MELOGRAPH: Multi-Engine WorkfLOw Graph Processing

Camelia Elena Ciolac[*]
Chalmers University of Technology
Gothenburg, Sweden
camelia@chalmers.se

ABSTRACT

This paper introduces MELOGRAPH, a new system that exposes in the front-end a domain specific language(DSL) for graph processing tasks and in the back-end identifies, ranks and generates source code for the top-N ranked engines. This approach lets the specialized MELOGRAPH be part of a more general multi-engine workflow optimizer. The candidate execution engines are chosen from the contemporaneous Big Data ecosystem: graph databases (e.g. Neo4j, TitanDB, OrientDB, Sparksee/DEX) and robust graph processing frameworks with Java API or packaged libraries of algorithms (e.g. Giraph, Okapi, Flink Gelly, Hama, Gremlin). As MELOGRAPH is work in progress, our current paper stresses upon the state of the art in this field, provides a general architecture and some early implementation insights.

Keywords

Big Data ecosystem; multi-engine workflow; graph processing tasks

1. STATE OF THE ART

The multitude of frameworks and datastores in the Big Data ecosystem, with their different data models, libraries of implemented algorithms, available connectors and performance profiles, make it challenging to select the right tools when building a Big Data storage and processing architecture. Instead, one can benefit from the open-source licenses and community editions to set up a polyvalent architecture, with polyglot persistence and multiple processing engines.

On the data management side, with the emergence of NoSQL datastores that usually complement relational databases in the enterprise data architecture, there has been a continuous interest for polyglot persistence (also referred to as "multiparadigm data storage" [7]). Besides explaining this data strategy as a specialization in data representation to better reflect the application data types, [7] highlighted the necessity of multiparadigm programming with Big data. At that time, in 2010, this collocation defined the lack of an unified query language among NoSQL datastores and the need develop an integration layer using the programming languages (e.g. Java, Python) in which these datastores provided query APIs.

This research topic remained an open challenge and few robust solutions have been provided to this date. From the industry, we cite Oracle Big Data SQL [13] with its "Query Franchising" strategy of unifying queries over Oracle databases, Hadoop and NoSQL datastores. From the open source community, we highlight the efforts of the Cascading Lingual project [1] which uses Apache Calcite (formerly named Optiq) to support SQL over a variety of data providers, by pushing as much as possible of the query processing to the datastores that manage the data. In the scientific community, the most recent and remarkable development in this field is the BigDAWG polystore[4] [6], promoting the "islands of information", each with its query language that finally maps to the underlying storage engines' native language through "shims" acting as translators.

Perhaps even more challenging than with storing and querying Big Data, developing complex analytics workflows needs choosing among the plethora of candidate frameworks. Several Big Data architecture patterns were promoted (e.g. [9]), in which scripting or a workflow manager orchestrates tasks on pre-established engines. Apache Oozie, Azkaban, Luigi are the most popular workflow managers in Hadoop, yet after inspecting their set of supported engines we conclude that they were rather designed for ETL pipelines, not complex analytics workflows. Let us underline that out-of-the-box they don't support scheduling on specialized graph processing frameworks.

A more mature approach to tackle the complexity of the problem is to enhance a workflow manager with some "intelligence", with the ability to decide between several frameworks and to generate all necessary ETLs between the chosen execution engines. Therefore nowadays an increasing focus is put on designing multi-engine workflows [17], addressing the various facets of this problem: process modelling and detail abstracting of execution, data formats, data sources [11]; execution engine selection modelled as a global optimization problem [12]; dynamic scheduling and resource management [3].

A couple of publications present prototypes of systems that address all these dimensions: QoX [14], IReS [3], Musketeer [8]. Common to all of them is the decoupling of

[*]Big Data @ Chalmers, ICT Area of Advance

the front-end from the back-end engine, yet each one has its specific approach; for example QoX exposes an XML-based "proprietary flow metadata language" (xLM) at the front-end and in the back-end uses a library of operations [14], while Musketeer exposes a SQL-like query language and a "Gather-Apply-Scatter style" domain specific language (DSL) for graph processing in the front-end and generates code based on code templates in the back-end [8]. Also, all three aforementioned systems use cost-based optimization, with specific search space and search algorithms for single- or multi- engine optimal scheduling of the workflow. Case studies and demonstrations of some of the aforementioned systems are anchored in the contemporaneous Big Data ecosystem: IReS with a practical demonstration over { Hadoop, Hama, Spark, PostgreSQL with HDFS, HBase, Elasticsearch} [3], Musketeer with an evaluation over {Spark, Hadoop MapReduce, GraphChi, PowerGraph, GraphLINQ} [8]. These proofs of concept did address graph processing tasks to a limited extent, without the intention to include a larger inventory of options currently available in the Big Data ecosystem.

Except from these studies, few practical results of integrating graph analytics in larger workflows are reported (e.g. graph processing and NLP [5]). In an extensive review of the current state of the art of graph processing on Big Data [2], the conclusion is that nowadays "most of the large scale graph processing platforms have the limitation that they are not able to connect their graph processing capabilities with the vast ecosystem of other analytics systems".They cite the industrial pioneering work of Teradata Aster 6.0 that "have started to tackle this challenge by extending its analytics capabilities with a multi-engine processing architecture"[2]. A deeper investigation into this system [15] reveals that graphs " can be derived, or projected, from many sources and types of business data" and that a "graph analytics program, or graph function, is modeled as a polymorphic table operator like Aster's existing SQL-MR analytics functions", making it possible to be invoked even from SQL queries.

Finally, let us highlight some differences between MELOGRAPH and the aforementioned systems. Compared to Aster's SQL-GR, MELOGRAPH not only aims to support a variety of data sources as input, but also to support various execution engines for each task; however MELOGRAPH's DSL doesn't make it embeddable in SQL queries. Compared to the other systems we innovate in the following aspects: 1) MELOGRAPH addresses specific graph processing tasks, the suitability of storing data in a graph database prior to processing, possible optimizations in cascaded graph tasks; 2) MELOGRAPH generates Java source code on-the-fly and does not treat operators as black boxes; 3) MELOGRAPH does not need any syntax validation since this is already enforced by the MPS editor (based on the DSL structure).

2. SOME MOTIVATING EXAMPLES

Before diving in the discussion about MELOGRAPH, let us first depict some examples of workflows containing graph processing tasks. In a first example, from the citations network in Computer Science we firstly retain a subgraph having in nodes only publications cited at least k times, then among them find communities based on papers' co-citation and lastly evaluate how close is each community to a clique structure. In this case we have a sequence of three graph processing tasks in the workflow: a k-core graph processing task, a label propagation (or other epidemics-based identification of network structure) and finally aggregations to compute local clustering coefficients.

Secondly, a use case from the entertainment industry, where one can use YouTube Data API to extract the network of videos (retaining video's unique identifier, metadata about its channel/user, video title, description) and their related-to relationships (a search for relatedToVideoId). Once the graph is obtained, two processing tasks are launched sequentially: a pattern matching query over the graph and a text semantic analysis over the resulting subgraphs video's description in order to extract entities (entity recognition task).

We end by presenting an example from [15]: a workflow from the marketing domain, where a PageRank graph analysis task identifies the most influential customers, a different task performs sentiment analysis of customers' reviews to discover the satisfied ones, and finally a join of the two resultsets is performed.

3. AN OVERVIEW OF MELOGRAPH

The main components that build up the MELOGRAPH internals are: the DSL kneaded in a language workbench, the Candidate Solutions Assembler, the Ranker and the Solution Packager. Now let us present some details for each component.

Our domain specific language, *MELOGRAPHy*, is designed and built to facilitate the definition of graph processing tasks in a manner agnostic to the execution engine. In a first stage, *MELOGRAPHy* supports functionality only based on the inventory of algorithms and queries available in the engines. However, in a second stage we want to extend the language to allow custom vertex-centric iterations using the Bulk Synchronous Parallel Model. We give some insights into our DSL in section 4 of the paper.

Based on the inventory of engines and on a set of patterns, the Candidate Solutions Assembler (CSA) builds the possible pipelines for solving the graph processing task. The CSA is concerned with the feasibility of the solution, not with its optimality, therefore some of the candidate solutions it produces may exhibit weak performance at runtime.

It is the role of the Ranker component to reward or to penalize candidate solutions based on a set of heuristics. Ideally, a cost-based model should be employed by the Ranker to build its final ranking of the candidate solutions; we'll address this aspect in a future study. From the cost perspective, for the moment we can anticipate adopting the approach to treat execution engines as black boxes; this idea already won consensus among both Big data management researchers (e.g. " A 'black box' approach makes a lot more sense when coping with disparate underlying engines" [16]) and multi-engine workflows researchers (e.g. [3] present an optimization of the workflow scheduling which is "orthogonal to (and in fact enhanced by) any optimization effort within a single engine").

Finally, the Solution Packager prepares all the necessary shell scripts to launch in execution the various tools found in each of the top-N ranked candidate solutions. Therefore, the final result of MELOGRAPH consists in a number of N folders in the local file system, each folder storing:

- the file containing the code source generated by the language workbench at compilation time;

- additional ETL scripts/programs to convert inputs to required format or to load data in a graph database;
- a driver script that orchestrates all the tools in that solution.

The next section presents how MELOGRAPH works, from receiving the input task to packaging a set of alternative solutions.

4. MELOGRAPH IN ACTION

Before presenting the processing workflow of MELOGRAPH, in current version, let us first provide some insights into basic MELOGRAPHy domain specific language.

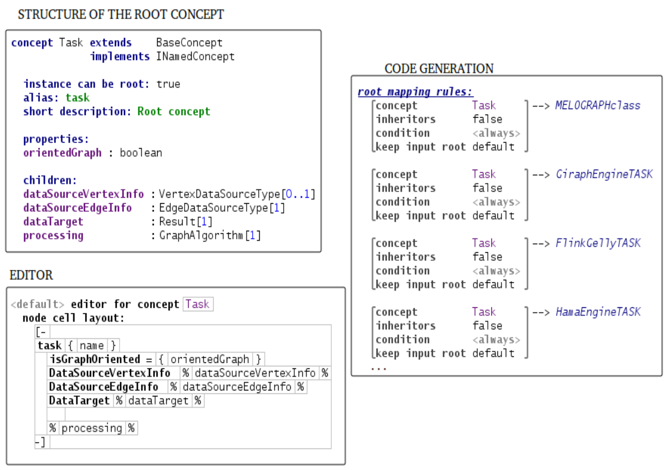
Regarding the DSL development environment we opted for the JetBrains MPS metaprogramming system[10] and made use of its model-to-model transformation approach to code generation. JetBrains MPS separates language development concerns into:

- structure (types of nodes in the Abstract Syntax Tree and their relationships), made of concepts organized in hierarchies;
- editor in charge of visualization of syntax from the user's perspective, optionally exhibiting some customized behaviour;
- generator which "defines the denotational semantics for the concepts in the language"[10].

The root concept in MELOGRAPHy's structure is the $Task$. For this concept, Figure 1 presents the three aforementioned concerns. Let us briefly make some comments.

Firstly, we highlight our choice to use the $GraphAlgorithm$ concept to abstract both queries (e.g. pattern matching) and graph analytics (e.g. PageRank).

Secondly, in terms of data sources, the minimal information needed to build and to analyze the graph is its structure, which is given by the edges together with their incident vertices identifiers. This is why $dataSourceEdgeInfo$ is mandatory, whereas $dataSourceVertexInfo$ is optional. We realize that this approach misses out the isolated nodes, but the user still has the option to load both the edges and the vertices data sets and hence include isolated vertices too. Besides eliminating unnecessary data loads, our approach has a more subtle benefit: it easily accommodates polyglot persistence, where vertex information is stored in a separate datastore than the information from which we build edges. One final comment is that we take into consideration adding one more child to the $Task$'s aforementioned structure, with the scope of empowering the user to suggest his/her preferred execution engine.



```
concept Task extends     BaseConcept
             implements INamedConcept

  instance can be root: true
  alias: task
  short description: Root concept

  properties:
  orientedGraph : boolean

  children:
  dataSourceVertexInfo : VertexDataSourceType[0..1]
  dataSourceEdgeInfo   : EdgeDataSourceType[1]
  dataTarget           : Result[1]
  processing           : GraphAlgorithm[1]
```

```
<default> editor for concept Task
  node cell layout:
    [-
    task { name }
      isGraphOriented = { orientedGraph }
      DataSourceVertexInfo  % dataSourceVertexInfo %
      DataSourceEdgeInfo  % dataSourceEdgeInfo %
      DataTarget % dataTarget %

      % processing %
    -]
```

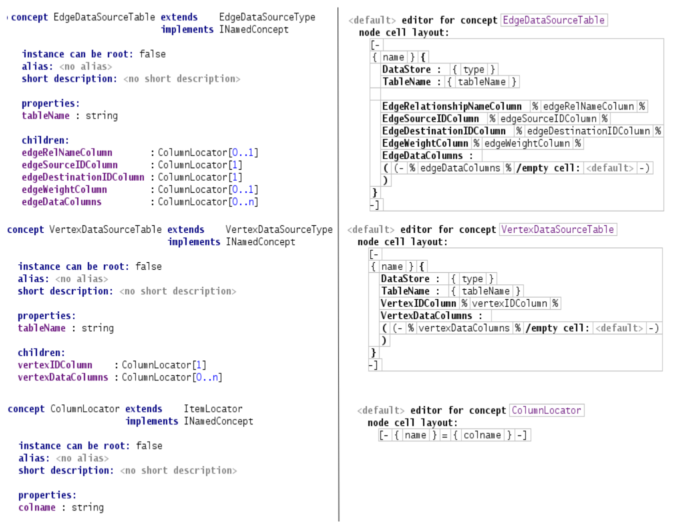
```
root mapping rules:
  [concept          Task     ] --> MELOGRAPHclass
   inheritors       false
   condition        <always>
   keep input root  default  ]

  [concept          Task     ] --> GiraphEngineTASK
   inheritors       false
   condition        <always>
   keep input root  default  ]

  [concept          Task     ] --> FlinkGellyTASK
   inheritors       false
   condition        <always>
   keep input root  default  ]

  [concept          Task     ] --> HamaEngineTASK
   inheritors       false
   condition        <always>
   keep input root  default  ]
  ...
```

Figure 1: Insights into MELOGRAPHy DSL



```
concept EdgeDataSourceTable extends     EdgeDataSourceType
                            implements INamedConcept

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  tableName : string

  children:
  edgeRelNameColumn       : ColumnLocator[0..1]
  edgeSourceIDColumn      : ColumnLocator[1]
  edgeDestinationIDColumn : ColumnLocator[1]
  edgeWeightColumn        : ColumnLocator[0..1]
  edgeDataColumns         : ColumnLocator[0..n]

concept VertexDataSourceTable extends     VertexDataSourceType
                              implements INamedConcept

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  tableName : string

  children:
  vertexIDColumn    : ColumnLocator[1]
  vertexDataColumns : ColumnLocator[0..n]

concept ColumnLocator extends     ItemLocator
                      implements INamedConcept

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  colname : string
```

```
<default> editor for concept EdgeDataSourceTable
  node cell layout:
    [-
    { name } {
      DataStore :  { type }
      TableName : { tableName }

      EdgeRelationshipNameColumn  % edgeRelNameColumn %
      EdgeSourceIDColumn  % edgeSourceIDColumn %
      EdgeDestinationIDColumn  % edgeDestinationIDColumn %
      EdgeWeightColumn % edgeWeightColumn %
      EdgeDataColumns :
      ( (- % edgeDataColumns % /empty cell: <default> -)
      )
    }
    -]

<default> editor for concept VertexDataSourceTable
  node cell layout:
    [-
    { name } {
      DataStore :  { type }
      TableName :  { tableName }
      VertexIDColumn % vertexIDColumn %
      VertexDataColumns :
      ( (- % vertexDataColumns % /empty cell: <default> -)
      )
    }
    -]

<default> editor for concept ColumnLocator
  node cell layout:
    [- { name } = { colname } -]
```

Figure 2: Illustration of edge and vertex data source concepts in MELOGRAPHy

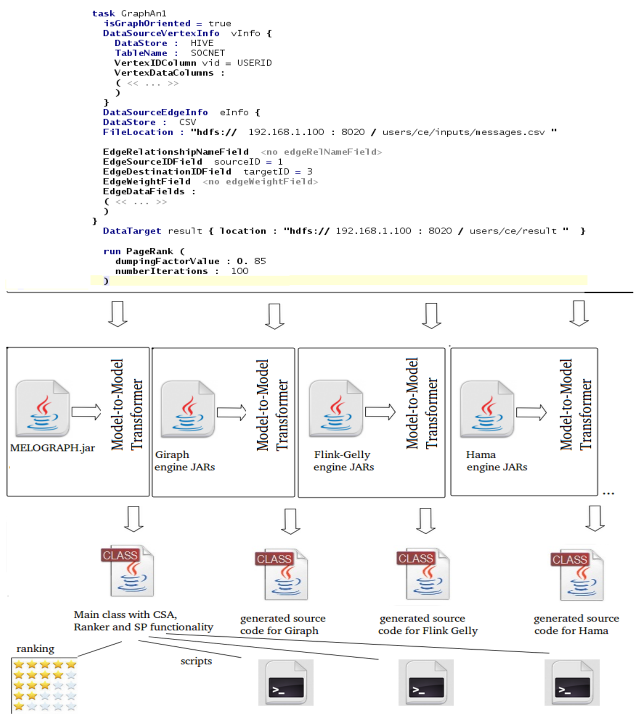
Figure 2 depicts the structure of concepts that extend $EdgeDataSourceType$ and $VertexDataSourceType$ for the specific case of table data source (e.g. in Hive; but also in a relational database in Oracle or MySQL). In a brief parenthesis we comment that given a relational database table as data source, MELOGRAPH will automatically include Sqoop in all candidate solutions pipelines and thus the actual input to the graph algorithm will be the HDFS file obtained after loading the data. Thus, in the source code generation templates we use file input directly in such cases.

Let us also remark that except from the case of graph databases (where this information is native), for the rest of the data sources it is mandatory to specify the means of accessing edges' endpoints. In the illustrated case of a table data source locators are column names, similarly in the case of a CSV file locators are field indices and in the case of HBase locators are fully qualified columns names from some column family.

We advance now to the MELOGRAPH workflow. As shown in Figure 1, the use of mapping configurations makes it possible to generate code for multiple engines at the same time according to a set of Java code templates developed by us. Details of this architecture are given in Figure 3.

Along with engines-specific code, we generate the class $MELOGRAPHclass$ that uses the functionality of all MELOGRAPH components discussed in the overview section of this paper. However, the ranking of the solutions can be obtained only when this Java program is run, consequently some of the heuristics need to be implemented in the model-to-model transformers, too.

With the data sources and the algorithm specified, MEL-



```
task GraphAn1
  isGraphOriented = true
  DataSourceVertexInfo  vInfo {
    DataStore :  HIVE
    TableName :  SOCNET
    VertexIDColumn vid = USERID
    VertexDataColumns :
    ( << ... >>
    )
  }
  DataSourceEdgeInfo  eInfo {
  DataStore :  CSV
  FileLocation : "hdfs://  192.168.1.100 : 8020 / users/ce/inputs/messages.csv "

  EdgeRelationshipNameField  <no edgeRelNameField>
  EdgeSourceIDField  sourceID = 1
  EdgeDestinationIDField  targetID = 3
  EdgeWeightField  <no edgeWeightField>
  EdgeDataFields :
  ( << ... >>
  )
}
  DataTarget result { location : "hdfs:// 192.168.1.100 : 8020 / users/ce/result "  }

  run PageRank (
    dumpingFactorValue : 0. 85
    numberIterations :  100
  )
```

MELOGRAPH.jar
Model-to-Model Transformer
Giraph engine JARs
Model-to-Model Transformer
Flink-Gelly engine JARs
Model-to-Model Transformer
Hama engine JARs
Model-to-Model Transformer
...
CLASS
CLASS
CLASS
CLASS
Main class with CSA, Ranker and SP functionality
generated source code for Giraph
generated source code for Flink Gelly
generated source code for Hama
ranking
scripts

Figure 3: MELOGRAPH generating multi-engine source code for a Task at runtime

OGRAPH's Candidate Solutions Assembler will emit all feasible pipelines to accomplish the task. This involves assembling tools based on predefined patterns. Except from the data movement scripts that MELOGRAPH will eventually generate, we need to embed these patterns in the model-to-model transformer for each individual engine too. This approach is required by the fact that all source files are generated at once and need to be consistent among them. MELOGRAPH's Ranker will then rank the solutions and output a list of the top-N solutions.

5. CONCLUSIONS

The current state of the art in multi-engine processing workflows on Big Data displays increasing interest in tackling the practical challenges raised by the contemporaneous Big Data ecosystem. We join this emerging research movement and present in this paper the first bricks that we've put in developing MELOGRAPH.

MELOGRAPH is specialized in graph processing tasks which are generally part of ampler workflows; however, the whole workflow needs to be developed in JetBrains MPS, too. In the design of MELOGRAPH we address heterogeneity at all levels, from the data sources, execution engines and runtime computing environment (through the heuristics used in ranking candidate solutions).

6. REFERENCES

[1] Cascading lingual, http://www.cascading.org/projects/lingual/.

[2] F. Bajaber, S. Sakr, O. Batarfi, A. Altalhi, R. Elshawi, and A. Barnawi. Big data processing systems: State-of-the-art and open challenges. In *Proceedings of the ICCC 2015*, pages 1–8, April 2015.

[3] K. Doka, N. Papailiou, D. Tsoumakos, C. Mantas, and N. Koziris. Ires: Intelligent, multi-engine resource scheduler for big data analytics workflows. In *Proceedings of the 2015 ACM SIGMOD*, SIGMOD '15, pages 1451–1456, New York, NY, USA, 2015. ACM.

[4] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The bigdawg polystore system. *SIGMOD Rec.*, 44(2):11–16, Aug. 2015.

[5] D. Ediger, S. Appling, E. Briscoe, R. McColl, and J. Poovey. Real-time streaming intelligence: Integrating graph and nlp analytics. In *Proceedings of IEEE HPEC, 2014*, pages 1–6, Sept 2014.

[6] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. Mattson, S. Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik. A demonstration of the bigdawg polystore system. *Proceedings of the PVLDB Endow.*, 8(12), 2015.

[7] D. Ghosh. Multiparadigm data storage for enterprise applications. *IEEE Softw.*, 27(5):57–60, Sept. 2010.

[8] 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 *Proceedings of the EuroSys '15*, pages 2:1–2:16, New York, NY, USA, 2015. ACM.

[9] M. Grover, T. Malaska, J. Seidman, and G. Shapira. *Hadoop Application Architectures*. O'Reilly, Beijing, 2015.

[10] JetBrains. Mps user's guide, https://confluence.jetbrains.com/display/mpsd32/mps+user

[11] V. Kantere and M. Filatov. Modelling processes of big data analytics. In J. e. a. Wang, editor, *WISE (1)*, volume 9418 of *Lecture Notes in Computer Science*, pages 309–322. Springer, 2015.

[12] G. Kougka, A. Gounaris, and K. Tsichlas. Practical algorithms for execution engine selection in data flows. *Future Generation Computer Systems*, 45(Complete):133–148, 2015.

[13] ORACLE. Unified query for big data management systems integrating big data systems with enterprise data warehouses. Technical report, ORACLE, ORACLE, Jan. 2015.

[14] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing analytic data flows for multiple execution engines. In *Proceedings of the 2012 ACM SIGMOD*, pages 829–840, New York, NY, USA, 2012. ACM.

[15] D. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Shenoi, M. Tan, and Y. Xiao. Large-scale graph analytics in aster 6: Bringing context to big data discovery. *Proc. VLDB Endow.*, 7(13):1405–1416, Aug. 2014.

[16] M. Stonebraker. The case for polystores, http://wp.sigmod.org/?p=1629, July 2015.

[17] D. Tsoumakos and C. Mantas. The case for multi-engine data analytics. In D. e. a. an Mey, editor, *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 406–415. Springer Berlin Heidelberg, 2014.