KAYAK: A Framework for Just-in-Time Data Preparation in a Data Lake

Antonio Maccioni[1(✉)] and Riccardo Torlone[2]

[1] Collective[i], New York City, USA
amaccioni@collectivei.com
[2] Università Roma Tre, Rome, Italy
torlone@dia.uniroma3.it

Abstract. A data lake is a loosely-structured collection of data at large scale that is usually fed with almost no requirement of data quality. This approach aims at eliminating any human effort before the actual exploitation of data, but the problem is only delayed since preparing and querying a data lake is usually a hard task. We address this problem by introducing KAYAK, a framework that helps data scientists in the definition and optimization of pipelines of data preparation. Since in many cases approximations of the results, which can be computed rapidly, are enough informative, KAYAK allows the users to specify their needs in terms of accuracy over performance and produces previews of the outputs satisfying such requirement. In this way, the pipeline is executed much faster and the process of data preparation is shortened. We discuss the design choices of KAYAK including execution strategies, optimization techniques, scheduling of operations, and metadata management. With a set of preliminary experiments, we show that the approach is effective and scales well with the number of datasets in the data lake.

Keywords: Data lake · Data preparation · Big data · Schema-on-read

1 Introduction

In traditional business intelligence, activities such as modeling, extracting, cleaning, and transforming data are necessary but they also make the data analysis an endless process. In response to that, big data-driven organizations are adopting an agile strategy that dismisses any pre-processing before the actual exploitation of data. This is done by maintaining a repository, called "data lake", for storing any kind of raw data in its native format. A dataset in the lake is a file, either collected from internal applications (e.g., logs or user-generated data) or from external sources (e.g., open data), that is directly stored on a (distributed) file system without going through an ETL process.

This work has been supported by the European Commission under the grant agreement number 774571 – Project PANTHEON.

J. Krogstie and H. A. Reijers (Eds.): CAiSE 2018, LNCS 10816, pp. 474–489, 2018.
https://doi.org/10.1007/978-3-319-91563-0_29

Unfortunately, reducing the engineering effort upfront just delays the traditional issues of data management since this approach does not eliminate the need of, e.g., data quality and schema understanding. Therefore, a long process of *data-preparation* (a.k.a. *data wrangling*) is required before any meaningful analysis can be performed [6,13,23]. This process typically consists of pipelines of operations such as: source and feature selection, exploratory analysis, data profiling, data summarization, and data curation. A number of state-of-the-art applications can support these activities, including: (i) data and metadata catalogs, for selecting the appropriate datasets [1,5,10,12]; (ii) tools for full-text indexing, for providing keyword search and other advanced search capabilities [9,10]; (iii) data profilers, for collecting meta-information from datasets [6,9,16]; (iv) processing engines like Spark [24] in conjunction with data science notebooks such as Jupyter[1] or Zeppelin[2], for executing the analysis and visualize the results. In such scenario, data preparation is an involved, fragmented and time-consuming process, thus preventing analysis on-the-fly over the lake.

In this framework, we propose a system, called KAYAK, supporting data scientists in the definition, execution and, most importantly, optimization of data preparation pipelines in a data lake[3]. With KAYAK data scientists can: (i) define pipelines composed by *primitives* implementing common data preparation activities and (ii) specify, for each primitive, their time *tolerance* in waiting for the result. This represents a mechanism to trade-off between performance and accuracy of primitives' results. Indeed, these primitives involve hard-to-scale algorithms that prevent analysis on-the-fly over new datasets [14,16–18], but often an approximate result is informative enough to move forward to the next action in the pipeline, with no need to wait for an exact result. KAYAK takes into account the tolerances by producing quick *previews* of primitive's results, when necessary. In this way, the pipelines are executed much faster and the time for data preparation is shortened.

On the practical side, each primitive in a pipeline is made of a series of tasks implementing built-in, atomic operations of data preparation. Each task can be computed incrementally via a number of steps, each of which can return previews to the user. KAYAK orchestrates the overall execution process by scheduling and computing the various steps of a pipeline according to several optimization strategies that balance the accuracy of results with the given time constraints. Another important feature of KAYAK is its ability to collect automatically, in a metadata catalog, different relationships among datasets, which can be used later to implement advanced analytics. The catalog also keeps the profile of each dataset and provides a high-level view of the content of the data lake.

We have verified the effectiveness of our approach with the first implementation of KAYAK and tested its scalability when the number and size of datasets in the lake increase.

[1] http://jupyter.org/.
[2] https://zeppelin.apache.org/.
[3] A demo of KAYAK has been shown in [15].

Currently, the system is used in operation within the PANTHEON project whose aim is supporting precision farming: it is charge of collecting and managing heterogeneous data coming from terrestrial and aerial robots moving in plantations, as well as from ground sensors and weather stations located nearby.

The rest of the paper is organized as follows. In Sect. 2 we provide an overview of our approach. In Sect. 3 we illustrate how KAYAK models data and, in Sect. 4, we describe our strategy for executing primitives. In Sect. 5 we discuss our experimental results and, in Sect. 6, some related works. Finally, in Sect. 7, we draw some conclusions and sketch future works.

2 Overview of the Approach

This section provides an overview, from a high-level perspective, of the main features of the system.

Pipelines, Primitives and Tasks. KAYAK is a framework that lies between users/applications and the file system where data is stored. It exposes a series of *primitives* for data preparation, some of which are reported in Table 1. For example, a data scientist can use primitive P_5 to find interesting ways to access a dataset. Each primitive is composed of a sequence of *tasks* that are reused across primitives (e.g., P_6 is split into T_b, T_c, T_w, while primitive P_7 uses T_c only). A task is atomic and consists of operations that can be executed either directly within KAYAK or involving external tools [16,24], as shown in Table 2.

Table 1. Example of primitives in KAYAK.

ID	NAME	TASKS
P_1	Insert dataset	T_a, T_p
P_2	Delete dataset	T_s
P_3	Search dataset	T_o
P_4	Complete profiling	T_a, T_b, T_c, T_d, T_m
P_5	Get recommendation	T_b, T_c, T_d, T_q
P_6	Find related dataset	T_b, T_c, T_w
P_7	Compute joinability	T_c
P_8	Compute k-means	T_g, T_n
P_9	Outlier detection	T_h, T_p, T_r, T_u
...	...	...

Table 2. Example of tasks in KAYAK.

ID	DESCRIPTION
T_a	Basic profiling of a dataset
T_b	Statistical profiling of a dataset
T_c	Compute Joinability of a dataset
T_d	Compute Affinity between two datasets
T_e	Find inclusion dependencies
T_f	Compute joinability between two datasets
...	...

A *pipeline* is a composition of primitives that is representable as a DAG (direct acyclic graph). As an example, Fig. 1 shows a pipeline composed by six primitives: P_1 inserts a new dataset in the lake and P_4 generates a profiles for it; then P_8 processes the dataset with a machine learning algorithm while P_7 identifies possible relationships with another dataset. Eventually, P_5 produces a query recommendation. Users can mark the primitives in the pipeline with a

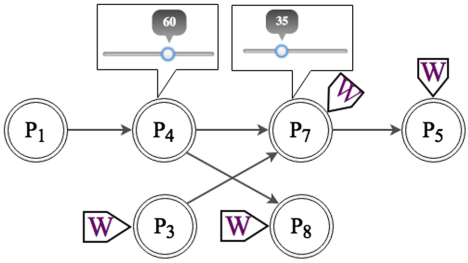


Fig. 1. Example of a data preparation pipeline.

watchpoint to inspect some intermediate result. For example, in Fig. 1 we have defined a watchpoint on P_7, P_5, P_3, and P_8.

Note that, we assume here that the output of a primitive is not directly used as input of the following primitive; they rather communicate indirectly by storing data in the lake or metadata in a catalog. Primitives can be *synchronous* when they do not allow the execution of a subsequent primitive before its completion, or *asynchronous*, when can be invoked and executed concurrently.

Metadata Management. KAYAK extracts metadata from datasets explicitly, with ad-hoc primitives (e.g., P_4), or implicitly, when a primitive needs some metadata and uses the corresponding profiling task (e.g., T_a in P_1). Metadata are organized in a set of predefined attributes and are stored in a catalog so that they can be accessed by any task.

Specifically, KAYAK collects *intra-dataset* and *inter-dataset* metadata. Intra-dataset metadata form the profile associated with each single dataset, which includes descriptive, statistical, structural and usage metadata attributes. Inter-dataset metadata specify relationships between different datasets or between attributes belonging to different datasets. They include integrity constraints (e.g., inclusion dependencies) and other properties proposed by ourselves, such as *joinability* (Ω) and *affinity* (Ψ) between datasets. Inter-dataset metadata are represented graphically, as shown in Fig. 2(a) and (b). Intuitively, joinability measures the mutual percentage of common values between attributes of two datasets, whereas affinity measures the semantic strength of a relationship according to some external knowledge. The *affinity* is an adaptation, to data lakes, of the entity complement proposed by Sarma et al. [21].

Time-to-Action and Tolerance of the User. Let us call *time-to-action* the amount of time elapsing between the submission of a primitive in a pipeline and the instant in which a data scientist is able to take an informed decision on how to move forward to the next step of the pipeline. To shorten primitive computation when unnecessarily long, we let the data scientist specify a *tolerance*. A high tolerance is set by the data scientist who does not want to wait for long and believes that an approximate result is enough informative. On the contrary, a low tolerance is specified when the priority is on accuracy. For instance, in the

pipeline of Fig. 1, primitives P_4 and P_7 have been specified with a tolerance of 60% and 35%, respectively.

Incremental Execution for Reducing Time-to-Action. In KAYAK, primitives can be executed incrementally and produce a sequence of *previews* along their computation. A primitive is decomposed into a series of tasks. Each task can be computed as a sequence of steps that returns the previews. A preview is an approximation of the exact result of the task and it is, therefore, computed much faster. Two strategies of incremental execution exist. A *greedy* strategy aims at reducing the time-to-action by producing a quick preview first, and then updating the user with refined previews within her tolerance. Alternatively, a *best-fit* strategy aims at giving the best accuracy according to the given tolerance. It generates only the most accurate preview that fits the tolerance of the user.

Confidence of Previews. Each preview comes with a *confidence* indicating the uncertainty on the correctness of the result with a value between 0 and 1. A confidence is 0 when the result is random and it is 1 when it is exact. A sequence of previews is always produced with an increasing confidence so that the user is always updated with more accurate results and metadata are updated with increasingly valuable information.

Extensibility. KAYAK provides a set of built-in, atomic tasks that can be easily extended for implementing new functionalities. Specifically, tasks implementing common activities of data preparation and therefore can be used by different primitives. For instance, referring to Table 2, task T_b is used by three primitives. In addition, a new task can be defined by the users, who needs to specify also the cost model for the computation of the task and all the possible ways to approximate it, as we will show next.

3 Modeling a Data Lake

In this section, we discuss on how data and metadata are represented and managed in our framework. Let us start with some basic notions.

Definition 1 (Dataset). *A dataset $D(X, C, R)$ has a name D, and is composed by a set X of* attributes*, a set C of* data objects*, and a* profile *R. Each data object in C is a set of attribute-value pairs, with attributes taken from X. The profile R is a set of attribute-value pairs, with attributes taken from a predefined set M of* metadata attributes*.*

A metadata attribute of a dataset D can refer to either the whole dataset or to an attribute of D. We use the dot notation to distinguish between the two cases. For instance, if D is a dataset involving an attribute *ZipCode*, the profile of D can include the pairs $\langle D.CreationDate : 11/11/2016\rangle$ and $\langle ZipCode.unique : true\rangle$. For simplicity, we assume that each dataset is stored in a file and therefore we often blur the distinction between dataset and file.

Definition 2 (Data Lake). *A data lake $\mathcal{D}$ is a collection of datasets having distinct names.*

Differently from a profile of a dataset, inter-dataset metadata capture relationships between different datasets and between attributes of different datasets. They are represented as graphs and are introduced next.

Affinity. In the affinity graph of a data lake $\mathcal{D}$, the nodes represent the attributes of the datasets in $\mathcal{D}$ and an edge between two attributes represents the presence of some time-independent relationship between them (e.g., the fact that they refer to the same real-world entity). Edges can have weights that measure the "strength" of the relationship.

Specifically, we consider a *domain-knowledge* affinity $\Omega(D_1.A_i, D_2.A_j)$ that measures the "semantic" affinity between attributes $D_1.A_i$ and $D_2.A_j$, which is computed by taking advantage of some existing external knowledge base (such as a domain ontology). The value assigned by $\Omega(D_1.A_i, D_2.A_j)$ ranges in $[0, 1]$ (i.e. 0 when there is no affinity and 1 when the affinity is maximum). Here, we take inspiration from the notion of *entity complement* proposed by Sarma et al. [21]. However, different kinds of affinity can be used such as those based on text classification.

We can now define the graph representing the affinity of the attributes $\mathcal{A}$ of the datasets in the data lake $\mathcal{D}$.

Definition 3 (Affinity Graph of Attributes). *The affinity graph of attributes in $\mathcal{D}$ is an undirected and weighted graph $G^{\Omega}_{\mathcal{A}} = (N_{\mathcal{A}}, E^{\Omega}_{\mathcal{A}})$ where $N_{\mathcal{A}}$ contains a node for each attribute A in $\mathcal{A}$ and $E^{\Omega}_{\mathcal{A}}$ contains an edge $(A_1, A_2, \Omega(A_1, A_2))$ for each pair of attributes A_1 and A_2 in $\mathcal{A}$ such that $\Omega(A_1, A_2) > 0$.*

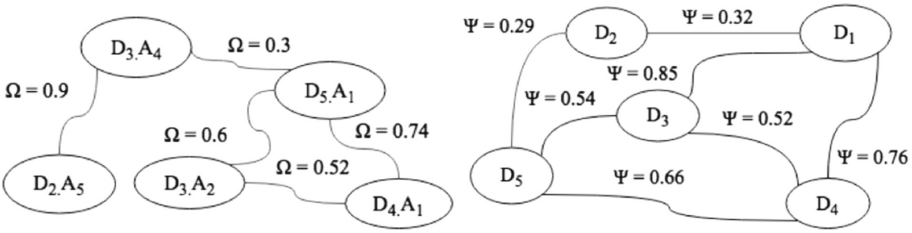
The notion of affinity between attributes can be used to define the affinity between two datasets D_1 and D_2.

Definition 4 (Affinity of Datasets). *Let X_1 and X_2 be the set of attributes of the datasets D_1 and D_2, respectively, and let $\hat{X} = X_1 \times X_2$. The affinity between D_1 and D_2, denoted by $\Omega(D_1, D_2)$, is defined as follows:*

$$\Omega(D_1, D_2) = \sum_{(A_j, A_k) \in \hat{X}} \Omega(A_j, A_k)$$

Analogously, we can define an affinity graph of datasets.

Definition 5 (Affinity Graph of Datasets). *The affinity graph of datasets for $\mathcal{D}$ is an undirected and weighted graph $G^{\Omega}_{\mathcal{D}} = (N_{\mathcal{D}}, E^{\Omega}_{\mathcal{D}})$ where $N_{\mathcal{D}}$ contains a node for each dataset D in $\mathcal{D}$ and $E_{\mathcal{D}}$ contains an edge $(D_1, D_2, \Omega(D_1, D_2))$ for each pair of dataset D_1 and D_2 in $\mathcal{D}$ such that $\Omega(D_1, D_2) > 0$.*



(a) Affinity of attributes A_j (b) Joinability of datasets D_i

Fig. 2. Inter-dataset metadata.

We are clearly interested in highly affine relationships. Therefore, we consider a simplified version of the graph where the edges have weights higher than a threshold τ, defined by the user. This is equivalent to consider irrelevant affinities below τ_Ω. An example of affinity graph of attributes is reported in Fig. 2(a).

Joinability. Another way to relate attributes and dataset is simply based on the existence of common values. We introduce the concept of *joinability* for this purpose.

Definition 6 (Joinability). *Given two attributes A_i and A_j belonging to the datasets D_1 and D_2, respectively, their joinability Ψ is defined as*

$$\Psi(D_1.A_i, D_2.A_j) = \frac{2 \cdot |\pi_{A_i}(D_1 \bowtie_{A_i=A_j} D_2)|}{(|\pi_{A_i}(D_1)| + |\pi_{A_j}(D_2)|)}$$

The joinability measures the mutual percentage of tuples of D_1 that join with tuples of D_2 on $D_1.A_i$ and $D_2.A_j$, and vice versa. This notion enjoys interesting properties, which we can discuss by considering the example in Fig. 3.

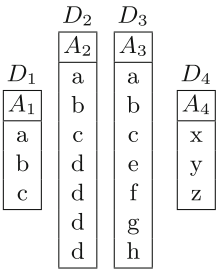


Fig. 3. Tabular datasets.

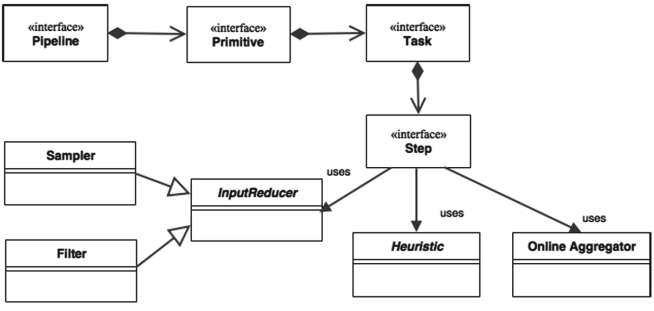


Fig. 4. Business logic of the framework.

The maximum joinability, (e.g., $\Psi(D_1.A_1, D_2.A_2) = 1$), is when each value of one attribute matches a value of the other attribute. If the result of the join

between the two attributes is empty, the joinability is 0 (e.g., $\Psi(D_2.A_2, D_4.A_4) = 0$). A joinability in $(0, 1)$ means that there are several matching values. The joinability takes also into account, for both attributes, the number of distinct values that do not match. For the dataset in Fig. 3 we have: $\Psi(D_1.A_1, D_2.A_2) > \Psi(D_1.A_1, D_3.A_3) > \Psi(D_1.A_1, D_4.A_4)$.

Similarly to the property of affinity, we can build a joinability graph of attributes and a joinability graph of datasets, where we represent only those edges whose joinability is higher or equal than a threshold parameter τ_Ψ. An example of a joinability graph is reported in Fig. 2(b).

4 Incremental Execution of Primitives

In this section, we describe the incremental execution of primitives, a mechanism that allows users to obtain previews of a result at an increasing level of accuracy.

Basic Idea. Users submit a primitive over an input I, which is typically a set of datasets. As we can see from Fig. 4, a primitive is composed of one or more tasks of type T. Each task type is associated with one or more *steps*. A step is an operation that is able to return a result for T over I. The result of a step can be either exact or approximate. We use t for indicating the step that computes the exact result r for t for T over I (i.e., $r = t(I)$). We use s_i^T for indicating the i-th approximate step of T, which returns a preview $p_i = s_i^T(I)$. Therefore, a preview p_i is an approximation of r.

We have several types of approximate steps, corresponding to different ways to approximate a task. For instance, some step reduces the input I (e.g., sampling), while other steps apply heuristics over I. In our framework, we have components that support the approximate steps, as shown in Fig. 4. The list of steps S_T for a task type T is declared in the definition of T. For simplicity, the following discussion considers a primitive with a single task type T, but this generalizes easily to primitives with many tasks. The incremental execution of a task type T over I is a sequence of m steps $s_1^T, \ldots, s_m^T$, where possibly the last step is the exact task t (i.e. $t = s_m^T$).

Each preview comes with a *confidence*, indicating the uncertainty on the correctness of the result with a number between 0 and 1. A confidence is 0 when the result is random and it is 1 when the result is exact. KAYAK computes the confidence executing a function embedded in the step definition. This function considers the confidence associated with the input of the step, e.g., the confidence of a metadata attribute. Note that, since previews are produced with an increasing confidence, metadata are stored with increasingly precise information and the user is always updated with more accurate primitive results.

In addition, each step is associated with a cost function that estimates its computational time over an input I, i.e. $cost(s_i^T, I)$. We have defined the cost functions using the Big-Θ time complexity. All other time-based measures like the load of the system and the tolerance have to be comparable with the cost and are therefore expressed in terms of the same virtual time. Moreover, we want

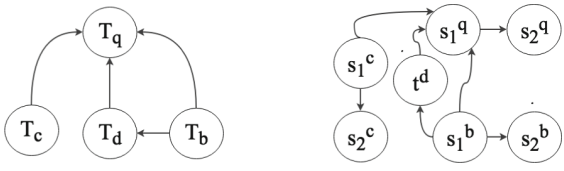
to underline that there is no formal relationship between the cost of a step and the confidence of the preview it produces.

Optimization. KAYAK needs to find a suitable incremental execution for a primitive, that is the order of the execution of steps. To this aim, KAYAK takes into account the tolerance of the user and the current workload. The tolerance is fixed by the user for each of the primitives in the pipeline. The workload is given by the sum of the costs of the steps of the primitives to be executed.

In our framework, we devised several incremental execution strategies and further strategies can be defined. For example, we have a so-called *best-fit* strategy that tries to generate only the most accurate preview that fits within the tolerance. This tends to limit the overall delay while still reducing the time-to-action according to the user's tolerance. Another strategy is the so-called *greedy* strategy that aims at minimizing the time-to-action and to update the user with subsequent previews. However, due to lack of space, we do not detail any strategy.

Step Dependency. At the end of the step generation, we set dependencies to enforce a correct execution of primitives composed of many tasks. Since we allow for a parallel execution of steps, a DAG of dependencies is considered. In the DAG, each node T_i is a task type and each edge (T_i, T_j) represents a dependency of a task T_j (the destination node) from another task T_i (the source node). A dependency indicates that T_j can start its execution only after T_i is completed.

In KAYAK, we do not have a centralized representation of the DAG, but dependencies are set within each task. For example, in Fig. 5(a) we have the DAG for the primitive P_5 that is composed of four tasks T_b, T_c, T_d and T_q, as in Table 2. The task T_q is the last task of the primitive and uses metadata provided by T_b, T_c and T_d. For this reason, T_q has a dependency with every other task. In addition, there is a dependency between T_d and T_b. This means that T_b and T_c can execute before the others, possibly in parallel.



(a) DAG of tasks (b) DAG of steps

Fig. 5. Dependencies among tasks of a primitive.

When a primitive is executed in incremental mode, the step generation phase produces a DAG that considers every single step. Let us suppose that T_b, T_c and T_q are executed incrementally in two steps each, while T_d is executed in a single step. Figure 5(b) shows the resulting DAG for this primitive execution. We set a dependency between two subsequent steps of the same task, such as (s_1^b, s_2^b), to

preserve the increase of the confidence of results. If this dependency is not set, then a preview might overwrite another preview with higher confidence.

We also need to set the inter-task dependencies. In this case, we set a dependency between the first generated steps of the two tasks. Using the example of P_5, the dependency (T_c, T_q) is established by the dependency (s_1^c, t_1^q). No inter-task dependency between following steps is considered. The reason behind this decision is, again, aimed at reducing the time-to-action.

There are two side effects that motivate this decision. The former is when, for example, t_1^c and t_2^c terminate before t_1^q has started. It follows that t_1^q will use metadata produced by t_2^c, resulting in a higher confidence. The latter side effect is when t_1^q is computed between t_1^c and t_2^c. The final result of t_1^q will be less accurate but the time-to-action is minimized. However, the user is notified of the fact that more accurate metadata is present for, possibly, refining the result of the primitive she just launched.

Scheduling of Steps. Dependencies are used for guaranteeing the consistency of primitives' results but they do not suffice to reduce the time-to-action of tasks coming from different primitives. To avoid a random order of execution, we use a step scheduling mechanism where the order of execution is done with respect to a priority assigned to each step. Steps with higher priority are executed first, while low priority steps are treated like processes to execute in background when the system is inactive. The priority function is defined as follows:

$$priority(s) = \frac{1}{cost(s)} + freshness(s) + completeness(s)$$

where:

- The *cost* is used to favor shorter steps, with the aim of reducing the time-to-action. It is given by the same function used in the previous sections.
- The *freshness* is used to avoid starvation. It uses the creation time of steps (with older steps having higher freshness). Let us explain the motivation behind this factor with an example. Let us consider the submission of a heavy task of type T_c followed by the submission of many shorter tasks of type T_a. If we consider only the cost factor, the task T_c will never be executed and it will starve in the queue.
- The *completeness* is used to balance the time-to-action across different primitives. It considers how many steps have already been instantiated for the task type. For instance, the completeness gives an advantage to the first step of task T_c over the second step of another task T_a. In fact, if we use only cost and freshness some step for T_c might not fulfill its time-to-action objective.

Note that our scheduling mechanisms do not conflict with mechanisms of cluster resource managers (e.g., Apache Mesos or Apache Yarn) used by data processing engines. We decide when a step can start its execution, while they schedule jobs of data processing only once their corresponding step has already started.

Use Case: Incremental Computation of Joinability. We now show the incremental execution of the task type T_c that computes the joinability of a

dataset against the rest of the datasets in the lake (see Definition 6). Since the data lake $\mathcal{D}$ can have a large number of datasets with many attributes, this task is computationally expensive and does not scale with the size of $\mathcal{D}$. Since this task is used by many primitives, it is often convenient to execute incrementally. Below we list the techniques used to generate previews of T_c.

1. *Joinability over frequent items.* Our least accurate step for T_c is given by a heuristic that takes the most frequent items present in the domain of two attributes along with the number of occurrences. This information is present in our metadata catalog and it is collected by other profiling tasks. We then compute the intersection between the two sets that allows us to determine the joinability of a small portion of the two attribute's domains in a constant time. The confidence is computed by considering the percentage of the coverage that the frequent items have over the entire domain.
2. *Joinability over a subset of attributes and sampled datasets.* This step uses some heuristics that aim at selecting those attributes that are likely to have higher values of joinability against the input dataset. It specifically selects a subset of attributes Z_i of the lake to be used in the computation of the joinability against attributes of D_i. Then, the datasets which the attributes of Z_i belong from are sampled to further reduce the cost of the operation. The approximation of joinability is similar to compute approximate joins [4,11]. The sample rate is chosen dynamically according to the size of the dataset, with lower sample rate for higher dataset size. The selected attributes Z_i are those having: (a) overlapping among the most frequent items of the attributes, (b) an inclusion dependency with D_i (we check from available metadata without computing T_e of Table 2), (c) high affinity with the attributes of D_i as taken from the *affinity graph of attributes*. The confidence of this level is given by the used sample ratios and the number of attributes that have been selected.
3. *Joinability over a subset of attributes.* This step selects the attributes of the previous case but does not apply any sampling over the datasets.
4. *Joinability over a subset of sampled datasets.* This step selects a set Y_i of datasets in $\mathcal{D}$ having high affinity with D_i by checking the *affinity graph of datasets*. Then, it computes the joinability between attributes of D_i and attributes of datasets in Y_i.
5. *Joinability over a subset of datasets.* This step selects the same set Y_i of datasets of the previous case but, differently from it, sampling is not applied.
6. *Joinability over a sampled data lake.* This step selects a sample from each of the datasets in $\mathcal{D}$ and then it applies the joinability between the attributes of D_i and any other attribute in $\mathcal{D}$.

We have described here the steps of the joinability task. The implementation of these steps makes use of other optimizations such as those in presence of inclusion dependencies or those that return zero when data types are different or domain ranges do not overlap. However, we do not discuss them in detail here because they do not deal with the approximation of the exact results.

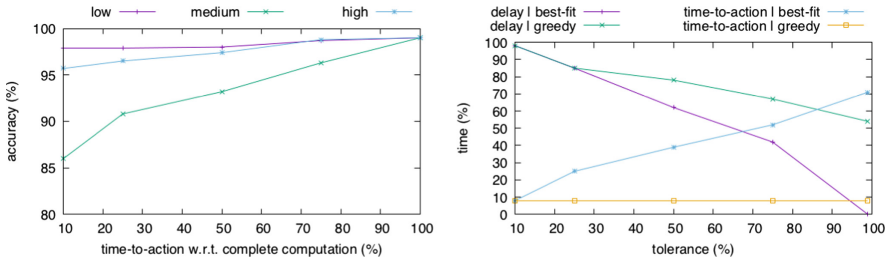
5 Experimental Results

5.1 Set-Up

The architecture of KAYAK is discussed in [15]. KAYAK is implemented in Java 8 and exploits *java.util.concurrent* library for the concurrent execution of tasks. The current version limits the use to *json* and to *csv* files only. The Metadata Catalog relies on two different database systems, namely MongoDB for the intra-dataset catalog and Neo4j for the inter-dataset catalog. The Queue Manager uses RabbitMQ as a messaging queue system. The User Interface is implemented using JSP pages and servlets on the web application server Tomcat 8. We also rely on external tools such as Spark 2.1 with MLib and SparkSQL add-ons for parallelizing operations on large datasets, and on Metanome[4] for some of the tasks for which the Metadata Collector is in charge.

5.2 Results

This section presents the experimentation that was conducted on a cluster of *m4.4xlarge* machines on Amazon EC2. Each machine is equipped with 16 vCPU, 64 GB and running a 2,3 GHz Intel Xeon with 18 cores. We created a data lake with 200 datasets ranging from hundreds of MBs to few TBs. We have taken datasets from the U.S. open data catalog[5] and from the NYC Taxi trips[6]. We have also generated synthetic datasets to create uses cases that were not covered with downloaded datasets.



(a) Accuracy of joinability (b) Incremental strategies comparison

Fig. 6. Incremental step generation at work.

Effectiveness. In this campaign, we have measured the trade-off between accuracy and time-to-action for the joinability task. It is a fundamental task in our framework that is used by many primitives. The results are in Fig. 6(a). We

[4] https://github.com/HPI-Information-Systems/Metanome.
[5] https://www.data.gov/.
[6] http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

have divided the tests in three sections according to the range of the joinability value Ψ: *low* ($0.0 \leq \Psi \leq 0.35$), *medium* ($0.35 < \Psi \leq 0.70$) and *high* ($0.70 < \Psi \leq 1.0$). The time-to-action is represented in terms of percentage of the time for the exact step. Each point of the graph represents the average of six joinability computations (we have six different tests for each of the sections). As we can see, the accuracy is constantly high for low values of joinability. This is due to the fact that a reduced input is already able to show that two attributes do not join well. A similar behavior is for medium values, although accuracy slightly degrades for low time-to-action. High values of joinability are more difficult to approximate with shorter time-to-action than previous sections, but we consider this accuracy still good for many practical situations.

Strategies Comparison. In this campaign, we test the differences between two incremental strategies, that we have briefly mentioned above. Again, we consider the joinability task. Let us consider the delay as the extra time spent on the incremental execution with respect to the non-incremental counterpart. We measure how the time-to-action and the delay vary with respect to the tolerance. Both the measures are taken in percentage with respect to the duration of the exact step. As we can see from the results in Fig. 6(b), the time-to-action for the greedy strategy is constant because the same short level is executed independently of the user's tolerance, while for the best-fit strategy the time-to-action increases linearly with the tolerance. However, the time-to-action is always lower than the tolerance due to a fragmentation effect that makes it hard to have the cost of a step that perfectly fits the tolerance. The delay of the greedy strategy is always greater than the delay of the best-fit strategy, because of all the short steps executed at the beginning. The delay for both strategies tends to diminish as the tolerance increases. The delay of the best-fit strategy has an opposite behavior w.r.t. the time-to-action. Indeed, the delay is inversely proportional to the tolerance. This is because as the tolerance increases, the best-fit strategy tends to schedule fewer and fewer steps.

6 Related Work

We divide related work of KAYAK into categories discussed separately.

Data Catalogs. There are several tools that are used for building repositories of datasets [1,3,5,10,12]. Basic catalogs like CKAN [1] do not consider relationships among datasets and metadata are mostly inserted manually. DataHub [5] is a catalog that enables collaborative use and analysis of datasets. It includes features like versioning, merging and branching for datasets, similarly to version control systems in the context of software engineering. GOODS is an enterprise search system for a data lake that is in use at Google [10]. It proposes, among the others, a solution with the semi-automatic realization of a metadata catalog, an annotation service, an efficient tracking of the provenance and advanced search features based on full-text indexing. All above catalogs use basic ways to understand relationships among datasets and give little support to users who are

unaware of the content of the datasets, though they are not explicitly designed for data preparation and exploration purposes.

Profiling Tools. In data science, tools such as R[7], IPython [19], Pandas-profiling[8] and notebook technologies[9] are extensively used for data exploration. They mainly compute statistical summaries integrated with some plotting features. More advanced data profiling consists on the discovery of constraints in the data [7,14,16,18]. Metanome, for instance, offers a suite of different algorithms for data profiling [16]. Some of these algorithms run by sharing pieces of computation [7] or by the aid of approximate techniques [14,18]. In KAYAK we have tasks that make use of these algorithms such as for example T_c in Table 2.

Data Wranglers. Schema-on-read data access has opened severe challenges in data wrangling [8,23] and specific tools are aimed at solving this problem [2,3,22]. Data TamR helps in finding insights thanks to novel approaches of data curation and data unification [2,22]. Trifacta is an application for self-service data wrangling providing several tools to the user [3]. All these systems provide features that can be embedded in KAYAK to be executed incrementally for minimizing the time-to-action.

Approximate Querying Systems. Another branch of work specifically focuses on approximating analytical query results [4,11,20]. Hellerstein et al. [11] propose an incremental strategy that aggregates tuples online so that the temporary result of the query is shown to the user, who can decide to interrupt the process anytime. Differently, when computing analytical queries with BlinkDB [4], users are asked the trade-off between time and accuracy in advance, and the system dynamically selects the best sample that allows replying the query under the user's constraints. This is similar to our best-fit strategy but we do not apply only sampling and we do not consider analytical queries. A critical aspect in all these works is the estimation of the error. To overcome these problems, DAQ [20] has recently introduced a deterministic approach to approximating analytic queries, where the user is initially provided with an interval that is guaranteed to contain the query result. Then, the interval shrinks as the query answering proceeds, until the convergence to the final answer. All these techniques work well on OLAP queries but since they require the workload in advance, they cannot be applied in our context where the user has usually not accessed the data yet and sampling cannot be the only technique for reducing the workload.

7 Conclusion and Future Work

In this paper, we have presented KAYAK, a end-to-end framework for data management with a data lake approach. KAYAK addresses data preparation, a crucial aspect for helping data-driven businesses in their analytics processes. KAYAK

[7] https://www.r-project.org/.
[8] https://github.com/JosPolfliet/pandas-profiling.
[9] http://zeppelin-project.org/.

provides a series of primitives for data preparation that can be executed by specifying a tolerance when the user prefers a quick result instead of an exact result. The framework also allows to define pipelines of primitives.

We have several future work directions in mind. We want to integrate the framework with components for supporting unstructured data, query expansion, and data visualization. We want to introduce a dynamic scheduling for the tasks and the possibility to set a tolerance for an entire pipeline. Finally, we would like to define a declarative language for designing primitives data preparation.

References

1. CKAN: The open source data portal software. http://ckan.org/. Accessed Nov 2017
2. Tamr. http://www.tamr.com/. Accessed Nov 2017
3. Trifacta. https://www.trifacta.com/. Accessed Nov 2017
4. Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., Stoica, I.: BlinkDB: queries with bounded errors and bounded response times on very large data. In: EuroSys, pp. 29–42 (2013)
5. Bhardwaj, A.P., Deshpande, A., Elmore, A.J., Karger, D.R., Madden, S., Parameswaran, A.G., Subramanyam, H., Wu, E., Zhang, R.: Collaborative data analytics with DataHub. PVLDB **8**(12), 1916–1927 (2015)
6. Deng, D., Fernandez, R.C., Abedjan, Z., Wang, S., Stonebraker, M., Elmagarmid, A.K., Ilyas, I.F., Madden, S., Ouzzani, M., Tang, N.: The data civilizer system. In: CIDR (2017)
7. Ehrlich, J., Roick, M., Schulze, L., Zwiener, J., Papenbrock, T., Naumann, F.: Holistic data profiling: simultaneous discovery of various metadata. In: EDBT, pp. 305–316 (2016)
8. Furche, T., Gottlob, G., Libkin, L., Orsi, G., Paton, N.W.: Data wrangling for big data: challenges and opportunities. In: EDBT, pp. 473–478 (2016)
9. Hai, R., Geisler, S., Quix, C.: Constance: an intelligent data lake system. In: SIGMOD, pp. 2097–2100 (2016)
10. Halevy, A.Y., Korn, F., Noy, N.F., Olston, C., Polyzotis, N., Roy, S., Whang, S.E.: Goods: organizing Google's datasets. In: SIGMOD (2016)
11. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online aggregation. In: SIGMOD, pp. 171–182 (1997)
12. Hellerstein, J.M., Sreekanti, V., Gonzalez, J.E., Dalton, J., Dey, A., Nag, S., Ramachandran, K., Arora, S., Bhattacharyya, A., Das, S., Donsky, M., Fierro, G., She, C., Steinbach, C., Subramanian, V., Sun, E.: Ground: a data context service. In: CIDR (2017)
13. Heudecker, N., White, A.: The data lake fallacy: all water and little substance. Gartner Report G 264950 (2014)
14. Ilyas, I.F., Markl, V., Haas, P.J., Brown, P., Aboulnaga, A.: CORDS: automatic discovery of correlations and soft functional dependencies. In: SIGMOD, pp. 647–658 (2004)
15. Maccioni, A., Torlone, R.: Crossing the finish line faster when paddling the data lake with KAYAK. PVLDB **10**(12), 1853–1856 (2017)
16. Papenbrock, T., Bergmann, T., Finke, M., Zwiener, J., Naumann, F.: Data profiling with metanome. PVLDB **8**(12), 1860–1863 (2015)

17. Papenbrock, T., Ehrlich, J., Marten, J., Neubert, T., Rudolph, J., Schönberg, M., Zwiener, J., Naumann, F.: Functional dependency discovery: an experimental evaluation of seven algorithms. PVLDB **8**(10), 1082–1093 (2015)
18. Papenbrock, T., Naumann, F.: A hybrid approach to functional dependency discovery. In: SIGMOD, pp. 821–833 (2016)
19. Pérez, F., Granger, B.E.: IPython: a system for interactive scientific computing. Comput. Sci. Eng. **9**(3), 21–29 (2007)
20. Potti, N., Patel, J.M.: DAQ: a new paradigm for approximate query processing. PVLDB **8**(9), 898–909 (2015)
21. Sarma, A.D., Fang, L., Gupta, N., Halevy, A.Y., Lee, H., Wu, F., Xin, R., Yu, C.: Finding related tables. In: SIGMOD (2012)
22. Stonebraker, M., Bruckner, D., Ilyas, I.F., Beskales, G., Cherniack, M., Zdonik, S.B., Pagan, A., Xu, S.: Data curation at scale: the data tamer system. In: CIDR (2013)
23. Terrizzano, I., Schwarz, P.M., Roth, M., Colino, J.E.: Data wrangling: the challenging journey from the wild to the lake. In: CIDR (2015)
24. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)