HAP: Building Pipelines with Heterogeneous Data

Damien Graux
Inria
damien.graux@inria.fr

Pierre Genevès
Cnrs
pierre.geneves@cnrs.fr

Nabil Layaïda
Inria
nabil.layaida@inria.fr

ABSTRACT

The increasing number of available datasets gives opportunities to build large and complex applications which aggregate results coming from several sources. These emerging usecases require new systems where combinations of heterogeneous sources are both allowed and efficient.
We propose a set of high-level primitives – called Hap – to facilitate the description of processing chains. Hap descriptions are formed from the combination of several queries written in popular query languages such as sparql and XPath. From any Hap description we generate a single sql query. This makes it possible to apply automatic optimizations on the whole pipeline description at compile-time. For the need of this demonstration, we generate queries in HiveQL form that we execute with Hive.

1. INTRODUCTION

The increasing availability of data under free licenses (open data) allows to develop innovative applications that combine and enrich data. These applications often have to deal with heterogeneous data – *i.e.* representing various kinds of information and structured using various standards – of diverse size – *e.g.* datasets size are spread over several orders of magnitude – and of various natures since some datasets are more dynamic than others.

The possible combinations of these three degrees of freedom conducted to designs of specific applications dedicated to each single case, for instance efficient evaluators of a chosen query language (*e.g.* sql, sparql...) in a distributed context. However, in some usecases, existing delineations of field have to be over crossed; indeed, aggregating results extracted from several datasets might be required to build more complex answers. Such a need implies to be able to efficiently query several kinds of data structures while being able to merge the obtained sub-results also efficiently.

For the need of this demonstration, we generate queries in the HiveQL form that is executed with Hive. Apache Hive [12] is an open-source data warehousing solution built on-top of Apache Hadoop [3]. As a consequence, it takes as file system the hdfs [11] and converts sql (technically Hive-QL – but the fragment we consider allow us to use the exact sql syntax –) queries in sequences of MapReduce jobs executed directly on Hadoop. Therefore, Apache Hive allows to query large datasets distributed across cluster of nodes using a relational language while providing resiliency thanks to Hadoop.

Proceedings of the VLDB Endowment, Vol. 10, No. 8


```
EVAL      id          ((columns))   [[query]]
CONNECT   id id id    ((columns))   [[conditions]]
FILTER    id id       ((columns))   [[filters]]
RETURN    id
```

Figure 1: Hap Syntax.

Contribution. In this demonstration, we present a set of high-level primitives called Hap for the description of processing chains. A Hap description is compiled into a single sql query. The advantages of using Hap are twofold. First, it allows to design pipelines dealing with heterogenous data (e.g. rdf, xml, csv) queried with their respective standard languages. Second, Hap makes it possible to generate a single optimized query for the whole pipeline, by applying automatic optimizations with rewriting rules and statistics on data.

2. HAP SYNTAX

Syntax. We propose four primitives, see Figure 1 for their syntax. Each primitive deals with a set of columns and defines also a unique identifier.

First, the initial instruction named `EVAL` allows to evaluate an existing query (see Section 3 for a description of accepted languages). Its syntax implies to give an ID to the task and to named the returned columns. Second, `CONNECT` gives the opportunity of combining sets of columns – results of queries by extension – according to keys. Third, `FILTER` allows to give conditions to refine a set of columns. Finally, `RETURN` is used to have a starting point in the compilation process and designates the set of columns (thanks to an identifier) that should be returned. The combination of these four primitives gives users the possibility of combining – in few lines – subresults of already existing queries they have without the need of rewriting them.

Technically, only one `RETURN` is tolerated per program. In addition, there must obviously be unicity of output identi-

```
EVAL     1       ((dep,arr,depHour,arrHour,stop))                [[ Q-plane ]]
EVAL     2       ((place,restau))                                [[ Q-diner ]]
EVAL     3       ((location,poi))                                [[ Q-tourism ]]
CONNECT  1 2 x   ((dep,arr,depHour,arrHour,stop,restau))         [[ place=stop ]]
FILTER   x y     ((dep,arr,depHour,arrHour,stop,restau))         [[ arrHour-depHour > k ]]
CONNECT  3 y f   ((dep,arr,poi,depHour,arrHour,stop,restau))     [[ location=stop ]]
RETURN   f
```

(a) HAP primitives of the Demonstration Example.

EVAL k ((name)) [[select x ...]]	RETURN i
(select x as name from (select x ...) as ini_k) as k	select * from [i]

CONNECT i j k ((name)) [[key]]	FILTER a b ((name)) [[condition]]
(select name from [i] join [j] on (key)) as k	(select name from [a] where condition) as b

(b) Partial Translations for each Primitive.

```
select *
from ( select dep arr poi depHour arrHour stop restau
       from ( select location poi
              from ( Q-tourism ) as ini_3
            ) as 3
       join ( select dep arr depHour arrHour stop restau
              from ( select dep arr depHour arrHour stop restau
                     from ( select dep arr depHour arrHour stop
                            from ( Q-plane) as ini_1
                          ) as 1
                     join ( select place restau
                            from ( Q-diner) as ini_2
                          ) as 2
                     on ( place=stop )
                   ) as x
              where arrHour-depHour > k
            ) as y
       on ( location=stop )
     ) as f
```

(c) "Naive" Translation using Figure 2b.

Figure 2: Demonstration Example.

fiers whereas it is not the case as input identifiers; indeed, a same result can be used at several places in the process, in other words a "split" of a branch can be done. Because of the restriction on the RETURN number, we are sure that the process can be translated into on single Hive query, which possibly contains nested sub-queries. Thereby, the translation algorithm is the following: starting from RETURN, it constructs the tree of sub-queries using the paths of identifiers defined by the CONNECT and FILTER primitives until it reaches a stop condition with an EVAL.

Demonstration Example. For instance, we consider the following process. Suppose one has a tourism agency with several already stored datasets in a Hive warehouse such as transportation timesheets (*e.g.* planes and/or trains), restaurant list, description of points of interest (POIs)... and several already existing services to query each single dataset for example "give me the next plane leaving London for NYC" or "list the 1-star restaurants in Paris". One possible new usecase could be: "I want to travel from one place to an other one as a tourist and if it exits a long enough connexion (more than k hours) I'd like to go to the restaurant." This application needs to combine results extracted from various datasets. Considering that Q-plane, Q-diner and Q-tourism respectively extract relevant information from the plane, the restaurant and the POIs databases, the final results might be obtained using our primitives as shown in Figure 2a.

These primitives make it possible to generate a single query directly executable by Hive. For example, the HAP demonstration example (Figure 2a) can be translated into the query of Figure 2c using the translation rules of Figure 2b. Their advantage is that they allow to apply a range of analysis and optimizations in the query generation process (see Section 4) while dealing with heterogeneous datasources with their dedicated query languages (see Section 3).

3. HETEROGENEOUS SOURCES

We extend the number of supported query languages which can be used in pipelines. Indeed, HAP allows to query other data structures (than relational) using the conventional language of each structure. HAP is then able to compiled into a single query this aggregation of different queries while optimizing (1) the translation of each non-SQL query and (2) the final output query (see Section 4).

RDF & SPARQL. The Resource Description Framework (RDF) is a language standardized by W3C to express structured information on the Web as graphs [7]. RDF data is structured in triples written (s p o). SPARQL is the standard RDF query language [10].
In this context, we propose and share RDFHive: a distributed RDF datastore benefiting from Apache Hive. RDFHive is designed to leverage existing Hadoop infrastructures for evaluating SPARQL queries. RDFHive relies on an optimized translation of SPARQL queries into SQL queries that Hive is able to evaluate. The sources of RDFHive are openly available under the CeCILL[1] license from: https://github.com/tyrex-team/rdfhive.

JSON & JSONPath. JSON[2] is an open-standard format that uses human-readable text to transmit data objects con-

[1]CeCILL v2.1: http://www.cecill.info/index.en.html
[2]JSON website: http://json.org/

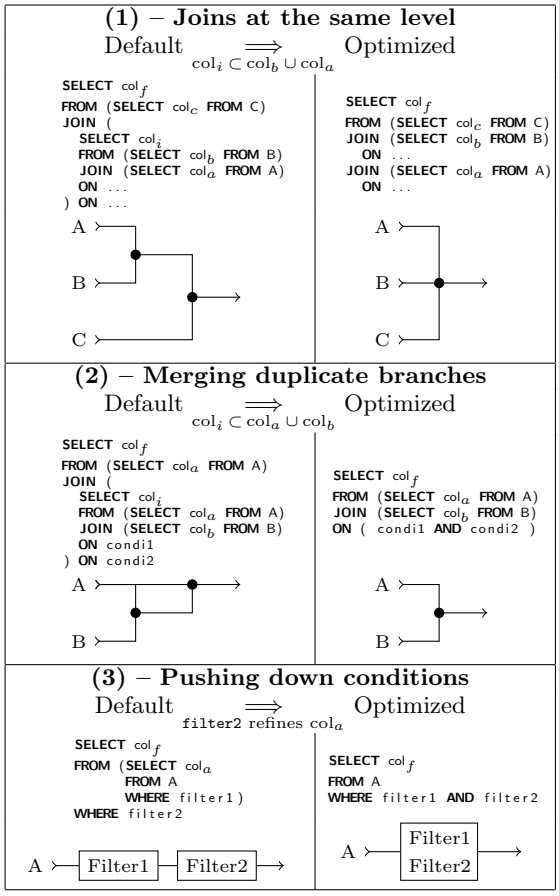

Figure 3: Pipeline Rewriting Rules.

sisting of attribute-value pairs. JSONPath [5] is a component allowing to find and extract relevant portions out of JSON structures. The Hive built-in `get_json_object` function supports a limited fragment of JSONPath. Thereby, HAP can also aggregate results extracted from JSON files.

XML & XPath. The Extensible Markup Language (XML) is a W3C markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable [1]. XPath [2] is a query language for selecting nodes from an XML document.
When XML documents are loaded as single string columns, HAP accepts the Hive built-in set of functions related to XPath *e.g.* `xpath(xml_string,xpath_expression_string)`.

4. OPTIMIZATIONS

An advantage of HAP is the possibility of setting up processes in few lines of code while allowing a global optimization of the whole pipeline. The optimizing compiler of HAP applies optimizations that cross the barriers of individual subqueries (it performs intra and inter-subqueries optimization). Using HAP makes it possible to further merge and reorder sub-queries, and to move filters out of the subquery in which they initially appeared, when appropriate. Obviously, the final generated query can still further benefit from optimizations performed by the execution engine.

```
select dep arr poi depHour arrHour stop restau
from ( select dep arr depHour arrHour stop
       from ( Q-plane ) as ini_1
       where arrHour-depHour > k
     ) as 1
join ( select place restau
       from ( Q-diner ) as ini_2
     ) as 2 on ( place=stop )
join ( select location poi
       from ( Q-tourism ) as ini_3
     ) as 3 on ( location=stop )
```

Figure 4: Optimized Query of the Example.

4.1 Using Statistics on Data

Hive translates its queries into sequences of MapReduce stages. As a consequence, it will have to decide for each MapReduce stage of a join which sequence is streamed through the reducers. Conventionally, the last specified table is always chosen to be streamed whereas the others are buffered. Therefore, it helps to reduce the memory needed in the reducer – for buffering the rows for a particular value of the join key – by organizing the tables such that the largest tables appear last in the sequence.

HAP attributes a weight $w(id)$ to each identifier id. These weights – which refer to the estimated size of the sets – are computed using statistics on data. To do so, HAP stores for each table T having a set of fields $\{f_1^T, \ldots, f_n^T\}$ the following information: the number of tuples in the table n_T, the numbers of distinct values in each field $v(f_1^T),\ldots,v(f_n^T)$. We assume that each value appears with equal probability (uniform distribution) in a column. Therefore, considering a `CONNECT` to obtain id_3 between id_1 and id_2 according to `[[` $f_i^{T_1}=f_j^{T_2}$ `]]`, we define the obtained weight $w(id_3)$ as follows:

$$w(id_3) = \min\left(\frac{w(id_1).w(id_2)}{v(f_i^{T_1})}, \frac{w(id_1).w(id_2)}{v(f_j^{T_2})}\right)$$

Similarly, the weight of an `EVAL` identifier is computed going directly in the query using the same strategy as above.

As a consequence, HAP can reorder the identifiers of a `CONNECT` using the respective weights to guarantee that the estimated largest table is the last of the sequence. Indeed, "`CONNECT` i j k ..." becomes "`CONNECT` j i k ..." if $w(i) > w(j)$.

4.2 Pipeline Rewriting Rules

A round of static rewriting is also realized. Actually, HAP tries to reorder the primitives according to the rules schematically presented in Figure 3.

Nested Queries. First of all, HAP tries to limitate the number of nested sub-queries in order to increase the Hive parallelism level. As shown in Figure 3, trying to group the connections and avoiding duplications can be done if the selected columns remain the same between levels *i.e.* no new column is created (by aggregation for instance).

Condition push down. In a second time, HAP tries to execute filters as soon as possible in order to limit (at most) the size of intermediate results. To do so, HAP pushes down filters while the columns involved in the conditions are located on the same branch.

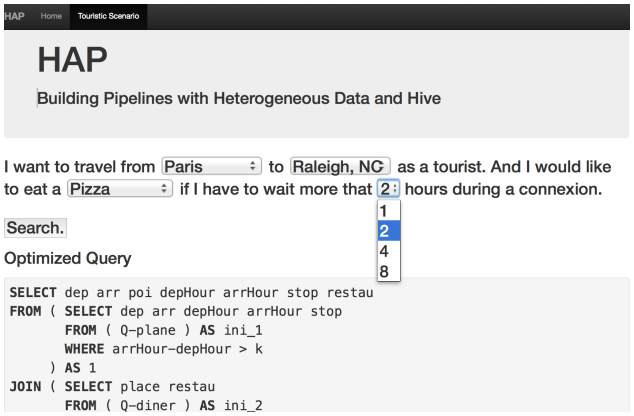

Figure 5: Application Screenshot.

Overall Guarantees. The following guarantees are offered for a query generated from a HAP description:

1. the final query minimizes the maximum depth of nested subqueries (HAP attempts to flatten subqueries as much as possible by joining on the maximum number of columns at a given level)

2. filters of the final query are as close to the sources as possible.

Considering the example shown Figure 2a, the previous optimization strategies lead to the query obtained Figure 4. Actually, compared to Figure 2c, the `FILTER` has been pushed closed to Q-plane, there is one nested query level less and the Q-tourism query is last since there are more POIs than planes or retaurants.

5. DEMONSTRATION DETAILS

The typical demonstration scenario is based on the touristic example introduced in Section 2 where information about planes, points of interest and restaurants are aggregated. This scenario, which widely extends the example previously presented, highlights several advantages of HAP:

1. Datasources have various structures which implies the use of various query languages *e.g.* POIs are stored in RDF – they should be queried with SPARQL – whereas restaurants are stored in relational CSV files.

2. Datasources have also different size spread over orders of magnitude *e.g.* GBs of POIs and only some kBs of planes.

3. The `FILTER` primitives needed in this usecase are complex *e.g* in "real" datasets, locations are given trough their latitude and longitude, thereby computing distances implies to use the Haversine formula.

4. The range of optimizations allows to avoid several subqueries joining at the same level the sub-processes which are initially written in various query languages.

Actually, attendees will be able to interact directly by writing HAP programs around this usecase. Moreover, the whole process will be runnable step-by-step in order to show the various optimizations realized, see *e.g.* Figure 5.

6. RELATED WORK & CONCLUSION

Accessing heterogeneous datasources can be done using multi-database systems [9] or data integration systems [4]. The typical solution is to define a common intermediate data model and also to provide a query language. The dominant state-of-the-art architectural model is the mediator/wrapper architecture: each datasource has an associated *wrapper* which is in charge of the translations between the datasets and the *mediator* which centralizes information. However, this architecture, used *e.g.* in [8], might suffer from the centralization of the mediator and the frequent translations done by the wrappers when datasources have to be distributed across a cluster. On the other hand, some systems – such as Hue[3] – aggregate only distributed components in order to have an end-to-end distributed pipeline. Finally, the HAP generated query can be plugged on tools such as [6] in order to decide the best execution engine.

HAP tries to benefit from both strategies: (1) the executions remain in a distributed context at any time since pipelines are *in fine* translated into MapReduce tasks, (2) it gets rid of wrappers/mediator bottlenecks by storing heterogeneous datasets directly in the Hive warehouse, (3) it uses a higher-level set of primitives to glue together heterogeneous datasets allowing sub-processes in various query languages and (4) it applies optimizations on the overall pipeline.

7. REFERENCES

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language. *W3C Rec. REC-xml-19980210*, 16:16, 1998.

[2] J. Clark, S. DeRose, et al. Xml path language, 1999.

[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] A. Doan, A. Halevy, and Z. Ives. *Principles of data integration*. Elsevier, 2012.

[5] S. Goessner. Jsonpath-xpath for json, 2007.

[6] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand. Musketeer: all for one, one for all in data processing systems. In *European Conference on Computer Systems*, page 2. ACM, 2015.

[7] P. Hayes and B. McBride. RDF semantics. *W3C Rec.*, 10, 2004.

[8] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, and J. Pereira. Cloudmdsql: Querying heterogeneous cloud data stores with a common language. *Distributed and Parallel Databases*, pages 1–41, 2015.

[9] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[10] E. PrudHommeaux, A. Seaborne, et al. SPARQL query language for RDF. *W3C Rec.*, 15, 2008.

[11] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies*, pages 1–10. IEEE, 2010.

[12] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *VLDB Endowment*, 2(2):1626–1629, 2009.

[3]Hue website: http://gethue.com/