Aura: A Flexible Dataflow Engine for Scalable Data Processing

Tobias Herb, Lauritz Thamsen, Thomas Renner, Odej Kao

Technische Universität Berlin
firstname.lastname@tu-berlin.de

Abstract. This paper describes Aura, a parallel dataflow engine for analysis of large-scale datasets on commodity clusters. Aura allows to compose program plans from relational operators and second-order functions, provides automatic program parallelization and optimization, and is a scalable and efficient runtime. Furthermore, Aura provides dedicated support for control flow, allowing advanced analysis programs to be executed as a single dataflow job. This way, it is not necessary to express, for example, data preprocessing, iterative algorithms, or even logic that depends on the outcome of a preceding dataflow as multiple separate jobs. The entire dataflow program is instead handled as one job by the engine, allowing to keep intermediate results in-memory and to consider the entire program during plan optimization to, for example, re-use partitions.

Keywords: Parallel Dataflow, Large-scale Data Analysis, Control Flow, Execution Plan Optimization

1 Introduction

More and larger datasets become available as, for example, sensing device data, user-generated content, and software logs. Gaining insights into this data is becoming increasingly important for many applications. For analyzing this data quickly and efficiently, different scalable data analytic frameworks have been developed. Prominent examples include MapReduce [5], Spark [16,15], and Flink [1]. In general, these systems share similar objectives, namely hiding the difficulty of parallel programming, providing fault tolerance, and allowing programs to be run efficiently on different cluster infrastructures. Developers write sequential programs and the processing framework takes care of distributing the program among the available compute nodes and executing each instance of the program on appropriate data partitions.

In most frameworks, the execution plan is static. Even if some frameworks support iterative programs, users cannot define arbitrary termination criteria. Furthermore, to the best of our knowledge there is no dedicated support for executing one of multiple branches of dataflows based on a condition. At the same time, analytical programs implementing, for example, graph or machine learning algorithms often require control flow. One approach to implement these structures in dataflow systems is to do it on client-side [16,15]. In this scenario, the client-side driver program creates and submits multiple jobs. However, even if the necessary dataflows jobs are generated from higher-level programming abstractions, executing a program as multiple jobs is often less efficient. It entails full data materialization between jobs, which often means writing these intermediate results to disk. Furthermore, the execution engines only knows about single jobs, not the entire dataflow program, limiting the scope of automatic plan optimization to parts of the programs.

With this motivation, we present Aura, a flexible dataflow engine for scalable data processing. Aura contains a novel mechanism for control flow that enables adjustments of the execution plans at runtime. This allows developers to express more complex analytic programs as a single job, allowing more optimized program execution. Aura realizes control-flow by the interaction between the centralized stepwise evaluation of control-flow logic and the thereby controlled distributed execution of the parallel dataflow.

Outline. The remainder of the paper is organized as follows. Section 2 describes related work. The system architecture and its components is presented in Section 3. Section 4 discusses Aura's program representation and translation. Section 5 presents its distributed execution model. Section 6 concludes this paper.

2 Related Work

This section describes the evolution of distributed dataflow engines and their programming abstractions. As the execution engines support more general job graphs and features like automatic plan optimization, the programming abstractions become more high-level and declarative.

2.1 Distributed Execution Engines for Parallel Dataflows

Aura builds upon the following engines and adds dedicated support for control flow.

MapReduce. The MapReduce [5] paradigm and its implementations allow to execute the two user-defined functions *Map* and *reduce* data-parallely on hundreds of commodity nodes. The execution is fault-tolerant as the results of each execution step are written to a fault-tolerant distributed file system and failing tasks are re-started.

Directed Acyclic Job Graphs. Systems like Dryad [7] and Nephele [13] build upon the data-parallel execution model, but allow more general job graphs than the two alternating functions with in-between data shuffling. As an example, tasks in Nephele can be connected in a general directed acyclic graph in which parallel tasks instances are connected either point-to-point or all-to-all. Furthermore, connections can be disk-based or in-memory.

Relational Operators and Plan Optimization. While tasks in Nephele and Dryad execute arbitrary user code, Scope [4] and Stratosphere/Apache Flink [3,1] provide a set of pre-defined operators, which include common database operators like joins, but also the second-order functions Map and Reduce. Both Scope and Stratosphere automatically compile and optimize logical query plans to physical plans from SQL-like programming abstractions.

Iterative Dataflows. Some systems, including Flink and Naiad [9,8] also provide dedicated support for iterative programs, allowing directed cyclyc job graphs. Both Flink and Naiad can also process datasets incrementally [6], allowing to exploit the sparse computational dependencies of many iterative algorithms.

Resilient Distributed Datasets. Spark [16] also allows scalable data analysis on commodity clusters, but is based on Resilient Distributed Datasets (RDD) [15]. RDDs are in-memory datasets that are distributed across nodes and on which operations are executed in parallel. The fault-tolerance in Spark does not rely on disk-based replication but on linage, so parts of RDDs can be recomputed when nodes fail.

2.2 Programming Abstractions for Parallel Dataflows

The programming abstractions for parallel dataflows are becoming more declarative and more high-level. At the same time, there is a development towards programming languages instead of mere library-based APIs. As such languages include control flow, they provide a direct motivation for supporting control flow on the level of the dataflow engine.

Declarative Programming Abstractions. Pig [10] and Hive [12] are SQL-like programming abstractions for MapReduce. Programs are arguably higher-level and more declarative. In fact, a single Pig or Hive programs usually results in multiple MapReduce jobs. Job compilation, thus, decouples the programming abstraction from actual dataflow jobs. DryadLINQ [14] and the Scope scripting language [4] are similar in that programmers are not limited to the two second-order functions Map and Reduce, but can also use typical relational operations.

Deep Language Embedding. While many programming abstractions for parallel dataflows are library-based, which are often tightly coupled to the underlying execution engine, Emma [2] uses *deep language embedding* to provide much more implicit parallelism. Emma compiles execution plans directly from code -rather than with an API- to enable a more holistic analysis of dataflow programs. Thus enabling additional optimizations, as well as a more complex control flow.

3 System Architecture

Figure 1 shows an overview of Aura's system architecture. As many scalable data processing frameworks [5,16,13], the architecture follows the master/worker pattern for parallelizing the execution over many nodes, as visible in (a) of the system overview. In general, Aura consists of three logical core entities:

- the master, called *Workload Manager* (WMs),
- one or more worker instances, called *Task Managers* (TMs), and
- one or more client instances

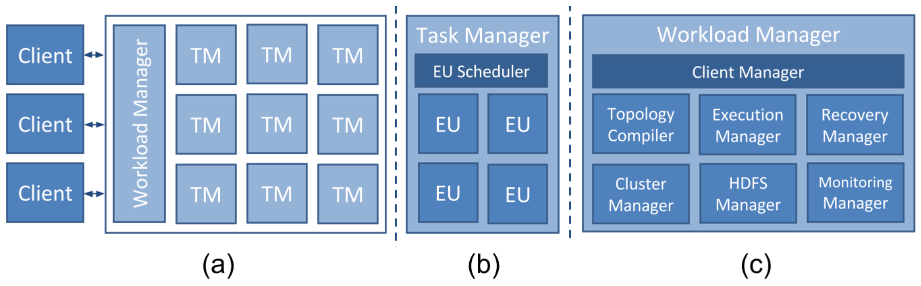

Fig. 1. Aura's system architecture: (a) overall master/worker architecture, (b) Task Manager component, and (c) Workload Manager component.

The following describes the inner structure and responsibilities of these components as well as the interactions between them in more detail.

3.1 Workload Manager

The central *Workload Manager* component manages and coordinates each running system instance of Aura. Its inner parts are shown in (c) of Figure 1. A fundamental service of the WM is the cluster management, which entails acquiring, coordinating, and monitoring available cluster resources. For instance, newly available or failed nodes are signaled to all other components by the central cluster manager.

Submitted dataflow programs are received by the WM. Afterwards, the WM internally creates a *Topology Controller* (TC) that takes care of the lifecycle management for the program. The lifecycle includes the parallelization, scheduling, deployment, monitoring and control-flow evaluation of the distributed dataflow program. Intermediate state and aggregated computation results of the executing dataflow are continously propagated back from the TMs to the coresponding TC. Thus, the TC is able to decide dynamically over the next execution steps of the program, depending on the gathered information. This novel feedback mechanism enables dynamic dataflow execution and allows to express more complex analytical programs than possible with static dataflows.

Another important function of the WM is the coordinated access to an underlying distributed file system such as the Hadoop Distributed File System (HDFS) [11]. The WM keeps track of all file resource locations and schedules
]

file access to co-located worker nodes to minimize network access. In case of failures, e.g. signaled by the cluster management, the TC triggers the necessary fault-tolerance logic to suspend execution, recover from failure, and resume distributed execution.

3.2 Task Manager

Computational tasks of a dataflow program are delegated and scheduled to *Task Managers* (TM). A TM consists essentially of a set of execution units, which perform tasks in parallel, as shown in (b) of Figure 1. Communication of dependent tasks is realized either via in-memory channels for tasks scheduled to different execution units of the same TM, or via network channels for tasks scheduled to execution units of different TMs. A Task Manager has as many *Execution Units* (TM-EU) as cores are available to ensure that each computational task is assigned to a core.

3.3 Client

The client is responsible for submitting dataflow programs to a runnning instance of Aura. It can obtain status updates and possibly failure messages (stack trace) from the WM during program execution. The engine supports parallel programs, submitted from different clients.

4 Program Representation and Translation

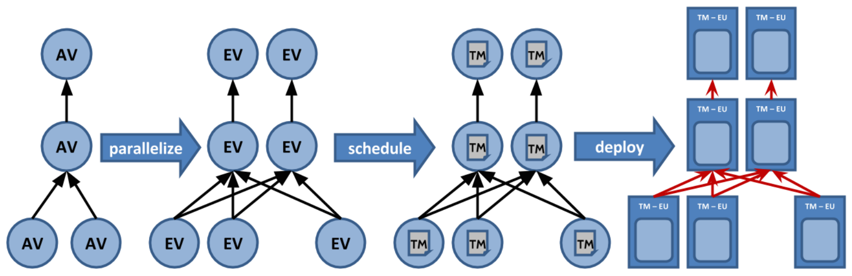

Fig. 2. Paralellizing, scheduling, and deployment of a physical plan.

This section treats the operator plan (i.e. logical plan) and physical plan (i.e. executable program representation) from the perspective of the execution engine. Further, it covers the translation process that generates the physical plan from the operator plan.

4.1 Operator Plan

An operator repesents a data-parallel function that transforms an individual data record or a collection of data records and returns the result. Typically, data records are ingested as arguments. Our system supports unary operators like map, filter, and aggregate functions as well as binary operators like a join or cartesian product. These operators are assembled to tree structures expressing a concrete analytical query, the so-called operator plan. The concrete processing takes place by streaming data records from bottom operators – i.e. the leafs in a concrete operator plan – along the edges up to the root operator, which receives the resulting data records of the processed query. The generation of a physical plan is realized by a query language compiler.

Additionally to the set of data-parallel operators we introduce two control-flow operators: (1) *select operator* and (2) *loop operator*.

The select operator acts as an if-statement in common programming languages and allows flexible dataflow execution depending on intermediate state of the current execution. It is parameterized with a predecessor plan, two successor plans, and a condition plan. The select operator first executes the predecessor plan and then the condition plan, which must produce a boolean result. Dependending on this result, the first (true) successor- or the second (false) successor plan is executed.

The loop operator realizes a loop-construct with termination condition like in an imperative programming language. The loop operator is parameterized with a loop plan and an termination plan. First, the loop plan is executed and then the termination plan is executed, which also has to yield a boolean result. Depending on the result the loop plan is re-executed (true) or not (false).

4.2 Physical Plan

The physical plan consists of multiple directed acyclic graphs (DAG), called partial physical plan, wired together logically by control-flow operators. The control-flow operators are parameterized with the corresponding partial physical plans and are composed together to a tree structure, comparable to an abstract syntax tree in compilers. Vertices of partial physical plans are distributable program units, containing one or more data-parallel operators. These vertices are deployed to execution units of the TMs. Edges between dependent vertices in the DAG are deployed as physical communication channels (TCP or in-memory). Edges between control-flow operators and vertices exchange only flow-logic information and no data records. An important feature of the channels is the so-called *intermediate cache*. This cache allows to materialize the data stream in memory and/or on disk either on the sender or receiver side. The intermediate cache is fundamental to enable control-flow, because the materialized data must first be evaluated by the condition plan of the control-flow operator and then be re-consumed by the selected successor plan.

4.3 Plan Translation and Deployment

The plan translation is the process where the physical plan is generated from the operator plan. In the first step operators are embedded in vertices. Embedding one operator in one execution vertex would lead, in the case of large operator plans, to a very inefficient execution because data records would have to be passed through many network channels. Under certain conditions several connected operators can be compactified to a macro operator which is then embedded in an single vertex. The necessary conditons for this compactification are:

– same partitioning keys,
– same partitioning strategy, and
– same degree of parallelism (DOP).

After the compactification (see figure 3) step the data-parallel instances of the vertices are generated, according to their DOPs. In the plan scheduling and deployment phase data-parallel vertex instances are assigned and shipped to the corresponding execution units of the TMs (see figure 2).

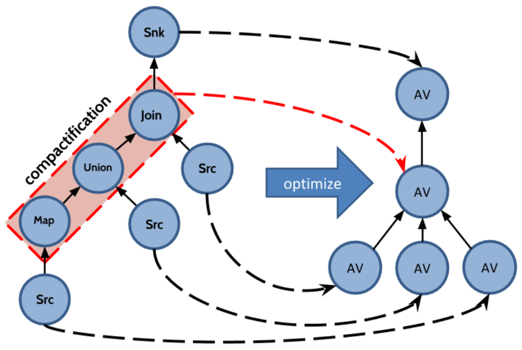

Fig. 3. Compactification of operators in the plan translation phase.

5 Distributed Execution

This section covers the push/pull-based data streams in our distributed execution model, which is shown in Figure 4, and describes the execution of control-flow operators.

5.1 Push/Pull-based Data Streams

Data records are either java objects or collections, e.g. sets or lists. For the external data transport, i.e. data transport between vertices, the records are serialized into transport buffers and pushed form sender to the receiver vertices. If

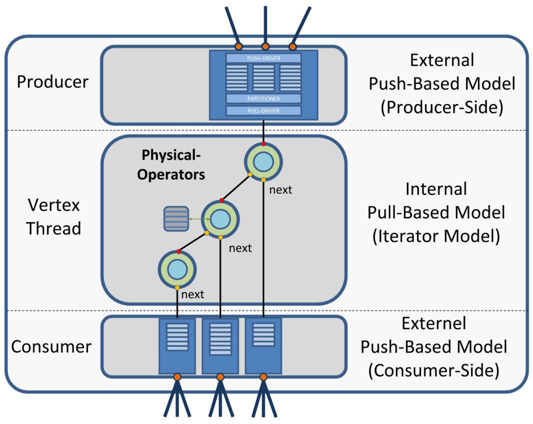

Fig. 4. Pull-based execution of data-parallel operators.

the receiver has not enough free memory it blocks the communication channel until enough memory is available. This receiver side blocking mechanism realizes the external back pressure behavior. The internal transport of data records between operators within a vertex follows a pull-based approach, where the root operators pulls individual records from the incoming transport buffer through all dependent operators and writes the results back to the outgoing transport buffer (see figure 4).

5.2 Control-Flow Execution

Control-flow operators can be seen as branching points between the execution of multiple partial physical plans. The control-flow tree resides centrally in the associated TC, where the evaluation of the current control-flow operation takes place. The processing of a dataflow program is carried out as a continous interplay between the execution of partial physical plans and the stepwise interpretation of the centralized control-flow tree. If the execution of a partial physical plan is terminated, the results are materialized in the outgoing intermediate caches of the sink vertices. The results are kept in the cache until one or more subsequent partial physical plans make an request. Point of cache expiration, i.e. when no more intermediate results are required, can be determined via static analysis of the complete physical plan. The runtime coordination between control-flow evaluation and distributed execution is realized via a so-called *activation token* which is passed back and forth between the TMs and the TC.

6 Summary

A large class of programs require control flow primitives such as iterations and conditional branches. Support for control flow on the level of a dataflow engine eliminates the need to express these programs as multiple jobs on the client-side. Furthermore, it allows to optimize programs in their entirety, across what previously were separate jobs: more intermediate results can be kept in memory, while the selection and configuration of physical operators–including data partitioning–can take the entire program into account.

Aura, the novel parallel dataflow engine presented in this paper, integrates dedicated support for control flow with automatic plan compilation and an efficient distributed execution model. This design makes our prototype an interesting platform to research the optimal execution of advanced dataflow programs as, for example, from the domains of graph analysis and machine learning.

Acknowledgements This work has been supported through grants by the German Science Foundation (DFG) as FOR 1306 Stratosphere and by the German Ministry for Education and Research as Berlin Big Data Center BBDC (funding mark 01IS14013A).

References

1. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinlaender, A., Sax, M.J., Schelter, S., Hoeger, M., Tzoumas, K., Warneke, D.: The stratosphere platform for big data analytics. VLDB Journal (2014)
2. Alexandrov, A., Kunft, A., Katsifodimos, A., Schüler, F., Thamsen, L., Kao, O., Herb, T., Markl, V.: Implicit Parallelism Through Deep Language Embedding. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. SIGMOD '15, ACM (2015)
3. Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., Warneke, D.: Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In: Proceedings of the 1st ACM Symposium on Cloud Computing. SoCC '10, ACM (2010)
4. Chaiken, R., Jenkins, B., Larson, P.A., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. Proc. VLDB Endow. 1(2) (2008)
5. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation. OSDI'04, USENIX Association (2004)
6. Ewen, S., Tzoumas, K., Kaufmann, M., Markl, V.: Spinning fast iterative data flows. Proceedings of the VLDB Endowment 5(11) (2012)
7. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In: Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. EuroSys '07, ACM (2007)
8. McSherry, F., Murray, D.G., Isaacs, R., Isard, M.: Differential Dataflow. In: Proceedings of the 6th Conference on Innovative Data Systems Research (CIDR). CIDR'13, ACM (2013)
9. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: A Timely Dataflow System. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM (2013)

10. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A Not-so-foreign Language for Data Processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. ACM SIGMOD (2008)
11. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (2010)
12. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: A Warehousing Solution over a Map-Reduce Framework. VLDB (2009)
13. Warneke, D., Kao, O.: Nephele: Efficient Parallel Data Processing in the Cloud. In: Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers. MTAGS '09, ACM (2009)
14. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K., Currey, J.: DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In: OSDI'08: Eighth Symposium on Operating System Design and Implementation. OSDI'08 (2008)
15. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. NSDI'12, USENIX Association (2012)
16. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster Computing with Working Sets. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing. HotCloud'10, USENIX Association (2010)