Benchmarking TPCH and comparing some machine learning algorithms on Spark and Flink

Hekmatullah Sajid
Seema Narasimha Swamy
Supervisor: Dr. Quoc Cuong To

Agenda

- ❏ Project Requirements
- ❏ TPCH Benchmarking in Flink and Spark
- ❏ Benchmarking Results
- ❏ Issues Faced/Limitations
- ❏ ML Algorithms Comparison and Benchmarking in Flink and Spark
- ❏ Summary

Project Requirements

- TPCH Benchmarking on Flink and Spark (completed)
- Comparison of some ML Algorithm on Flink and Spark(To be done)

Objective:

- Comparing the performance of Flink and Spark in terms of scalability ,efficient optimization and execution of SQL queries
- Comparing the performance of Flink and Spark in terms of scalability of certain Machine algorithms and its accuracy

Part 1: TPCH Benchmarking

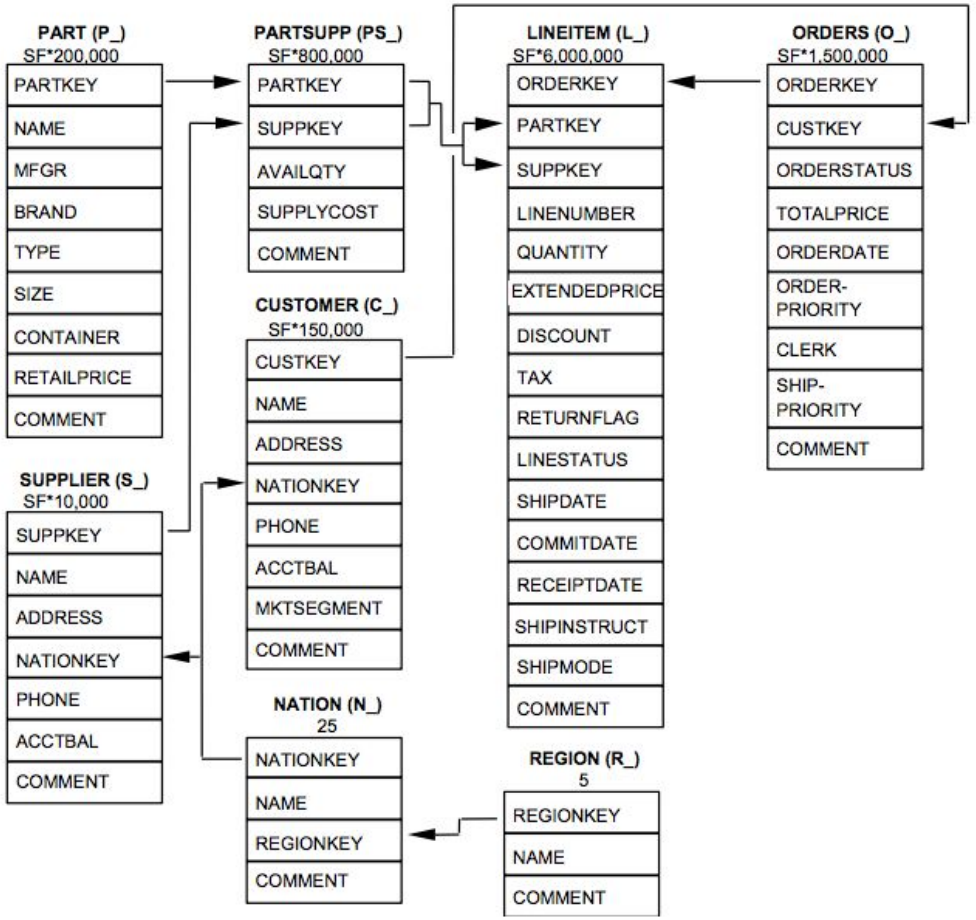
About TPCH Benchmark

- TPCH is the benchmark for decision support.
- It consists of 22 queries to cover various business questions.
- This benchmark illustrates decision support systems that
  - Examine large volumes of data
  - Execute queries with a high degree of complexity
  - Give answers to critical business questions
- TPCH Benchmark queries(22 queries) are implemented on Flink and Spark and the results are compared

Data Set For TPCH Benchmarking

- DBGen - a TPC provided software package could be used to produce the datasets
- Datasets can be scaled from 1 GB to 100 TB
- Supports query result validation
- At scale of 1.0 the source tables of TPCH schema holds
  - lineitem.tbl (742.0 MB)
  - orders.tbl (167.9 MB)
  - partsupp.tbl (116.2 MB)
  - customer.tbl (23.8 MB)
  - part.tbl (23.6 MB),
  - supplier.tbl (1.4 MB)
  - nation.tbl (3 KB)
  - region.tbl (1 KB)

TPCH-Schema



http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.2.pdf

TPCH Benchmark Queries on Flink

- Flink supports scalability of SQL Queries(still under development)
- Table API supported by Flink is used to implement queries
- Table API - SQL-like expression language for relational stream and batch processing
- Supports Java and Scala
- Tables can also be queried with regular SQL
- The logical plan of query execution optimized using Apache Calcite and transformed into a DataSet or DataStream program.

Example: TPCH Implementation

- Some of the queries can be executed directly as a SQL Query(if all the operations in the query are supported by Table API)
- For example, the query3 is quite straightforward which does basic operations like sum, average,count, groupby, orderby

```
String SQLQuery = "select l_orderkey, sum(l_extendedprice * (1-l_discount)) as revenue, o_orderdate, o_shipriority
        + "FROM customer, orders, lineitem "
        + "WHERE c_mktsegment = '" + segment +"' and "
        + "c_custkey = o_custkey and  l_orderkey = o_orderkey and "
        + "o_orderdate < '" + date.toString() +"' and "
        + "l_shipdate > '" + date.toString() +"' "
        + "GROUP BY l_orderkey, o_orderdate, o_shipriority ORDER BY revenue desc, o_orderdate";

Table res = env.sql(SQLQuery);
```

Example- SQL format Not Supported

- The Table api was not able to create logical plan for the query.
  - Joins and inner query were causing the issue(Query 2)

```
Table supplier = env.scan("supplier");
Table partsupp = env.scan("partsupp");
Table part = env.scan("part").filter("p_size = " + pSize)
        .filter("LIKE(p_type, '%" + pType + "%')");
Table nation = env.scan("nation");
Table region = env.scan("region").filter("r_name = '" + rRegion + "'");

Table minCost = part.join(partsupp).where("p_partkey = ps_partkey")
        .join(supplier).where("s_suppkey = ps_suppkey")
        .join(nation).where("s_nationkey = n_nationkey")
        .join(region).where("n_regionkey = r_regionkey")
        .groupBy("p_partkey")
        .select("min(ps_supplycost) as mincost");
Table outterQ =  part.join(partsupp).where("p_partkey = ps_partkey")
        .join(supplier).where("s_suppkey = ps_suppkey")
        .join(nation).where("s_nationkey = n_nationkey")
        .join(region).where("n_regionkey = r_regionkey");
Table res = outterQ.join(minCost).where("ps_supplycost = mincost")
        .orderBy("s_acctbal, n_name, s_name, p_partkey")
        .limit(100)
        .select(" s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment ");
```

TPCH Benchmark queries on SPARK

- Spark SQL allows to query structured data inside Spark programs
  - Spark SQL or Dataframe API
- Spark SQL provide Spark with more information
  - About the structure of data and the computation being performed.
  - Used for extra optimization
  - Same execution engine is used
- Supports Java, Scala, R , Python
- DataFrame is conceptually equivalent to a table in a relational database
  - Provides functionalities like select, filter, groupby

Example: Query implementation

- Spark supports a wide range of SQL operations when compared to Flink.
- SQL Query can be directly executed as shown below:

```
return spark.sql("select l_returnflag, "
        + "l_linestatus, "
        + "sum(l_quantity) as sum_qty, "
        + "sum(l_extendedprice) as sum_base_price, "
        + "sum(l_extendedprice*(1-l_discount)) as sum_disc_price, "
        + "sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge, "
        + "avg(l_quantity) as avg_qty, "
        + "avg(l_extendedprice) as avg_price, "
        + "avg(l_discount) as avg_disc, "
        + "count(*) as count_order "
        + "from lineitem "
        + "where l_shipdate <= '" + dateThreshold + "' "
        + "group by l_returnflag, l_linestatus "
        + "order by l_returnflag, l_linestatus").collectAsList();
```

Example - Where SQL not Supported

- Some SQL operations such as views, built-in functions and parsing inner queries in some cases not supported in Spark
- In such cases, DataFrames are used to implement the query (Below example from Query 7)

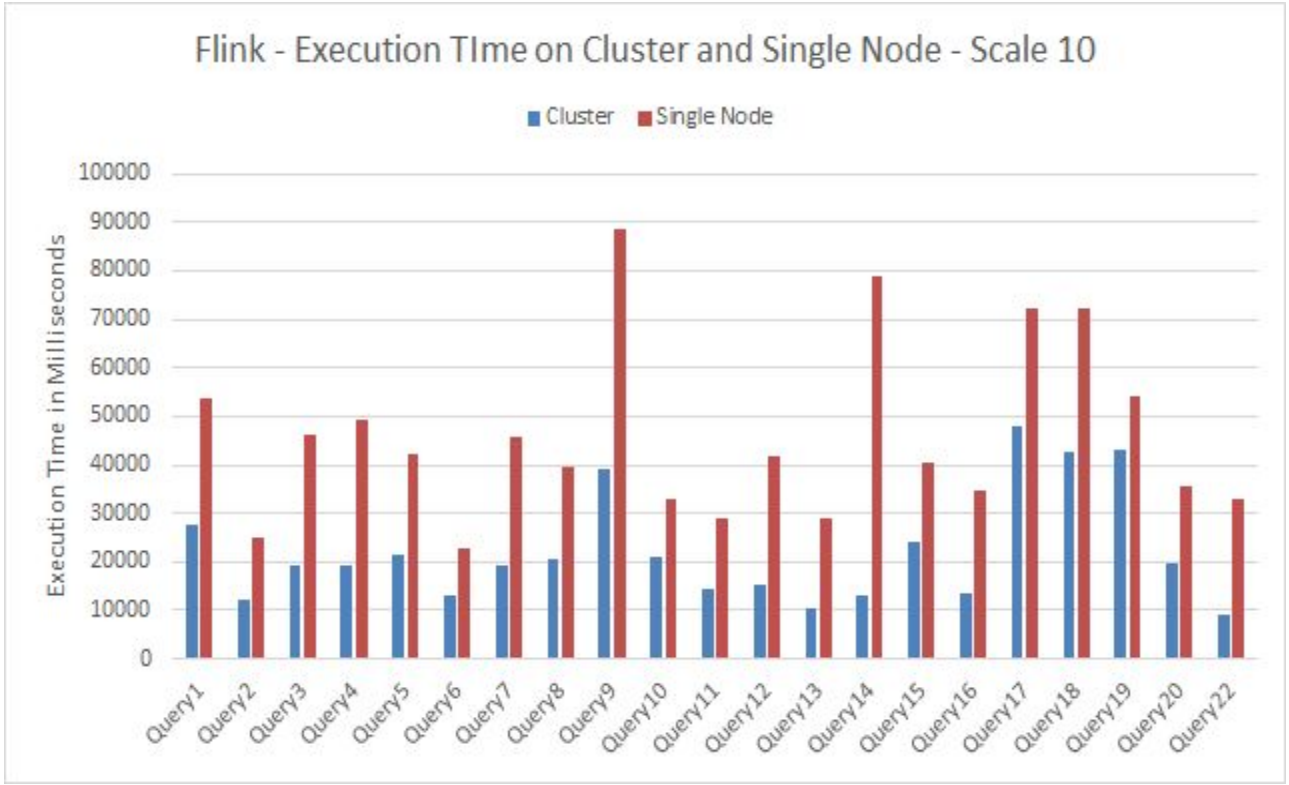
```
spark.udf().register("getYear", new UDF1<String,String>() {
    private static final long serialVersionUID = 7916629676707870056L;
    @Override
    public String call(final String date) {
        return date.substring(0, 4);
    }
}, DataTypes.StringType);

Dataset<Row> lineitem = spark.table("lineitem")
        .filter("l_shipdate >= date '1995-01-01'")
        .filter("l_shipdate <= date '1996-12-31'");

Dataset<Row> supplier = spark.table("supplier");
Dataset<Row> orders = spark.table("orders");
Dataset<Row> customer = spark.table("customer");
Dataset<Row> nation1Table = spark.table("nation")
        .toDF("n1_nationkey", "n1_name", "n1_regionkey", "n1_comment")
        .filter("n1_name = '" + nation1 + "'");
Dataset<Row> nation2Table = spark.table("nation")
        .toDF("n2_nationkey", "n2_name", "n2_regionkey", "n2_comment")
        .filter("n2_name = '" + nation2 + "'");

return supplier.join(nation1Table).where("s_nationkey = n1_nationkey").join(lineitem)
        .where("s_suppkey = l_suppkey").join(orders).where("o_orderkey = l_orderkey")
        .join(customer).where("c_custkey = o_custkey").join(nation2Table)
        .where("c_nationkey = n2_nationkey")
        .select(nation1Table.col("n1_name").as("supp_nation"),
                nation2Table.col("n2_name").as("cust_nation"),
                callUDF("getYear", lineitem.col("l_shipdate")).as("l_year"),
                callUDF("volume", lineitem.col("l_extendedprice"), lineitem.col("l_discount"))
                .as("volume"))
        .orderBy("supp_nation", "cust_nation", "l_year")
        .groupBy("supp_nation", "cust_nation", "l_year").sum("volume")
        .collectAsList();
```

Flink Scalability - Single Node vs Cluster

Speedup =
Single Node Execution Time/Cluster Execution Time

Query1	1.928704
Query2	2.010176
Query3	2.392159
Query4	2.5551
Query5	1.966603
Query6	1.76055
Query7	2.366001
Query8	1.91834
Query9	2.261517
Query10	1.582159
Query11	2.027584

Query12	2.716791
Query13	2.772829
Query14	6.011066
Query15	1.683007
Query16	2.577941
Query17	1.500306
Query18	1.685734
Query19	1.264395
Query20	1.801401
Query22	3.632736



Cluster Environment contained 6 Worker Nodes

Flink Version: 1.2.0

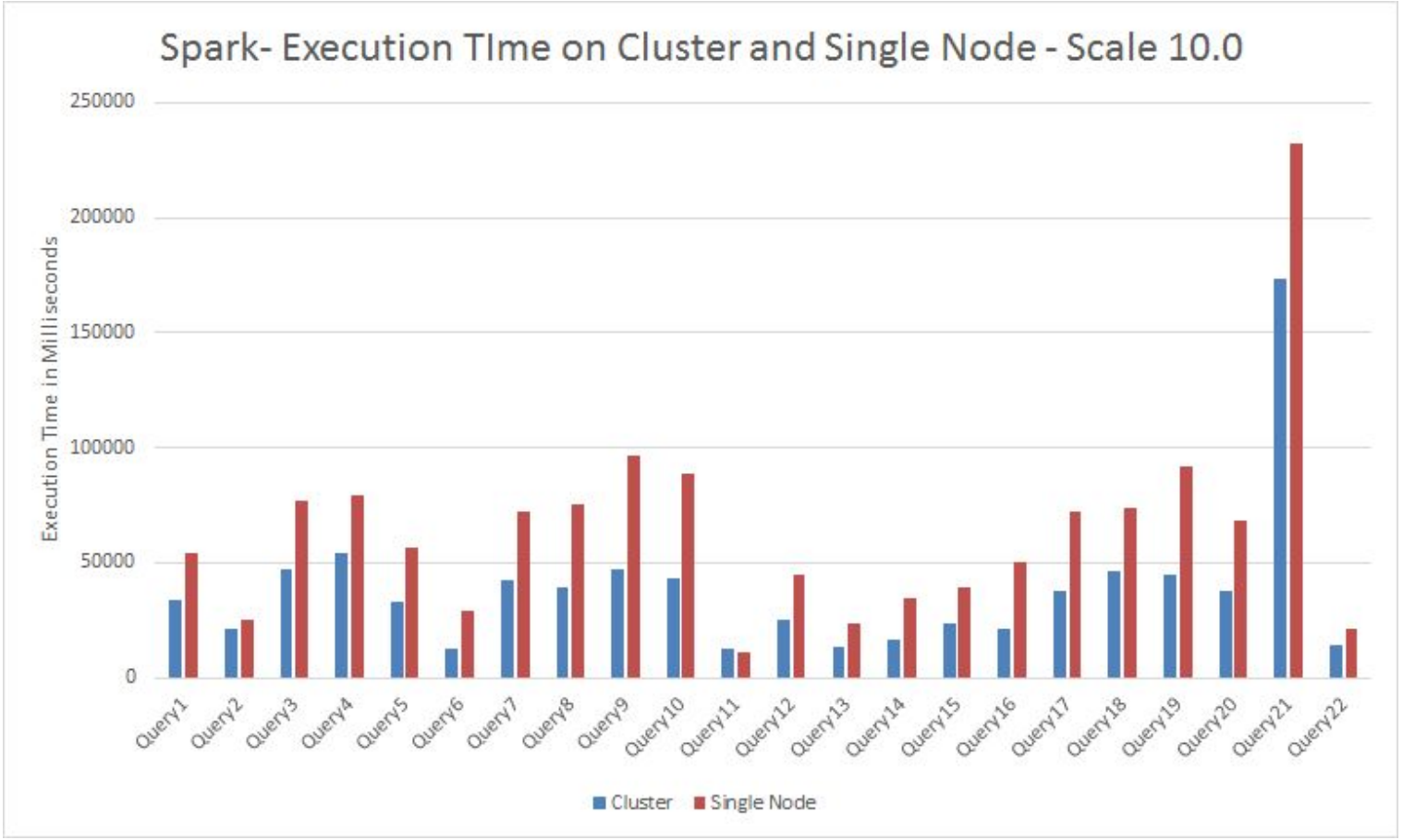
Average Speedup on cluster with 6 worker nodes 2.31

Spark Scalability - Single Node vs Cluster

Speedup =
Single Node Execution Time/Cluster Execution Time

Query1	1.622942	Query12	1.771221
Query2	1.209228	Query13	1.731222
Query3	1.632352	Query14	2.031795
Query4	1.465442	Query15	1.692472
Query5	1.725402	Query16	2.352842
Query6	2.246241	Query17	1.919377
Query7	1.694341	Query18	1.606503
Query8	1.932259	Query19	2.043679
Query9	2.053911	Query20	1.810036
Query10	2.054539	Query21	1.337525
Query11	0.863977	Query22	1.507296

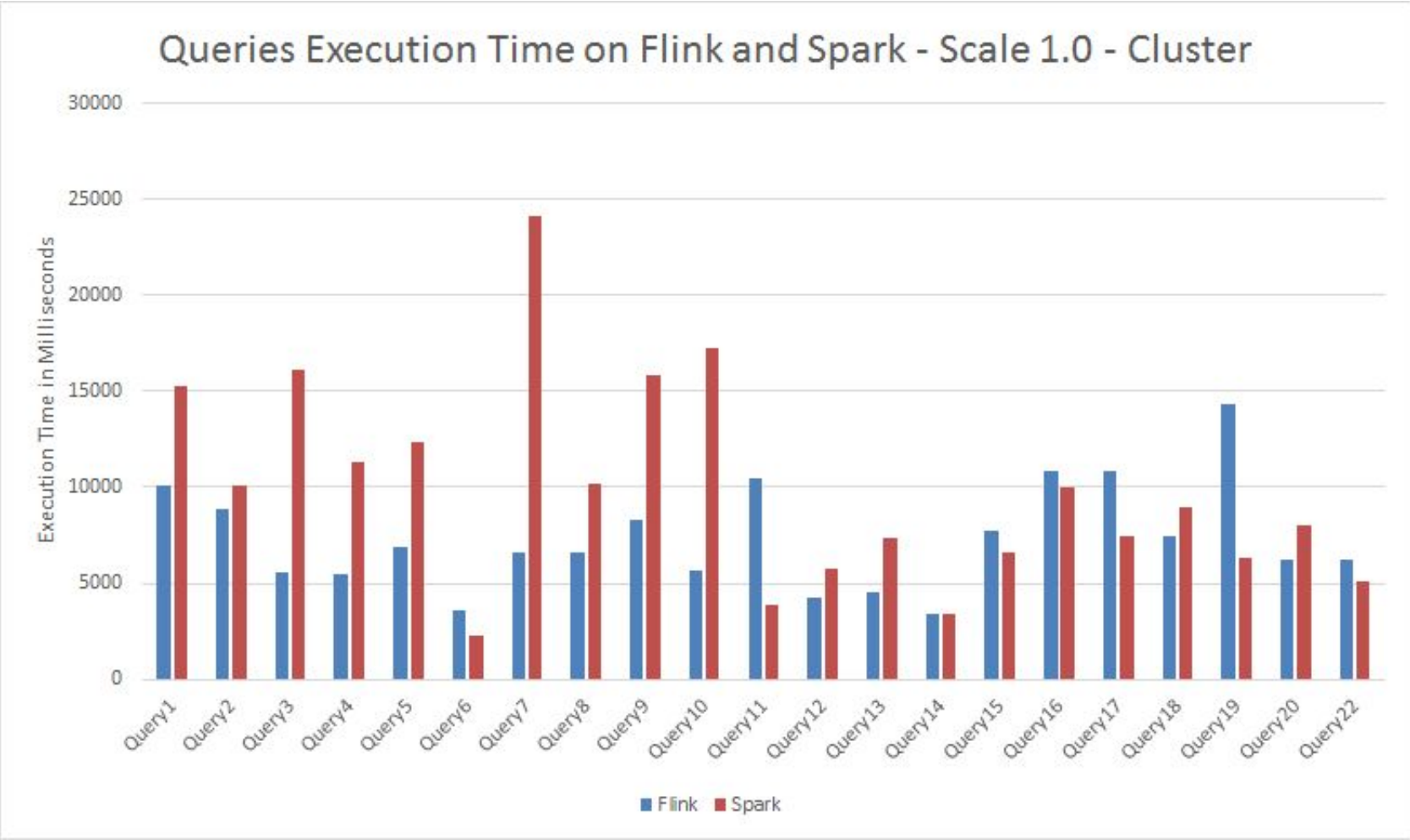
Cluster Environment contained 6 Worker Nodes



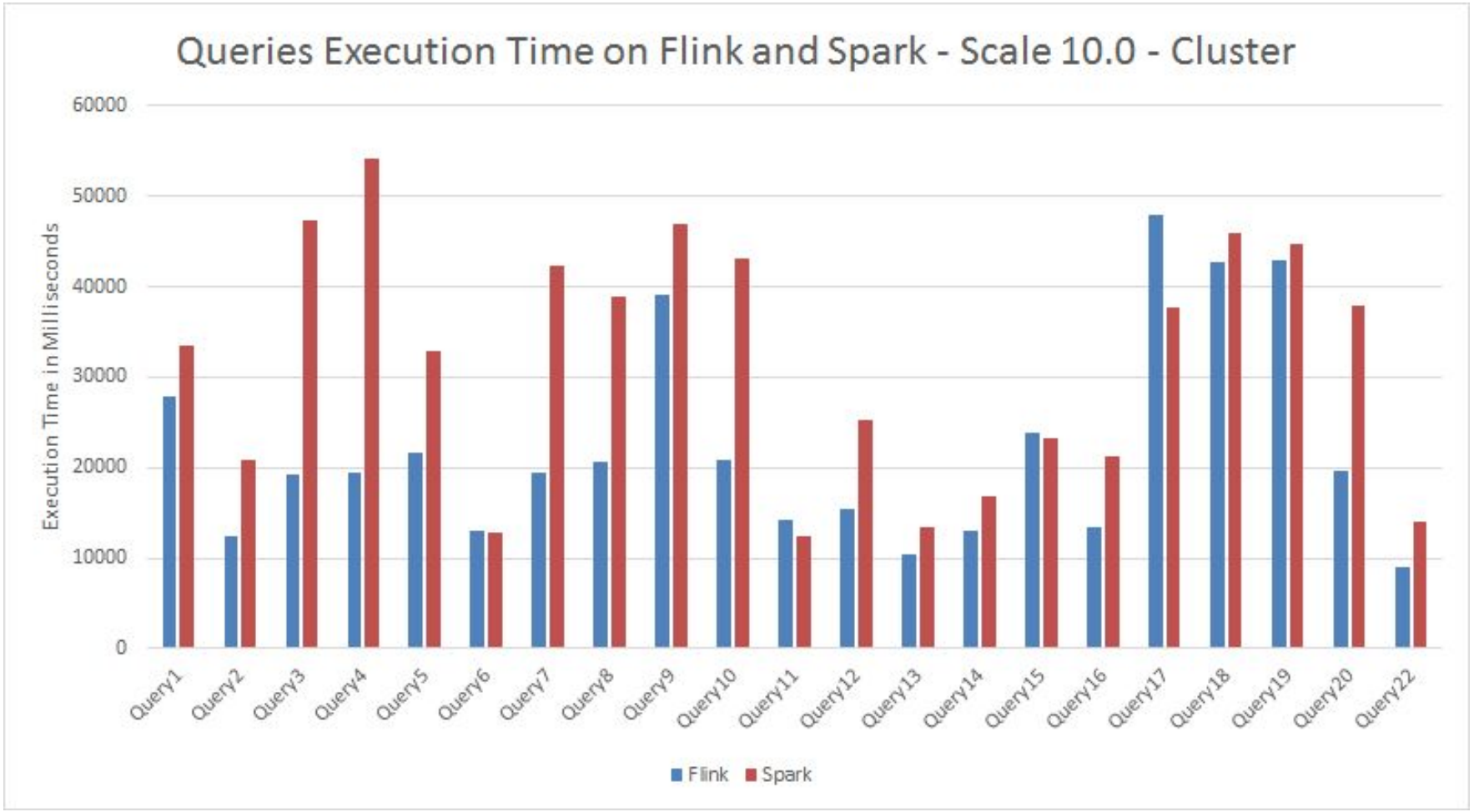
Spark Version: 2.1.0

Average Speedup on cluster with 6 worker nodes 1.74

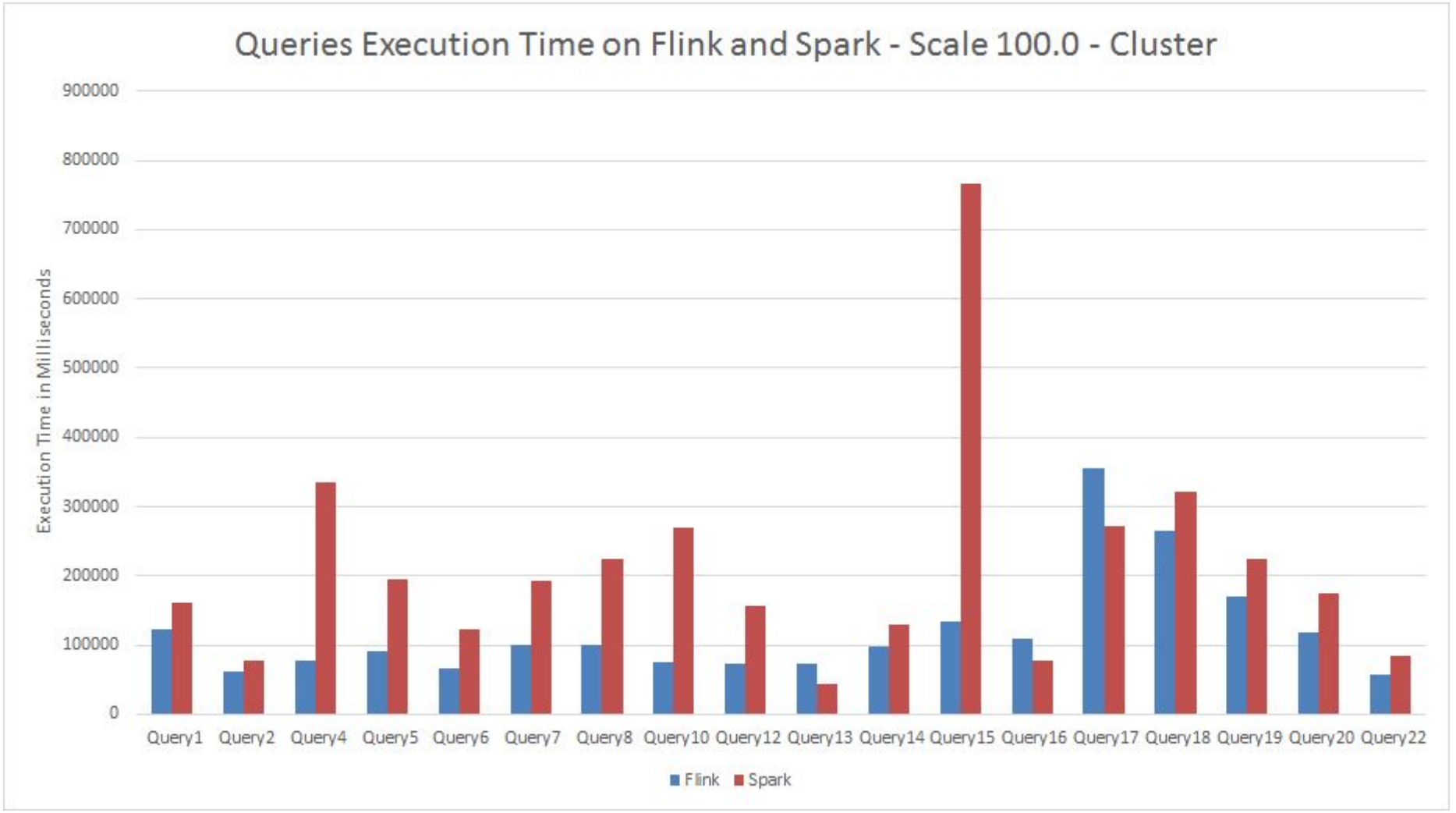
Comparison of Flink and Spark (Scale 1.0)



Comparison of Flink and Spark (scale 10.0)



Comparison of Flink and Spark (scale 100.0)



Results Interpretations

- Junit tests are used to validate Queries and compared against sample output
- Overall, Execution time of Queries is faster in Flink compared to Spark
- Queries where Flink's performance is slow
  - Query 13
    - SQL is supported in both (inner query with left outer join)
  - Query 16
    - Flink SQL does not support "*count(distinct ps_suppkey)*"
    - New table, using Table API *distinct* operator.
  - Query 17
    - SQL supported, outer and inner queries with aggregation
- Query 15, Execution Time is much higher in Spark
  - View is not supported in both Flink and Spark
  - Table and DataFrame are created and registered

Issues Faced - Flink

- Flink SQL fails to generate logical plan for some queries
  - Example: Query2, which has inner query
  - Supports *IN*, cannot create logical plan for *NOT IN* (Query16)
  - Inner queries in *WHERE* separated by logical *OR*
    - Executed separately and summed using *UNION* (Query19)
- "*Hash join exceeded maximum number of recursions, without reducing partitions enough to be memory resident. Probably cause: Too many duplicate keys*"
  - Query 21 at scale 1.0
  - Query 9 and Query 11 at scale 100.0
- For Scale Factor(100.0) - Query 3 failed(An open issue)
  - Fails to generate result and process hangs.
  - Worked with Scale Factor 1.0 and 10.0

Issues Faced - Spark

- Does not have strong support of built-in functions
  - For example extract year from a date, date between two dates
- Only 3 out of 22 queries failed to parse SQL statement
  - Does not support views (Query15)
  - Fails to support some complex Inner Queries

Part 2: ML Benchmarking

ML Algorithms in Flink

FlinkML

- provides scalable ML algorithms
- iterative processing of algorithms which are common in ML
- Still under development and aims to add many more algorithms in the roadmap.
- Currently supported ones

- Supervised Algorithms:
  - Support Vector Machines
  - Linear Regression
  - K-nearest neighbours join
- Recommendation
  - Alternating Least Squares(ALS)

ML Algorithms in Spark

MLLib

- Apache Spark's scalable machine learning library
- Usable in Java, Scala, Python, and R.
- excels at iterative computation
- supports wide range of ML algorithms for classification, regression, clustering, and collaborative filtering

❑ Supervised Algorithms:

- ❑ Support Vector Machines
- ❑ Linear Regression
- ❑ Decision Tree Classifier
- ❑ Naive Bayes
- ❑ Random Forests
- ❑ Multilayer Perceptron Classifier
- ❑ K- nearest neighbours join

❑ Recommendation

- ❑ Collaborative FIltering using ALS

❑ Unsupervised Algorithms:

- ❑ K-Means

Data Source

To perform evaluation of ML Algorithms, we will use publicly available datasets. Examples of common data sources include:

- UCI Machine Learning Repository
- Amazon AWS public datasets
- Kaggle
- KDnuggets

Benchmark Measurements for ML Algorithm

- Runtime of the algorithm is compared between Flink and Spark
- Accuracy of the algorithm is compared between Flink and Spark

Summary

So far, TPCH Benchmarking for Batch processing in Flink and Spark are completed

Plan for the future:

ML Algorithms in Flink and Spark to be tested on local(by August Mid)

Algorithms to be compared on Clusters(by August End)

Documentation to be done(work on it parallely)

References

- http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.2.pdf
- https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/index.html
- https://spark.apache.org/docs/latest/sql-programming-guide.html
- https://spark.apache.org/docs/latest/ml-guide.html