Crossing the finish line faster when paddling the Data Lake with KAYAK

Antonio Maccioni
Collective[i]
New York City, U.S.A.
amaccioni@collectivei.com

Riccardo Torlone
Roma Tre University
Rome, Italy
torlone@ing.uniroma3.it

ABSTRACT

Paddling in a data lake is strenuous for a data scientist. Being a loosely-structured collection of raw data with little or no meta-information available, the difficulties of extracting insights from a data lake start from the initial phases of data analysis. Indeed, data preparation, which involves many complex operations (such as source and feature selection, exploratory analysis, data profiling, and data curation), is a long and involved activity for navigating the lake before getting precious insights at the finish line.

In this framework, we demonstrate KAYAK, a framework that supports data preparation in a data lake with ad-hoc primitives and allows data scientists to cross the finish line sooner. KAYAK takes into account the tolerance of the user in waiting for the primitives' results and it uses incremental execution strategies to produce informative previews of these results. The framework is based on a wise management of metadata and on features that limit human intervention, thus scaling smoothly when the data lake evolves.

1. INTRODUCTION

A data lake is a loosely-structured collection of data at large scale which is usually fed with almost no requirement of data quality. Modern data analytics toolkits operate on data lakes in a schema-on-read fashion, by accessing directly the datasets, usually stored in a distributed file system. Data scientists adopt this approach because they can collect a larger amount of data while dismissing any human effort prior the actual analysis of data. Unfortunately, problems are only delayed since, given the heterogeneity of data and the absence of meta-information, preparing, curating, exploring, and querying an even small portion of a data lake is often a hard job. In particular, a schema-on-read access does not circumvent the need for data quality or schema understanding.

Therefore, analysts have to go through a long pipeline of data preparation tasks (a.k.a. data wrangling or DataOps) when they want to gain insights into the available data [4, 8, 11]. Currently, this is done by combining a multitude of tools. For instance, data and metadata catalogs are used to facilitate the fishing of datasets of interest across the organization [1, 3, 6, 7]. However, these catalogs first require data profilers to collect meta-information from underlying data [4, 5, 9]. Among them, R[1] and pandas-profiling[2] are extensively used for computing statistical summaries, often in conjunction with Data Science Notebooks like Project Jupyter[3] or Apache Zeppelin[4]. In addition, specialized tools like Metanome [9] and Data Civilizer [4] are used for discovering data constraints, whereas tools like Constance [5] explicitly focus on collecting and managing structural and semantic metadata for heterogeneous data sources.

Given that all these tools rely on hard-to-scale algorithms, the "time-to-action" of data scientists is exceedingly long for data preparation. Moreover, the issues are made worse when the number of datasets of the lake increases and new datasets, for which no metadata is available, are included in the pipeline and maybe are the next to be accessed.

To expedite data preparation in a data lake we propose KAYAK, a data management framework that implements ad-hoc primitives and execute them with an efficient strategy. The main features of our system are the following.

- KAYAK has a metadata catalogue that not only keeps a profile for each dataset, but also includes meta-information on how different datasets are related to each other. These metadata are essential to suitably contextualize each dataset in the lake.
- KAYAK takes into account the "tolerance" of the user in waiting for the result of a primitive execution. Then, it tries to produce significant previews of the result within the user's tolerance so that she can start thinking about the next step in the pipeline. In general, previews are enough informative to proceed in an analysis process and are rapidly computed via an incremental execution of the basic operators, which relies on heuristics to approximate the results of our primitives. These heuristics differ from traditional methods for approximating query answering that use online aggregation or knowledge of the workload [2].
- KAYAK provides a rather extensive set of predefined primitives for managing a data lake and executes them

[1] https://www.r-project.org/
[2] https://github.com/JosPolfliet/pandas-profiling
[3] http://jupyter.org/
[4] https://zeppelin.apache.org/

Proceedings of the VLDB Endowment, Vol. 10, No. 12

in parallel making sure that, given a tolerance, the most accurate results are returned to the user on even just-inserted datasets in the lake.

2. OVERVIEW OF KAYAK

In this section, we give a high-level overview of the system.

Primitives and Tasks. KAYAK is a framework that lies between users/applications and the file system used to store data. We call *kayakers* the users of the framework. KAYAK exposes a series of *primitives* for data preparation, exploration and analysis to kayakers: some of them are reported in Table 1. For example, a kayaker can use primitive P_5 to find interesting ways to access a dataset. Each primitive is decomposed into *tasks* that are reused across many primitives: Table 2 shows some of them. For instance, P_5 is split into T_b, T_c, T_d, while primitive P_7 uses T_c only. A task is atomic and involves a sequence of operations that are implemented in KAYAK or are calls to external tools [9, 13].

KAYAK has *synchronous* and *asynchronous* primitives, which generate synchronous and asynchronous tasks, respectively. A synchronous primitive prevents the user from submitting another primitive before its completion and it is typically used for fast operations, while an asynchronous primitive can be submitted concurrently with others to further reduce the duration of data preparation.

Table 1: Example of Primitives in Kayak.

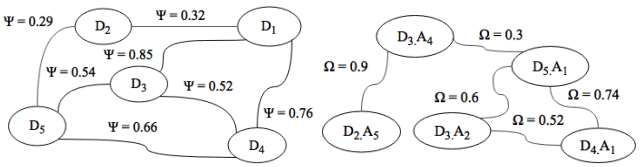
PRIMITIVE	NAME	USES_TASK
P_1	Insert dataset	T_a, T_p
P_2	Delete dataset	T_s
P_3	Search dataset	T_o
P_4	Complete profiling	T_a, T_b, T_c, T_d, T_m
P_5	Get recommendation	T_b, T_c, T_d, T_q
P_6	Find related dataset	T_b, T_c, T_w
P_7	Compute joinability	T_c
P_8	Compute k-means	T_g, T_n
...	...	...

Table 2: Example of Tasks in Kayak.

TASK TYPE	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
...	...

Metadata management. KAYAK collects metadata explicitly, using ad-hoc primitives (e.g., P_4), or implicitly, when a primitive needs some metadata attribute and uses the corresponding profiling task (e.g., T_a in P_1). Metadata attributes are stored in a centralized catalog so that they can be accessed by any task.

KAYAK collects *intra-dataset* and *inter-dataset* metadata. Intra-dataset metadata constitute the profile associated with each single dataset. They include attributes describing content, statistical, structural, and usage information about the dataset. Inter-dataset metadata specify relationships between different datasets or between attributes belonging to different datasets. They include data constraints (e.g., inclusion dependencies) and other properties that we have



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

Figure 1: Inter-dataset metadata.

introduced such as *joinability* (Ω) and *affinity* (Ψ) between datasets. Inter-dataset metadata are represented graphically, as shown in Figure 1. Intuitively, joinability measures the mutual percentage of common values, while affinity measures the semantic strength of a relationship according to external knowledge. The *affinity* is an adaptation, in the context of data lakes, of the notion of "entity complement" proposed by Das Sarma et al. [10].

Time-to-action and Tolerance of the user. We call *time-to-action* the amount of time elapsing between the submission of a primitive and when the user is able to take an informed decision on how to proceed in the analysis process. To shorten primitive computation when unnecessarily long, we let the data scientist to specify a *tolerance*. A low tolerance is set by the user who does not intend to wait too long and accepts an approximate result, assuming that it is enough informative to continue the preparation. Conversely, a high tolerance is specified when accuracy is a priority.

Incremental execution for reducing time-to-action. In KAYAK, primitives can be executed incrementally, producing a sequence of *previews* of the computation results. A primitive is decomposed into a series of tasks that can be computed in steps, each of which returns a 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 are possible. A *greedy* strategy aims at reducing the time-to-action by producing a quick preview first, and then updating the user with many refined previews within her tolerance. Alternatively, a *best-fit* strategy aims at giving the best approximation according to the given tolerance. It generates only the most accurate preview that fits within 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 increasing confidence so that the user is always updated with more accurate previews and metadata are updated with increasingly valuable information.

3. ARCHITECTURE OF KAYAK

This section shows the components of KAYAK and the way in which they interact with each other (see Figure 2).

- *User Interface and APIs:* this component allows users to submit primitives in a user-friendly fashion and to visualize the results. In addition, it exposes primitives with a series of APIs so that third-party applications can easily interact with KAYAK (e.g., data mining or visualization applications).

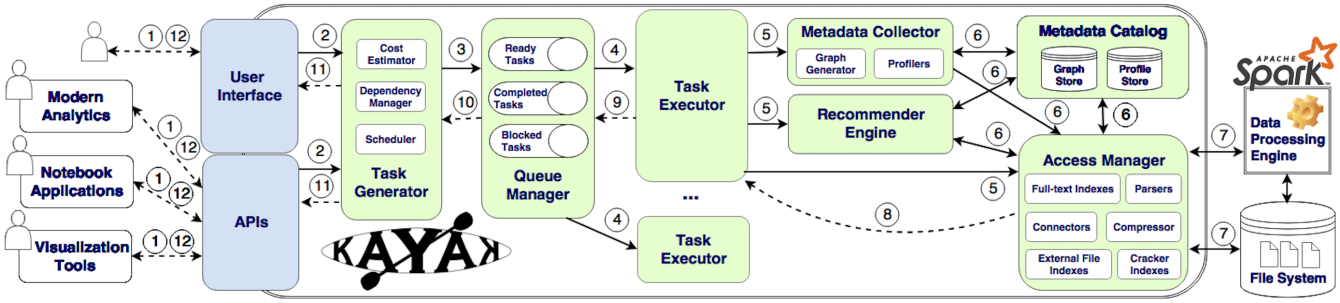


Figure 2: Logical architecture of Kayak.

- *Task Generator:* this component is the gateway for the whole framework. It receives calls of primitives and generates the corresponding tasks deciding whether execute them incrementally.
- *Queue Manager:* this component supports the asynchronous execution of tasks by decoupling their invocation from their execution. For this, it makes use of a message-oriented broker.
- *Task Executor:* this component processes tasks. In a scalable deployment of KAYAK, this component is typically instantiated multiple times for a parallel and distributed computation. Figure 3 shows the physical deployment of the system, with multiple users and multiple executors.
- *Metadata Catalog:* this component stores metadata. It is divided into a *profile store*, which keeps intra-dataset metadata, and in a *graph store*, which keeps inter-dataset metadata (e.g., Figure 1). The content of the Metadata Catalog is divided into categories inspired by existing metadata schemes (e.g., [1, 12]) but it is however extensible.
- *Metadata Collector:* this component is in charge of populating the metadata catalog. It has a *profiler* for extracting intra-dataset metadata and a *graph generator* for inter-dataset metadata.
- *Access Manager:* this component constitutes the interface between the storage system and the engine for data access. We make use of Spark for this task [13].
- *Recommender Engine:* this component provides precious insights by suggesting to users queries of potential interest for them.

Kayak at work. We show how the components of the architecture interact in an asynchronous and non-incremental execution. Let us consider the submission of the primitive P_5 (action ① of Figure 2). The *Task Generator* receives the primitive and the input (②). Then, it determines the tasks for computing P_5. It specifically instantiates t_b, t_c, t_d and t_q from the task types T_b, T_c, T_d and T_q, respectively. The generator also sets a series of information on each task: i) the cost, using a *cost estimator*, ii) the dependencies with other tasks, if any, using a *dependency manager*, and iii) the priority of the task, using a *scheduler*. The priority is used by the *Queue Manager* to determine the position of a task into the priority queue of *ready tasks* (③). If the task has dependencies with tasks that have not been executed yet,

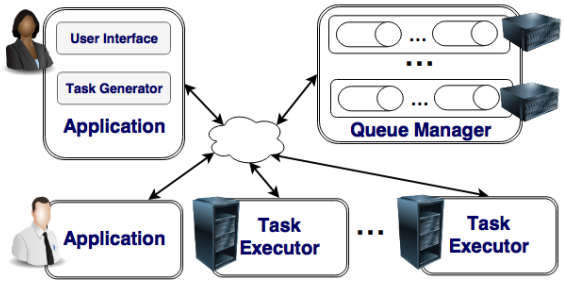


Figure 3: Physical deployment of Kayak.

it is first inserted into a queue of *blocked tasks* until its dependencies are solved. This is tested looking into a queue of *completed tasks*. In our specific case, we have that t_d depends on t_b whereas t_q depends on t_b, t_c and t_d. Therefore, t_q remains in the queue of *blocked tasks* until all the tasks it depends on are terminated.

Since KAYAK can have multiple executors (see Figure 3), tasks of the same primitive can run in parallel. Taking into account the task dependencies, we can exploit some possible degree of parallelism such as, for instance, running together t_b and t_c. However, each task is eventually consumed by one and only one *Task Executor*. Specifically, when an executor is free, it takes the task with the highest priority from the queue of *ready tasks* (④). The execution of the task involves one or many components of KAYAK (⑤ and ⑥). If the task needs to access a dataset, then the *Access Manager* is used for this purpose, possibly exploiting a suitable data processing engine (⑦). In our example, t_b, t_c and t_d collect some of the required metadata, while t_q produces the result to return to the user, as represented by dotted lines (⑧-⑨). The *Task Generator* extracts the results from the completed tasks (⑩) and makes them available to the user (⑪-⑫). It is worth noting that while P_5 is being processed, the user can submit other primitives and further tasks can be executed simultaneously.

The workflow of execution for synchronous primitives is simpler. A *Task Executor* processes synchronous tasks only. The *Task Generator* sends synchronous tasks directly to this executor and waits for their completion before accepting further requests.

4. DEMO SETTING

We propose to the audience the following scenarios which are aimed at giving a comprehensive view of the system.

They demonstrate a typical end-to-end workflow of the kayaker using KAYAK.

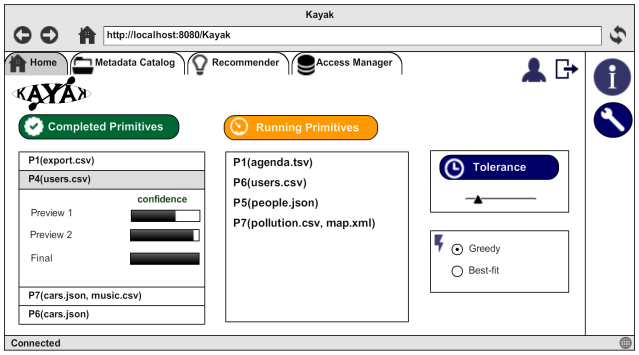


Figure 4: Interface of Kayak.

Scenario A: Focus on the usability. In this initial demo scenario, we overview KAYAK showing the user interface in Figure 4. At the bootstrap of the application, a kayaker can login and change some parameters such as the number of parallel executors to use. Once KAYAK is running, we show how to insert and remove datasets from the lake. At this point, we can launch the primitives exposed by the framework over the datasets in the lake. Primitives are grouped in macro-areas, each one provided with its own tab window. A macro-area roughly corresponds to one component of KAYAK. We will provide some datasets in advance for guaranteeing a meaningful demonstration.

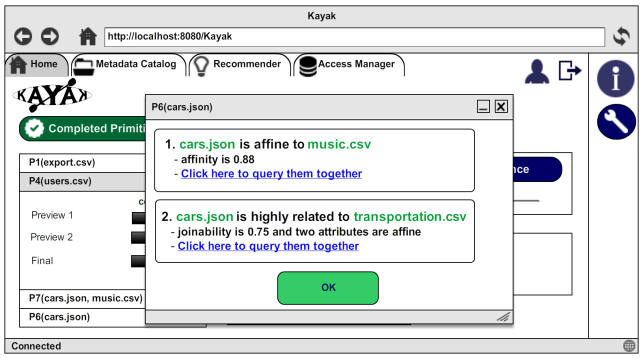


Figure 5: Preview visualization.

From the interface, the kayaker can set her tolerance and decides the incremental strategy among those available: greedy or best-fit. Lists "completed" and "running" show the completed and running primitives, respectively. In the former it is possible to see, for each submitted primitive, the list of computed previews ordered by confidence. The user can click on a preview to obtain, via a pop-up window, the approximate result. KAYAK allows also to see the "explanation" of the primitive execution. For instance, Figure 5 shows a pop-up of the primitive P_6 for finding datasets related to *cars.json*. In this case, KAYAK has found two datasets that are related with *cars.json*. Additionally, the user can also submit a query involving the two datasets by just clicking on the proposed link, which corresponds to another primitive call.

Scenario B: Focus on the architecture. In this scenario, we delve into the architecture of the system. This involves the showcase of the primitives associated with each of the components in the architecture. Subsequently, we demonstrate the workflow that involve these tasks, as discussed in the previous section.

Scenario C: Focus on the incremental execution. In this scenario, we focus on the incremental execution of primitives within KAYAK. We submit the primitives by varying the tolerance so that users can understand how KAYAK is able to accommodate different user needs. This highlights the time-to-action gain with respect to the non-incremental execution.

Scenario D: Focus on the metadata catalog. In this scenario, we show how we designed the metadata catalog and the original metrics to assess the relationships (e.g., affinity and joinability as in Figure 1) among different datasets or attributes. It will be possible to navigate the catalog in a visual manner directly from KAYAK. In addition, we simulate a collaborative scenario in which many kayakers are using the application and one user can benefit from metadata that were computed by primitives submitted by another user.

5. REFERENCES

[1] CKAN: The open source data portal software. http://ckan.org/, (last accessed May, 2017).

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[3] A. P. Bhardwaj, A. Deshpande, A. J. Elmore, D. R. Karger, S. Madden, A. G. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang. Collaborative data analytics with DataHub. *PVLDB*, 8(12):1916–1927, 2015.

[4] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The data civilizer system. In *CIDR*, 2017.

[5] R. Hai, S. Geisler, and C. Quix. Constance: An intelligent data lake system. In *SIGMOD*, pages 2097–2100, 2016.

[6] A. Y. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing Google's Datasets. In *SIGMOD*, 2016.

[7] J. M. Hellerstein, V. Sreekanti, J. E. Gonzalez, J. Dalton, A. Dey, S. Nag, K. Ramachandran, S. Arora, A. Bhattacharyya, S. Das, M. Donsky, G. Fierro, C. She, C. Steinbach, V. Subramanian, and E. Sun. Ground: A data context service. In *CIDR*, 2017.

[8] N. Heudecker and A. White. The data lake fallacy: All water and little substance. *Gartner Report G*, 264950, 2014.

[9] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with metanome. *PVLDB*, 8(12):1860–1871, 2015.

[10] A. D. Sarma, L. Fang, N. Gupta, A. Y. Halevy, H. Lee, F. Wu, R. Xin, and C. Yu. Finding related tables. In *SIGMOD*, 2012.

[11] I. Terrizzano, P. M. Schwarz, M. Roth, and J. E. Colino. Data wrangling: The challenging journey from the wild to the lake. In *CIDR*, 2015.

[12] S. Weibel, J. Kunze, C. Lagoze, and M. Wolf. Dublin core metadata for resource discovery. Technical report, Internet Engineering Task Force, 1998.

[13] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.