MuSQLE: Distributed SQL Query Execution Over Multiple Engine Environments

Victor Giannakouris*, Nikolaos Papailiou*, Dimitrios Tsoumakos§ and Nectarios Koziris*
*Computing Systems Laboratory, National and Technical University of Athens, Greece
{vgian, npapa, nkoziris}@cslab.ece.ntua.gr
§Department of Informatics, Ionian University, Corfu, Greece
dtsouma@ionio.gr

***Abstract*—Multi-engine analytics has been gaining an increasing amount of attention from both the academic and the industrial community as it can successfully cope with the heterogeneity and complexity that the plethora of frameworks, technologies and requirements have brought forth. It is now common for a data analyst to combine data that resides on multiple and totally independent engines and perform complex analytics queries. Multi-engine solutions based on SQL can facilitate such efforts, as SQL is a popular standard that the majority of data-scientists understands. Existing solutions propose a middleware that centrally optimizes query execution for multiple engines. Yet, this approach requires manual integration of every primitive engine operator along with its cost model, rendering the process of adding new operators or engines highly inextensible. To address this issue we present MuSQLE, a system for SQL-based analytics over multi-engine environments. MuSQLE can efficiently utilize external SQL engines allowing for both intra and inter engine optimizations. Our framework adopts a novel API-based strategy. Instead of manual integration, MuSQLE specifies a generic API, used for the cost estimation and query execution, that needs to be implemented for each SQL engine endpoint. Our engine API is integrated with a state-of-the-art query optimizer, adding support for location-based, multi-engine query optimization and letting individual runtimes perform sub-query physical optimization. The derived multi-engine plans are executed using the Spark distributed execution framework. Our detailed experimental evaluation, integrating PostgreSQL, MemSQL and SparkSQL under MuSQLE, demonstrates its ability to accurately decide on the most suitable execution engine. MuSQLE can provide speedups of up to 1 order of magnitude for TPCH queries, leveraging different engines for the execution of individual query parts.**

***Keywords*-Polystore, Multi-Engine Optimization, SQL, Cost-models, Big Data**

I. Introduction

Big Data analytics constitutes a large fraction of modern datacenter workloads with numerous applications in most aspects of business and everyday life. Current methods and tools for Big Data analysis are quite diverse, being dictated by the heterogeneity of use cases that operate over different data formats, computational and functional requirements. This reality has brought forth an abundance of: i) computation languages and models (e.g., SQL, MapReduce, BSP, etc), ii) data store technologies (e.g., NoSQL stores [1], [2], columnstores [3], distributed FSs [4], etc), and iii) execution engines (e.g. [5], [6], [7], [8], etc).

While all these systems have been successful, they still showcase their advantages on a subset of analytics applications. Their striking limitation is that they require specific data formats and query inputs, being able to utilize only their custom execution engine. The need for a *multi-engine* approach, that splits and coordinates workflow execution among multiple engines has been recently recognized and is gaining increasing attention ([9], [10], [11], [12], [13]).

Regarding a standardized querying language, SQL, the mainstay query language for RDBMSs, is generally acknowledged as a top in-demand skill for this new era, with new platforms constantly seeking its support ([14], [15], [16], [13], [17]). SQL emerges as the de facto language for big data due to its extensibility, its ability to naturally represent queries that can be optimized, and the vast number of products and developers that already support it. These points highly suggest a multi-engine approach that allows SQL analytics over multiple data formats and uses the most appropriate engines.

Recent attempts along this line ([9], [18], [11]) tackle the issue by producing federated systems: There exists a custom subsystem where the different engines' operators and cost models are integrated and optimized planning is performed. While this is achieved for the already included systems and operators, this setup lacks extensibility as it requires a lot of manual work in order for one to incorporate a new analytics engine or support new operators on already integrated ones. Moreover, it frequently proves suboptimal to define a single, global optimization layer, and disregard the capabilities of the underlying engines to locally optimize.

In this work we present MuSQLE, an open-source framework[1] for high-performance SQL-based analytics over different data sources and execution engines. MuSQLE is able to overcome the aforementioned deficiencies and optimize simple or complex SQL queries. Our solution adopts a novel API-based strategy for integrating runtimes: Instead of manual integration, MuSQLE utilizes standardized, API-based cost-model and execution endpoints from the participating engines. Our system is able to optimally disseminate parts of the initial query (including the appropriate data movements between stores) using state-of-the-art planning

[1] https://github.com/gsvic/MuSQLE

and letting individual optimizers handle the respective subqueries. MuSQLE utilizes Spark [14] as an executor and orchestrator layer, extending its current functionality as well as providing it with a native cost-model.

In summary, our work makes the following contributions:

- We propose a generic SQL engine API that can facilitate multi-engine query optimization. The API is based on well-documented SQL functionality and can be implemented using generic, engine-agnostic interfaces like JDBC and ODBC.
- We integrate this API into a state-of-the-art query optimizer, allowing for external, multi-engine cost-based query optimization. Our optimizer runs on the logical level, allowing the connected engines to have full control of physical optimization and join execution. This approach avoids the detailed enumeration of all physical operators on the external optimizer and thus further facilitates the integration of a new SQL engine.
- We compile and utilize a cost model for the SparkSQL operators. This model is used within our query planner to achieve query optimization for SparkSQL.
- We present a fully functional system that integrates three popular engines: SparkSQL [14], PostgreSQL [19] and MemSQL [20]. We describe our system architecture and components in detail, as well as extensively evaluate the utility and efficiency of our scheme.
- Our detailed experimental evaluation showcases that MuSQLE can accurately decide on the most suitable execution engine and provides speedups of up to 1 order of magnitude for TPCH queries, leveraging different engines for the execution of individual query parts.

II. Related Work

We now present some of the most relevant approaches in multi engine analytics focusing on SQL-based solutions. We emphasize on the pros and cons of each approach, distinguishing between two categories: *Research works* and *Production-level systems*.

Production Systems: *SparkSQL* [21] provides a complete in-memory query execution engine using Resilient Distributed Datasets (RDDs), allowing remote data manipulation via DataFrames. For example, it is possible to query tables stored in an RDBMS or a main memory store (e.g., MemSQL [20]) as well as Parquet files in a HDFS cluster. The user can query these tables using traditional SQL via the SQLContext or by using methods of the DataFrame interface. However, in such a case, SparkSQL needs to fetch and distribute every external table into its worker nodes in order to perform data operations. As a result, optimization and processing capabilities of the external stores are ignored (e.g., index scans in case of filters).

PrestoDB [16] is a popular system for distributed, analytical query execution over heterogeneous datastores developed at Facebook. It provides a distributed execution model using similar algorithms with SparkSQL for querying data across multiple datastores by providing an engine-specific connector for each external system (e.g., Kafka, Cassandra, Hive, etc.) in order to be integrated to the core system. PrestoDB also needs to fetch each table involved in the query without pushing any operation to the underlying runtimes.

Apache Drill [22] enables SQL-based querying over unstructured, schema-free datastores (e.g., HDFS, MongoDB, Azure). However, Drill does not utilize individual cost models, statistics and estimates in the planning phase. This prevents it from executing one or more subqueries locally even if the local execution would be faster.

Research Works: BigDAWG [13] addresses the problem of query resolution over heterogeneous environments by executing queries using *islands of information*, where each island refers to a data model, a query language and a set of data management systems. In BigDAWG, a query is optimized using either *Single-* or *Multi-Island Planning*. While the system supports native subquery execution inside the underlying engines, it treats each engine as a black-box. As a result, local optimizers are ignored.

A different approach is followed by CloudMdsQL [23] which provides a functional SQL-like language for querying data stored in heterogeneous datastores within a single query, focusing on the ad-hoc usage of each datastore's native query language and engine. CloudMdsQL supports the local execution of a subquery and shipping of intermediate results. As it focuses on integration, it requires custom-made wrappers that translate from source to target language and provide cost models for the optimizer to use. Furthermore, the optimization is more rule-based, using selection and join condition pushdowns. The MuSQLE optimizer is more refined: Statistics injection, when moving an intermediate result into another store, is utilized for increased accuracy.

A different data integration approach is followed by *QUEPA* [24], which focuses on data integration over polystores. QUEPA introduces two new query methods: *Augmented search* and *augmented exploration*. In summary, this approach enables the user to query the polystore without knowing the exact structure of each individual database. Using record linkage, QUEPA will return records enriched with relevant (similar) tuples of other databases of the polystore. In *exploratory mode*, the user is prompted to select the relevant information retrieved that she wishes to explore. However, QUEPA only focuses on integration and does not provide any optimization for the execution phase.

SQL++ [17] provides a semi-structured data model, which combines the traditional SQL language with JSON extensions making it easy to query NoSQL databases by embedding JSON queries inside SQL code. However, SQL++ focuses only on the proposed language without providing query planning optimizations.

MISO [11] focuses on the tuning of the physical design of polystores in order to minimize data movements of

intermediate results between the underlying stores. MISO aims at optimizing the performance of ad-hoc, big data query processing by deciding where data is best to reside in. Yet, MISO also needs to maintain and calibrate its own cost functions for estimating the cost of operations. As a result, for every new engine to be added there is an integration overhead to generate the appropriate cost estimators.

III. Architecture

Figure 1 depicts MuSQLE's architecture. Our system is designed to facilitate the execution of multi-engine SQL queries. Such queries can be executed over tables that reside in multiple engines. The *Metastore* module is responsible for storing the schema and location information for each table. Our *SQL Parser* communicates with the *Metastore* in order to validate a user query and create the query graph. After parsing the query, our *Multi-engine Optimizer*, discussed in Section V, finds the optimal execution plan taking into account logical operator ordering, engine selection for query subgraphs as well as the required intermediate result movements. The generated execution plan is a tree of SQL and move operators. Each SQL operator is bound to a specific engine and refers to a subgraph of the initial query. The move operators handle the transfers of intermediate results between different SQL engines.

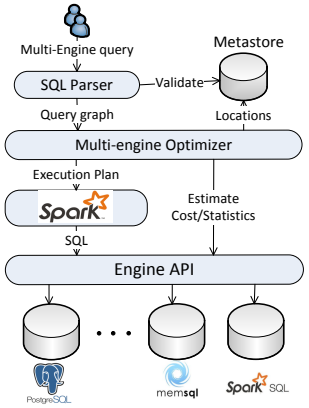


Figure 1: MuSQLE system architecture

In order for our optimizer to interact with the various engines, we introduce an *Engine API*, presented in Section IV. The API contains methods for the estimation of execution cost and intermediate result statistics which are used by our optimizer. It also contains SQL query execution methods as well as methods for retrieving and loading intermediate tables. We opt for utilizing Spark's engine as an execution framework for our multi-engine plans. Using Spark, MuSQLE provides scalable, main-memory based interaction between the connected SQL engines as well as primitives for fault-tolerance. Spark also provides a SQL interface allowing us to also utilize it as an alternative execution engine. We implement our engine API for three state-of-the-art engines: i) PostgreSQL, ii) MemSQL and iii) SparkSQL. The selected engines provide a diverse set, ranging from centralized to distributed execution, row- and column-oriented storage and disk-based to main-memory data indexing and join execution.

IV. Engine API

In this section, we describe in detail our *Engine API* that undertakes the integration between MuSQLE and the execution engine stack. To make our API generic and easy to implement, we devise five basic functions. Our functions are, in most cases, an extension of the already provided SQL functionality and require limited work to be implemented for a new engine. Our API functions are categorized in two groups: *Execution* and *Estimation*.

Execution functions: The following functions are used for executing our multi-engine query plans using Spark:

- `def execute(sqlQuery: String): DataFrame:`

This method sends a SQL query for execution to the specific engine. This is the most basic operation and can be implemented by extending well-known and massively used interfaces like JDBC and ODBC. The result of the query is loaded in a Spark DataFrame and can be subsequently moved to another engine for the execution of a query plan.

- `def loadTable(table: String, df: DataFrame): Unit:`

This method takes as argument a Spark DataFrame, which is an intermediate result table, and loads it in the specific engine. Again, this interface requires limited implementation overhead for most SQL engines.

Estimation functions: These functions are used inside our optimizer to estimate the execution cost and statistics of query subgraphs. In detail, the methods are:

- `def getStats(sql: String): Stats:`

This method is used to obtain an estimation of: i) the execution time of a specific SQL query and ii) the statistics of the result table. This functionality is already provided by many SQL engines in the context of the *EXPLAIN* statement. However, the results returned by different engines do not have a standard format. In most cases, they return the selected execution plan, the number of result rows as well as an execution cost measured in disk or cpu operations. To foster the integration of SQL engines in a multi-engine environment, we believe that such methods should follow a standard output format and return values that are comparable. Especially in the case of engines that use cost-based query optimization, such functionality already exists but is sometimes not exposed by the *EXPLAIN* output. To tackle this problem, we parse the output provided by the *EXPLAIN* statements of both PostgreSQL and MemSQL and use it to implement our API function. For Spark SQL, currently not utilizing a cost-based optimizer, we developed custom cost models for each operation as well as custom statistics, described in Section VI. Details on how we combine the

returned values in order to achieve unbiased query planning are presented in Section V-B.

- `def getLoadCost(stats: Stats): Double:`

This method returns an estimation of the time required to load a table from a Spark DataFrame to the specific engine. The statistics of the table, which contain its number of rows and columns are provided in the Statistics object. Cost functions similar to the following equation can be used for this task: $C_{move} = B_t \cdot T_{eng}$, where B_t is the size of the input table and T_{eng} is the transfer rate for the engine eng.

- `def injectStats(table: String, stats: Stats): Unit:`

This function is required when trying to obtain execution statistics for SQL queries that utilize intermediate results, not present in a specific engine. In a what-if optimization style, our optimizer injects all intermediate result statistics in the required engines, before calling the `getStats` method for queries that contain them. We analyze in detail how this is achieved by our optimizer in Section V. In brief, the `injectStats` method creates a "fake" table using the name and the statistics provided as argument. We have created custom code for doing this operation in all the integrated engines. However, assuming the multi-engine execution scenario is beneficial, such APIs can be easily implemented by the engine developers.

V. Multi-Engine Query Optimization

A. Optimization Algorithm

Finding the optimal join plan for complex queries has always been a major research challenge in optimizing database systems. One of the oldest and most efficient dynamic programming algorithms for join planning is *DPsize* [25], widely used in commercial databases like IBM's DB2. *DPsize* limits the search space to left-deep trees and generates plans in increasing order of size. A more recent approach, *DPccp* [26] and its variant *DPhyp* [27] are considered to be the most efficient, state-of-the-art dynamic programming algorithms for query optimization. They reduce the search space by examining connected subgraphs of the query in a bottom-up fashion. *DPccp* bases its enumeration procedure on finding all *csg-cmp-pairs* in the SQL join graph, where each table is represented by a vertex and join conditions are recorded using edges.

Definition 1: (csg-cmp-pair) Let $G = (V, E)$ be a join graph and S1, S2 two subsets of V such that S1 $\subseteq$ V and S2 $\subseteq$ (V $\setminus$ S1) are a connected subgraph and a connected complement respectively. If there further exists an edge (u, v)$\in$E such that u$\in$S1 and v$\in$S2, we call (S1, S2) a *csg-cmp-pair*.

In essence, *csg-cmp-pairs* are pairs that contain a connected subgraph (csg) of the query graph and one of its connected complement subgraphs (cmp). Each *csg-cmp-pair* corresponds to a 2-way join between the csg and the cmp graphs. This property ensures that the enumeration of all *csg-cmp-pairs* checks all possible 2-way join plans, ensuring the optimality of the selected plan. We extend the *DPhyp* algorithm in order to find the optimal join plan of a multi-engine SQL query. Our main extensions are:

Location-based optimization: One of the major differences of our algorithm compared to *DPhyp* is the structure of the dpTable used. To leverage the strengths of multiple engines, the optimizer needs to take into account the location of each intermediate result. A result with a certain location, while more expensive to generate than another, can be more efficiently used in a subsequent join leading to a better join plan for the query. To cover this case, we change the structure of our dynamic programming table: While in *DPhyp* only one plan is kept for each query subgraph, our dpTable maintains, for each query subgraph, a list of plans that contains the best join plan for each possible location (i.e., integrated SQL engine).

Multi-engine execution cost and statistics estimation: We integrate our generic SQL engine API with the *DPhyp* optimizer, allowing for the discovery of the optimal plan without depending on hand-coded or statically-integrated cost models. This approach separates the task of plan enumeration from the task of cost estimation, facilitating 'out of the box' integration and optimal utilization of new engines.

Algorithm 1 presents the pseudocode of our extended $emitCsgCmp(S_1, S_2)$ function. We maintain the same notation as used in [27], allowing interested readers to easily point to this paper for the details of the baseline algorithm.

ALGORITHM 1: $emitCsgCmp(S1, S2)$

```
1  plans_1 = dpTable[S_1];
2  plans_2 = dpTable[S_2];
3  S = S_1 ∪ S_2;
4  p = ⋀_{(u_1,u_2)∈E,u_i⊂S_i} P(u_1, u_2) ;
5  for (e_1, plan_1) ∈ plans_1 do
6      for (e_2, plan_2) ∈ plans_2 do
7          for e ∈ engines do
8              //execute query in engine e
9              c_1 = 0; c_2 = 0;
10             newPlan_1 = plan_1; newPlan_2 = plan_2;
11             if e_1 != e then
12                 table_1 = newTempTable();
13                 c_1 = e.getLoadCost(plan_1.stats);
14                 e.injectStats(table_1, plan_1.stats);
15                 newPlan_1 = move(newPlan_1, e, table_1);
16             if e_2 != e then
17                 table_2 = newTempTable();
18                 c_2 = e.getLoadCost(plan_2.stats);
19                 e.injectStats(table_2, plan_2.stats);
20                 newPlan_2 = move(newPlan_2, e, table_2);
21             newPlan = newPlan_1 ⋈_p newPlan_2;
22             sql = toSQL(newPlan, e);
23             stats = e.getStats(sql);
24             stats.cost = stats.cost + c_1 + c_2;
25             if dpTable[S][e] = ∅ ∨
                 stats.cost < dpTable[S][e].cost then
26                 dpTable[S][e] = newPlan;
```

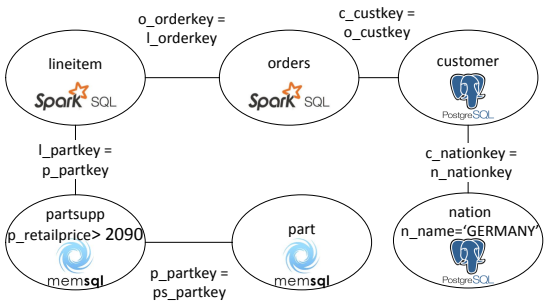


Figure 2: Join graph for Q_e

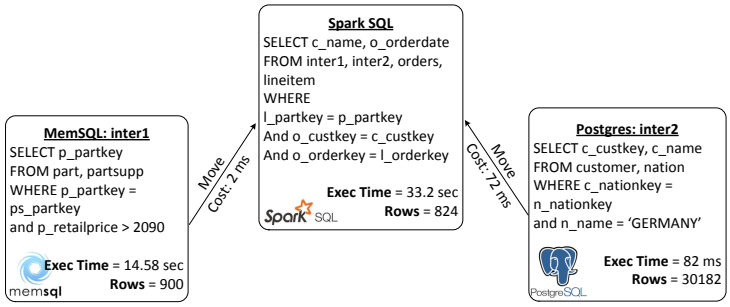


Figure 3: Multi-engine execution plan for Q_e

The $emitCsgCmp(S_1, S_2)$ function is called for each *csg-cmp-pair* and is responsible for checking the cost of the specific join using the optimal plans for S_1 and S_2 as well as the cost model.

As mentioned above, our dpTable is a two-dimensional array having one more dimension corresponding to the engine location of each intermediate result. Therefore, the $emitCsgCmp(S_1, S_2)$ function needs to evaluate the utilization of multiple plans, retrieved in lines 1-2, for both input query subgraphs. Additionally, we must check the cost of using these results in *all* the connected engines. Even if both intermediate results are not in a specific engine, loading them and continuing the query execution in that engine might be beneficial. To check all possible execution plans, we consider all combinations of left plans, right plans and execution engines (lines 5-26). For each of them, we check the locations of the left and the right plan, applying move operations if required. To apply move operators, we call the *getLoadCost* method for engine e in order to estimate the required time (lines 13 and 18). We also inject an intermediate temporary table, in engine e, using the API method *injectStatistics* (lines 14 and 19). This table will be used later on for query estimation.

After checking for move operations, we compose the result plan (line 21) and transform it to a SQL query string (line 22). The method to produce this string recursively iterates through the plan nodes stopping at move operators. This allows us to create a query string that refers to intermediate moved tables with their temporary names. The generated SQL query contains only the join information assigned for execution in engine e and is sent for estimation to the respective engine through the *getExecutionStats* API method. The dpTable record for $S = S_1 \cup S_2$, i.e., the resulting subgraph of the query, is updated only if the estimated cost is lower than the one contained in the dpTable for the specific combination of (S, e) (line 26).

To facilitate the understanding of our algorithm, we present a detailed query optimization example for the following query Q_e:

```
SELECT c_name, o_orderdate
FROM part, partsupp, lineitem, orders,
customer, nation WHERE
p_partkey = ps_partkey AND
c_nationkey = n_nationkey AND
l_partkey = p_partkey AND
o_custkey = c_custkey AND
o_orderkey = l_orderkey AND
p_retailprice > 2090 AND
n_name = 'GERMANY'
```

The above query is based on TPC-H data and returns all customers from Germany that ordered a part with retail price higher than 2090. For this example, we assume that the tables `lineitem` and `orders`, due to their large size, are stored in a Spark cluster, using HDFS files. Accordingly, the `customer` and `nation` tables are stored in a PostgreSQL server and the `part` and `partsupp` tables in a MemSQL cluster. The join graph of Q_e, along with the table location information is depicted in Figure 2. Our optimizer uses the *DPhyp* algorithm to enumerate all *csg-cmp-pairs* of the join graph. One possible *csg-cmp-pair* (S_1, S_2) is the one with $S_1 = \{part, partsupp\}$ and $S_2 = \{lineitem, orders\}$. When *DPhyp* calls the *emitCsgCmp* method for this pair, the optimal plans for both S_1 and S_2 can be found in the dpTable. Algorithm 1 checks all combinations of plans for S_1, S_2 and execution engines. This process results in selecting SparkSQL as the execution engine while moving the result of the optimal plan for S_1 from MemSQL to SparkSQL. In this case, the *toSQL* method (line 22 of Algorithm 1) refers to the intermediate result table coming from MemSQL with its temporary name "inter1" and therefore the remaining SQL query is:

```
SELECT o_custkey, o_orderdate
FROM inter1, lineitem, orders WHERE
l_partkey = p_partkey AND
o_orderkey = l_orderkey
```

This query is sent for estimation to the SparkSQL API. Its estimated cost is added to the move cost of "inter1" and the resulting plan is inserted in the dpTable for $S = \{part, partsupp, lineitem, orders\}$. Later on, the *csg-cmp-pair* (S'_1, S'_2) with $S'_1 = \{part, partsupp, lineitem, orders\}$ and $S'_2 = \{customer, nation\}$ is considered. This time *emitCsgCmp* selects SparkSQL as the execution engine while moving the plan of S'_2 from PostgreSQL to SparkSQL. The *toSQL* method

generates the following query, referring to both "inter1" and "inter2" coming from MemSQL and PostgreSQL:

```
SELECT c_name, o_orderdate
FROM inter1,inter2,lineitem, orders WHERE
l_partkey = p_partkey AND
o_custkey = c_custkey AND
o_orderkey = l_orderkey
```

Again, the estimation API of SparkSQL is called and the resulting plan is inserted to the dpTable, adding the required move cost. After the enumeration of all possible *csg-cmp-pairs*, the optimal plan of the query is discovered. The selected multi-engine plan is depicted in Figure 3 and consists of SQL and move operators. SQL operators are bound to specific engines and their estimated cost and number of results is depicted inside the respective boxes. The tree-based multi-engine plan is executed in a bottom-up fashion, using the Spark processing framework as well as our execution-related, engine API methods.

B. Comparing query estimations of different engines

Query optimization and execution time estimation are challenging tasks and are based on both cardinality estimations and operator cost models. In principle, as long as the cardinality estimations and the cost models of an engine are accurate, a good estimation of the execution time can be obtained for the query in hand. In reality, cardinality estimates are usually computed based on simplifying assumptions like uniformity and independence. Furthermore, cost model functions are oversimplified and do not take into account important parameters (e.g., the server load during query execution). In most cases, cost is measured in primitive operations like disk fetches or CPU cycles. While these measurements are considered to have linear correlation to the actual execution time, the correlation factors depend on hardware-specific parameters like the disk throughput or the CPU speed. A recent survey [28] illustrated that state-of-the-art SQL engines can easily misestimate costs by a factor of 1000 or more.

In this landscape, a major challenge for our system is how to compare and utilize the estimations provided by our user-implemented estimation APIs. To achieve an unbiased optimization procedure, we use the Metastore to record all query estimations, retrieved by the various APIs during query optimization. We also maintain, for each executed query, a detailed log of execution time for both the total query as well as it subqueries executed on different engines. This set of measurements is used to train machine learning models for tweaking the accuracy of the provided APIs. Our models target two challenges:

1) Transforming the costs measured using primitive operations (e.g., disk fetches) to estimations for the execution time. For example, the PostgreSQL EXPLAIN API returns the cost of the query in page fetches. Assuming a linear connection between the disk cost and the execution time, we use the previously described set of measurements to train a linear model that maps disk cost to execution time.

Symbol	Description
D_r	Cost of a single row read
D_w	Cost of a single row write
t_h	Cost of hashing a single value
t_{br}	Cost of broadcasting a single row
C_{cpu}	Cost of a single CPU comparison
$cores$	The number of cores in the cluster
$Part(s)$	The number of partitions of the relation s
$R(s)$	The number of rows of the relation s
S_p	spark.sql.shuffle.partitions

Table I: Cost Model Parameters

2) Due to inaccurate engine predictions or faulty API implementations, an engine can consistently fail to reasonably predict query execution time. To handle this case, we perform an accuracy analysis on top of all our query estimation measurements. This analysis computes the correlation between the estimated and the actual execution times for each engine. The computed correlation is used to adjust our confidence on a specific estimation API. Our optimizer uses a probability, proportionate to the measured correlation, to randomly discard the API estimation results. Therefore, in the case of an API that fails to achieve sufficient correlation to its actual execution times, the entire engine will be discarded from the optimization process.

VI. SparkSQL Cost-Based Query Optimization

Our engine API requires the implementation of cost estimations for each integrated SQL engine. While PostgreSQL and MemSQL provide such functionality through their internal cost-based optimizers, SparkSQL opts for heuristic-based optimization, without implementing cost models for its various operators. In this section, we present a cost model for SparkSQL which can be used to estimate the execution time of a SparkSQL physical plan. We also add support for injection of table statistics and utilize intermediate result cardinalities in order to provide more accurate execution time estimations.

Cost models: We have integrated a set of cost models into SparkSQL, extending the logic and formulas described in [29]. We present the cost models for three important operators: Sort-Merge Join, Broadcast-Hash Join and Exchange. Detailed formulas exist for all operators, yet we omit them due to space limitations. Table I presents the notation used in our cost models. First, we define the number of rounds required for an operation to run. The number of parallel tasks depends on how many CPUs a task uses. This can be set by modifying the `spark.task.cpus` parameter. For the rest of our discussion we assume that this parameter is set to 1. Thus, the number of tasks which can be run in parallel is equal to the number of cores in the cluster. The number of rounds required for a task of p partitions is thus:

$$Rounds(p) = \left\lceil \frac{p}{cores} \right\rceil$$

Exchange: This operation performs a shuffle operation on the data and partitions the results. The operation will be executed into $Part(s)$ tasks, where each task will process $R(s)/Part(s)$ rows. Each row will be hashed on the same column and will be sent to the corresponding partition according to the resulting hash value. The resulting cost is thus:

$$C_{exch}(s) = \frac{R(s)}{Part(s)} \cdot (C_{cpu} + D_w) \cdot Rounds(Part(s))$$

Broadcast-Hash Join: First, each row of the "small" table is hashed on the join condition attribute. This process takes place in the Spark driver node. Then, the hashed relation is broadcasted to all the workers. The cost of this operation equals: $C_{broadcast}(s) = R(s) \cdot (t_h + t_{br})$. After all nodes receive the hashed relation, a local join is performed for each partition of the large relation with the small relation. Thus, the total cost of a broadcast-hash join equals:

$$C_{bhj} = C_{broadcast}(s) + \frac{R(s) \cdot R(l) \cdot C_{cpu}}{Part(l)} \cdot Rounds(Part(l))$$

Sort-Merge Join: This is the SparkSQL distributed implementation of the traditional Sort-Merge Join algorithm. Before the actual join execution, each relation involved is first shuffled and sorted. Thus, for relation s the sorting cost equals: $C_{sort}(s) = R(s) \cdot \log R(s) \cdot C_{cpu} \cdot Rounds(s)$
After the two relations are sorted, they need to be merged. The merge cost is:
$C_{merge}(s,t) = R(s) \cdot R(t) \cdot Rounds(S_p) \cdot C_{cpu}$
Summing up, the cost of a sort-merge join is defined as:

$$C_{smj}(s,t) = C_{exch}(s) + C_{sort}(s) + C_{exch}(t) + C_{sort}(t) + C_{merge}(s,t)$$

VII. Statistics Injection

PostgresSQL Statistics Injection: In PostgreSQL, the $reltuples$ and $relpages$ columns of the pg_class system table represent the number of rows and pages respectively. Even if $relpages$ change manually, the planner checks the actual number of pages using the *RelationGetNumberOfBlocks()* method. Thus, in order to modify the statistics, we used *pg_dbms_stats*[2], an open source framework which provides functionality for "freezing" and modifying table statistics of PostgreSQL.

SparkSQL Statistics Injection: We extend SparkSQL, adding support for statistics injection and cardinality estimation. The existence of statistics lead to better cost estimations as well as to more accurate physical operator selection. For example, when planning queries that include tables stored in external data sources (e.g., MemSQL, PostgreSQL, etc), SparkSQL chooses explicitly the Sort-Merge Join algorithm even though the external table may be small. We solve this problem by *injecting* the input size of an external relation using the MuSQLE optimizer. Therefore, when SparkSQL is integrated with MuSQLE, it is able to adaptively select the use of *Broadcast Hash Join* for small external tables.

[2]https://github.com/ossc-db/pg_dbms_stats

VIII. Spark-Based Query Execution

After generating the multi-engine execution plan described in Section V, we need to execute this plan over a Spark cluster. SparkSQL uses Catalyst as its query optimizer. Thus, we need to transform our query plan into a SparkSQL equivalent using Catalyst's expressions. To do so, we developed a method which takes as input a multi-engine plan (i.e., see Figure 3) and transforms it into a SparkSQL native execution plan. We achieve that by traversing our plan in a bottom-up manner. At each leaf, using pattern matching, we match each operator of our plan (move, SQL) to the SparkSQL's Catalyst equivalent one. Finally, this method returns our execution plan represented in SparkSQL operations.

IX. Experimental Evaluation

In this section, we present a detailed evaluation of our multi-engine SQL platform. We integrate MuSQLE with a diverse set of engines, consisting of three state-of-the-art systems: PostgreSQL, MemSQL and SparkSQL. The selected engines excel on different and complementary data and query use cases, allowing MuSQLE to adaptively provide a combination of their advantages.

A. Cluster Setup and Compared Systems

All our experiments use Virtual Machine resources deployed on a private Openstack cluster, consisting of 8 VM containers. Each VM container has a 26-core Intel Xeon® CPU at 2.67GHz, 48 GB of RAM and two 2TB disks setup with RAID 0. Our experiments run on 13 Virtual Machines each with 8G RAM, 4 VCPUs and 100G of disk storage. The VMs are organized as follows:

- SparkSQL: We setup a Spark cluster consisting of 1 master and 5 worker VMs. The master VM runs the Spark driver(master) and the HDFS namenode. Each of the worker VMs runs a Spark worker and a HDFS datanode.
- PostgreSQL: One Virtual Machine is used for running a PostgreSQL server.
- MemSQL: We use a cluster of 1 master and 5 worker VMs. The master VM runs MemSQL's Master Aggregator while each worker runs a MemSQL leaf node.

There is currently limited work on the multi-engine SQL optimization landscape as mentioned in the related work section (code for the most relevant [11] and [13] was not available till the time of publication). To showcase the advantages of our system, we compare it against the performance of the three underlying SQL engines, if they functioned individually.

B. Datasets & Query Sets

Datasets: We evaluate MuSQLE using synthetic data generated from the popular TPCH benchmark [30]. In order to test the scalability of our system as well as its optimization

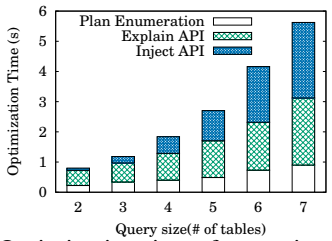


Figure 4: Optimization times for queries with variable sizes, using SparkSQL, PostgerSQL and MemSQL.

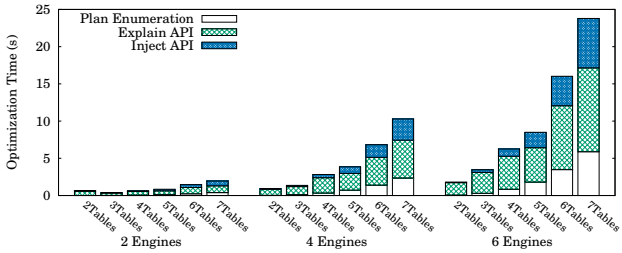
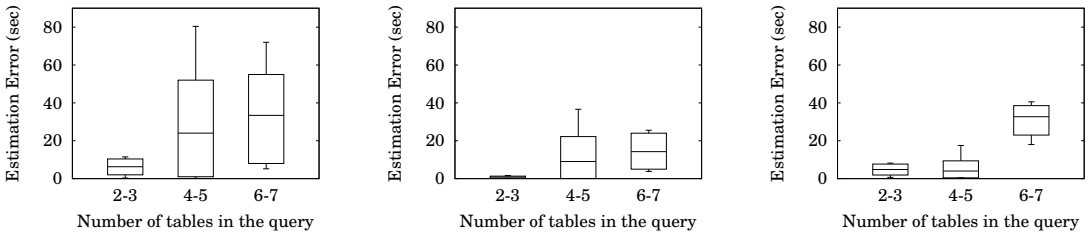


Figure 5: Optimization times for queries with variable sizes, using 2-6 fake SQL engine APIs.



(a) (b) (c)

Figure 6: Absolute estimation errors of execution time for (a) PostgreSQL, (b) MemSQL and (c) SparkSQL.

potential, we use three different TPCH scale sizes: 5GB, 20GB and 50GB.

Query sets: Our system focuses on SQL queries that include multiple tables residing in different engines. To present a detailed evaluation of such query scenarios, we generate a custom query set[3], extending the original TPCH benchmark queries. Our set consists of 18 queries, each one referenced as Qn, $0 \leq n \leq 17$. We classify our queries in two categories: i)**join-only** queries (Q0 - Q8) and ii)**join-filter** queries (Q9 - Q17). Queries of the first category contain multiple joins, producing large output sizes as they combine all information of the primitive tables without applying any filtering operation. In contrast, the second category includes queries with ranging selectivity, containing various filtering predicates. The selected categories can showcase the benefit of pushing the execution of subqueries to the individual engines. In brief, queries with large joins and small selectivity need to transfer large intermediate results between the connected engines and thus present small improvements. When queries have joins or subqueries with high selectivities, it is far more beneficial to push their execution to individual engines, transferring only their small intermediate results.

C. Query Optimization

The time required for optimizing queries that contain a variable number of tables, using our three integrated engines, is presented in Figure 4. As mentioned in Section V, our optimizer utilizes the engine APIs to estimate the cost and statistics of the various intermediate execution plans. External API calls can insert arbitrary large overheads on the optimizer's execution time. To measure this impact we break down the total optimization time into: 1) the plan enumeration time, 2) the time spent in external cost estimation APIs ("EXPLAIN API") and 3) the time spent on statistics injection ("INJECT API"). As depicted in Figure 4, we are able to find optimal plans for all our multi-engine queries within 6 seconds. However, the majority of the optimization time is spent on the external engine APIs. While the actual plan enumeration cost, introduced by our optimizer, is less than 1 sec for all queries, the total optimization time ranges between 1 and 6 seconds. This is largely attributed to the complexity of the external API implementations. For the purposes of this paper, we provided basic implementations for all engine APIs. However, the benefits of multi-engine execution can push individual engine experts to fine-tune the performance of the respective APIs.

To test the impact of adding a larger number of SQL engines on the optimization time, we simulate multiple engine API implementations. All methods of this API insert a delay, randomly selected from the distribution of delays of the actual engine API calls. Figure 5 presents the optimization times required for various numbers of connected engines. We note that the number of engines affects the performance of our optimizer. However, as presented in the following sections, this overhead is usually alleviated by the large improvements on the query execution times.

D. Cost Model Accuracy

Figure 6 presents the estimation accuracy for our integrated engines. We use a box plot in order to capture the average, standard deviation, min and max values of the error. As mentioned in Section V-B, we train regression models in order to translate the cost estimations of MemSQL and PostgreSQL into execution time estimations. As expected, the query estimation error increases with the query size, due to the propagation of erroneous cardinality and cost

[3]https://github.com/gsvic/MuSQLE/blob/master/Queries.scala

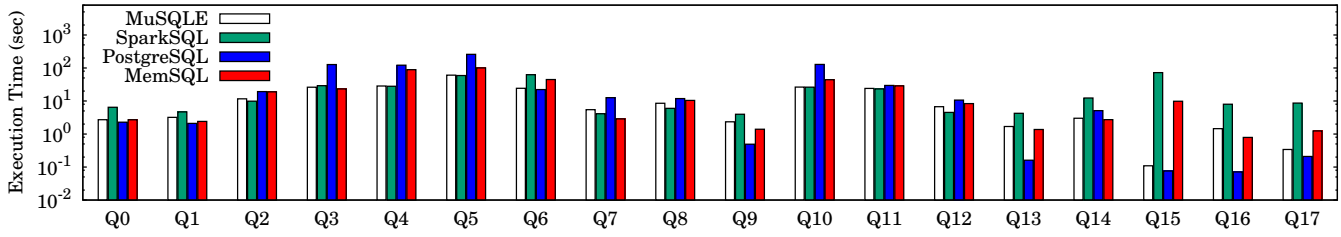


Figure 7: TPCH 5GB, all tables are stored in all engines

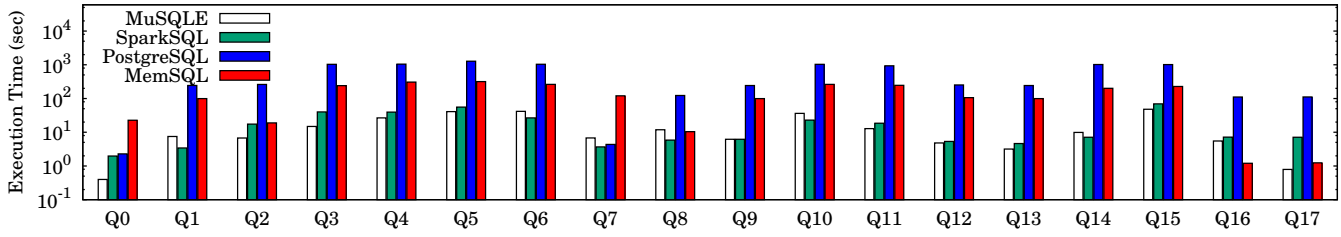


Figure 8: TPCH 5GB, each table is stored in a specific engine

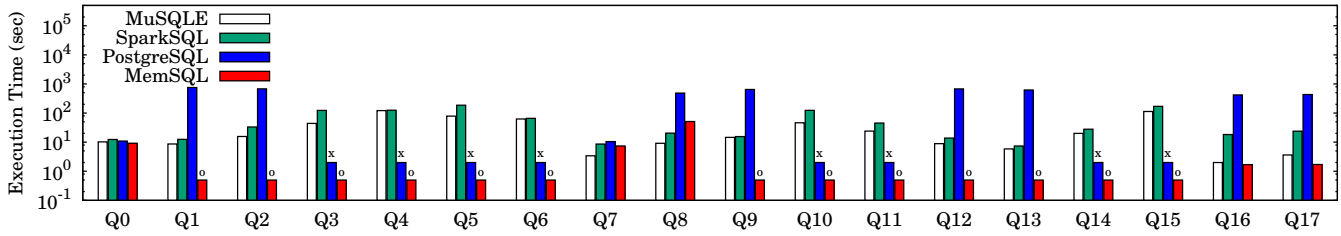


Figure 9: TPCH 20GB, each table is stored in a specific engine.
(**x**: execution time $>$ 20 minutes, **o**: out-of-memory)

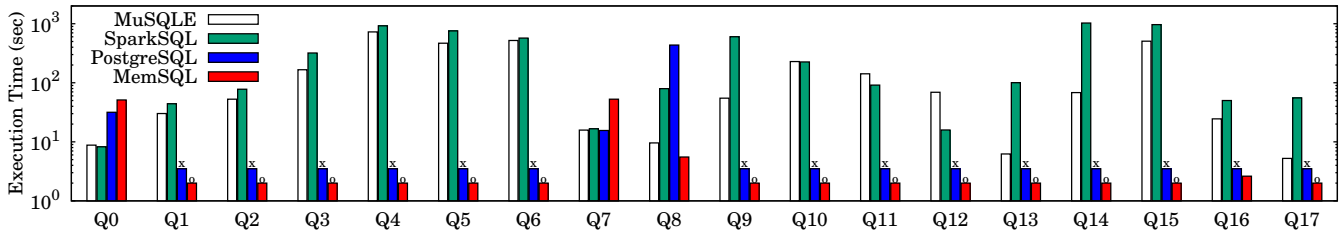


Figure 10: TPCH 50GB, each table is stored in a specific engine.
(**x**: execution time $>$ 20 minutes, **o**: out-of-memory)

estimations. However, we note that our proposed SparkSQL cost model, in conjunction with our proposed statistics, achieves good estimation accuracy.

E. Performance Comparison

To showcase MuSQLE's multi-engine benefits, we run the experiments under the following scenarios:

All tables stored in all engines: In this case, we assume that all tables are stored in all connected engines. Due to the existence of data in all engines, MuSQLE cannot achieve significant performance improvements with respect to the best underlying engine. However, this scenario nicely demonstrates the accuracy of our optimizer in such cases. As presented in Figure 7, our optimizer manages to select the best execution engine for most query cases. For $Q9$, $Q13$ and $Q16$ wrong cost estimates lead to sub-optimal execution plans.

Different Table Locations: In most real-life multi-engine query scenarios, tables will be stored in different engines according to their characteristics. For example, a table with high frequency of updates would be stored in an OLTP database, while a large log table in a plain HDFS file. To test this scenario, we select the following location for the TPCH tables: PostgreSQL stores the small sized tables (customer, nation, region). MemSQL stores the medium sized ones (part, partsupp, supplier), while the larger tables (lineitem, orders) are stored in HDFS.

TPCH 5GB: Figure 8 depicts the respective results for the TPCH 5 GB dataset. The performance improvement for this dataset is not significant due to the small amount of input data. This suggests that, in most cases, the optimal execution plan is to move the tables and execute the whole query in a single engine in order to prevent the intermediate result data movements. However, we note that our system, correctly

estimating the execution times for the different engines, selects the most profitable. For example, the execution of Q12 takes place in PostgreSQL while query Q17 runs in MemSQL. Again, this experiment showcases our optimizer's decision accuracy.

TPCH (20, 50)GB: For larger dataset sizes, all individual engines incur significant overheads when loading external tables. In such cases, plans that perform local processing inside the individual engines while moving small sized intermediate results prove largely beneficial. Specifically, Figures 9 and 10 illustrate that most of the queries could not be completed in MemSQL due to the large intermediate results which lead to an out-of-memory error. In the case of PostgreSQL, it took more than 2000 sec to complete the execution of several queries, requiring over twenty minutes to fetch the external tables. SparkSQL, taking advantage of its distributed execution engine, succeeds in handling all the queries of this scale. However, MuSQLE not only manages to select the most efficient execution engine, but also achieves better response times than SparkSQL by pushing local processing on the other engines. For example, $Q14$ contains one filter on $lineitem$ table. MuSQLE's execution plan pushes a subquery containing the filter into SparkSQL. The small sized intermediate results are then moved in MemSQL, where they are joined with the smaller $part$ and $partsupp$ tables. Similar improvements can be also observed for queries $Q13$, $Q15$, $Q16$ and $Q17$, which represent the cases that MuSQLE outperforms even the best individual engine, resulting in up to one order of magnitude speedups.

X. Conclusions

In this paper we presented MuSQLE, a multi-engine SQL executor. MuSQLE introduces a generic engine API that needs to be implemented for each integrated engine. We extend a state-of-the-art query optimizer, adding support for location based optimization and individual engine cost estimation. We have integrated MuSQLE with PostgreSQL, MemSQL and SparkSQL. Our detailed experimental evaluation proves that MuSQLE can accurately select the best execution engine for a large set of queries and provides speedups of up to 1 order of magnitude, leveraging different engines for the execution of individual query parts.

Acknowledgments

This work has been partly supported by the European Commission in terms of the ASAP FP7 ICT Project under grant agreement no 619706. Nikolaos Papailiou has received funding from IKY fellowships of excellence for postgraduate studies in Greece - SIEMENS program.

References

[1] "Apache HBase," http://hbase.apache.org/.
[2] "neo4j," http://neo4j.com/.
[3] "monetdb," https://www.monetdb.org/.
[4] "Hadoop Distributed File System," http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[5] "Apache Hadoop," http://hadoop.apache.org/.
[6] "Apache Spark," https://spark.apache.org/.
[7] "Apache Hama," https://hama.apache.org/.
[8] "Apache Flink," https://flink.apache.org/.
[9] A. Simitsis, K. Wilkinson, U. Dayal, and M. Hsu, "HFMS: Managing the Lifecycle and Complexity of Hybrid Analytic Data Flows," in *ICDE*. IEEE, 2013.
[10] D. DeWitt, A. Halverson, R. Nehme *et al.*, "Split query processing in polybase," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013.
[11] J. LeFevre, J. Sankaranarayanan, H. Hacigumus *et al.*, "MISO: Souping up big data query processing with a multistore system," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2014.
[12] K. Doka, N. Papailiou, D. Tsoumakos, C. Mantas, and N. Koziris, "IReS: Intelligent, Multi-Engine Resource Scheduler for Big Data Analytics Workflows," in *SIGMOD*, 2015.
[13] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The bigdawg polystore system," *ACM Sigmod Record*, 2015.
[14] "Spark SQL," https://spark.apache.org/sql/.
[15] "Apache Impala," http://impala.io/.
[16] "Presto," http://www.teradata.com/Presto.
[17] K. Ong, Y. Papakonstantinou, and R. Vernoux, "The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases," *CoRR*, 2014.
[18] A. Simitsis, K. Wilkinson, and P. Jovanovic, "xPAD: A Platform for Analytic Data Flows," in *SIGMOD 2013*.
[19] "PostgreSQL," https://www.postgresql.org/.
[20] "MemSQL," http://www.memsql.com/.
[21] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015.
[22] "Apache Drill," https://drill.apache.org/.
[23] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, and J. Pereira, "CloudMdsQL: querying heterogeneous cloud data stores with a common language," *Distributed and Parallel Databases*, 2015.
[24] A. Maccioni, E. Basili, and R. Torlone, "QUEPA: Querying and exploring a polystore by augmentation."
[25] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang, "Query optimization in the IBM DB2 family," *IEEE Data Eng. Bull.*, 1993.
[26] G. Moerkotte and T. Neumann, "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products," in *VLDB*, 2006.
[27] G. Moerkotte and T. Neumann, "Dynamic programming strikes back," in *SIGMOD*, 2008.
[28] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, 2015.
[29] D. Taniar, C. H. Leung, W. Rahayu, and S. Goel, *High performance parallel database processing and grid databases*. John Wiley & Sons, 2008.
[30] T. P. P. Council, "TPC-H benchmark specification," *Published at http://www. tcp. org/hspec. html*, 2008.