Deconstructing the Performance of Big Data Analytics Frameworks

ABSTRACT

Over the last decade, several frameworks have been designed to aid in conducting analytics over large data sets. Unfortunately, the underlying complexity of these frameworks has made it incredibly hard to come to informed conclusions as to how they perform given an input workload (or a query).

In this paper, we compare the performance of four popular "SQL-on-Hadoop" frameworks – Hive, Pig, Presto, and Spark SQL. We conduct a large set of cluster experiments using a TPC-H like benchmark and study different aspects which can impact the overall performance, with a special focus on the query execution plan generated by these frameworks; such plans can often be views as DAGs.

Our results show that in-memory frameworks (Presto and Spark) perform significantly better than traditional systems at the expense of lack of fault-tolerance and/or significantly higher network traffic. We also find that standard metrics such as depth, width or number of stages do not explain the jobs' performance in these frameworks.

1. INTRODUCTION

In the last decade, the amount of data to be analyzed increased dramatically.

and new programming paradigms such as MapReduce were rdesigned to distribute data processing across multiple machines using big data execution frameworks.

paragraph 1 - context; large amount of data; MapReduce as a platform to execute the queries; data analytics jobs have evolved and MR converts them to DAG's because queries become complex

paragraph 2 - however the execution frameworks became very different. single job vs. multiple jobs(Hive, Pig on Hadoop -¿ sequence of MapReduce jobs), Hive, Pig on Tez/Hadoop, Spark, Dryad, Presto, single job with a DAG, cosmos). The main reason is because they target various improvements and push various query optimizations and system opimizations.

paragraph 3 - in general every framework follows a certain pattern in creating the DAG; query plan, optimizations and then the query execution plan which is executed.

paragraph 4 - however, it is hard to tell how all these frameworks perform and what aspects are important for performance. In this paper, we do a comparative evaluation of the various frameworks. we specifically focus on their e2e performance for DAGs using TPC-DS and TPC-H benchmarks.

```
SELECT
    c_custkey, c_name, c_acctbal
    n_name, c_address, c_phone, c_comment
    sum(l_extendedprice * (1 - l_discount)) as rev,
FROM
    customer, orders, lineitem, nation
WHERE
    c_custkey = o_custkey,
    and l_orderkey = o_orderkey,
    and o_orderdate >= '1993-07-01'
    and o_orderdate < '1993-10-01'
    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
    rev desc
limit 20;
```

Figure 1: TPC-H query 10. It scans the *lineitem* table, applies an inequality predicate, projects a few columns, performs an aggregation and sorts the final result.

2. MOTIVATION AND BACKGROUND

In recent years, a variety of frameworks have been developed for running rich analytics on massive volumes of data. The underlying data can be derived from diverse sources such as network and server logs, web and social network crawls, sensor data, etc., and the analytics can range from text analytics over unstructured or semi-structured data (e.g., logs or configuration files) or SQL-like processing over structured or semi-structured data. The underlying data is often stored in and managed used Hadoop [**?**].

Our work focuses on native Hadoop frameworks for SQL processing. Such systems take as input SQL queries such as the one shown in Figure 1, which are then converted into a sequence of compute tasks that run across a large cluster of machines. Over many iterations, these tasks read (intermediate/input) data from the underlying file system, and process it to generate intermediate results which are then exchanged across

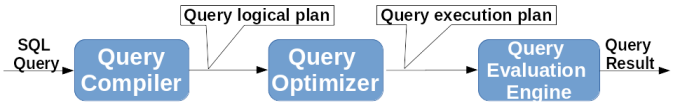


Figure 2: Standard steps in processing a query

tasks, eventually producing a response to the query.

In what follows, we provide more details regarding the end-to-end query processing pipeline common to these frameworks before going into the specific of individual frameworks.

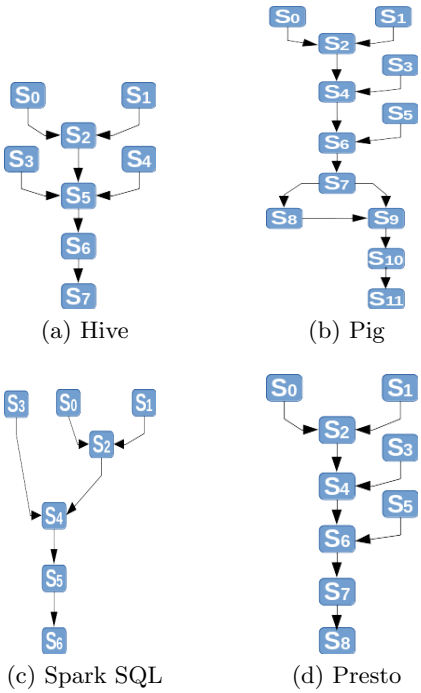
2.1 Query Processing

SQL processing in big data analytics frameworks follows a set of steps which are described in Figure 2. When a query is submitted for execution, a query compiler parses the user specified SQL and generates an internal tokenized representation of the query in the form of a physical operator tree. In this stage, checks are also made to determine if columns and tables identified in the query exist in the database and if the query has been formed correctly with the appropriate keywords and structure.

The goal of the optimizer is to determine the most efficient way to execute a query by estimating the costs of alternative query plans. It rearranges the operator tree by analyzing each part of the query and picks the best query plan. One of the key sets of factors that determine the performance of a query plan is the order in which the tables are joined and the type of join algorithm used because these choices control the amount of data transferred across machines and hence shuffle cost.

For example, if a query has three tables to join A, B, C of size 10 rows, 10,000 rows, and 1,000,000 rows, respectively, a query plan that joins B and C first can take several orders-of-magnitude more time to execute than one that joins A and C first. The improvement is mainly due to the fact that the later join is between two smaller tables.

Different join algorithms can be used based on the query semantics and the size of the tables. In a*distributed hash join algorithm*, multiple map tasks read the input tables, emitting (key, value) pairs, where the key is the join column. These pairs are shuffled and distributed over the network to reduce tasks which perform the actual join operation. The benefit of such a join is that it works regardless of the data size at the expense of greater resource use. A more optimized form of join is the *broadcast join* whose goal is to eliminate the shuffle and reduce phases of the repartitioned join. It is fast and works when one table is small enough to fit in the memory and a map only job can perform the join processing by doing a single scan through the largest table. Several extensions of these joining algorithms exists.



(a) Hive (b) Pig

(c) Spark SQL (d) Presto

Figure 3: DAGs generated for TPC-H Q10

Finally, the chosen query plan is converted to a query execution plan which is evaluated and produce the final result. A query execution plan is represented as a Direct Acyclic Graph(DAG), where nodes are stages in the query which perform processing on the input data and generate intermediate data. A stage consists of a set of tasks which represent the atomic entity for processing the data. The edges represent the logical connections between stages and describe the flow and the type of data communication in the DAG. The most common types of communications used in practice are scatter gather, one to one and broadcast.

Figure 3 shows the query execution DAGs generated by different frameworks for query 10 from the TPC-H [**?**] workload described in Figure 1.

We can see that the DAGs are pretty diverse, which is a direct consequence of the internal optimizations each framework is adopting. In what follows, we describe these internal optimizations along with other key details that set the different frameworks apart in terms of the performance they offer for a given input query.

2.2 Frameworks in Detail

In addition to support query processing, most frameworks require support for fault tolerance and high availability. This is generally support by the underlying execution engine. The frameworks' organization can be viewed logically as shown in Figure 4.

2.2.1 Apache Hive

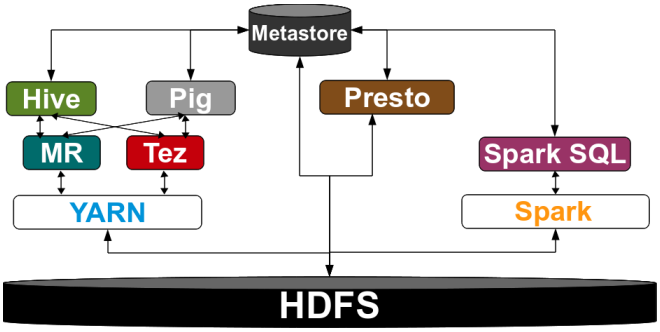


Figure 4: Setup

Hive [?] is one of the most popular system for querying and analyzing large datasets stored in shared-nothing frameworks such as Hadoop. It leverages Hadoop's scalability, fault tolerance and highly availability.

Hive stores data inside the Hadoop Distributed File System (HDFS) in a structured format, and it allows users to query it using a language very similar to SQL, called HiveQL. Typically, the query execution plan generated from the HiveQL statements uses MapReduce [?] as a processing engine by converting the DAG into a sequence of MapReduce jobs, which significantly degrades the performance due to unnecessary writes to HDFS for every intermediate job output in the pipeline.

The Tez [?] framework was introduced as a replacement for MapReduce as a primitive for Hive [?], Pig [?], Cascading [?], etc. A HiveQL query is executed as a single Tez job which *pipelines* the data through the execution stages and hence it provides smaller latency for interactive queries and higher throughput for batch queries.

The main aspects every query framework is concerned are the set of optimizations and the types of storage formats supported. Hive mainly supports two types of join algorithms: by default a distributed hash join is used, but a broadcast join can be used as an optimization. However, in the absence of a query optimizer component, it is hard for the user to infer the right types of joins to use because they require deep understanding of the query semantics and the properties of the data from the tables to join. Similarly, the user has to manually rewrite the query in order to pick the right join order by inspecting the sizes of the joining tables. This can be troublesome in a production cluster where new queries can arrive anytime and it is almost impossible to do manual query inspection and optimization-tuning. However, with Hive 0.14, a new cost based optimized (CBO) component was introduced. CBO [?] collects various statistics in terms of both resource usage and semantics of the data and it automatically optimize the query plan to decide the join order and what algorithm to use for a given join.

aditya: not sure we need to talk about storage. this whole para can go? Columnar storage formats significantly improves the analytical queries because the data is stored in columns instead of rows which reduces disk I/O and enable better compression and encoding schemes. Hive implements an Optimized Row Columnar(ORC) format which is an optimized version of the Row Columnar(RC) file format. Some of the most important characteristics are a light-weight built-in index that allows skipping row groups that don't pass a filtering predicate, generating a single file as the output of each task or by providing data encoding and block-level compression. A different columnar storage format, but widely popular is Parquet which is mainly designed to support complex nested data structures, to exploit efficient encodings and to allow compression schemes to be specified on a per-column level.

2.2.2 *Apache Pig*

Unlike Hive which relies on an execution planner to automatically translate HiveQL statements to an execution plan, Pig provides a data flow language called Pig Latin for creating and executing data flows atop Hadoop. Pig is aimed at experienced programmers; essentially it relies on the user handcrafting the query execution plan, as we explain below. Pig does give users more control over the flow of the data than Hive does, leading to better optimizations in some cases.

Like Hive, the Pig query execution plan can be executed through MapReduce framework or through the Tez framework in the latest Pig version.

Pig's optimizer is rule based and a set of optimizations rules are implemented and enabled by default. In particular, Pig supports predicate pushdown and partition elimination; these optimizations are supported by Hive as well. The lack of a cost based optimizer means the join ordering selection has to be handled manually in the handcrafted query. Pig implements a suite of specialized join algorithms similar to Hive; by default, distributed hash join algorithm is enabled.

aditya: cut this In terms of storage formats, Pig supports Parquet columnar format while ORCFile is under development. However, evidence shows that Pig is mainly used atop row-oriented data.

2.2.3 *Apache Spark SQL*

Spark [?] uses Resilient Distributed Datasets (RDDs), a novel distributed memory abstraction that lets programmers perform in-memory computation. In addition, it enables efficient data reuse and better fault tolerance in a broad range of applications.

Spark SQL [?] brings native SQL support to Spark. It streamlines the process of querying data stored both in RDDs and in external sources (for example from Hive catalogs**aditya: what are catalogs**). It derives from

Spark key properties including scalability, fault tolerance, high availability and even some simple query optimizations such as pipelining operations.

One of the most important component of Spark SQL is the underlying optimizer called Catalyst which supports both rule-based and cost-based optimizations. Catalyst uses features of the Scala programming language, such as pattern-matching, to express composable rules and transforms SQL queries into RDD actions. In a nutshell, Spark SQL takes an unoptimized logical plan from the HiveQL parser, applies various optimizations and creates an optimized logical plan which is then converted to multiple physical plans, using physical operators that match the Spark execution engine. It then selects a plan, based on a cost model. At the moment, the cost-based model is used to select the join algorithms; for relations that are known to be small it uses a broadcast join algorithm using a peer to peer facility available in Spark. Otherwise the distributed hash join algorithm is used as default. In addition, physical planning also performs rule-based physical optimizations, such as predicate pushdown.

aditya: cut this In terms of storage formats, Spark SQL supports various input formats but it is heavily promoting Parquet.

2.2.4 Presto

Presto is an in-memory distributed SQL query engine optimized to satisfy low-latency queries. Instead of relying on another execution framework (like Hive or Pig does), Presto provides its own long running daemons on each node of the cluster. In Presto, a coordinator daemon not only parses, analyzes and generates the query plan for a SQL query submitted from the client, but it also takes care of the scheduling process and monitors the job's progress.

Similarly with Tez or Spark SQL, it adopts a pipelined execution model where multiple stages run at once and stream data from one stage to the next as it becomes available. In addition, Presto does all processing in memory and dynamically compiles certain portions of the query plan down to the byte code which let JVM optimize and generate native machine code. Another distinction from other execution engines such as Hadoop or Spark is that Presto has multi-threaded tasks, typically defined as one task per stage per machine. These threads can handle multiple splits of data in parallel in the same task.

Presto does not support a cost-based optimization component. Joins are not automatically reordered based on the table size and the user manually needs to make sure that smaller tables are on the right hand size of the join and they must fit in the memory. Presto implements the distributed hash join as its default join algorithm.

aditya: cut this? Although we may need it for the methodology section The presence of different connectors, enable Presto to access data from different data sources(for example Hive, Cassandra, MySQL, etc.) and hence supports a variety of storage formats including columnar formats such as ORCFile, RCFile or Parquet.

Despite its major benefits due to in-memory capabilities, Presto has several major limitations. First, a query can be executed only if the amount of data processed can fit in the memory. Second, Presto lacks built-in fault tolerance. For example if a task or a machine where a query is executing stalls, the whole query fails and has to be re-executed. Also, User Defined Functions(UDFs) are much harder to deploy/develop/build than in Pig or Hive. **aditya: why is it harder?** Finally, it has limited support for predicate pushdown, which is well supported in other frameworks. **aditya: check this**

In many ways, Presto is similar in design to Cloudera's Impala. They are both MPP(Massively Parallel Processing) databases, both can run on top of HDFS and conceptually bypass MapReduce. The main difference are runtimes. Impala is built with C++ and Presto is written in Java. Also Impala can only work with HDFS while Presto is more interoperable.

3. MEASUREMENTS METHODOLOGY

In this section we present the environment used to run our experiments.

3.1 Hardware Configuration

We deployed our experiments in a 10 node cluster on CloudLab [**?**]. One of the nodes is designated as a master node and it hosts all the master daemons required by different frameworks. The other 9 nodes are designated as "compute" nodes. Every node in the cluster has 2x Intel Xeon CPUs @ 2.6 GHZ, with 8 physical cores each (16 physical cores in total), 2 x 1TB SATA disks (7.2K rpm), 1 x 10 Gb Ethernet card, and 64 GB of RAM. Out of the 2 SATA disks we use one for data storage in HDFS, to store Spark RDDs, logs and other data maintained by different frameworks. Each node runs 64-bit Ubuntu Linux 14.04.

3.2 Software Configuration

Figure 4 shows the main software components deployed in our testbed. We are using Hadoop 2.6.0 to store the data in the Hadoop's Distributed File System (HDFS) and YARN as one of the resource management frameworks used. HDFS Namenode and SecondaryNameNode processes run on the master machine while the DataNodes managing the distributed data run on the other machines. We configured the block data size to 256 MB and the replication factor is set to 3.

The YARN's ResourceManager process deployed on the master machine handles resource requests from applications, tracks the available resources on the compute machines where NodeManagers are running and does the matching between the two. YARN assigns resources in terms of containers and we configured each NodeManager to run up to 8 containers in parallel (1 per core). The size of every container is set to 7 GB of memory accounting for a total of 56 GB of memory to be handled by YARN on every compute node. We reserved the remaining memory to be available for the operating system. In our experiments, YARN is using the Capacity Scheduler as the cluster wide resource scheduler and all the jobs are submitted to the same queue.

aditya: the following paragraph needs to be completely taken out. Seems repetitive, and incorrect To run Hive and Pig SQL-like queries atop Hadoop as a single job, we installed Tez 0.7.0. In YARN's terminology, Tez acts as an ApplicationMaster and enables the execution of complex jobs represented as DAGs against YARN. It has its own optimization techniques and a scheduler to handle the resource requests/assignments to tasks in various stages in the DAG. In our experiments, we disabled the data compression and enabled various Tez properties such as speculative execution, pipelining and container reuse.

aditya: where is the metastore in the picture? Query engine frameworks require a metastore entity. Metastore constitutes of the metastore service and the database. The metastore service provides the interface to the SQL-like frameworks and the database stores the data definitions, and the mapping to the data stored in HDFS. For our experiments, we configure a MySQL server to hold the database and we deployed Hive 0.14.0 on the master machine which uses the JDBC driver to access the metastore. Similarly we deployed Pig 0.15.0 which uses the same Hive metastore.

aditya: many things here are not defined. E.g., Thrift server. Also it would be good to show such things, and others like the Hive Connector in the picture We configured Presto to query the data stored in the Hive metastore using the thrift server and a Hive connector. We run one PrestoServer process on each machine and set the one running on the master node to act as a coordinator while the others as workers.

Presto's definition of tasks is different than in systems like Hadoop. For example, the coordinator will create a single task on a worker per stage in a query and the number of threads used for running splits concurrently inside the task is configurable. In our experiments we set the number of maximum threads per task to 8 in order to achieve the same parallelism across all the evaluated frameworks. Also, Presto requires that the amount of data to process to fit in memory. For this reason, based on the queries we run we configure the task.max-memory parameter to 50 GB.

aditya: again, what are thrift and beeline? what is interrogation? Lastly, we deployed Spark as a different resource management framework. It uses HDFS as a storage system and Spark SQL as a library to run SQL-like queries. We configured Spark to use thrift server to connect to the Hive metastore and beeline to interrogate the queries. The Spark master process is deployed on the master machine while a worker process is running on every compute machine. To achieve the same level of parallelism as Hadoop or Presto, we set the number of cores per worker to 8 and the amount of memory to 56 GB. **aditya: why 56? why not 50?** We also properly configured the storage directories for shuffle and RDD data.

In all cases, we diabled compression. **aditya: say why?**

aditya: cut the below We also did a set of other tuning. For example we disabled compression. In Tez we set to false tez.runtime.intermediate-output.should-compress and tez.runtime.intermediate-input.is-compressed properties. Similarly in Spark we set spark.shuffle.compress, spark.shuffle.spill.compress and spark.broadcast.compress to false. For the other systems, the compression is disabled by default. Also, we had to enable the cost based optimizer in Hive which if turned off by default. Lastly, for frameworks which does not have a query optimizer such as Pig and Presto we carefully checked that the distributed hash join algorithm(the default option) is enabled.

To monitor the resource usage, we developed and deployed an underlying framework which captures various OS counters on every compute machine, in order to extract relevant information such as CPU cycles, memory used, number of bytes read/written from/to network/disk.

3.3 Workload

Our experiments use the TPC-H workload [**?**], a well-known set of benchmarks for query engins performance. It contains 22 business oriented ad-hoc queries and concurrent data modifications. **aditya: more clearly justify the choice of the workload**

Most of our experiments show results for a subset of queries that could run successfully across all frameworks. **aditya: show table of failed queries? Is this the main reason we did not use all queries?** For example query 13 fails in Presto because non-equi joins are not supported(**robert: add more examples**). More precisely, most of our results are based on 7 distinct queries which are diverse enough to cover all the different characteristics available in TPC-H workload. Table 1 describes the main features of every query we used. **aditya: this para needs to be rewritten as it is not well justified why we pick a subset.**

	#tables	#joins	filtering	column projection	aggregation	sort	nested query	has views
Q1	1	0	Y	Y	Y	Y	N	N
Q3	3	2	Y	Y	Y	Y	N	N
Q7	5	5	Y	Y	Y	Y	Y	N
Q10	4	3	Y	Y	Y	Y	N	N
Q12	2	1	Y	N	Y	Y	N	N
Q15	2	2	Y	Y	Y	Y	N	Y
Q20	5	5	Y	N	Y	Y	N	Y

Table 1: Characteristics of the TPC-H query set used in the paper.

Can we explain this more systematically?

Although multiple versions of TPC-H scripts are available online, every version contains query scripts which have different variations in order to leverage the benefits of a framework or another. For example, (pig jira tpch citation) contains Pig TPC-H scripts, carefully written in order to maximize the benefits of Pig, while (impala tpch) provides Impala optimized TPC-H query scripts. It does not help that each framework comes with its own query set or benchmark on which it performs well. For the purpose of this paper, we want to understand the performance of the benchmarked frameworks using the same set of query scripts. We picked the standard queries available in (hortonworks tpch) which are runnable by any framework except Pig. Pig requires the query scripts to be translated into PigLatin. For this reason, we manually rewritten a handful of the queries in PigLatin such that their semantics are as close as possible by the standard SQL queries provided in (hortonworks tpch).

We generated the data files using the TPC-H generator at 200 GB scale factor and copied them into HDFS as plain text. All of the query scripts we run are using the same underlying data.

3.4 Limitations

aditya: this whole subsection is confusing. What optimizations are we turning off? We need to be clear. We *are* enabling some optimizations, especially pertaining to query opt, but turning off others. We need to be more precise in describing these

As we described in subsection 2.2 different frameworks have various optimizations which make them performing well. For example Spark SQL is best optimized to work on data stored in Parquet format, while Hive atop ORCFile format. Similarly, both frameworks have a query optimizer. However, in order to be efficient it requires manual tuning and collection of statistics which are fed to the optimizer. In addition, different compression algorithms are indicated to be used. For example Snappy for Spark and Zlib or Snappy for Hive. In this paper, we disable all the optimizations describe above and leave their usage as a future work.

Another limitation of this work is that we run a single query at a time. However, this lets us to get a better understanding of the framework behavior by canceling the noise produced by external factors. For example, running a realistic workload will impede us measuring the amount of resources used by a specific query or to understand from where the gains per query comes from. Also, we minimize the variations in different optimizations an underlying framework may do. As an example, Hive or Pig set a level of parallelism in the query execution plan based on various hints like the amount of input data. However, when Tez process the plan, it might change the parallelism dynamically based on the workload in the cluster.

4. CLUSTER EXPERIMENTS

In this section we study the impact of various characteristics which affect applications performance. First, through empirical evidence we show why careful construction of the query execution plan as a single DAG is important. Second, given that different frameworks creates different DAGs, we deconstruct their performance by quantifying various metrics which reflects the impact of the query optimizer and the query execution engine. Last, but not least important we provide a discussion based on our findings.

Every experiment consists of a single query, run in isolation and it is repeated 3-4 times.

4.1 Single DAG vs. Multiple DAGs

Experiment Goal. Typically, in order to execute a SQL-like query atop analytics systems which runs MapReduce, the query framework translates the query execution plan which is a DAG in a sequence of MapReduce jobs. Recently, with the development of new frameworks capable of expressing jobs as arbitrary DAGs the query execution plan can be expressed as a framework specific DAG as well. Seeing the job representation as a single DAG can significantly improve performance. With this experiment, our goal is to quantify these benefits.

One of the most important characteristic of a single DAG is the ability to do pipelining. Pipelining enables efficient data transfers among dependent stages. By efficient we mean holding the intermediate data in memory or local disks, and data transfers as soon as it becomes available in the upstream stages. At the other extreme,

a sequence of dependent jobs lack this characteristic. For example, the output of a job is persistently written to HDFS before the dependent jobs are able to launch and read their input. This process can significantly delay the job completion time due to the burden of the replication mechanism in HDFS and the slowdown start of a MapReduce job.

Experiment Description. In the following when we refer to executing jobs as a single DAG we refer to their execution atop Tez and pipeline is enabled. Otherwise it is using MapReduce. We run the set of queries described in Table 1 using Hive and Pig, as a single versus multiple DAGs. We configured MapReduce framework to request the same container size as Tez in order to achieve the same parallelism in the cluster, and we enabled speculative execution. We also let Pig and Hive to decide the right number of reducers to use, based on their optimizations. Compression is disabled by default in MapReduce.

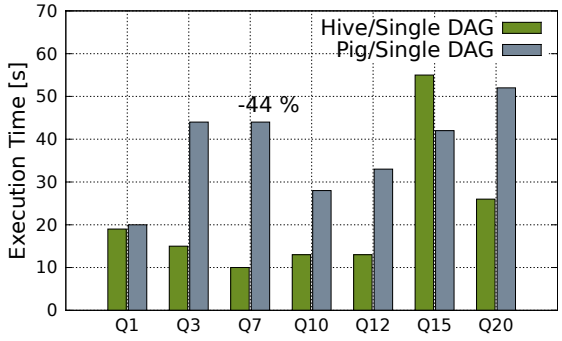


Figure 5: Job completion time improvement when pipeline is enabled.

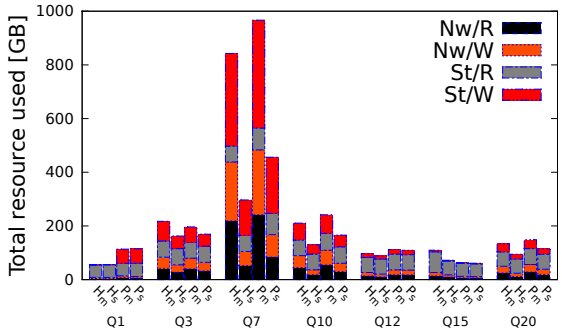


Figure 6: Pipeline impact over the amount of storage and network used. H_m **- Hive / Multiple DAGs;** H_s **- Hive / Single DAG;** P_m **- Pig / Multiple DAGs;** P_s **- Pig / Single DAG**

Experiment Discussion. Figure 5 shows the improvements in job completion time when the query is run as a single DAG. As we can see, the query performance improves on average by 22% for Hive and 38% for Pig. In practice we expect these gains to be even more dramatic, where contention among queries will exacerbate the inefficiencies due to multiple DAGs creation.

	Q1	Q3	Q7	Q10	Q12	Q15	Q20
Hive	25	18	7	8	19	26	13
Pig	11	21	−4	11	15	16	17

Table 2: Memory efficiency improvements of single DAG jobs [%].

In order to understand the cost of pipelining in overall performance, we also captured the amount of resources used during the job execution. Figure 6 shows the amount of data read and written from/to network/storage. On average, Hive as a single DAG transfers 40% less data over the network and 48% less data is written to disks comparing with Hive running as multiple DAGs. Similarly, Pig as a single DAG does 27% less network traffic and 23% less disk writes comparing with its alternative approach. The additional amount of data generated is due to the lack of pipelining which requires storing data persistently and being replicated. We observe that these numbers are significantly and becomes very problematic in practice where thousands of queries can run simultaneously, the network is not full bisection bandwidth or the data is geo-distributed across multiple sites based on current trends(cite GDA papers).

We also measured the cluster efficiency. We define the efficiency of a resource A, as the fraction of A used from the total amount of A available during the experiment. Table 2 shows the improvements in terms of memory efficiency when the queries are run as a single DAG. On average, Hive single DAG is 17% more memory efficient than Hive running as multiple DAGs, while Pig single DAG is 12% more efficient for the same. These gains are mainly due to pipelining effect which enables better buffering of the intermediate data in memory for direct transfers.

	Type	Q1	Q3	Q7	Q10	Q12	Q15	Q20
Hive	SD	.90	.37	.43	.39	.56	.47	.25
	MD	.32	.09	.17	.12	.06	.10	.06
Pig	SD	.25	.25	.26	.24	.25	.25	.26
	MD	1.41	.53	1.43	.59	.13	.07	.24

Table 3: Ratio of aggregate stages versus input stages. SD - Single DAG; MD - Multiple DAGs [%].

Furthermore, we are looking at other characteristics which can explain the performance of running jobs as single DAGs or multiple DAGs. For example, we observed that while MapReduce allows only for all-to-all communication between stages, in single DAG frameworks multiple communication patterns are possible to reduce the expense of data transfers when possible. In Tez, the current supported patterns are all-to-all, scatter-gather, one-to-one and custom where the application developer can specify its own communication

model. We also analyzed the ratio of tasks which aggregates data versus the number of tasks which reads data from HDFS. As we can see in Table 3, Hive single DAG under-parallelize the aggregate stages whereas the ratio numbers are much smaller than Hive multiple DAGs. This behavior might also explains the smaller improvements in job completion time for Hive single DAG. On the other hand, Pig provides larger ratio variations across different queries.

While Pig single DAG improves performance for most of the queries, it is not the case for Q7 where the performance degrades by 44% instead. Further inspection of the DAG reveals that Pig is parallelizing very poorly some of the stages. In Q7, we found 2 stages which account for most of the time penalty. These stages have 59 and 79 tasks respectively, while only 2 tasks in each stage are processing most of the data(12.8 GB and 15 GB of input data which accounts for over 98% of the input processed). We found that this is a known issue, and Pig/Tez currently exaggerates the parallelism for some stages and might be costly. A JIRA(cite JIRA) is pending and a patch will be available in the next Pig release.

4.2 Diversity in DAG generation

Experiment Goal. As we discussed in section 2, different "SQL-on-Hadoop" frameworks generates the query plan as a single DAG. Nonetheless, the DAG for the same query might be very different based on framework specific optimizations. In addition, query performance depends neither only on the generated DAG nor only the underlying execution engine, but it is tightly related to both. In this section we present a set of comprehensive measurements which decouples the impact of the query planning and the query evaluation engine on the overall performance. Finally, we provide a set of insights which hopefully will help the research community and frameworks developers to design/tune better query engines.

Experiment Description. We analyze the performance of four different "SQL-On-Hadoop" frameworks: Hive and Pig running atop Tez and use Hadoop as a query execution engine; Spark SQL over Spark and Presto. To provide a fair comparison we run the set of TPC-H queries described in Table 1. However, for some experiments we extend this set with queries that are able to succeed for particular frameworks. As in 4.1 we also monitor the resource usage encountered during every experiment, where an experiment consists of a single query run in isolation. At the beginning of every run, we clear the cache and remove any file created by frameworks from the past runs(for example RDD files).

4.2.1 *Query Performance*

Figure 7 shows the query completion time across different frameworks. In-memory frameworks improve the performance the most, but there is no clear winner between Spark SQL and Presto: in Q3, Presto outperforms Spark SQL by 47% while for Q20 Spark SQL is doing better by 36%. Also, Hive is better than Pig for most of the queries, except Q3. In some cases(for example Q15 or even Q12), none of the frameworks improves performance substantially.

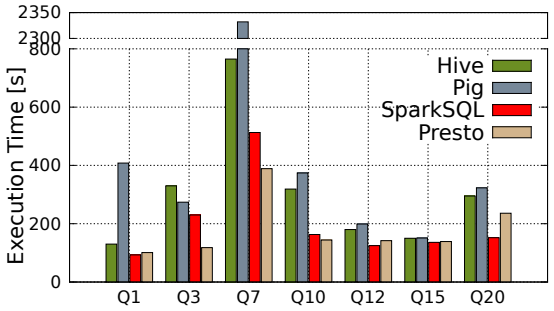


Figure 7: TPC-H: Job Completion Time.

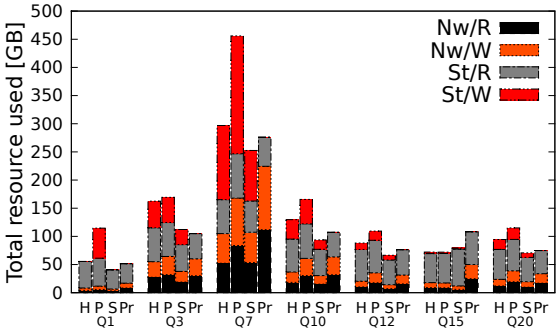
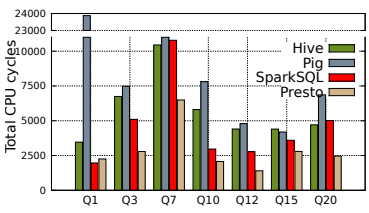


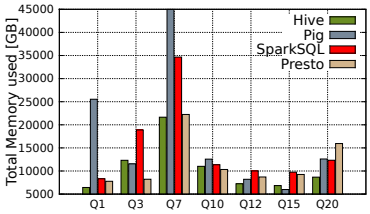
Figure 9: Total amount of network and storage used. H-Hive, P-Pig, S-Spark SQL, Pr-Presto

We computed the cluster memory efficiency and show the results in Figure 8(c). As expected, Spark SQL and Presto are using the memory more efficient and on average above 50%. Surprisingly, Spark SQL is more efficient on average by 23% than Presto, although it writes the intermediate data to disk. We believe this is due to the RDD data which is persistently kept in memory during the job runtime and push the memory usage higher. We also computed the CPU cluster efficiency. However, the numbers are not insightful and below 15% in most of the cases with no clear distinction among frameworks. One of the reason is we run a single query at a time which makes difficult for the CPU to be highly utilized during the execution, due to the existence of choke points at various levels in the DAG.

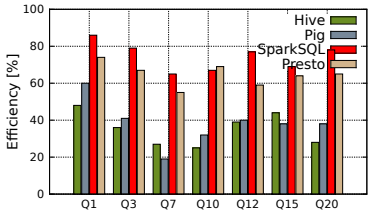
We also plotted the resource footprint for every run. Figure 8(a) and Figure 8(b) show the amount of CPU cycles and the total amount of memory used while Figure 9 the amount of data transferred from/to



(a) Total CPU cycles used.



(b) Total memory used. (GB)



(c) Cluster memory efficiency. (%)

Figure 8: TPC-H: Resources Used

disk/network.

For ***Q1***, Pig takes 3-4 more times to finish it than the other approaches. This translates to highest number of CPU cycles used and the largest amount of memory consumed. These values are correlated with the observation that Pig does 300 times more disk writes than any other framework. We believe most of these writes happens during the processing phase where Pig generates temporary data which is written to disk. Our finding is also confirmed by the small amount of network traffic which is comparable with the other frameworks and which makes sense considering that Q1 has no join operations.

Q3 and ***Q10*** queries look very similar. Both have joins on the same tables except Q10 which has an additional join. For *Q3* Presto improves job completion time by at least 49% comparing with the other frameworks at the expense of more network traffic. Hive and Pig performs very similar and their resource footprint look similar despite the fact that their DAGs are different as we will see in 4.2.2. The same observations applies to Q10 except that Spark SQL and Presto achieves similar performance.

For ***Q12*** and ***Q20***, Spark SQL performs the best. Looking at the resource footprint we observe that Spark SQL generates the fewest amount of network traffic among all the frameworks, which might be one of the major cause of better performance.

Q15 is an example of a query where none of the frameworks outperforms the others. While there is some variety in the DAGs generated, as we will see in 4.2.2, it impacts less the performance. We believe the main reason of this characteristic is the nature of the query which is highly selective.

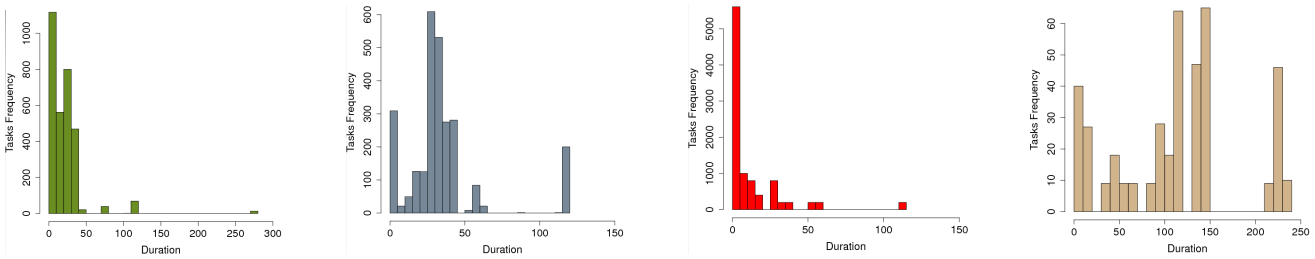
An interesting query is ***Q7***. Pig takes 3 times more than Hive to complete it, despite the similarity in the cluster memory efficiency and the total number of CPU cycles used. One of the penalty factor is the disk whereas Pig generates the largest amount of data which is written to disk. It is also interesting to observe that Presto is completing the query in the shortest time. However, the amount of network traffic generated is the largest and closer to the amount generated by Pig. Q7 is a perfect example to understand the benefits of in-memory frameworks. Presto does not generate almost any disk write operations even if the amount of intermediate data transferred over the network is high, and the amount of disk reads is also small. The Spark SQL number of reads from disk is comparable with Presto and we believe the main performance penalty comes from generating data to disks although smaller values comparing with Pig and Hive but much larger than Presto.

Next, we do a better investigation of these gains analyzing various metrics which characterize the query plan and the query execution engine.

4.2.2 *Impact of Query Planning*

Tasks Distribution. One of the primary effect of various DAG generation schemes is the variety in tasks distribution. Hive and Pig generates the number of tasks per stage based on hints regarding the amount of data to process and to be generated for later stages. Spark SQL is highly parallelizable and assigns the same fixed number of tasks for every stage disregarding the amount of work or data to be processed. Presto assign a task per stage per machine and the tasks are multi-threaded. Figure 10 shows the tasks distribution among the evaluated query set from Table 1. Hive and Pig generates tasks with duration mainly in the range 1 second to 50 seconds with Pig on the higher end of this range and Hive on the higher end on the number of tasks generated. Pig also has more variety in terms of tasks duration, observation which can lead to two conclusions: there is a lot of straggling effect which is less possible considering that both Hive and Pig use Hadoop as a query execution engine. The second thought is the data is more skewed in Pig than Hive, observation which is confirmed by Figure 11.

At the other extreme, Spark SQL generates the largest number of tasks among all the frameworks. However most of the tasks are tiny and in the range 1 second to 10 seconds. Presto query planning produces very few tasks as expected but the tasks have very high variation with durations in the range 1 second to 150 seconds for most of the tasks, with a significant fraction of tasks on the higher end. Presto tasks



(a) Hive (b) Pig (c) Spark SQL (d) Presto

Figure 10: Tasks distribution across different frameworks

are longer because they are multi-threaded and process larger amounts of data than what a single-threaded Hadoop or Spark task does.

		Q1	Q3	Q7	Q10	Q12	Q15	Q20
Hive	25%	200	441	472	442	400	426	427
	50%	63	39	36	3	21	32	24
	75%	1	1	13	48	1	7	6
	100%		1	2	2	1	1	1
Pig	25%	200	280	242	283	240	205	267
	50%	140	80	127	70	31	6	14
	75%	141	4	11	18	2	2	2
	100%	1	1	1	20	1	1	1
Spark SQL	25%	400	600	1002	601	600	800	1001
	50%	205	200	400	400	200	400	600
	75%		200	400	200	200	201	400
	100%		200	200	200	3	202	
Presto	25%	9	27	52	28	18	27	37
	50%	9	18	36	27	9	18	36
	75%	1	9	18	9	9	10	9
	100%		1	1	1	1	1	1

Table 4: Tasks distribution by different frameworks. 25% × depth; 50% × depth; 75% × depth; 100% × depth.

To provide a more detailed explanation we also computed the distribution of tasks inside every query. Table 4 shows the number of tasks at various levels in the generated DAGs by the frameworks. For any particular DAG and framework, the largest number of tasks are located in the top stages. This is explained through the fact that most of these stages are extractors, which process the input data which in general is the largest amount of data provided as input to any stage in a DAG. Later in the DAG the number of tasks decreases. For example in Hive and Pig only few tasks are produced in any stage later in the DAG for most of the queries, while for Spark SQL and Presto the trend is smoother. Also, it is interesting to observe that Spark SQL has a number of tasks at every level as a multiple of 200 and this is because as default, 200 tasks are generated for every stage.

DAGs Characterization Metrics. Tables 5 (a), (b) and (c) do a characteristics of the generated DAGs by different frameworks using standard metrics such as number of stages, depth and width. In every table, the red number represents the maximum value across all the frameworks for the same query. As we can see, there is a significant variation in number of stages across frameworks. For example in Q10, Hive generates 8 stages while Pig 12 stages. Also, Pig produces the most number of stages across all the queries, except Q15 while Hive and Spark the smallest number.

We define the depth of a DAG as the maximum number of hops required to traverse the DAG from a top level stage to a leaf stage. For our query set, Pig is consistently generating DAGs with largest depth among all the analyzed frameworks.

We also define the width of a DAG as the number of stages which can be categorized on the same level. A stage is located on the level with highest label where one of his parent is located, plus one. For our query set, DAGs generated by different frameworks have different width for different queries. For example, for Q1 and Q20 SparkSQL generates the largest width, for Q3 and Q7 by Presto, while Q10 by Hive. Hence there is no clear "winner" in terms of the framework which produces the most wide DAG.

	Depth	Width	Num. stages	Num. tasks
Hive	−0.10	0.63	0.62	0.82
Pig	0.33	0.20	0.35	0.71
Spark SQL	0.877	0.59	0.61	0.46
Presto	0.126	0.68	0.699	0.24

Table 6: Correlation of various metrics to runtime

To get a better understanding of how these metrics are correlated with the query completion time, we compute the correlation coefficient using Pearson algorithm. For every framework, we run additional queries from the TPC-H benchmark in order to have more sample points to provide more accurate correlation results. For Hive, we were also able to run Q4, Q5, Q6, Q11, Q13, Q14, Q16, Q17, Q18, Q21, Q22, for Pig Q4, Q22 while for Spark SQL Q6, Q13, Q14, Q22, and Presto Q5, Q6, Q14, Q16, Q21 respectively. As we can see in Table 6, there is significant correlation between runtime and various metrics. Hive query runtime is highly correlated with the number of tasks but it also has good correlation with the width and number of stages. Pig

	H	P	S	Pr
Q1	3	**5**	4	3
Q3	7	**10**	6	7
Q7	11	**13**	12	**13**
Q10	8	**12**	8	9
Q12	5	**7**	6	5
Q15	8	9	**10**	8
Q20	12	**15**	11	11

(a) Number of stages

	H	P	S	Pr
Q1	2	**4**	1	2
Q3	4	**7**	3	4
Q7	5	**7**	6	6
Q10	4	**8**	4	5
Q12	3	**5**	2	3
Q15	4	**7**	3	4
Q20	6	**8**	3	4

(b) Depth

	H	P	S	Pr
Q1	1	1	**2**	1
Q3	2	**3**	2	**3**
Q7	**6**	4	5	**6**
Q10	**4**	**4**	3	**4**
Q12	2	2	**3**	2
Q15	3	2	**4**	3
Q20	4	5	**6**	5

(c) Width

Table 5: Characterization of simple metrics.

runtime has good correlation with the number of tasks while Spark SQL has high correlation between job completion time and the depth of the DAGs. Lastly, Presto runtime is correlated with the width and the number of stages. Driven by these observations, we conclude that there is no single metric to correlate with the job completion time. Different frameworks have good correlations with different metrics.

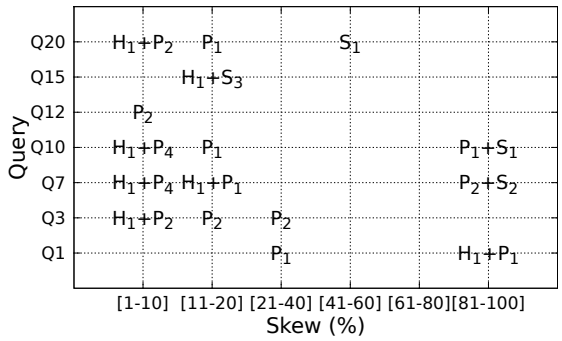


Figure 11: Data skew distribution. H-Hive, P-Pig, S-Spark SQL [%]

Data Skew. Another important characteristic of the query planning is the amount of data skew generated. Data skew represents the difference in the input processed by tasks of the same stage. We were able to extract this information for all the frameworks except Presto. Figure 11 shows for every query the number of stages skewed by every framework and the amount of skew in percentages. For example for Q15, Hive has a stage with a data skew between 11% and 20% while Spark has 3 stages with skew values in the same interval. As we can see, Hive, Pig and Spark create skewed stages. For most of the queries, Hive skew the input less, while Spark SQL do the most. Also, Hive and Spark SQL creates skew only for few stages in most of the queries while Pig generates skew for most of them. This behavior directly impacts the query performance due to long tasks which impede the query to do progress or to complete in a timely manner.

Amount of Data Generated by a Query Plan. A query performance depends also on the amount of data generated during query execution. More data requires more resources for processing it, more disk reads and writes operations and larger network transfers. In this experiment we computed the amount of input data for all the "interior" stages in the DAG, as a fraction of the input data for top stages. This metric quantifies how much of the input data is regenerated later in the DAG and hence how complex can be the query execution. Figure 12 shows our results. Spark SQL is highly selective and generates the least amount of intermediate data which is in the flight. This also might explain why Spark SQL performs well without stressing none of the resources too much. On the other hand, surprisingly Presto generates the largest amount of intermediate data, despite its best performance in terms of job completion time. However, Presto's ability to perform in-memory computation and minimizing the number of operations to disks makes it a feasible solution when network is not a bottleneck. Hive and Pig generate similar amounts of data at runtime which translates to similar overall performance.

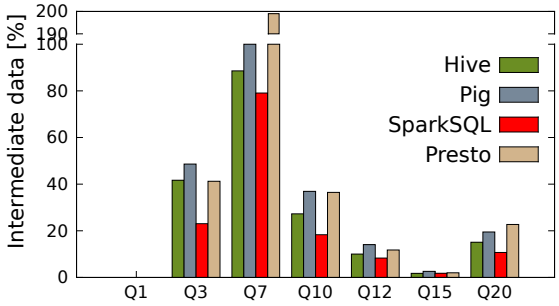


Figure 12: Fraction of input data of intermediary stages over the input data for top stages. [%]

4.2.3 *Impact of Query Execution Engine*

Execution Skew. While data skew can significantly degrades query performance, tasks with long duration can happen due to others effects as well, which we call execution skew. One of the important cause is the straggler effect. Stragglers are typically tasks which takes longer than usual due to various scheduling aspects which can impede them to do progress. For ex-

	Hive				Pig				Spark SQL				Presto			
	25%	**50%**	**75%**	**100%**	**25%**	**50%**	**75%**	**100%**	**25%**	**50%**	**75%**	**100%**	**25%**	**50%**	**75%**	**100%**
Q1	34	33	0	**33.3**	40	20	0	**40**	0	0	75	25	100	0	0	0
Q3	57.2	0	**42.8**	0	**30**	20	**30**	**20**	0	16.7	33.3	50	71.6	14.2	14.2	0
Q7	36.4	**36.4**	**27.2**	0	46.7	**13.3**	**20**	**20**	16.7	0	33.3	**50**	61.5	15.4	15.4	7.7
Q10	62.5	0	**37.5**	0	33.3	**8.3**	**25**	**33.3**	12.5	0	25	**62.5**	100	0	0	0
Q12	40	20	40	0	57.2	**14.2**	14.2	**14.2**	0	0	100	0	100	0	0	0
Q15	37.5	**25**	37.5	0	77.8	11.1	11.1	0	10	10	20	**60**	75	12.5	12.5	0
Q20	58.3	8.4	**33.3**	0	66.7	**20**	**6.6**	**6.7**	9	0	27.3	**63.7**	63.8	18.1	18.1	0

Table 7: Variation in completion time inside a stage. 25% × #stages; 50% × #stages; 75% × #stages; 100% × #stages.

ample due to a bad task placement with respect to the data or the machine is too busy, due to failures, etc. We computed the variation in tasks duration across tasks in the same stage and we show the results in Table **??**. For every query and framework we computed the percentiles of stages who have variation in tasks duration in certain range. For example, Q1 executed by Hive has 34% of the stages with tasks variation between 0% and 25%, 33% of the stages with variation in the range 26% and 50%, 0 stages in 51%-75% and 33.3% of them have larger variation in between 76% and 100%. We marked with red the percentiles of stages where the stages with data skew are located. We observe that Pig has most of the stages with high execution skew due to data skew. However, for Hive and Spark SQL there is significant execution skew which is not correlated with the data skew. Despite this behavior, we believe the execution skew metric is not very important in overall jobs performance. For example Hive has high stage variation on the lower end while Spark SQL on the higher end which is a direct consequence of the underlying execution framework, Hadoop or Spark. However, Spark SQL performance is better than Hive for all the analyzed queries.

	Q1	**Q3**	**Q7**	**Q10**	**Q12**	**Q15**	**Q20**
Hive	92	92	90	92	91	93	91
Pig	89	90	90	90	88	91	89
Spark SQL	96	81	92	92	95	95	86
Presto	100	100	100	100	100	100	100

Table 8: Data locality.Fraction of tasks executed data local over the total number of tasks which can achieve data locality (%).

Data Locality. Another important metric which can impact query performance at runtime is the data locality which is defined as the amount of tasks which are executed on the same machine where the data resides or as close as possible in terms of network conditions. Not all the tasks in a DAG can be data-local. For example tasks which aggregates data from multiple machines. In this experiment we quantified the percentiles of tasks which were assigned local as a fraction of the total tasks which can be local. This metric characterize the underlying execution framework capability to achieve data locality. Table 8 shows our findings.

	JCT(%)	Nw/R(%)	Nw/W(%)	St/R(%)	St/W(%)
H_{25}	35	13	13	−8	4
H_{50}	32	12	11	3	5
H_{75}	65	17	16	4	9
P_{25}	38	15	15	3	1
P_{50}	32	15	15	5	5
P_{75}	42	12	12	−3	1
S_{25}	7	6	6	0	4
S_{50}	18	6	6	10	9
S_{75}	23	11	11	4	11
Pr_{25}	∞	∞	∞	∞	∞
Pr_{50}	∞	∞	∞	∞	∞
Pr_{75}	∞	∞	∞	∞	∞

Table 9: Overhead due to failures. H_{25} **- fail the machine when the query in Hive reached 25% of its lifetime.**

Hive and Pig achieves nearly 92% data locality while Spark SQL near the same. Unsurprisingly, Presto achieves 100% data locality which is mainly due to its capability to enforce tasks to execute locally, at the expense of some delay. However, we expect more variations if a complete workload is run instead.

Fault Tolerance. Last but not least important, the query performance is also determined by the ability of the underlying execution frameworks to cope with failures. In this experiment, we run Q3 and we fail one of the compute machines when the job progress is at 25%, 50% and 75% respectively in its lifetime.

Table 9 shows our results. Every row in the table represents the penalty in job completion time, and the amounts of network/storage read/writes due to failure. Every framework impacts query performance due to the failure event. Hive and Pig delays the job completion time on average by 44% and 38% respectively. Spark SQL also has performance loss, but significantly lower. This is mainly due to the robustness of Spark against Hadoop. It is also interesting to observe that the loss in performance is a function of the time when the failure happens. For example the performance is highly impacted when failures happens at 75% at its lifetime for Hive, Pig and Spark SQL while the least impacted due to the early stages of the job. This is explained due to the fact that later in the job lifetime the aggregate stages are running, and failures of a machine results in recomputing of the intermediate data lost due to the failed machine. This observation is also confirmed by

larger amounts of network traffic generated due to recomputation when the failure happens later in the job lifetime. At the other extreme, Presto does not support fault tolerance at all and the failure of the machine translates into the query failure. We were not able to successfully run Q3 when Presto faces failures. However, this is in agreement with its specification and design choices.