveness. The edges connect only two columns to express binary relationships such as content similarity—the values of the columns are similar—schema similarity, the attribute names being similar, or the existence of a PK/FK between them. Hyperedges are necessary to allow users query at different granularities, e.g., columns and tables. A discovery query, then, filters the nodes and edges in the EKG according to property and relation constraints.

C. Building, Maintaining, Querying

Our contribution is AURUM, a system to *build*, *maintain* and *query* the EKG; the three processes are depicted in Fig. 1.

Build the EKG. (Section III-A) A key goal of building the EKG is to minimize the time-consuming access to underlying data sources. To build the EKG in as few passes over the data as possible, we separate its construction into two stages (the **Two-step process**), a *signature-building* stage and a *relationship-building* stage. During the first stage, a **profiler** (box 3 in the figure) produces signatures that represent the underlying data and stores them in profiles. These profiles are built by *reading the data only-once*—we use sketches as explained in the next section—and form the nodes of the EKG. During the *relationship-building* stage, a **graph builder** component (box 2 in the figure) computes the relationships

between the columns of the data by using the signatures in the profiles, avoiding reading the data again. To avoid the expensive all-pairs comparison ($O(n^2)$) process, the graph builder computes the edges of the EKG in linear time by transforming the problem into one of approximate nearest neighbors ($O(n)$).

Maintain the EKG. (Section III-B) When data changes, we want to keep the EKG up-to-date without recomputing from scratch and while minimizing the access to the underlying data sources. Our **resource efficient signature sampling (RESS)** determines what data has changed by reading only a sample of the data. AURUM triggers then a re-indexing of these sources, computing their new relationships and updating the EKG.

Query the EKG. (Section IV) To query the EKG, users express property and relation constraints to obtain the data they need through a set of composable primitives. Although the primitives can be implemented in any graph language such as SPARQL, Cypher, etc., we identified a few additional requirements that justified a separate implementation. Ranking results, access to provenance, debugging, and human-latency path query execution all justify the design of an execution engine for SRQL, which is the name of the data discovery language we implement in AURUM. SRQL consists of the discovery primitives, a *discovery results set (DRS)* to support the above functionality, as well as an index, **G-Index**, which we use to speed up path queries.

III. BUILDING AND MAINTAINING THE EKG

In this section, we explain how to *build* the EKG (III-A) and how to *maintain* it (III-B). Last we discuss how the EKG evolves in the broader context of an enterprise in III-C.

Why is building the EKG difficult? To build the EKG, a naive process would need to read a column in a table of a data source, and then iterate over all the other columns of all the other tables, each time comparing both the values—to compute *content similarity* and *PK/FK candidates*—as well as the names, to compute *schema similarity*. This process is clearly infeasible because it requires multiple access rounds to the data (heavy I/O) and a quadratic number of operations in the number of columns (heavy CPU). So even when the resulting graph may fit in memory, its construction is expensive because it involves an all-pairs comparison for each relationship of interest (we consider here content similarity, schema similarity, and candidate PK/FK), and because it incurs too much IO overhead to the external source systems.

The key to building an EKG efficiently and minimizing IO overhead is to avoid the all-pairs comparison. We achieve this by using a *Two-step process*. During the first step the process uses sketching techniques to summarize the data in one-pass only, reducing IO overhead. During the second step, it employs locality-sensitive hashing (LSH) techniques to transform the all-pairs problem into an approximate nearest neighbor one.

A. Two-Step Process: Building the EKG

Our two-step process divides the EKG building process into a *signature-building* stage, and a *relationship-building* one.

Algorithm 1: Two-Step Process to build the EKG

input : C, collection of columns from all databases and tables,
store, a store to save the profiles,
sim_thresholds, a list of similarity thresholds
output: $H = (V, E)$, where V is the nodes of the EKG.
$E = (e_r | r \in R)$, with R the set of syntactic relationships. H is the EKG

```
// Step 1: signature-building stage
1  for c ∈ C do
2      profile← compute_profile(c);
3      store[c]← profile;
4      add_node(H, profile); //A node is represented with its profile
// Step 2: relationship-building stage
5  index_name← create_indexer(sim_thresholds);
6  index_content← create_indexer(sim_thresholds);
7  for p∈ store.profiles do
8      name← p.name;
9      signature← p.signature;
10     uniqueness_ratio← p.uniqueness_ratio;
11     index_name.index(p, name);
12     index_content.index(p, signature);
13 for p ∈ store.profiles do
14     attr_sim_candidates = index_name.query(p);
15     add_edges(H, attr_sim_candidates, type="attr_similarity");
16     content_sim_candidates = index_content.query(p);
17     add_edges(H, content_sim_candidates, "content_similarity");
18     for candidate ∈ content_sim_candidates do
19         if pkfk_candidate = is_pkfk_candidate(candidate,
           uniqueness_ratio) then
20             add_edges(H, pkfk_candidate, "pkfk_candidate");
21 return H;
```

Both are shown in Algorithm 1. The signature-building stage (step 1) is carried out by a **profiler**, which summarizes the data into *profiles* that contain signatures with enough information to compute the syntactic relationships. The relationship-building stage (step 2) is carried out by the **graph builder**, which uses the *profiles* to compute the syntactic relationships that populate the EKG in $O(n)$ time.

1) Signature-Building Stage: Profiler: The profiler summarizes each data source (column in tables) into a *profile*, which maintains information such as content sketches (MinHash), cardinality, data distributions, types, etc. The profiler must be: i) scalable, because of the sheer amount of data it must analyze; and ii) IO-efficient, because it interacts directly with data sources that may be used by other production systems. Our profiler builds the profiles by reading data only-once and scales to large amounts of data by using data parallelism.

The profiler consists of a pipeline of stages which are wrapped up in a function `compute_profile()` (see line 2 in Algorithm 1). Each stage in the pipeline computes a part of the profile for each column. There are two special stages, *source* and *sink*. The *source* is used to read data from files, RDBMS, etc., and provides the input to the `compute_profile()` function. The *sink* stores the computed profiles (line 3), so that they are accessible to the graph builder during the second stage of the building process.

Those operations that are shared across profiler stages, such as hashing, are placed at the beginning of the pipeline to save computation downstream.

Parallelism. The profiler exploits parallelism at three different levels. First, multiple profilers can run in a distributed fashion, each processing a disjoint set of data sources. Second, the profiler supports *inter-task parallelism*, assigning an instance

of the processing pipeline to one thread and running multiple such threads on each machine. It also supports *intra-task parallelism* with a single pipeline instance per machine, but with multiple threads assigned to one pipeline or stage.

Task grain. A natural choice is to assign one pipeline per data source (i.e., file or table). This allows for efficient sequential reads from the data source. In addition, it reduces the complexity of the profiler because a task corresponds directly to a table. Unfortunately, this design leads to under-utilization of processing resources, because table sizes in real datasets are highly skewed. This leads to long-running stragglers that hamper full utilization of the available hardware.

A more efficient approach can be achieved with finer-grain tasks, by partitioning data at the column level. With this approach, the I/O threads that read the data are now decoupled from the processing stages through a queue. A *task creator* component partitions each column into subtasks, that are processed by the processing stages. Each thread keeps track of the partial profile computed for each subtask. When the thread finishes processing a subtask, it sends the partial profile to a *merger* component, which is responsible for merging all the subtasks for each column and creating the final profile. This design is robust to data skew and achieves full utilization of the available hardware, so it is our preferred approach.

2) Relationship-building Stage: Graph Builder: The graph builder computes the syntactic relationships between the columns in the database using the profiles created in the signature-building stage. The idea is that a relationship between two profiles reflects a relationship between the underlying data. The main problem the graph builder solves is to compute such relationships in $O(n)$, with n the number of columns, avoiding the cost of an all-pairs comparison, for which it uses locality-sensitive hashing [4].

Building EKG syntactic relationships. We want to create a relationship between two nodes of the EKG (e.g., columns) if their signatures are *similar* when indexed in LSH, i.e., they hash into the same bucket. Signatures can be similar according to two criteria, Jaccard similarity—for which we use the MinHash signature—or cosine similarity, with a TF-IDF signature. We can then differentiate both similarity criteria as two different relationships in the EKG, such as MinHash and TF-IDF, but we will only talk about one to simplify the presentation (`signature` in algorithm 1 see line 9). As these signatures are computed in the profiling stage, the graph builder can access them directly from the store; line 7). None of the relationships computed at this point are binary, they all have some associated score that indicates how similar two columns are, how likely two columns are a PK/FK, etc. Such score, the *relationship strength* allows users to refine their queries by, for example, using a threshold that all results must surpass. We explain next how we compute such score.

Relationship strength score. Each edge in the EKG has a weight that indicates the strength of the relationship, e.g., how much a column is similar to another. LSH, however, will return elements that are similar based on a pre-determined, fixed threshold. To obtain an approximation of the relationship strength when using LSH, we create an indexer structure that internally indexes the same signature multiple times, in multiple LSH indexes configured with different similarity thresholds, and configured to balance the probability of false positives and negatives. We create such objects in lines 5 and 6 of Algorithm 1 passing as parameters a list of configurable thresholds. Both name and content signatures are indexed in lines 11 and 12 respectively. The next step is then to iterate a second time over the profiles (line 13), querying the indexer objects (lines 14 and 16) and retrieving candidates. When querying the indexer objects, they, internally, will iterate over the multiple LSH indexes, starting with the one configured with the highest similarity threshold. As the indexer objects return candidates together with their weights (the similarity threshold of the LSH index), they keep the candidates in internal state, so that they can filter them out when they appear in LSH indexes with smaller thresholds. Thus the indexer avoids duplicates and obtains an approximate weight for the relationship, which is then materialized in the EKG (lines 15 and 17).

Alternative LSH techniques: The above description uses a collection of traditional LSH indexes. Since it must use several indexes for different thresholds, the storage needs increase with the number of thresholds desired. Although in our deployments storage overhead is not a problem, there are alternative LSH indexes which help with reducing such overhead, such as LSHForest [5] and MultiProbe-LSH [6]. Integrating these indexes in AURUM is straightforward.

Approximate PK/FK relationships. To compute PK/FK-candidate relationships, we first identify whether a column is an approximate key, which we do by measuring the ratio of unique values divided by the number of total values. The profiler computes this *uniqueness_ratio* ratio during the first stage, and the graph builder retrieves it (line 10 of the algorithm). A true primary key will have a uniqueness_ratio of 1, but because the profiler uses the Hyperloglog sketch to compute this, we may have a small error rate, so we simply check that the value is close to 1. When we retrieve the content-similar candidates, we iterate over them (line 18) and check whether they are PK/FK candidates (line 19), in which case we add the candidate PK/FK relationship to the EKG (20). This method is similar to PowerPivot's[7] approach, and works well in practice, as shown in the evaluation.

B. RESS: Maintaining the EKG

We discuss how to maintain the EKG when data changes without re-reading all data from scratch:

Incrementally maintaining profiler. Consider a column, c, for which we compute a MinHash signature, m_t at time t. At time $t+1$, we can compute the signature m_{t+1}. If m_{t+1} is not identical to m_t, i.e., if the Jaccard similarity is not 1, we know the data has changed, and, by obtaining the magnitude of the change, i.e., the distance, $1 - JS(m_t, m_{t+1})$ between the signatures, we can determine whether it is necessary to

Algorithm 2: Resource-Efficient Signature Sampling

input : C, collection of columns from all databases,
store, a store to retrieve the profiles,
γ, the maximum magnitude of change tolerable before triggering a request to recompute profile
output: M, where $M \in C$ is the set of columns that need a new profile

1 **for** $c \in C$ **do**
2 $s \leftarrow$ random_sample(c);
3 $x \leftarrow$ store[c].num_unique_values;
4 $JS_{max} = |s|/x$;
5 $s_{mh} \leftarrow$ minhash(s);
6 $x_{mh} \leftarrow$ store[c].content_signature;
7 JS' = jaccard_sim(s_{mh}, x_{mh});
8 $\delta = 1 - JS'/JS_{max}$;
9 **if** $\delta > \gamma$ **then**
10 trigger_recompute(M, c);
11 **return** M;

recompute the signature and update the EKG, by checking if this difference is larger than a given threshold. The challenge of maintaining the EKG is to detect such changes without computing m_{t+1}, because computing this means we need to read the entire column again.

Using our Resource Efficient Signature Sampling (RESS) method, given in Algorithm 2, we compute an estimate of the magnitude of change by using only a sample, s, of c, instead of the entire data. To do this, we assume that the data has not changed, i.e., that MinHashes m_t are identical to those at m_{t+1}, indicating that the Jaccard similarity of the old and new columns is 1. We then try to reject this assumption. The Jaccard similarity of two columns, or sets, is expressed as their intersection divided by their union. We observe that, if we have the cardinalities of the columns available (we have the number of unique values at time t and because we assume the columns has not changed, the number of unique values remains unchanged at $t+1$) then we know that the maximum Jaccard similarity, JS_{max}, can be expressed as:

$$JS = \frac{|x \cap y|}{|x \cup y|}; \quad JS_{max} = \frac{min(|x|, |y|)}{max(|x|, |y|)}$$

Note that when $|x| = |y|$, then the intersection and the union are the same, so $JS_{max} = 1$. Using this observation, and the fact that the number of unique values of a sample, s, is always lower than that of the original data, c, we expect that, after sampling, JS_{max} is given by $JS_{max} = |s|/x$, where x is the number of unique values of c, which can be obtained from the previously computed profile. This expression gives us the maximum Jaccard similarity based on the sample.

The RESS algorithm operates as follows. We obtain a sample of c (line 2), proportional to the percentage of the number of unique values for this column (line 3). Then we obtain the JS_{max} under the assumption the data did not change using the observation above, see line 4. We now compute the MinHash signature of s, s_{mh} (line 5) and retrieve the MinHash signature of c (line 6), which is already available because the profiler computes it to estimate the *content-similarity* relationship. The algorithm then computes the new estimate of Jaccard similarity (line 7), and scales it to a distance between 0 and 1 (line 8) so that it can be compared with the maximum tolerable magnitude of change, γ and input parameter to the algorithm. If the estimated change is above γ (line 9) then the algorithm indicates that it is necessary to recompute c, in line 10.

When sampling, RESS relies on the sampling mechanism of the underlying source, although when possible, it is easy to add a connector with sampling features.

C. Additional EKG Features

In addition to changing data, other factors contribute to the continuous evolution of the EKG. We summarize them next:

•**Offline component:** Syntactic relationships computed with approximate methods, such as the ones explained above, will contain false positives. It is possible to run an offline component that checks the exact measurements for the relationships existing in the EKG, therefore filtering out the false positives and increasing the precision of the existing ones. Because this component must check only a subset of all pairs of columns, its IO demand is lower. There are two more scenarios which benefit from the offline component: i) users require exact relationships, i.e., no loss in recall; ii) users want a syntactic relationship which is not compatible with LSH, and therefore must be computed exhaustively.

•**User Feedback:** Along with the SRQL query language (presented next) we incorporate a set of *metadata* utilities that allow users to annotate existing relationships and nodes in the EKG, as well as manually creating and removing relationships from the EKG. In practice, users cannot directly modify the EKG (unless given specific permissions), but their annotations are visible to other users. Users with the right permissions can later materialize users' feedback, thus modifying the EKG.

IV. Querying the EKG with SRQL

In this section we present the SRQL language IV-A and its implementation in IV-B.

A. The Source Retrieval Query Language

In this section we give a formal definition of the language primitives of SRQL in IV-A1, followed by a running example in IV-A2 and concluding with an in-depth discussion of the Discovery Result Set (DRS) (in IV-A3).

1) SRQL Language: The SRQL language consists of two concepts: *discoverable elements* (DE), which are the nodes of the EKG, and *discovery primitives* (DP), which are functions used to find DEs. DPs are divided into two groups: pure and lookup. Pure DPs are closed on the DE set, that is, they receive a set of DEs as input and return a set of DEs as output. Lookup primitives also return a set of DEs, but receive a string as input. A SRQL query typically consists of a combination of pure DPs, sometimes preceded by lookup DPs. To deal with important aspects of the SRQL language such as result navigation and ranking, as well as debugging, the SRQL language implementation uses the concept of a *discovery result set* (DRS), which wraps up the output DEs with additional metadata used to support additional functionality.

2) Running Example: Here we show how an analyst can use SRQL in the different stages of discovery via an example. Suppose a bonds analyst finds out an inconsistent value for *profits*, reported in last quarter's sales report. The analyst wants to verify the hypothesis that the error was caused by using two redundant copies of a table that diverged over time.

1. Broad search of related sources. Retrieve tables with columns that contain the keywords *"Sales", "Profits"*:

```
results = schemaSearch("Sales", "Profits")
```

The `schemaSearch` primitive finds columns across all databases that contain the input keywords. Other **lookup primitives** are available to search, e.g., values. All SRQL queries return a **discovery result set (DRS)** object, which permits inspecting the results as *columns*, *tables*, or *paths*. When inspecting the *tables* in the result the analyst sees a *"Profits_3Term_17"* and decides to inspect the schema:

```
results = columns("Profits_3Term_17")
```

`columns` show the schema of the table, which is *Tx_ID, product, qty, profit*; it seems relevant. Other primitives would allow to inspect the *tables*, instead of the *columns*. The next step is to find *similar* tables to *"Profits_3Term_17"*, that may have caused the error in the report.

2. Searching for similar content. SRQL provides several primitives to explore the different syntactic relationships:

```
contentSim(table: str) =
    drs = columns(table)
    return jaccardContentSim(drs) OR cosineSim(drs)
           OR rangeSim(drs)
results = contentSim("Profits_3Term_17")
```

For composability, all SRQL primitives accept *columns* unless explicitly changed. The first operation in `contentSim` is to apply `columns` to the input table. Next, each of the three primitives, `jaccardContent`, `cosineContent`, and `rangeSim` apply the primitive logic to each input column individually; the results per column are then combined. Then, an `OR` primitive is used to combine the results of the primitives—the analyst is interested in any relationship that indicates similarity. SRQL has other primitives to combine results with other semantics, such as `AND` and `SET-DIFFERENCE`. All primitives operate at the column level. If what is desired is to obtain intersection or union of tables, then the input `DRS` can be configured to provide this, as we show later.

In summary, the query returns columns from the different sources in the organization that are similar—according to any of the primitives used—to any column in the input table. To narrow down the result, the analyst refines the query to find tables with similar content, and also similar schemas:

```
match(columns: drs) = contentSim(columns)
                      AND attributeNameSim(columns)
table = "Profits3Term2017"
results = match(columns(table))
```

The primitive `attributeNameSim` returns columns with similar names to the input columns. Combining it with the `AND` primitive yields the intersection of both queries: i.e., only columns that are content—and attributeName—similar.

By adding the `attributeNameSim` primitive, the results improve, but there are still too many: presumably there are many sources in the databases with columns that have content and attribute names similar to *Tx_ID*—which seems to be an ID—but with different semantics.

3. Incorporating additional intuition. Semantic ambiguity is a fact of life in data discovery and integration applications. SRQL users can write queries that limit ambiguity in some cases. For example, the analyst may know some values in the original table that should also appear in a potential copy, such as *LIBORAdjustedBond*, which is one product the bond analyst knows about. This value should appear in tables that refer to products, differentiating them from other tables that also contain a *Tx_ID* field. This additional intuition can be used to refine the previous query with the aim of reducing the ambiguity of the results, as follows:

```
matchAndDisambig(columns: drs, value: str) =
    table(match(columns)) AND table(valueSearch(value))
value = "LIBORAdjustedBond"
res = matchAndDisambig(columns("Profits_3Term_17"), value)
```

The analyst wants tables with similar columns to the input table and also contain the value *"LIBORAdjustedBond"*. To not require both conditions to apply to a single column, it is possible to configure `AND` to take the intersection of the results at the table label and not at the columns level, by using the primitive `table` on the inputs to `AND`. Unfortunately, many results match the query, making it difficult to navigate the results. Fortunately, the analyst can rank the results:

4. Ranking the results. To find candidate copies, the analyst wants to see the tables that contain the highest number of columns similar to a column in the input table. Fortunately, `DRS` objects (which are explained in the next section) can be sorted according to a number of ranking options (developers can add new options), and are ranked by a default policy. In this case, however, the analyst chooses to rank the results of the last query according to **coverage**, i.e., tables having the largest overlap with the source table. This makes the cause of the misleading value in profits apparent: a file named *"ProfitsSecondTerm17.csv"*—with the same columns and very similar content—appears in the output. Maybe a user copied the table from the database to do some analysis, and subsequent updates made this copy of the table diverge. The wrong table was used for the report, leading to the initial error.

5. Recovering missed information. After more inspection, the analyst realizes that the file contains values that do not appear in the original table: removing the copy would potentially delete data from the organization. The analyst thinks that if the data in the file comes from other sources, it may be possible to recover that information from the origin sources that contribute values to these tables. To verify this, the analyst wants to know whether there is a join path between the *table* and the *file*, with the following SRQL query:

```
t1 = "Profits_3Term_17", t2 = "ProfitsThirdTerm17.csv"
res = path(table(t1), table(t2), pkfkCandidate(), hops=3)
```

The `path` primitive finds paths between the first and second arguments if they are transitively connected with the relation represented by the primitive in the third argument. In this case, if both tables are connected with a candidate PK/FK relationship—that exposes candidate primary-key/foreign key relations—in fewer than 3 hops, then the path will appear in

the output DRS. Similar to the OR and AND primitives, path can interpret the input DRS as columns or tables. In this case both input DRS objects are configured as *tables*. To access the paths, the DRS provides a third convenient view that we mentioned above, paths, that contains all possible 3-hop paths between t1 and t2 connected with PK/FK candidates.

Without SRQL, the analyst would have spent large amounts of time asking other people and manually exploring the myriad of sources. AURUM simplifies the discovery task and executes the SRQL queries in human-scale latencies by using the EKG. We describe in more detail the role of the DRS object next:

3) Discovery Result Set: If we trace the execution of a SRQL query in the EKG, we obtain a directed graph from the input DEs of the query to the output DEs. The discovery result set (DRS) keeps track of this *provenance graph*, which we use to answer SRQL queries, navigate the results with different granularities, and to rank results on-the-fly. Although in theory the provenance graph could be as large as the EKG, in practice its size is much smaller. For example, none of the queries we used in our evaluation section required more than 500MB of memory (section V-B). We explain next how we build and use the provenance graph:

Recording Provenance. Recording provenance is crucial for ranking query results. It consists of storing how the SRQL query traversed the EKG from the input DEs to produce the output data. The provenance is thus a directed subgraph of the EKG that indicates a path from input to output DEs.

The provenance graph has source nodes, which are the input DEs, or *virtual nodes* to indicate the input parameters of a lookup primitive, and sink nodes, which correspond to the output. Every discovery primitive modifies this graph deterministically. Primitives that query relationships add downstream DEs to existing nodes in the provenance graph—the neighbors that satisfy the constraint or the DEs within the same hierarchy. Path queries add entire chains of nodes, one per DE involved in the path. Finally, set primitives create joins in the graph: they take 2 input DEs as input and produce one.

Obtaining SRQL query results. Relationship primitives are answered by returning the *sink* nodes of the provenance graph. Path primitives must return paths that connect a set of source DEs with target DEs; these can be answered by obtaining those paths from the DRS. The DRS knows the granularity at which the DEs it represents must be interpreted; columns and tables primitives change this granularity.

Explaining SRQL results. The provenance graph is sufficient to answer **why** and **how** provenance questions, as defined in the literature [8]. It is possible to identify what input DEs determine the existence of a DE in the output by asking a *why* query on the DRS. Asking *how* will return the entire path, from source DEs to sink DEs, along with the primitives that led to its existence. This feature is practically useful for debugging purposes—when writing a new SRQL query—and helpful to shed light on the results of complex SRQL queries.

B. SRQL Execution Engine

We have two requirements for our SRQL execution engine. First, queries must run efficiently. Despite having access to the EKG—which means relationships are pre-computed and materialized in the graph—SRQL queries become expensive when they involve path queries or large combinations of other primitives. We explain our solution, based on a **G-Index** in section IV-B1. Second, the engine must support ranking results in a flexible manner. For that, we explain our **on-the-fly** ranking in section IV-B2.

1) G-Index: Building the SRQL execution engine on top of a relational database has the benefit of having the indexing capabilities already available to answer edge primitives, as well as the chance to store properties on both edges and nodes by using the relational model. Executing path primitives, however, can become slow. A path primitive in SRQL returns all the paths between a source and target DE. The results of the primitive can be limited to paths with a configurable maximum number of hops. These kinds of graph traversal queries can be expressed in a relational database via self-joins on a relation with *source* and *target* node attributes. In practice, however, even when both attributes have indexes, the query can become slow, so we have built G-Index to speed up path primitives.

The G-Index is a space-efficient in-memory representation of the EKG. It is a map, int->(int, bitset), in which the nodes of the EKG are represented with their id as int, and map to tuples of (int, bitset), in which the first component is the id of the target node, and the bitset stores the edge information. Specifically, each bit with value 1 in the bitset indicates a different kind of relationship between the nodes—the EKG presented in this paper requires then only 3 bits for the 3 relationships we explain.

The G-Index can be constructed incrementally along with the EKG. In case of a shutdown, it can be efficiently reconstructed from scratch, as we report in the evaluation section. Last, no concurrent writes and reads occur in the structure, which is frozen to reads while being constructed. To support evolution of EKG, we can answer queries using an old copy of EKG while building the new one.

2) On-the-Fly Ranking: Every SRQL query returns a set of DEs, which are wrapped up in a DRS object, as we have seen above. Before presenting the results to users, the DRS ranks the results based on a default policy. Despite the default behavior, we want to allow power users to select the most appropriate ranking criteria for the discovery problem they have, and we want to avoid re-running queries when users change their criteria, making it easy to explore different ranking strategies. In general, a ranking strategy sorts the output results by some measure of relevance with respect to some input query or data. In the context of SRQL, this means a ranking criteria sorts the output DEs based on the input DEs. The provenance graph is key to enable this functionality. Because it has the results, it enables decoupling query execution from result ranking. That is, by default any SRQL query returns an unordered set of DEs, and then a ranking strategy ranks them according to a policy; this is analogous

Use Case	Usefulness	Time Savings
Lookup primitives		
Combiner primitives		
Similarity primitives		
Path primitives		
Ranking		

TABLE I
SURVEY RESULTS. LEFT TO RIGHT: SCORE OF 0 TO 5

to the ORDER BY operator in SQL. The provenance graph is sufficient to express ranking strategies that we have found in practice, and it serves as a framework to write new strategies when necessary. Next, we describe two criteria for ranking results:

Examples of Ranking Strategies. The EKG edges, and by extension the provenance graph edges, have a weight that indicates the strength of each relationship. One useful way of ranking results is to use these weights, traversing the provenance graph from sinks to sources, and aggregating the weights we find; when reaching a join, we select the highest weight of the upstream connections, and follow that path. We call this **certainty ranking**, and we set it up as the default strategy. It is trivial to aggregate these scores per table to rank tables instead of columns.

A more sophisticated ranking strategy for SRQL queries that return tables as output is **coverage ranking**, where output tables of a query are ranked with respect to the input DEs. One example is showing which output table has the highest number of columns that are similar to columns in an input table. For example, if we find two tables that are content-similar to a table of interest, we may be interested in knowing how many of the columns of the newly found tables are similar to the input, and rank the results based on that. The provenance graph keeps the necessary data to allow this.

V. EVALUATION

To evaluate the value of AURUM for data discovery, we deployed the system with three collaborators with real data discovery problems and surveyed their experience as reported in section V-A. We then discuss the performance of AURUM at querying (section V-B), building (section V-C) and maintaining (section V-D) the EKG.

A. Aurum: Data Discovery in Context

We deployed AURUM in 3 corporate scenarios. In each scenario we had to find an aligned interest with the company, develop a use case scenario based on the companies' discovery needs and then perform the actual deployment, fixing the problems that each new environment brings. Finally, we conducted a survey with the help of a few of the users who had used the system consistently and who had suffered directly the discovery problems of the company before. First we describe the discovery use cases that AURUM helped to solve (V-A1) and then we report the results of the survey in which we asked them to evaluate the value of AURUM (V-A3).

1) Real Use Cases: **University DWH.** DWH analysts solve customers' questions by finding relevant data and manually creating views. We used AURUM to automate this process. We deployed AURUM on an Amazon EC2 instance from which the analysts had access to the system.

The analysts highlighted the benefits of AURUM to quickly navigate data and inspect schemas. The analysts noted a novel use case of detecting copied tables in the DWH. Copies occur because the DWH is built through several ETL processes that integrate data from various places in the university, leading to duplication, which in turn may lead to errors. By writing a SRQL query that finds tables with high content and schema similarity, we found four tables that were candidate copies.

Pharma company. Our pharmaceutical user ran AURUM on 6 different public chemical databases (ChEMBL [9], DrugBank [10], BindingDB [11], Chebi [12], KiDatabase) and an internal database accessed by over 1000 analysts. These databases contain domain-specific information on diverse chemical entities. The users remarked that AURUM helped them identify known entities in unknown databases, by using the relationships in the EKG. Although initially only interested in finding join paths, they became later interested in writing variations of *schema complement* in SRQL, again, with the aim of learning about the schema of other unfamiliar databases.

Sustainability. The sustainability team of our university wanted to enrich their own datasets. They used AURUM to navigate a 250GB repository of CSV files from *data.gov*. They were interested in finding data about building energy consumption, water use intensity, density of LEED buildings, power plant CO2 emissions, and waste diversion rates—what is diverted from landfill and incinerators. In a 1.5h hands-on session during which we assisted them to use AURUM, they found 2 relevant sources for each of the use cases. When they found semantic ambiguity—data.gov is full of information about diverse appliances efficiency—they refined their SRQL queries to avoid such ambiguities, e.g., instead of looking for *energy efficiency*, combine the query with a requirement for *KwH* in the schema name, avoiding spurious results. As they explored the datasets and learned about how different agencies referred to certain efficiency metrics, they refined their SRQL queries, leading to more accurate datasets. They complemented the handful of relevant datasets they had found with similar content, which they found using AURUM. They highlighted the benefits of quickly exploring and refining SRQL queries.

2) Survey Results: We surveyed 4 users (2 from pharma and 1 from each of the other deployments) that used AURUM. We asked them to rate: i) the *usefulness* of AURUM's discovery primitives from 0 (not useful) to 5 (very useful); and, ii) time savings—how much time they saved with AURUM in comparison with the manual approach from 0 (no savings) to 5 (very significant savings).

The results are in table I. All users found the different

EKG nodes/edges	Neo4J (avg/95/99)			Janusgraph (avg/95/99)			Virtuoso (avg/95/99)			PostgreSQL (avg/95/99)			SRQL G-Index (avg/95/99)		
Query	**Edge**	**2-hop**	**5-hop**	**Edge**	**2-hop**	**5-hop**	**Edge**	**2-hop**	**5-hop**	**Edge**	**2-hop**	**5-hop**	**Edge**	**2-hop**	**5-hop**
1M/20M	8.1 10.5 22.7	4.4 7.8 8.7	9,600 13,200 14,200	2.3. 4.0 8.0	97.4 194.0 228.4	✗	4.2 2.3 8.4	221.3 1,100 1,700	3,900 5600 6000	0.8 1.1 1.5	33 45 48	18,300 19,700 19,900	0.7 1.1 1.5	2.1 2.1 2.1	10,900 11,300 11,300
100K/20M	8.1 10.4 23.5	99 128 145	✗	4.2 7.0 9.0	1,500 1,900 2,100	✗	1.0 1.8 2.7	179.5 714.0 1,100	✗	0.3 1.4 1.9	108 127 130	✗	0.3 1.4 1.8	112 113 113	18,100 18,400 18,400
100K/80M	9.4 13.3 23	1,400 3,100 4,400	✗	3.6 6.0 10.0	8,500 9,800 10,400	✗	2.5 5.2 9.7	753.9 778.4 778.7	✗	0.4 2.5 3	3,200 3,800 3,900	✗	0.4 2.4 3	1,000 1,100 1,100	64,400 66,100 66,300
10K/20M	8.3 10.2 23.7	7,000 10,900 11,600	✗	6.1 10.0 28.0	✗	✗	1.6 2.5 8.0	4,800 5,300 5,600	✗	0.2 0.2 0.9	12,300 13,000 13,300	✗	0.1 0.2 0.9	3,300 3,400 3,400	13,800 13,900 13,900
DWH	8.7 15.4 32.9	32.9 41.4 42.0	42.2 138.1 206.9	1.6 3.0 5.0	129.4 299.8 399.2	✗	0.7 1.1 1.4	6.7 7.8 8.1	18.5 62.6 85.1	0.4 0.9 1.2	11.6 13.6 14.1	73,500 92,500 95,400	.02 .03 .04	.02 .06 .08	.01 .01 .01
ChEMBL	8.3 11.0 17.5	10.8 14.5 14.8	1,500 3,000 3,100	0.7 2.0 3.0	23.4 62.4 80	340 860 1,000	0.5 0.9 1.1	3.7 5.2 5.8	25.6 43.8 45.0	0.2 0.4 0.4	2.4 4.0 4.7	72.2 144.7 148.8	0.2 0.4 0.4	.02 .05 .07	.02 .04 .04
MassData	12.0 14.0 20.1	64 85 90	✗	1.6 3.0 6.0	330.7 802.5 947.6	✗	1.0 2.4 3.4	25.7 49.6 65.6	✗	1.8 2.2 2.5	199.2 251.1 279.7	✗	0.7 1.5 2	.02 .06 .08	.01 .01 .01

TABLE II
EDGE AND PATH PRIMITIVE PROCESSING TIMES OF DIFFERENT SYSTEMS ON DIFFERENT GRAPHS. TIMES IN MILLISECONDS.

primitives useful to express their discovery queries. Not all users used all of them, though. For example, in the case of sustainability, no path primitives were necessary. In that case, however, ranking was crucial, as the number of sources was larger than in other use cases. In general, we had very positive experience working with the users and AURUM.

How do they solve discovery today? We asked them to describe how they solve data discovery problems today, choosing from 6 options (they could select more than one choice). The options were *asking other people or more senior employees where is the data* (3 votes), *writing SQL queries and use database metadata to understand the content* (3 votes), *Manual work: writing scripts, visual inspection* (3 votes), *Use some commercial tool to navigate the sources* (1 vote), *use some open source tool to navigate the sources* (2 votes), and *other* (0 votes).

3) End-to-end deployment: In the next section, we evaluate the performance of each of Aurum's stages in isolation. In all the deployments we have described here, most time is spent during the profiling stage, i.e., the first step of our Two-step process. In addition to being the most computationally expensive process, reading from source systems is often limited to avoid cluttering other workloads.

Conclusion. Finally, when asked how likely they are to use AURUM in their organization, three users gave a 4 and one of gave a 5. We are currently working with them to achieve a deeper integration in their environments, with the hope of discover more details and related problems. The experiments so far with AURUM **helped us to confirm that AURUM is useful for varied discovery needs**.

B. Query: SRQL Execution Engine Performance

Next we analyze how the G-Index speeds up discovery path primitives. Our goal is to show that G-Index on top of a relational database is sufficient to cope with the discovery needs of querying the EKG, and that specialized graph systems that run general graph workloads are not needed. We compare with other specialized graph systems to support this claim. Also note that specialized graph systems do not support some of our requirements, e.g., provenance. We evaluate the performance of G-Index with 3 queries: a 2-hop and 5-hop path query, and an edge primitive. We compare the runtime with other 4 state-of-the-art systems.

Datasets. We use 7 different EKGs, three of them (DWH, CHEMBL and MASSDATA) are built on real datasets, and the other 4 are generated following the Erdos-Renyi model for random graphs, which we use to expose the limits of the systems. DWH is a public subset of the data warehouse of our university with around 200 tables. CHEMBL is a well known public chemical database [9] with around 70 tables. MASSDATA is a repository of open governmental data for the state of Massachusetts, with around 300 tables.

Setup. We compare 5 systems in total, including ours. Before running a system, we spent time tuning it until we achieved the best results. Neo4J [13] is a state-of-the-art single-node graph processing system. We configured the database with indexes to speed up the different queries. All the queries are written in Cypher, Neo4J's language. JanusGraph [14] is an open source version of Titan, a distributed graph database system. We configured it to run on top of Cassandra, because it was the storage backend that yielded the best results. For JanusGraph, we used Gremlin as the query language. Virtuoso is a well-known database for processing RDF data. In the case of Virtuoso, we serialized the EKG as a repository of triples in RDF format, as this was the format in which we found Virtuoso to perform the best. We query the system using SPARQL. Last, we run the queries on standalone PostgreSQL configured with indexes. All systems run on a MacBook Pro with 8GB of memory.

Note that, SPARQL in Virtuoso uses an abstraction called *property graph*—not to be confused with the property graph model [15]—to express path queries. This abstraction does not support path primitives defined on more than one edge type: this means the system is not suitable for SRQL. We included

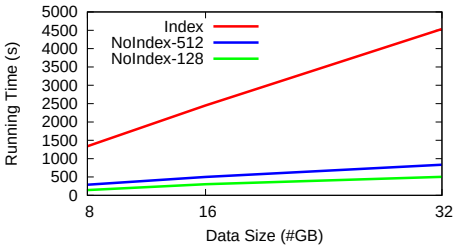

Fig. 2. Profiler runtimes for 3 configurations

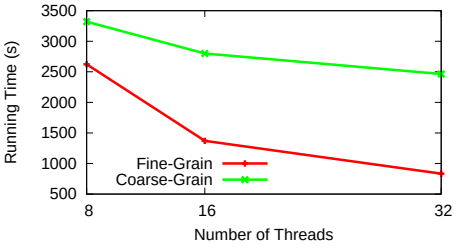

Fig. 3. Profiler runtime with two grain sizes

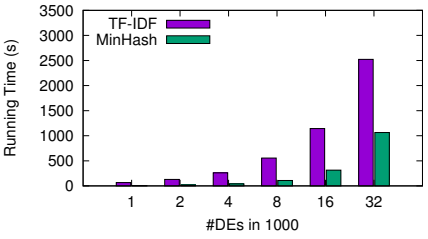

Fig. 4. Graph builder runtimes with increase DEs

the results for completeness.

Results. We report average, 95, and 99 percentile runtime over 10 runs for each query in Table II. A cross (✗) indicates a system run out of memory or 2 min were elapsed.

Edge primitives SRQL G-Index is 2 orders of magnitude faster than Neo4J, and 1 order of magnitude faster than Virtuoso—edge queries are efficient in the underlying PostgreSQL. Although the absolute runtime is low (10ms), the runtime will multiply with the total number of input DEs, so it is important to execute these efficiently.

The EKG structure has limited impact on the performance of the edge primitives, but significantly impacts the runtime of path primitives: we ran two discovery path queries, with a maximum of 2 and 5 hops and present the results next.

2-Hop Path primitives On the real data EKGs, G-Index is 20x faster than Virtuoso, which is the second fastest system. On the synthetic EKGs, G-Index performs better, but on-par with Virtuoso and Neo4J. G-Index speeds up short path queries over PostgreSQL (by several orders of magnitude on real EKGs), while PostgreSQL is competitive with Neo4J and even with Virtuoso in the case of the real EKGs.

5-Hop Path primitives The biggest impact of G-Index is shown with the more complex 5-hop path queries. None of the other systems completed all the queries over all the EKGs, while G-Index finished them successfully. When the other systems finished the queries, the G-Index performed significantly better. For example, for the real data EKGs, G-Index is 2 orders of magnitude faster. For the synthetic ones the time is similar to Virtuoso and Neo4J for the least connected EKG, but these systems did not finish the query for any of the other synthetic EKGs.

Loading time and memory consumption We measured the loading times for the different systems. This times include creating the necessary indexes, for example, creating the G-Index in the case of our solution. PostgresSQL achieves the fastest loading times, and G-Index adds negligible overhead on top when created incrementally—note we do not need transactional writes to G-Index because the EKG is written first and read later, so the process is very efficient. Including the incremental creation of G-Index, PostgreSQL takes 42s to load the smallest EKG and 160s to load the largest one. The second fastest system, Neo4J, took 134s and 669s, respectively. Virtuoso, which creates many indexes to speed up query execution took 162s and 2234s, which is 14x more time than PostgreSQL. G-Index is an in-memory structure so after a shutdown, the index must be recomputed. The time to recompute G-Index from scratch—as opposed to incrementally—was of 20s for the largest EKG. Finally, G-Index's memory footprint is on the order of 200MB in the case of the smallest EKG and below 2GB in the case of the largest. This makes sense, the index, written in C++, must only keep an efficient bitmap to represent the connections between nodes, which are itself represented with integers.

Conclusion. With G-Index, discovery primitives, including path queries, execute within human-scale latencies.

C. Build: Two-step Process

We measure the performance of the profiler and graph builder to build the EKG (V-C1) as well as the quality of the links produced (V-C2).

1) Profiler and Graph Builder efficiency: To evaluate the profiler, we measure the time it takes to index varying amounts of data. We ran 3 configurations of the profiler. Using the Index configuration, we ran the profiler and also indexed the data into elasticsearch—necessary to answer lookup primitives. With NoIndex, we only measure profiling time; we ran this configuration with a MinHash signature of 512 (NoIndex-512) and 128 permutations (NoIndex-128). We used the DWH dataset, which we replicated to achieve the target input data size, while keeping data skewness. We ran these experiments on a machine with 32 cores and SSD disk.

The results in Fig. 2 show that all modes scale linearly with the input data size. The Index mode takes the longest time because the indexing speed of elasticsearch becomes the processing bottleneck. To factor out this bottleneck and evaluate the performance of the profiler we built, we run the two NoIndex configurations. In the case of NoIndex-512, the limiting factor is computing the signature; when we run NoIndex-128, which is 4x cheaper to compute than NoIndex-512 due to the reduced number of hashing operations, we become limited by the data deserialization routines we use. Nevertheless, these results are very positive. For example, the profiler took only slightly above 1.5 hours to build signatures for the 250GB of data.gov that we used in our deployment with the sustainability team.

What about task granularity? We want to evaluate the benefits of using finer-grain tasks (Fine-Grain) in comparison with creating a task per table (Coarse-Grain). Fig. 3 shows how, for the same dataset used above, Fine-Grain achieves better performance than Coarse-Grain; real datasets are skewed and this justifies the design of our profiler.

Graph Builder. To evaluate the performance of the graph builder, we measure the runtime as a function of the number

of input profiles with the *content-similarity* relationship (DEs in the x axis).

The results in Fig. 4 show linear growth of the graph builder when using both MinHash and TF-IDF—as expected. To obtain the TF-IDF signature, the graph builder must read data from disk, so as to calculate the IDF across the profiles—which explains its higher runtime. MinHash signatures are created as a part of profiling process, hence its lower runtime.

2) Accuracy Microbenchmarks: **Approximate PK/FK relationship.** We measure the precision and recall of candidate PK/FK generated by AURUM in 4 different datasets for which we had ground truth. The results are in Fig. 7.

For TPC-H, our approach achieved similar quality as reported by PowerPivot [7] and better than reported by *Randomness* [16]. Similarly, our approach achieved good results for the FDB dataset, which consists of financial information from our university. Chembl [9] contains a larger number of declared PK/FKs in their schema. In this case, we used the technique presented by PowerPivot [7] of filtering PK/FKs by those fields with similar attribute names, which was straightforward to express in a simple SRQL query with 2 primitives. After further inspection of the results, we found that some of the candidate PK/FKs found by AURUM are in fact semantically reasonable joins, even though they are not included in the official PK/FK constraints defined by the schema (confirmed by an expert). When they were false positives, users found it easy to ignore. Finally, we evaluated our method on a dataset used by one of our collaborators, and accessed by hundreds of analysts daily (Pharma). In this case both the precision and recall of the relationships we found was above 90%, despite the large number of relationships available.

Content similarity relationship. We compare the quality of the content similarity relationships generated with MinHash and LSH with ground truth obtained by running an exact all-pairs similarity method. We use 3 datasets (we did not use Pharma because we could not run the expensive all-pairs method in the production environment). The results are in table III. We do not show the results for for TPC-H and FDB, for which our approach achieved 100% precision and recall. Instead, we add the MASSDATA dataset and show results for MinHash with 512 and 128 permutations. For all three datasets we achieve a high recall, with 100% in the case of ChEMBL and 88% in the case of MASSDATA. The precision for MASSDATA is also high, while for DWH and CHEMBL we achieved a precision of around 50%—note that we are measuring exactly those relationships above the configured 0.7 similarity threshold—we verified that many of the false positives actually have a high similarity score.

Even when we use the cheaper-to-compute MinHash signature with 128 permutations, we still achieve high quality results. This is important, because computing the MinHash signature takes a considerable time during the profiling stage as we have seen in the previous section.

In conclusion, our two-step process to build the EKG scales linearly with the input data by means of the profiler and graph builder, which computes signatures and relationships in linear time. Despite using approximate methods, the quality of the relationships in the EKG suggests a reasonable approximation trade-off against the expensive exact method.

Dataset	#Sim. Pairs actual ($\delta = 0.7$)	Precision/ Recall K=512	Precision/ Recall K=128
DWH	8872	56%/92%	66%/71%
Chembl	11	57%/100%	42%/100%
MassData	28297	90%/88%	89%/91%

TABLE III
ACCURACY RESULTS FOR CONTENT SIMILARITY

D. Maintain: RESS

In our final experiment, we evaluate the efficiency of our resource-efficient signature sampling (RESS) method. Our goal is to understand whether we can efficiently detect data changes, so that we can keep the EKG up-to-date at low cost.
Synthetic experiment: In the first experiment (Fig. 5), we created 100 columns with 1K values drawn from a uniform distribution. We then change 50 of the columns by uniformly altering in a varying degree the size of the changes (shown in the X axis). We then run RESS and report the precision and recall of the method when compared with the real changes. Both metrics are higher when the data has changed more often. Even when the data does not change much, the recall is very high, and the precision means that only a few additional data must be re-read to verify. To further evaluate RESS, we run a second experiment with real data.
Real data: We obtained two versions of the CHEMBL database. We used version 21 (which we have used in the rest of this evaluation) and version 22. ChEMBL updates their database version every few months, adding and removing new data to the existing relations, so we built an EKG using ChEMBL21 and then changed the underlying data to ChEMBL22, therefore simulating data evolution. At this point, we run EKG to identify the magnitude of change for the columns in the dataset. We then run RESS with different sample sizes and compute the absolute error between the estimated magnitude of change and the true change—which we compute and use as ground truth (420 of the underlying columns changed). The results in Fig. 6 show the error in the x axis and the percentage of columns in the y axis.

RESS identifies 90% of the modified datasets by reading only 10% of the original data, with a maximum error of less than 15%. When using 25% of the data the error reduces to 10%. The baseline error is of 5%. For the rest of sample sizes shown in the figure, RESS behaves as expected: the bigger the sample size the lower the error.

VI. RELATED WORK

Enterprise search systems. LinkedIn has open sourced WhereHows [17]. The Apache foundation offers Atlas [18] and Google has Goods [1]. All these systems have a strong emphasis on recording the lifecycle of datasets through lineage information, which is orthogonal to the problem we solve

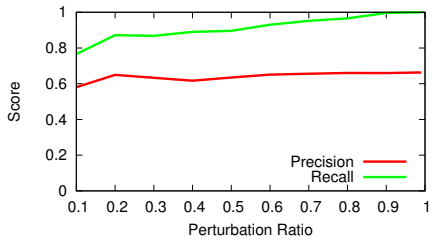

Fig. 5. Precision and Recall as a function of change with 10% sample

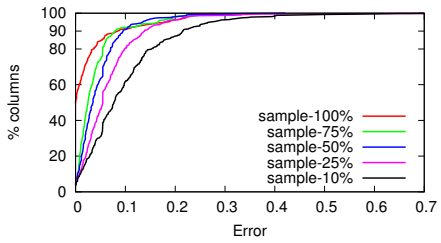

Fig. 6. Absolute error for different sample sizes.

Dataset	#FKs actual	Precision/Recall
TPC-H	11	100%/81%
FDB	7	100%/100%
Chembl	85	35%/67%
Pharma	431	100%/91%

Fig. 7. Effectiveness of the PK/FK discovery

in this paper. These systems provide some subset of search or discovery features, such as finding similar datasets, or inspecting existing ones, but none of them permit users to change the discovery query on-demand.

Exploratory systems. Octopus [3] and Infogather [2] solve the problem of *schema complement*. Finding related tables [19] focuses on finding tables that are candidates for joins, similar to schema complement, or candidates for union. Finally, in [20] the authors build a system with a particular set of operations that solve use cases in the context of oceanographic data. All the above systems rely on custom indexes, specific to each use case. In contrast, we take a more general approach to the data discovery problem, building an EKG that we use to flexibly answer varied discovery needs, and that can then be used as the basis for new, more complex relationships.

IR and databases. There is a plethora of systems to perform keyword search in RDBMS [21], [22], [23], [24], [25], [26]. Most of these systems can be seen as a constrained implementation of our discovery primitives, on an EKG with DE granularity of values. Then, most of these systems propose to connect the results through a Steiner tree, so that they can rank them appropriately. Since we maintain a provenance graph of the execution of SRQL queries, we could find a Steiner tree in such graph, therefore offering similar functionality. We have built AURUM for more general discovery cases that cannot be solved with keyword search alone.

Other links for the EKG. We designed Aurum to support relationships useful to discover and connect datasets within organizations. We discussed some syntactic relationships but left many others out. For example, work on finding functional dependencies [27], as well as correlation among columns [28], [29] is complementary to our work.

VII. CONCLUSION

We presented AURUM[1], a data discovery system that builds, maintains and allows users to query an enterprise knowledge graph to solve diverse discovery needs. We have used AURUM with several companies. Many others are in the process of onboarding the technology, and we have already identified some lines of future work, such as the inclusion of other kinds of non data-driven relations. We see AURUM as a stepping stone towards addressing the substantial challenges that the modern flood of data present in large organizations.

[1]https://github.com/mitdbg/aurum-datadiscovery

Acknowledgements: We thank the many users and industrial collaborators that participated in the user studies we conducted for the evaluation.

REFERENCES

[1] A. Halevy *et al.*, "Goods: Organizing Google's Datasets," in *SIGMOD*, 2016.
[2] M. Yakout *et al.*, "InfoGather: Entity Augmentation and Attribute Discovery by Holistic Matching with Web Tables," in *SIGMOD*, 2012.
[3] M. J. Cafarella, A. Halevy *et al.*, "Data Integration for the Relational Web," *VLDB*, 2009.
[4] A. Andoni and P. Indyk, "Near-optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions," *Commun. ACM*, 2008.
[5] M. Bawa, T. Condie, and P. Ganesan, "LSH Forest: Self-tuning Indexes for Similarity Search," in *WWW*, 2005.
[6] Q. Lv, W. Josephson *et al.*, "Multi-probe LSH: Efficient Indexing for High-dimensional Similarity Search," in *VLDB*, 2007.
[7] Z. Chen, V. Narasayya *et al.*, "Fast Foreign-key Detection in Microsoft SQL Server PowerPivot for Excel," *VLDB*, 2014.
[8] J. Cheney, L. Chiticariu *et al.*, "Provenance in Databases: Why, How, and Where," *Found. Trends databases*, 2009.
[9] E. L. Willighagen, A. Waagmeester *et al.*, "The chembl database as linked open data," *Journal of Cheminformatics*, 2013.
[10] D. S. Wishart, C. Knox *et al.*, "DrugBank: a knowledgebase for drugs, drug actions and drug targets." *Nucleic Acids Research*, 2008.
[11] T. Liu *et al.*, "BindingDB: a web-accessible database of experimentally determined proteinligand binding affinities," *NAR*, 2007.
[12] J. Hastings *et al.*, "The ChEBI reference database and ontology for biologically relevant chemistry: enhancements for 2013," *NAR*, 2012.
[13] Neo4J, "Neo4J," https://neo4j.com, 2017.
[14] JanusGraph, "JanusGraph: Distributed graph database," https://janusgraph.org, 2017.
[15] W. Sun, A. Fokoue *et al.*, "SQLGraph: An Efficient Relational-Based Property Graph Store," in *SIGMOD*, 2015.
[16] M. Zhang, M. Hadjieleftheriou *et al.*, "On multi-column foreign key discovery," *VLDB*, 2010.
[17] LinkedIn, "WhereHows: A Data Discovery and Lineage Portal," 2016.
[18] Apache, "ATLAS: Data Governance and Metadata framework for Hadoop," http://atlas.incubator.apache.org, 2016.
[19] D. Sarma *et al.*, "Finding Related Tables," in *SIGMOD*, 2012.
[20] V. M. Megler *et al.*, "Are Data Sets Like Documents?: Evaluating Similarity-Based Ranked Search over Scientific Data," *TKDE*, 2015.
[21] K. Chen, J. Madhavan, and A. Halevy, "Exploring schema repositories with schemr," 2009.
[22] S. Agrawal, S. Chaudhuri *et al.*, "DBXplorer: Enabling Keyword Search over Relational Databases," in *SIGMOD*, 2002.
[23] B. Aditya, G. Bhalotia *et al.*, "BANKS: Browsing and Keyword Searching in Relational Databases," in *VLDB*, 2002.
[24] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases," ser. VLDB, 2002.
[25] A. Simitsis, G. Koutrika *et al.*, "Précis: from unstructured keywords as queries to structured databases as answers," *VLDB*, 2008.
[26] Z. Abedjan *et al.*, "DataXFormer: A robust data transformation system," in *ICDE*, 2016.
[27] T. Bleifu, S. Kruse *et al.*, "Efficient denial constraint discovery with hydra," *VLDB*, 2017.
[28] I. F. Ilyas, V. Markl *et al.*, "CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies," in *SIGMOD*, 2004.
[29] H. V. Nguyen, E. Müller *et al.*, "Detecting Correlated Columns in Relational Databases with Mixed Data Types," in *SSDBM*, 2014.