When to Use a Distributed Dataflow Engine: Evaluating the Performance of Apache Flink

Ilya Verbitskiy[*], Lauritz Thamsen[†], Odej Kao[†]
Technische Universität Berlin
[*]ilya.verbitskiy@campus.tu-berlin.de
[†]{firstname.lastname}@tu-berlin.de

***Abstract*—With the increasing amount of available data, distributed data processing systems like Apache Flink and Apache Spark have emerged that allow to analyze large-scale datasets. However, such engines introduce significant computational overhead compared to non-distributed implementations. Therefore, the question arises when using a distributed processing approach is actually beneficial. This paper helps to answer this question with an evaluation of the performance of the distributed data processing framework Apache Flink. In particular, we compare Apache Flink executed on up to 50 cluster nodes to single-threaded implementations executed on a typical laptop for three different benchmarks: TPC-H Query 10, Connected Components, and Gradient Descent. The evaluation shows that the performance of Apache Flink is highly problem dependent and varies from early outperformance in case of TPC-H Query 10 to slower runtimes in case of Connected Components. The reported results give hints for which problems, input sizes, and cluster resources using a distributed data processing system like Apache Flink or Apache Spark is sensible.**

I. Introduction

Helping businesses and researchers to discover relevant information in overwhelming collection of data is an important task. The digital age, however, generated an explosion of available data [1]. Datasets are getting increasingly bigger such that they do not fit into the memory of a single computer [2].

One way to approach this problem is to use distributed data processing. The Google File System (GFS) [3], a reliable and scalable file storage, and Google's MapReduce [4], a distributed data processing engine, presented a model for distributed data processing engines. Similar ideas are found in Hadoop[1], which includes a distributed file storage (HDFS) [5] and a distributed data processing engine (Hadoop MapReduce) [6]. Frameworks like Apache Spark[2] [7], Apache Storm[3] [8], and Apache Flink[4] improved upon the programming model and allow to describe arbitrary workflows. These frameworks orchestrate multiple computer nodes organized in a cluster. The nodes communicate with each other in order to distribute the workload.

However, such systems introduce significant computational overheads in the form of communication, serialization, scheduling, deployment, and synchronization. Because of this is often difficult to decide when to use such a distributed system. The question arises for which data sizes and cluster resources the computational advantages of a distributed system outweigh its overheads. Knowing which problems are better suited for non-distributed environments enables users to save costs on cluster resources. Moreover, this knowledge can also reduce development costs when the most cost-effective solution is chosen right away, so that only a single implementation has to be developed. In addition, using the comparison of single-threaded benchmarks against distributed scale-outs the user can also choose the system which better suits his financial or temporal constraints. He can decide whether performance gains of distributed systems, if any, justify their increased cost.

In this paper we present a performance evaluation of the data processing framework Apache Flink by comparing single-threaded implementations against their distributed counterparts. For our approach we followed the idea of the Configuration that Outperforms a Single Thread (COST) [10], which seeks to find a constellation of cluster resources and problem input sizes such that the distributed data processing engine shows performance gains over an efficient serial implementation. The comparison of distributed and serial implementations gives insights into the true performance benefits of a distributed data processing engine. Since Apache Flink and other distributed data processing systems like Apache Spark exhibit similar runtime behavior [11, 10, 12] the results are representative for a wider range of systems. We chose the three benchmarks, TPC-H Query 10, Connected Components, and Gradient Descent for the evaluation. For the cluster implementations, we used the examples provided with Apache Flink. The single-threaded versions used Objective-C and a single-threaded database engine. These benchmarks were chosen to showcase the performance of Apache Flink for three different typical application domains of distributed data processing.

Outline. The remainder of this paper is structured as follows. Section II provides some background on the Apache Flink framework and the COST metric. Section III presents the approach and explains the selection of the benchmarks. Section IV provides information on the benchmark implementations as well as the used datasets. Section V summarizes and Section VI discusses the evaluation results. Finally, Section VII concludes the paper and provides ideas for future work.

[1]Apache Hadoop, http://hadoop.apache.org/ [Accessed May 23, 2016]
[2]Apache Spark, http://spark.apache.org/ [Accessed May 23, 2016]
[3]Apache Storm, http://storm.apache.org/ [Accessed May 23, 2016]
[4]Apache Flink, https://flink.apache.org/ [Accessed May 23, 2016], originated from [9]

II. Background

This section provides information on the benchmarked distributed data processing engine Apache Flink. Subsequently, the ideas and motivations behind the COST metric are presented.

A. Apache Flink

Apache Flink is an open source platform for distributed data processing which embraces the Google Dataflow model [13]. It enables the user to write programs that can be distributed over a number of worker nodes. This makes it possible to process large-scale datasets faster than a single computer could.

Internally, Apache Flink represents job definitions using directed acyclic graphs (DAGs) [14]. The nodes of the graph are either sources, sinks, or operators. Source nodes read in or generate the input data, while sink nodes produce the output. The inner vertices are operators which execute arbitrary user-defined functions (UDFs) that consume input from incident nodes and provide input for adjacent nodes. The DAGs generated from the user's job definitions are then transformed into the more concrete *Execution Graphs*, which contain the necessary information for running the job on a cluster. Data partitioning enables the data-parallel execution of the subtasks. During the transformation the UDFs are split up into multiple parallel subtasks. Each subtask executes the same UDF. However, each processes different parts of the input data.

This setup makes clear where overheads might emerge. Firstly, the task scheduling as well as the deployment of the tasks on the respective machines introduce overhead. Moreover, the nodes have to communicate with each other in order to distribute the workload. The communication between different nodes demands for serialization and transport buffering, which adds to the overhead.

On top of the possibility to define a job by using a DAG, a layer of second-order functions is implemented in order to simplify the development for the user [15]. Apache Flink provides APIs for implementing batch as well as stream processing. The exposed functions for dataset and data stream transformations are similar to functions known from functional programming (e. g. map, reduce, and filter). Additionally, the DataSet API provides transformations known from relational databases like joins and grouping. The DataStream API provides additional operators which are useful in the streaming context. These operators include the definition of windows and window-based aggregations.

Apache Flink further provides support for iterations in the dataflow. A special case are the Delta Iterations [12]. Delta Iterations exploit the fact that for some computations not every data item is updated at every iteration step. They work on a working set and a solution set. The working set is what drives the iterations. At every step a new working set is computed and fed back into the iteration. A shrinkage of the working set improves the efficiency by leaving converged subinstances of the problem untouched. The solution set can be modified at each iteration step. The output of the Delta Iteration is the solution set after the last iteration. The Delta Iteration terminates either when the working set is empty or a maximum number of iterations is reached.

An important feature of data processing frameworks is that such jobs can be easily distributed without further changes to the program code. The jobs can be executed locally on a single computer or run on a cluster with hundreds or thousands of workers.

B. The COST Metric

One fundamental aspect in the development of scalable systems is to assess their performance. In the context of parallel systems the performance metric of parallel speedup has emerged [16]. Several such speedup metrics were defined [17], including:

- The *Relative Speedup* which is defined as

$$S = \frac{\begin{array}{c}\text{time to solve problem P using}\\ \text{algorithm A and 1 processor}\end{array}}{\begin{array}{c}\text{time to solve problem P using}\\ \text{algorithm A and N processors}\end{array}}$$

  The baseline is defined as using the same algorithm which, however, executes using only one processor.
- The *Real Speedup* which is defined as

$$S = \frac{\begin{array}{c}\text{time to solve problem P using}\\ \text{best serial algorithm A and 1 processor}\end{array}}{\begin{array}{c}\text{time to solve problem P using}\\ \text{algorithm A and N processors}\end{array}}$$

  Here, the parallel algorithm is compared to the best serial algorithm. Since the best serial algorithms might not be known, a state-of-the-art algorithm can be used in practice.

Similarly, these concepts can be used for distributed data processing systems.

The COST metric suggests a new baseline for evaluating the performance of scalable systems against an efficient single-threaded implementation. The baseline is similar to the baselines defined for real speedup. However, the COST metric does not explicitly demand the benchmark to be executed on the same computer. In fact the original single-threaded COST benchmarks were executed on a commodity laptop. The COST of a data processing system is defined as the required resources (e. g. amount of workers, number of CPU cores, CPU speed, RAM size) such that the system outperforms the baseline and, therefore, shows real speedup.

The COST metric is motivated by the fact that relative speedup and strong scalability do not imply good overall performance, that is, no real speedup. Imagine a benchmark of a data processing system which does show strong scalability for a given problem as the number of resources increases. This sign of scalability might be misleading. The data processing engine might have significant overhead which is distributed with the increasing amount of workers. Thus, the benchmark would only document the scalability of the computational overhead and not the actual performance gains. The COST metric deals with this problem by providing a baseline which does not reward “substantial but parallelizable overheads” [10, 1].

III. Methodology

In order to evaluate the performance benefits of the distributed data processing engine Apache Flink, the COST metric was used. For this, distributed benchmarks are executed on a cluster and the respective performance is compared to the single-threaded counterpart which is run on a commodity laptop. Due to the disk constraints only those datasets were selected that could fit into the laptop's memory.

A. Benchmark Selection

The benchmarks were chosen such that they cover different application domains. Three popular use cases were selected.

The TPC-H Query 10 benchmark is an example of a relational query that analyzes historical data. Executing such queries was among the key requirements for parallel databases.

Traversing huge graphs is another prominent use case for distributed systems. Distributed graph algorithms are known for their extensive intra-node communication [18]. Therefore, Connected Components was selected to gain insights into the performance implications when run on a cluster.

Lastly, machine learning methods demand for distributed systems as they often work with immense training datasets. Consequently, the optimization method Gradient Descent was included.

B. Benchmarking Setup

The distributed implementations were run on a cluster whereas a consumer laptop was used for the single-threaded benchmark. Every run was measured three times and the median was reported.

The cluster consists of 50 machines. Each computer in the cluster is equipped with an Intel Xeon X3450 @2.67GHz CPUs (4 physical cores, 8 hardware contexts) and 16 GB of RAM. Each node runs Linux (kernel version 3.10.0), Java 1.8.0, and Apache Flink 0.10.1. We configured Flink to use HDFS 2.7.1, to allocate 10 GB of the main memory, and to provide eight task execution slots per worker.

The local computer is a laptop MacBookPro12,1 with an Intel Core i5 5257U @2.7GHz CPU (2 physical cores, 4 hardware contexts) and 8 GB of RAM.

IV. Benchmarks

This sections presents the implementations of the benchmarks. The local and the cluster implementations are available online[5]. The Apache Flink jobs are based on the example implementations provided with the framework. When selecting the single-threaded implementation, care was taken that the selected implementations were not unnecessary inefficient. However, guaranteeing the most efficient implementation was not the aim.

[5]Available on GitHub at https://github.com/verbit/cost-flink-local and https://github.com/verbit/cost-flink-cluster [Accessed May 23, 2016]

Listing 1
TPC-H Query 10

```
SELECT c.custkey, c.name,
    SUM(l.extendedprice * (1 - l.discount)) AS
        revenue,
    c.acctbal, n.name, c.address, c.phone, c.
       comment
FROM customer c, orders o, lineitem l, nation
    n
WHERE c.custkey = o.custkey
    AND l.orderkey = o.orderkey
    AND o.orderdate >= DATE '[DATE]'
    AND o.orderdate < DATE '[DATE]' + INTERVAL
        '3' MONTH
    AND l.returnflag = 'R'
    AND c.nationkey = n.nationkey
GROUP BY c.custkey, c.name, c.acctbal, c.phone
    , n.name,
    c.address, c.comment
ORDER BY revenue DESC LIMIT 20;
```

A. TPC-H Query 10

The Transaction Processing Performance Council (TPC) concentrates on the development of benchmarks for databases and transaction processing systems[6]. One such benchmark suite is TPC-H[7]. The suite consists of a data population and a query component which are claimed to have industry-wide relevance.

The aim of the data scheme is to portray a real-world business scenario. In supplement to the data scheme, the TPC-H suite provides the DBGen tool[8] for data generation. The program generates comma separated value (CSV) datasets. The amount of the data generated can be adjusted by adjusting the Scale Factor (SF). A Scale Factor of N corresponds to roughly N GB of data.

TPC-H defines further business-oriented queries. Listing 1 shows the definition of query number 10. The query finds customers which might have had problems with ordered parts and thus returned them. The customers are sorted in terms of their contribution to the lost revenue and the top 20 are returned.

The implementations of the TPC-H Query 10 were slightly adjusted in order to match the implementation provided with Apache Flink. They, therefore, differ from the original definition. Firstly, the implementations do not output the `c.phone` and `c.comment` columns. What is more, the orders are filtered only by years greater than 1990 as opposed to a period of three months. In addition to that, the result is not sorted by revenue and is further not limited to 20 entries.

Apache Flink: The implementation in the Apache Flink framework is straightforward. It makes use of Flink's DataSet API which enables an easy translation of the original query

[6]TPC organization, http://www.tpc.org/ [Accessed May 23, 2016]

[7]TPC-H Specification 2.16.0 http://www.tpc.org/tpch/spec/tpch2.16.0v1.pdf [Accessed May 23, 2016]

[8]The tool is available for download on https://github.com/electrum/tpch-dbgen [Accessed May 23, 2016]

into a Flink job. An advantage of the Flink implementation is that it does not need to transform the data into an internal representation as opposed to database systems. Instead it operates directly on the generated CSV files.

Single-Threaded: The SQLite database[9] was chosen for the single-threaded implementation. SQLite is a software library which implements a SQL database engine in a single-threaded manner. The local TPC-H Query 10 implementation uses the `sqlite3` command line tool for creating a database from the generated CSV data and executing the actual query against the created database. Both, creating and querying the database, are implemented using ordinary SQL scripts with some SQLite extensions for importing and exporting CSV files. The scripts are executed by the command line tool.

B. Connected Components

A connected component of a given undirected graph is a subgraph in which the vertices meet the following two conditions:

1) Two arbitrary vertices are connected by a path.
2) The vertices are only connected to vertices in the subgraph.

There exist multiple techniques for finding the connected components of a graph. In the following we present two algorithms. While the Flink job is implemented using label propagation due to its scalability, the Objective-C implementation makes use of the disjoint-set data structure enabling fast execution.

Apache Flink: The label propagation algorithm begins by assigning every vertex an unique identifier. Afterwards, at each iteration step every vertex propagates its current identifier to all neighbors. If a vertex receives an identifier smaller than it currently has, the identifier is updated. This process iterates until there are no changes to the identifiers of the vertices.

The Apache Flink job uses the DataSet API for implementing the operations needed by the label propagation algorithm. The propagation of a vertex' current identifier is implemented using SQL-like operations (e. g. joins and aggregation). For the iteration process, Flink's Delta Iterations are used.

Single-Threaded: The disjoint-set data structure interface can be split up in two primal methods, namely *Union* and *Find*. To begin with, every vertex is assigned his own identifier. The Find method takes a vertex as input and outputs its root node. The Union method takes two vertices as input and assigns an identical identifier to both. Thus, effectively making the two vertices belong to the same connected component. Listing 2 shows the implementation of the naive Union-Find interface.

A problem of the naive implementation is that it might degenerate into a linked list due to the behavior of the Union method. In order to improve the computational complexity of the naive implementation the following two modifications are employed:

- *Union-by-Rank*: Adjust the Union method by attaching the smaller tree under the bigger one.

[9]SQLite project, https://www.sqlite.org/ [Accessed May 23, 2016]

Listing 2
NAIVE UNION-FIND

```
uint32_t uf_find(union_find *uf, uint32_t n) {
  uint32_t r = uf->root[n];
  while (r != uf->root[r]) r = uf->root[r];
  return r;
}
void uf_union(union_find *uf, uint32_t n1,
    uint32_t n2) {
  uint32_t n1_root = uf_find(uf, n1);
  uint32_t n2_root = uf_find(uf, n2);
  uf->root[n1_root] = n2_root;
}
```

- *Path Compression:* Extend the Find method such that after every invocation for a given vertex the found root is directly assigned to all transitive parents of that vertex.

Listing 3 displays the optimized Union-Find interface.

Listing 3
OPTIMIZED UNION-FIND

```
uint32_t uf_find(union_find *uf, uint32_t n) {
  uint32_t r = uf->root[n]; // find root
  while (r != uf->root[r]) r = uf->root[r];
  if (n != r) {
    uint32_t j = n;
    uint32_t j_root = uf->root[n];
    while (j_root != r) { // compress path
      uf->root[j] = r;
      j = j_root;
      j_root = uf->root[j_root];
    }
  }
  return r;
}
void uf_union(union_find *uf, uint32_t n1,
    uint32_t n2) {
  uint32_t n1_root = uf_find(uf, n1);
  uint32_t n2_root = uf_find(uf, n2);
  if (n1_root == n2_root) return;
  // union by rank
  if (uf->rank[n1] < uf->rank[n2]) {
    uf->root[n1_root] = n2_root;
  } else if (uf->rank[n1] > uf->rank[n2]) {
    uf->root[n2_root] = n1_root;
  } else {
    uf->root[n2_root] = n1_root;
    ++(uf->rank[n1_root]);
  }
}
```

The implementation computes the connected components by traversing an edge list and applying the Union method for each edge. After all edges are processed the Find method returns for every vertex the identifier of the connected component. The program supports ASCII encoded edge list files as well as binary representation. In case of a textual edge list representation the program has to deal with file sizes of tens of gigabytes. Therefore, special care was taken in implementing the input

data processing in order to ensure efficient file traversal and parsing.

C. Gradient Descent

Gradient descent is a technique to solve optimization problems. Given an objective function, gradient descent searches for the minimum of that function. The algorithm iteratively computes the gradient at the current point and moves in the negative direction of that gradient.

For large-scale learning problem calculating the gradient for the whole training dataset might be unfeasible. The stochastic gradient descent simplifies the calculation by only looking at one example at a time. The weight vector is updated as

$$w_{t+1} \coloneqq w_t - \gamma \nabla Q_i(w_t)$$

where i is chosen randomly in every iteration step. It can be shown that stochastic gradient performs well in large-scale applications [19].

The efficiency of the stochastic gradient descent can be improved by calculating the gradient of a small subset of random data points, so called mini-batches, rather than for a single point. That is, the weight vector is updated by

$$w_{t+1} \coloneqq w_t - \gamma \frac{1}{|B|} \sum_{b \in B} \nabla Q_b(w_t)$$

where B is a set of randomly chosen indices at every iteration step. The mini-batch stochastic gradient descent enables more efficient calculation compared to single point updates since it enables the usage of efficient vectorized implementations.

Apache Flink: The Apache Flink implementation uses the FlinkML library[10] which contains several predefined routines for machine learning applications. The stochastic gradient descent method is part of the optimization package. Listing 4 shows how the library enables a concise definition of the Apache Flink job.

Listing 4
GRADIENT DESCENT JOB

```
val initialWeights = ...
val trainingDataSet = ...
// create optimizer
val sgd = SimpleGradientDescent()
  .setIterations(50)
  .setStepsize(0.1)
  .setLossFunction(
      GenericLossFunction(SquaredLoss,
          LinearPrediction))
// calculate weights
val weights = sgd.optimize(trainingDataSet,
    initialWeights)
weights.print()
```

At the time of this writing, however, the FlinkML implementations do not include true stochastic gradient descent. Instead whole partitions of the dataset are used to compute the gradient. Thus, effectively leading to regular batch gradient descent. Later releases of the library should provide the correct implementations[11].

Single-Threaded: The single-threaded implementation is implemented in Objective-C and makes use of efficient implementations of the BLAS interface[12] [20]. Similar to the implementation of connected components, the implementation has to deal with file sizes of tens of gigabytes. The program has therefore to implement efficient data reading. The implementation accepts ASCII encoded as well as binary dataset implementations. Randomly shuffled data as recommended in [19] is an important requirement of the program. By assuming random order in the input data, the implementation can traverse the data sequentially leading to performance boosts. In case of ASCII encoded datasets the files are transformed into a binary representation before the computation starts. This does not only reduce the file size but moreover eliminates decoding overhead.

V. RESULTS

This section presents the results from running the implementations on the cluster and on the local machine, respectively.

A. TPC-H Query 10

The DBGen tool was used to generate the input data for both implementations. The generator gives the possibility to vary the generated data size by adjusting the Scale Factor (SF). A Scale Factor of N corresponds to a data size of roughly N GB. This benchmark includes two comparisons. The single-threaded versions used the SQLite database. Before the actual query is run the generated data has first to be read in into the database. The load time as well as the query execution time were measured. The measurements were repeated with two different Scale Factors. Table I illustrates the results.

TABLE I
LOCAL TPC-H QUERY 10 PERFORMANCE

Implementation	Load time	Query time
SQLite (SF 1)	74 s	7 s
SQLite (SF 10)	790 s	202 s

The benchmark shows that for smaller input sizes the query time is reasonable small at seven seconds. However, increasing the input data by a factor of ten significantly increases the query time at 202 seconds. In addition, the time it takes SQLite to read in the data is significantly high compared to the time it takes to execute the query. In both cases the load time is higher than the query time.

The second benchmark shows how the single-threaded implementation compares to the cluster scale-out. Figure 1 summarizes the outcome.

[10] Apache Flink's machine learning library FlinkML, https://ci.apache.org/projects/flink/flink-docs-master/libs/ml/ [Accessed May 23, 2016]

[11] Flink ML optimization package, https://ci.apache.org/projects/flink/flink-docs-release-0.10/libs/ml/optimization.html [Accessed May 23, 2016]

[12] BLAS routines, http://www.netlib.org/blas/ [Accessed May 23, 2016]

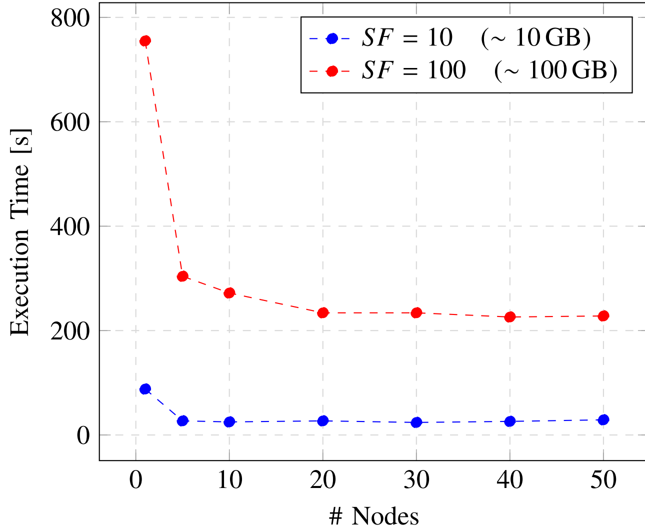

Fig. 1. Execution time of the TPC-H Query 10 on the cluster implemented in Apache Flink.

Comparing the runtime of the 10 GB dataset on one computer to the runtime of the single-threaded implementation yields interesting results. The one node scale-out with a runtime of 88 seconds already outperforms the single-threaded version. This holds true even if the load time of SQLite is disregarded.

However, note that the TPC-H Query 10 benchmark exhibits bad strong scaling behavior. For the 10 GB dataset the job finishes in 27 seconds when using 5 nodes. Further increasing the amount of workers does not improve the runtime. When performing the query using the 100 GB dataset a similar picture emerges. With 20 nodes the benchmarks needs 234 seconds. Larger scale-out do not improve the runtime. However, with a runtime of 304 seconds using 5 nodes the cost of bigger scale-outs might not justify the runtime improvements.

B. Connected Components

For the calculation of the connected components the `uk-2007-05` graph was used [21, 22]. Table II summarizes the basic quantities of that graph.

TABLE II
BASIC QUANTITIES OF THE `UK-2007-05` GRAPH

	uk-2007-05
Vertices	105 896 555
Edges	3 738 733 648
File Size (ASCII)	66.9 GB
File Size (Binary)	15.7 GB

Both implementations were run using the ASCII encoded version of the graph's edge list. In addition to that, the local implementation was also benchmarked using the binary format. Table III summarizes the outcomes on the local machine. Figure 2 visualizes the execution time on the cluster for different amount of worker nodes.

TABLE III
LOCAL CONNECTED COMPONENTS PERFORMANCE

Implementation	Execution Time
Local (Binary Input)	52 s
Local (ASCII Input)	148 s

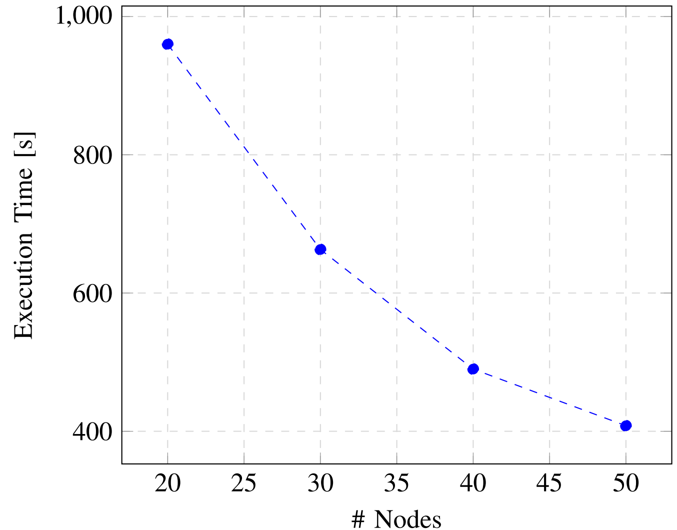

Fig. 2. Execution time of the Connected Components benchmark on the cluster implemented in Apache Flink.

The cluster consisting of 50 nodes was not able to outperform the local Connected Components implementation. The fastest runtime was measured with all the 50 nodes at 408 seconds. The local implementation that processed the ASCII encoded input with a runtime of 148 seconds was faster by more than a factor of two.

C. Gradient Descent

The input data was generated by using a polynomial function and adding Gaussian error. In order to simulate multi-dimensional workload the Vandermonde matrix for a 20 dimensional polynomial was generated. Effectively, this makes the Gradient Descent fit a polynomial function of order 20 to the generated data. The Gradient Descent benchmark was run with 50 iterations over the whole dataset with no convergence criterion.

The benchmark was measured with a dataset of 20 000 000 data points. This translates to roughly 10 GB of data. Table IV summarizes the performance of the local implementation. In addition to that, the distributed implementation ran on different cluster scale-outs. Figure 3 displays the results.

Even if the transformation time is disregarded, the single-threaded is outperformed beginning with a cluster size of five nodes and a runtime of 152 seconds. Increasing the amount

TABLE IV
LOCAL GRADIENT DESCENT PERFORMANCE

Dataset Size	Conversion	Execution
$m = 20\,000\,000$ (~ 10 GB)	209 s	180 s

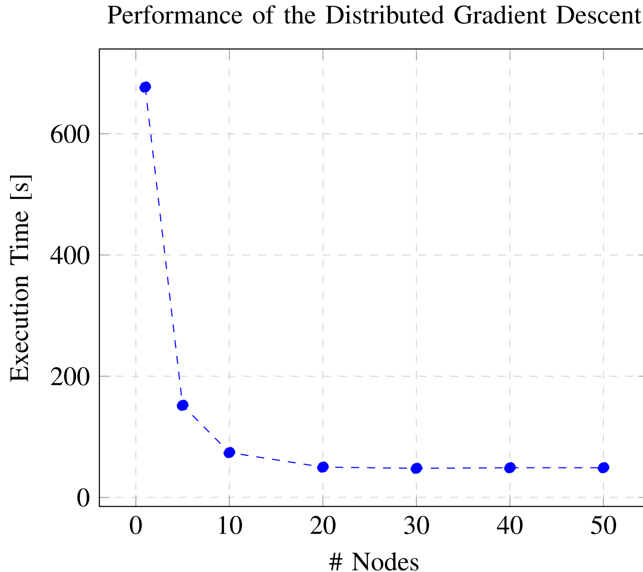

Fig. 3. Execution time of the Gradient Descent benchmark on the cluster implemented in Apache Flink.

of workers shows even higher outperformance. However, beginning with 20 nodes and a runtime of 50 seconds no further improvements could be measured by increasing the amount of workers.

VI. DISCUSSION

As the reported measurements suggest, whether using Apache Flink on a cluster is beneficial over a single-threaded implementation is very dependent on the problem at hand. The single-threaded TPC-H Query 10 is outperformed even by a one node scale-out. Further increase in nodes shows how strongly the data processing framework outperforms the single-threaded database. However, it should be noted that the systems have different use case. The SQLite is a database used for storing and querying data stored on a single computer. Apache Flink, on the other hand, provides the facilities for massively parallel data processing. Both systems have distinct feature sets and target different user groups. This might contribute to the large discrepancy in running times. The choice of SQLite might not be the right choice to truly evaluate a baseline for Apache Flink. But still, it provides insights into how a simple single-threaded database performs compared to a dataflow system.

The Connected Components benchmark shows a completely different picture. Even the cluster setup with 50 nodes could not outperform the local implementation, independent of the input data format. The benchmark displays that some problems might not map well into the distributed environment. Distributed graph algorithms are known for extensive intra-node communication adding significant overhead to the scalable implementation.

For the Gradient Descent benchmark the performance differences were notably smaller when compared to the previous two problems. The single-threaded implementation spends most of the time reading the input file. Thus, the execution time is primarily limited by the disk drive speed. Furthermore, the program spends more than half of its time transforming the dataset into a binary representation. The Apache Flink implementation, on the other side, benefits from distributed file reading through HDFS and distributed parsing.

VII. CONCLUSION AND FUTURE WORK

The paper presented a performance comparison between the distributed data processing framework Apache Flink and a single computer to help answer the question for which problems such a distributed system is a sensible choice. The used COST metric aims to provide a better baseline for the performance evaluation of distributed data processing systems. By using single-threaded baselines, systems that introduce significant but distributable overhead are not rewarded. Furthermore, the user can gain insights into the real performance benefits of the distributed system. In order to gain representative COSTs, benchmarks with different application domains were selected. The three benchmarks consist of TPC-H Query 10, Connected Components, and Gradient Descent.

The evaluation on a cluster of 50 nodes and a commodity laptop showed a notable variety for the gained COSTs. While the local TPC-H Query 10 implementation was outperformed even by a Apache Flink installation running with one worker node, the local Connected Components implementation was not outperformed by the cluster.

The results can be used by a user to make more educated choices of systems for his problems at hand. Primarily, this allows the user to select the faster system or the best system given financial or temporal constraints. In addition, wrong system decision are minimized and costs can be saved in cases where the distributed system is not appropriate.

Several extensions to the work presented in this paper would be interesting. Firstly, using SQLite as the single-threaded implementation of Connected Components might not have been the perfect choice. A single-threaded dataflow implementation might yield more comparable results. In addition to that, the range of benchmarked problems and algorithms could be further extended in order to provide an even broader portfolio of comparisons. This could help users in mapping problems at hand more easily to the benchmarked problems. Furthermore, comparing other factors than the runtime can provide useful insights for the user as well. Tradeoffs like the ease of implementation and the monetary cost between the system could as well be considered. Lastly, adding comparisons to multi-threaded implementations that take full advantage of the computing capabilities of a single node would give users a better impression of the system alternatives.

VIII. Acknowledgments

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 (BMBF) as Berlin Big Data Center BBDC (funding mark 01IS14013A).

References

[1] J. Gantz and D. Reinsel, "The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East." IDC iView, Tech. Rep., 2012.

[2] M. Cox and D. Ellsworth, "Application-controlled Demand Paging for Out-of-core Visualization," in *Proceedings of the 8th Conference on Visualization '97*, ser. VIS '97. IEEE Computer Society Press, 1997, pp. 235–ff.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. ACM, 2003, pp. 29–43.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. USENIX Association, 2004, pp. 137–150.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. IEEE Computer Society, 2010, pp. 1–10.

[6] M. Olson, "HADOOP: Scalable, Flexible Data Storage and Analysis," *IQT Quarterly*, vol. 1, no. 3, pp. 14–18, 2010.

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. USENIX Association, 2010, pp. 10–10.

[8] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@Twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. ACM, 2014, pp. 147–156.

[9] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The Stratosphere Platform for Big Data Analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014.

[10] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at what COST?" in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, 2015.

[11] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl, "Implicit Parallelism Through Deep Language Embedding," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. ACM, 2015, pp. 47–61.

[12] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning Fast Iterative Data Flows," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1268–1279, 2012.

[13] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," *Proceedings of the VLDB Endowment*, vol. 8, pp. 1792–1803, 2015.

[14] D. Warneke and O. Kao, "Nephele: Efficient Parallel Data Processing in the Cloud," in *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*, ser. MTAGS '09. ACM, 2009, pp. 8:1–8:10.

[15] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. ACM, 2010, pp. 119–130.

[16] X.-H. Sun and J. L. Gustafson, "Toward a Better Parallel Performance Metric," *Parallel Computing*, vol. 17, no. 10-11, pp. 1093–1109, 1991.

[17] S. Sahni and V. Thanvantri, "Performance Metrics: Keeping the Focus on Runtime," *IEEE Parallel Distrib. Technol.*, vol. 4, no. 1, pp. 43–56, 1996.

[18] I. Stanton and G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12. ACM, 2012, pp. 1222–1230.

[19] L. Bottou, "Stochastic Gradient Descent Tricks," in *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 421–436.

[20] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, 1979.

[21] P. Boldi and S. Vigna, "The WebGraph Framework I: Compression Techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. ACM, 2004, pp. 595–602.

[22] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. ACM, 2011, pp. 587–596.