CS 838: Project Report
Group 3: Margaret Pearce and Subasree Venkatsubhramaniyen

Bootstrap-SQL Performance Evaluation

I. Problem Statement

The problem statement started off with an aim to evaluate the performance of approximation system namely BlinkDB on benchmark TPC-DS and TPC-H queries. However, given that BlinkDB has already been tested on benchmark queries, we found a relatively new extension to BlinkDB system called bootstrap-sql, which is integrated with Spark 1.4 and Hive.

Bootstrap-sql is an approximation system that runs an input query over multiple batches (configurable) to provide the users with multiple intermediate results of varying accuracy. Each batch gets to process its share of unique rows independent of other batches by selecting a sample of data (without replacement) and also aggregates the results with the previous batch, thereby improving the accuracy. The system eventually has processed the whole data set when the final batch of a multi-batch mode is reached. Batches are executed as Spark jobs and the aggregates across batches are computed over the in-memory RDDs of the corresponding previous batches.

The codebase for bootstrap-sql is a private repository and has not been widely distributed/ tested yet. Hence, the goal of the experiment is to understand the design and working of the system and evaluate its performance with TPC-DS queries on the following parameters:

- Response time for the first result set of the query as the system returns results in multiple batches
- Accuracy of the final result set
- Amount of storage I/O and Network I/O
- Types of queries which can be tested using the approximation system
- Understand the impact of various parameters that could be tuned and study the performance of the system
- Identify the use cases where the system could be useful
- Infer interesting results on the system behavior on different queries

II. System Setup

The codebase of bootstrap-sql is integrated with Spark 1.4 and Scala 2.10 with a custom version of Hive including Hive Online. We decided to setup the system on our CloudLab VMs, which are preconfigured with HDFS 2.6, Spark and Hive.

The setup steps are stated below:

- Clone private repository of the codebase from github in master and slave machines
- Repoint the environment variable SPARK_HOME to the spark folder in the cloned repository
- Navigate to the cloned folder and invoke the shell script “make-distribution.sh” from the master VM with the following arguments:
  - -DskipTests
  - -Pyarn
  - -Phadoop-2.4 -Dhadoop.version=2.4.0
  - -Phive
  - -Phive.0.12.0
  - -Phive-thriftserver

- Create a new script for sourcing environment variables different from existing run.sh script to reflect changes in spark variables and start up commands
- We decided to use the existing TPC-DS database of size 50GB

We started with the public repository of the approximation system called BlinkDB as it had better documentation. However, upon exploration, we figured out that the system lacked built-in capabilities for creating offline samples of data and required a separate optimization piece from the user to accomplish this. We moved on to the private repository of the codebase, which is less documented. The private repository was significantly different from the public repository, making it difficult to connect the two versions. We figured out that the working of the system is not in line with the BlinkDB paper discussed in class. However, we could trace it to an extended version of BlinkDB called bootstrap-sql and noted that the system works in an altogether different fashion.

III. Implementation

Testing Script

To evaluate the bootstrap-sql project, we modified an existing test harness ("SuiteHarness838.scala"). We revised the code to facilitate executing queries under the settings we were interested in exploring. Specifically, we added support for the following command line arguments:

- Testing mode: 0 for no online mode (regular Hive execution), 1 for online mode with a single batch, 2 for online mode with multiple batches
- Query number: The TPC-DS query to run. Note that each query was modified for compatibility with Hive Online mode.
- Number of batches: The number of batches to split the query into. Each of the *b* batch processes roughly *n/b* of the total input data of size *n*.
- The comma-separated name(s) of the tables to sample
- A suffix for output file names for organizational purposes

Based on the mode specified, the test harness will set appropriate properties in the Hive context. It will then run the query under the indicated settings. As the query progresses (in a multi-batch mode), output from each batch will be displayed.

When running the script, we typically piped the output to a text file as a log. This file contains the results for each batch, including the time it took for each batch to execute and return results.

TPC-DS Queries

In addition to our testing harness, we found that we needed to modify the standard TPC-DS queries we selected for our experiments to adhere to limitations of Hive Online. We removed the minimum number of lines in each query that resulted in a query accepted by Hive Online. In particular, we needed to remove instances of "PARTITION BY", "ORDER BY", "UNION ALL" and "LIMIT" from each query. Hive Online limitations are discussed in Section VI.

Utilities

Lastly, we wrote a few utility scripts to help us collect data for each experiment from the Spark UI by parsing the HTML data. We used this to easily generate graphs and compute metrics.

IV. Experiments

The test harness we wrote takes as parameters the type of batch (baseline, single batch online and multi-batch online), the number of batches in case of multi-batch and the tables to be sampled in case of online run.

Baseline (Control Setting)
The baseline mode of execution describes the situation of the query run without any specific settings, just like any other Hive query. This result is important for us to compare the results of experiments with specific settings.

Selected Queries
With given query limitations with Online Hive, we picked a handful of TPC-DS queries, namely Query 12, Query 50 and Query 85. These queries cover a variety of scenarios that were interesting to us:

- Query 12: This query involves a single large fact table, two small dimension tables, and does one aggregation over a single column. The query also includes date range filters and limits results to specific categories. This is a simple-case query that covers join and aggregations.
- Query 50: This query computes several aggregate sums based on different cases. It also uses two large fact tables and three small dimension tables. We were interested in exploring what the consequences would be of only sampling one of the two fact tables involved in this query.
- Query 85: Similarly, this query involves two large fact tables and five smaller dimension tables. However, this query has very complex join conditions involving logical operations. We were interested in seeing the effects of the complicated join coupled with only sampling a single fact table on the online query.

Initially we also planned on evaluating queries 21 and 71. However, we were unable to run query 71 due to limitations with Hive Online. Query 71 used the “UNION ALL” operator on a sub-query, which is not accepted by Hive Online. We could not find an equivalent way to rewrite the query, so we decided to remove it from our test cases. We were interested in testing query 21 specifically to see how it would perform when sampling multiple tables since this query uses two fact tables. However, support for sampling multiple tables appears incomplete in Hive Online. Without this capability, there was nothing unique that we wanted to explore with query 21, so we opted to remove it from our test cases.

Parameter Settings (Variable Settings)
For each of the above queries, we wanted to test the system by varying the number of batches (1, 5, 10 ,15, 20, 50, 100) to study how performance varies as the number of batches is increased. We also vary the number and choice of tables to be sampled to study the system behavior in terms of accuracy of the results and the response time. In particular, we wanted to observe the effect of sampling only the fact tables and tables which contribute to the columns being aggregated in the query.

Spark Settings
We used the same Spark settings for each test case we ran:

- Spark master: Private IP of our master VM, port 7077 (10.0.1.31:7077)
- Driver memory: 1 GB
- Event log enabled: true
- Event log directory: /home/ubuntu/storage/logs
- Executor memory: 21 GB
- Executor cores: 4
- Task CPUs: 1

V. Results

Query Time

We measured query completion time for all queries under each batch setting. As the number of batches increased, total query time (from submitting the query to getting the last results) increased nearly linearly.

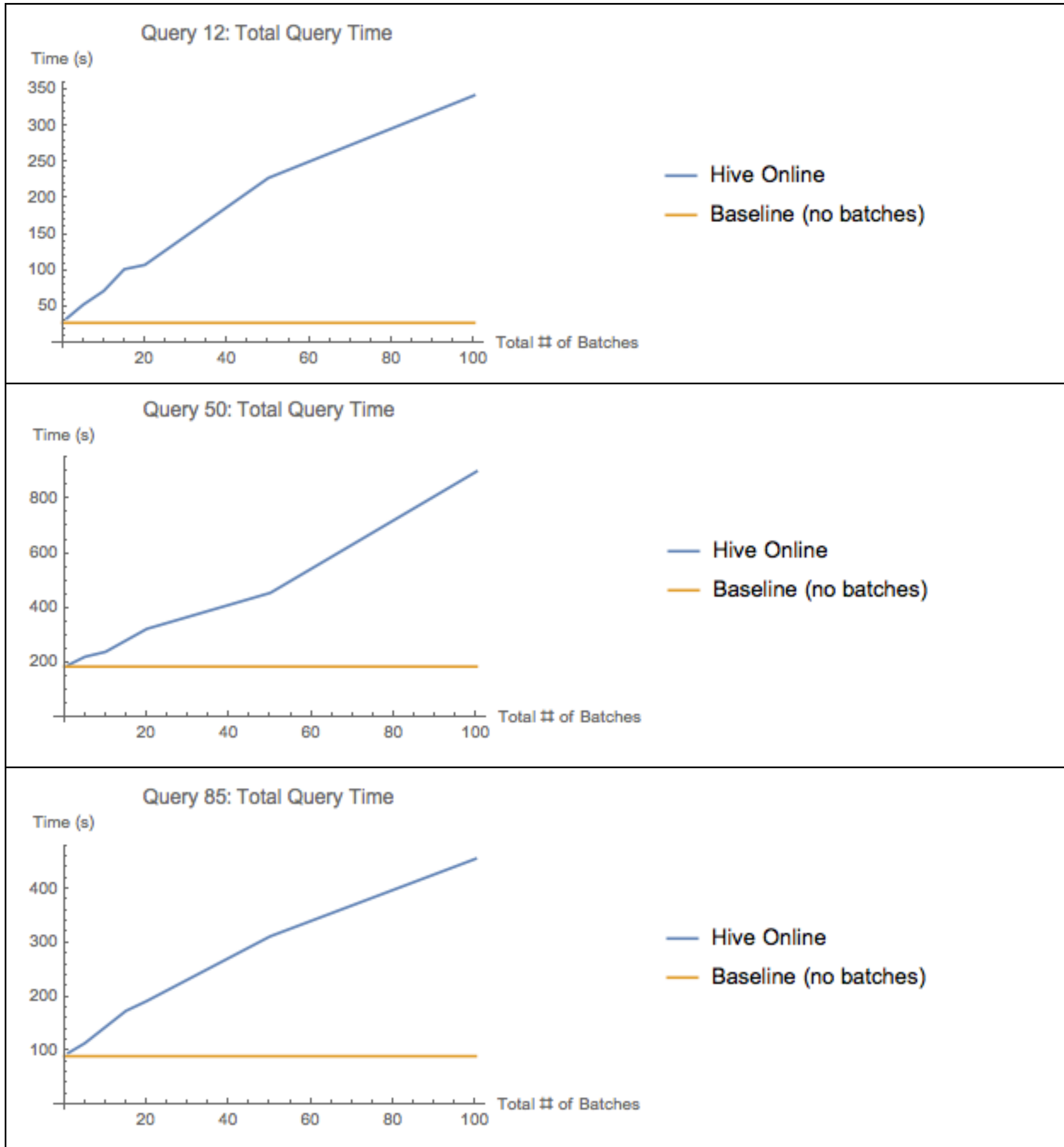
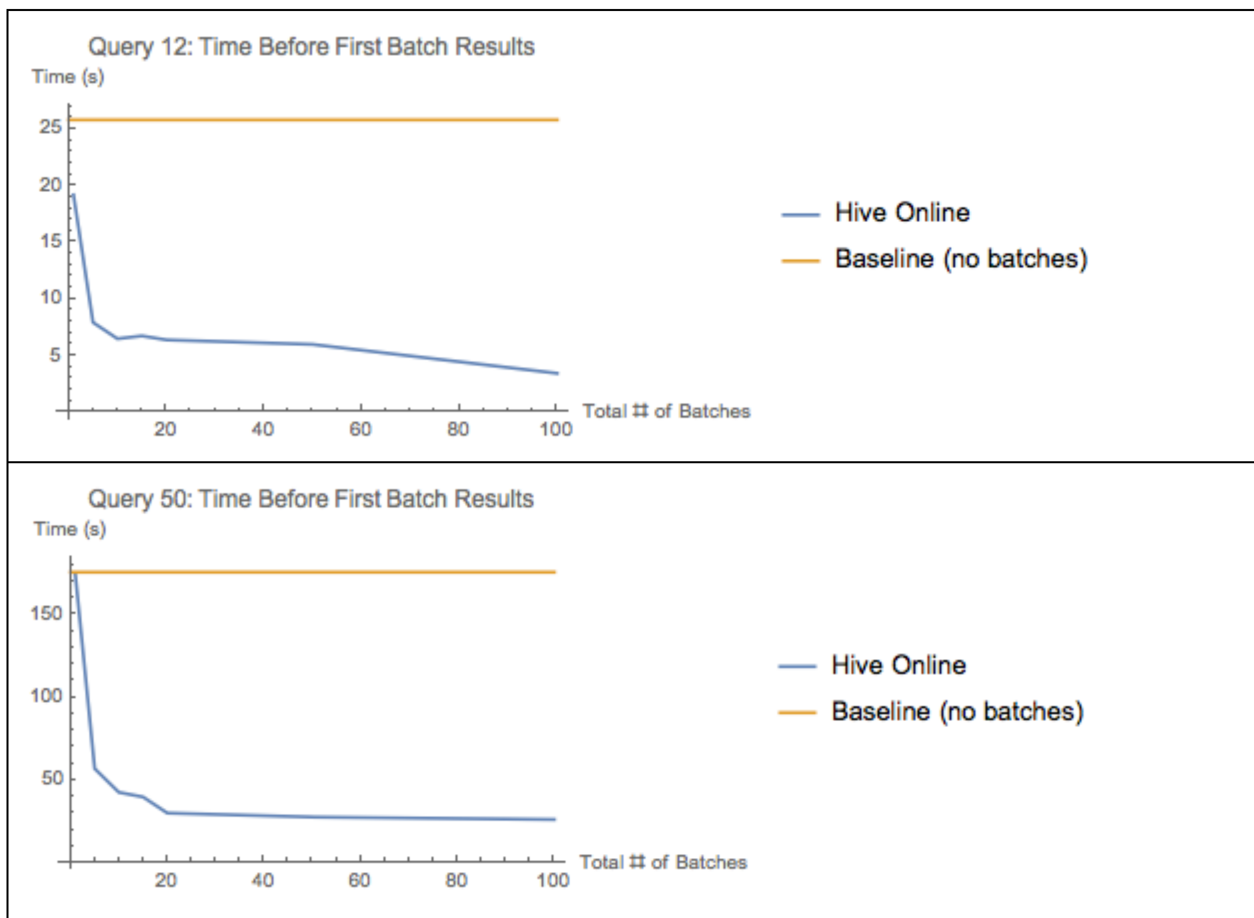


Figure 1: Total query time vs. number of batches

Batch setting	Query 12: Total query time (s)	Query 50: Total query time (s)	Query 85: Total query time (s)
Baseline	28	186	90
1	34	192	96
5	53	222	114
10	72	240	144
15	102	282	174
20	108	324	192
50	228	456	312
100	342	900	456

Table 1: Total query time by number of batches

By contrast, as the number of batches increases, the time before the *first batch* returns its results decreases. This is an intuitive finding based on the realization that each batch processes a roughly equivalent amount of data, and the total amount of data processed by a Hive Online query is the same amount of data processed in a non-approximate setting. Therefore, the first batch of a multi-batch query has monotonically decreasing amounts of data to perform calculations on as the number of batches increases, and it is able to compute results quicker. The time for the first result set appears inversely proportional to the number of batches.



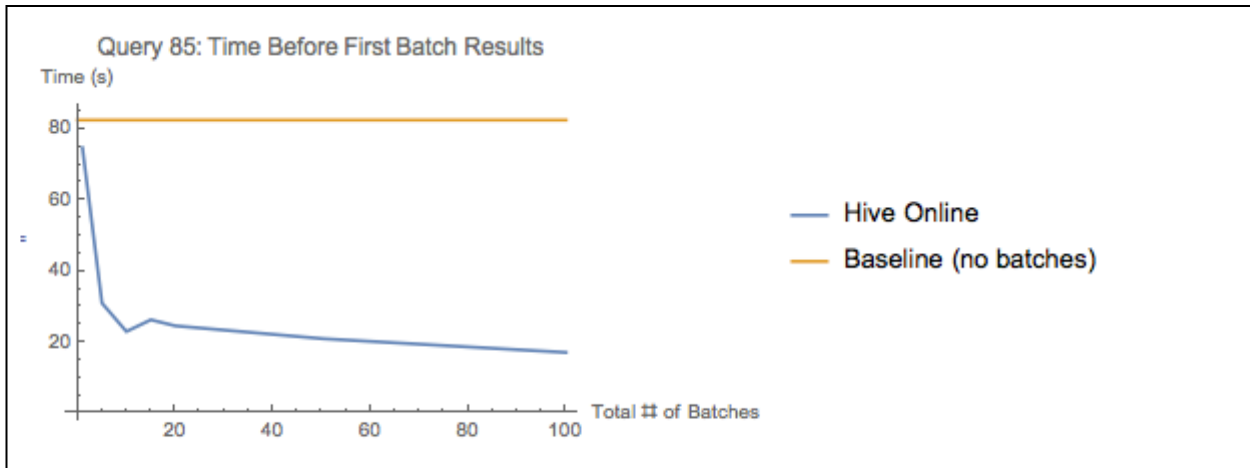


Figure 2: Time before first batch results vs. the number of batches setting for the query

Batch setting	Query 12: Time before first batch (s)	Query 50: Time before first batch (s)	Query 85: Time before first batch (s)
Baseline	25.8	175.5	82.6
1	19.2	174.1	74.8
5	7.9	57	31
10	6.5	42.7	23
15	6.8	39.9	26.3
20	6.4	30.2	24.6
50	6	27.8	21
100	3.4	26.3	17.1

Table 2: Time before first batch results across batch settings

Looking strictly at the total time results, it is unclear why increasing the number of batches results in a linear increase in the total query time. We looked at query 12 in more detail to help explain this observation. Query 12 is the simplest case of the queries that we explored: it includes a single aggregate column from a fact table and a handful of join conditions over two dimension tables.

When query 12 is executed in multi-batch mode, each batch is submitted as a separate Spark job. The job for each of the *b* batches consists of three stages. At the DAG level, we observed a pattern among all batches: the first stage gets the sample for this batch and performs Hive Online-specific optimizations. The next two stages appear to involve aggregating the results for the current sample with the results from previous batches.

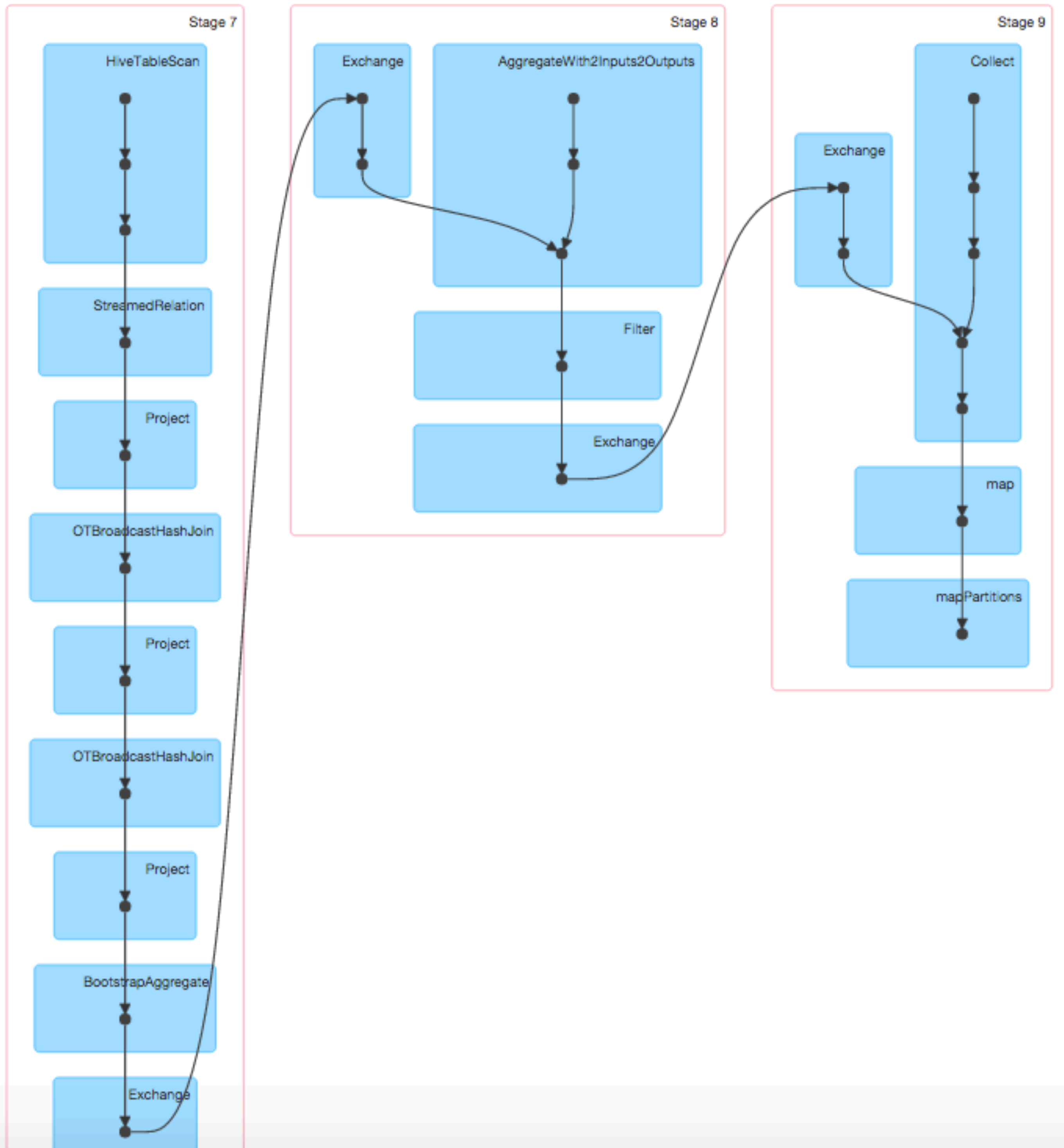


Figure 3: Query 12: DAG for a single job (corresponding to a single batch)

The graphs below show the total time for each batch within a multi-batch query broken down by the time for each stage. We compared setting the number of batches *b* to 5 (low end) and 100 (high end). In both cases, the majority of the time tends to be spent in Stage 1. Therefore, we conclude that sampling adds some overhead for each batch. As the number of batches increases, this overhead adds up and increases the overall query time. Even though Stages 2 and 3 take less time as the number of batches increases because these stages process smaller amounts of

data, the gains don't make up for the increased time due to the sampling overhead in Stage 1. Furthermore, batches are not processed in parallel: batch *b* cannot start until batch *b-1* completes.

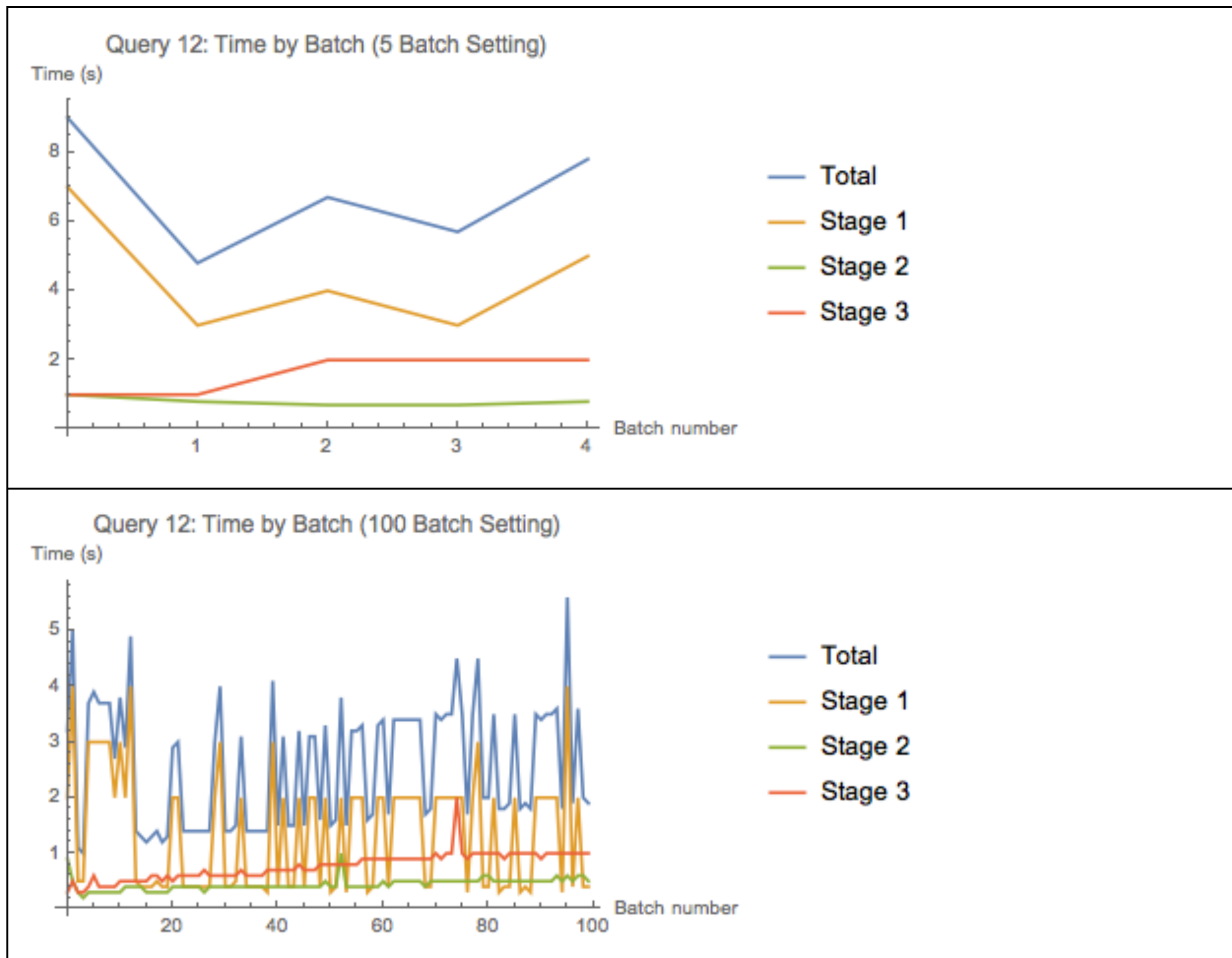


Figure 4: Time per batch for query 12, 5 batch setting (top) and 100 batch setting (bottom). Tabular data available in CSVs submitted with this report.

Storage Read

We were interested in measuring storage read for each query/ setting as a means to help explain how sampling behaves. We observed the following behavior:

- Across all number of batch settings, the total amount of data read from storage remains fairly constant.
- Within each batch of a multi-batch query, all storage read occurs in Stage 1, reaffirming our hypothesis that Stage 1 is where sampling takes place.
- Looking only at storage read showed us that the same amount of data is processed regardless of what batch setting is used.

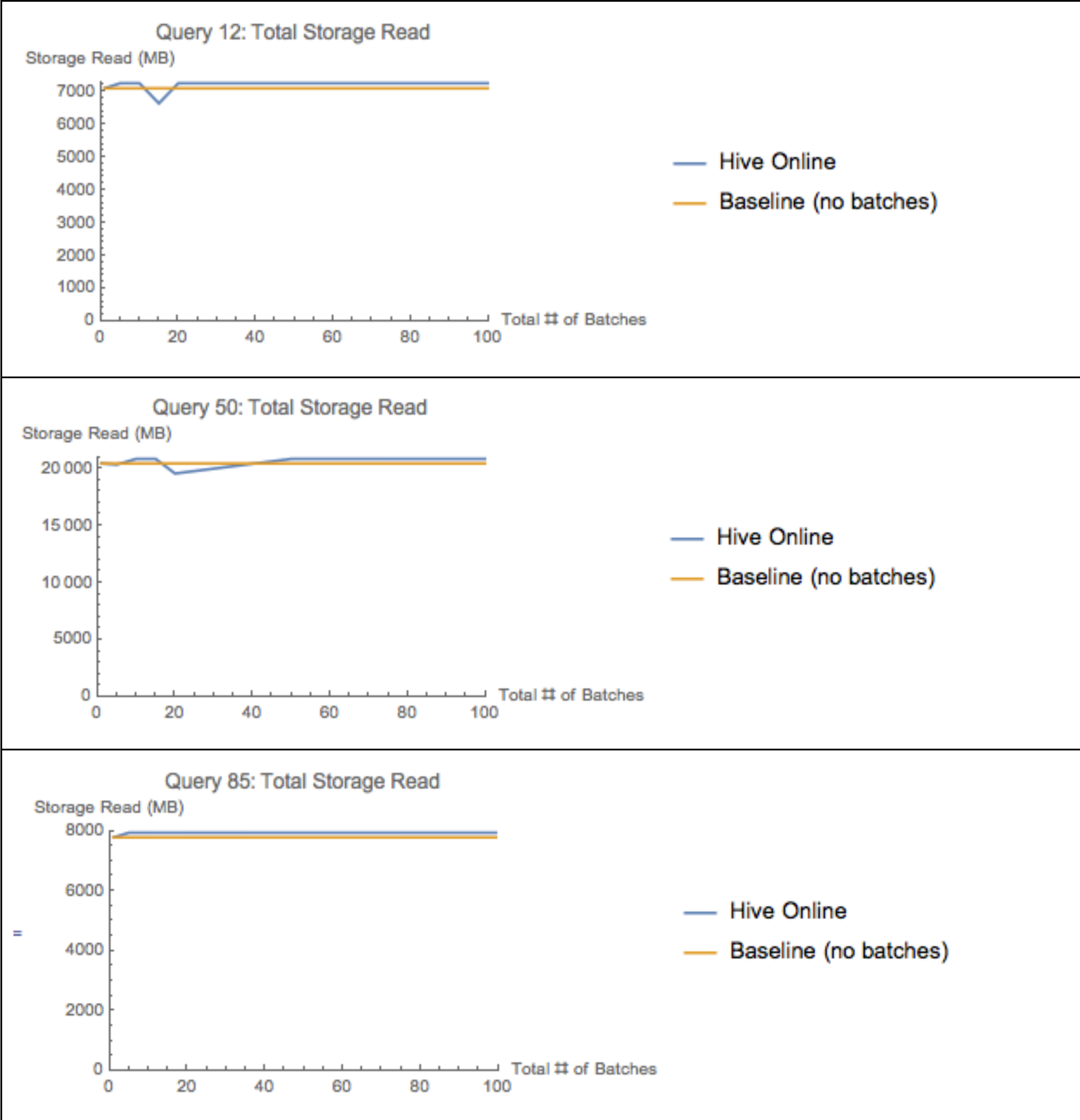


Figure 5: Storage read by batch level setting

Batch setting	Query 12: Total storage read (MB)	Query 50: Total storage read (MB)	Query 85: Total storage read (MB)
Baseline	7100	20437.9	7786.8
1	7100	20437.9	7786.8
5	7254.8	20337.9	7941.6
10	7254.6	20841.1	7941.6
15	6641.4	20841.1	7941.4
20	7254.7	19561.3	7941.6
50	7254.7	20841.1	7941.4
100	7254.5	20841.1	7941.2

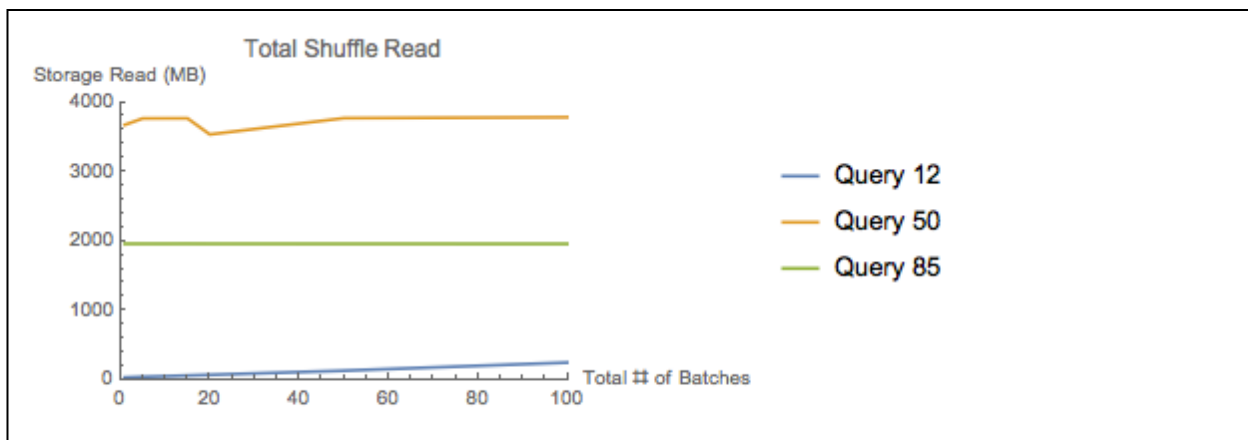
Table 3: Total storage read for each query across batch settings

Accuracy

We compared the results from multi-batch queries with results we obtained from the baseline queries. In each query, we discovered that the multi-batch setting converged to the same values calculated with the baseline queries. This had an important implication for sampling: multi-batch queries process the exact data set as the baseline queries. Each row is sampled exactly once (sampling without replacement), and therefore by the end of the final batch, the full data set has been processed. We would expect to see differences in one or more values if sampling with replacement was used due to the possibility that a single row could be read more than once or not at all.

Shuffle Read and Write

Finally, we measured the total amount of data read and written in shuffle phase for all queries/ settings. We found that within stages of a single batch, data is shuffled to compute the aggregate for the current sample. As the number of batches increases, each batch is merging its results with the output from the previous batch. We see small fluctuations in the total shuffle read/ write size, but overall the amount of data read/written in the shuffle phase appears constant across batch settings. This is consistent with our observation that the same amount of data is read by the query overall (see storage read results).



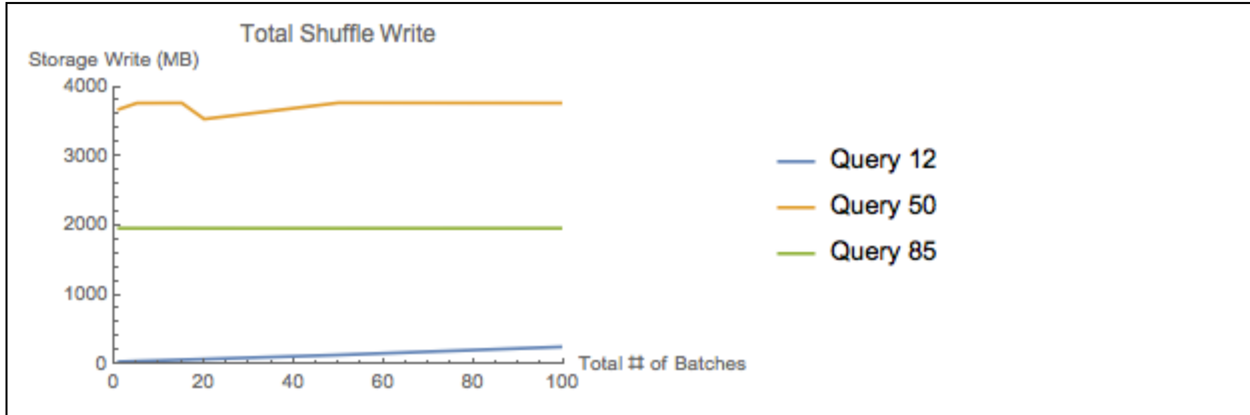


Figure 6: Shuffle read and write by query/ batch setting

Batch setting	Query 12: Total shuffle read (MB)	Query 50: Total shuffle read (MB)	Query 85: Total shuffle read (MB)
Baseline	10.8	2651.1	1698
1	39.1	3675.5	1963.4
5	48.3	3767.7	1963.4
10	57	3768.7	1963.3
15	67.4	3769.6	1963.8
20	77	3537.5	1963.1
50	137.1	3772.2	1962.6
100	254.7	3785.4	1962.6

Table 4: Total shuffle read for each query across batch settings

Batch setting	Query 12: Total shuffle write (MB)	Query 50: Total shuffle write (MB)	Query 85: Total shuffle write (MB)
Baseline	10.8	2654.5	1697.9
1	39.2	3678.9	1963.4
5	48.3	3767.7	1963.4
10	57	3768.7	1963.3
15	67.4	3769.7	1963.8
20	77	3537.5	1963.2
50	137.4	3772.4	1962.7
100	254.7	3786	1963.5

Table 5: Total shuffle write for each query across batch settings

VI. Limitations and Future Work

From the above experiment results, we found that the approximation system applies the given operator (query) on incremental datasets (smaller subset) so as to produce meaningful intermediate results to the input query. The system increases the accuracy of the results across iterations/batches by aggregating data across batches.

Limitations

We identified several limitations of bootstrap-sql throughout our experiments:

- There is no support for the following query operators: LIMIT, PARTITION BY, ORDER BY, UNION ALL. This list is not exhaustive, but represents only the unsupported operators that we encountered.
- The implementation for sampling multiple tables for a given query is incomplete.

- We are unaware of a method to automatically return the batch-level accuracy in a multi-batch setting. This would be particularly important when determining an appropriate number of batches to ensure that the accuracy of the current batch is acceptable for a given application.

Future Work

We faced a lot of challenges in the installation of the system as there was little or no documentation in the codebase. We observed that the system creates samples of data on the fly as the query is executed and hence offline sampling is not implemented in bootstrap-sql. Also, we noted that there is a HiveTableScan operation in each batch of the query that contributes to generating a sample for the batch, adding to the overall response time.

There is an inherent tradeoff between computation time and storage costs for different methods of sampling. Offline sampling systems such as BlinkDB use more storage because they maintain samples that are generated offline and persist them across the cluster. By contrast, bootstrap-sql creates samples at runtime by assuming that data is uniformly distributed across cluster nodes and processes data in small batches by reading subsets at a time, increasing query time. Hence, given the tradeoff between offline sampling and online sampling in terms of storage cost and query execution time, a sampling system that optimizes for both scenarios could be useful to strike a balance.

As the system takes queries with no choice on the accuracy or latency (unlike BlinkDB), it would be useful if the system could display accuracy and confidence level of the result set in each batch (given that the accuracy improves across batches). The current system accepts the number of batches and name of tables to be sampled as parameters, and hence the system design would be even better if this is abstracted and that the system could intelligently pick choices for batches and tables. We noted that the batches are executed in a sequential fashion aimed to increase the accuracy of the results eventually. However, there could be some parallelism in the batch execution mode to speed up total query time.

VII. Appendix

The following table lists the files included in a tarball submitted with this report.

File name	Description
SuiteHarness838.scala	Testing harness (see Section III)
query12_project.sql	TPC-DS query 12 with modifications for Hive Online compatibility
query50_project.sql	TPC-DS query 50 with modifications for Hive Online compatibility
query85_project.sql	TPC-DS query 85 with modifications for Hive Online compatibility
ScenarioMetricsGeneric.py	A python script to extract Spark UI data for each experiment setting (time, input, shuffle)
ScenarioMetrics.py	A python script to extract additional stage-level data specifically for query 12.
MathematicaPoints.py	A script to generate appropriate lists of data from a CSV that can easily be copied to Mathematica for graphing purposes
Query12Graphs.nb	Mathematica code for query 12 graphs
Query50Graphs.nb	Mathematica code for query 50 graphs
Query85Graphs.nb	Mathematica code for query 85 graphs

***.csv**	Results sets from ScenarioMetrics*.py (for each experiment setting)
***_log.txt**	Piped output from running queries under a variety of settings (see Section IV)

Table 6: Supplemental files