Distributed Implementation of BG Benchmark Validation Phase

Dimitrios Stripelis \( ^\uparrow \), Sachin Raja \( ^\uparrow \)
\( ^\uparrow \{\text{stripeli, raja}\}@usc.edu \)

1. BG BENCHMARK OVERVIEW
BG is a state full benchmark used to evaluate the performance of a data store for interactive social networking actions and sessions. These actions and sessions either read or update a very small amount of the entire data set. BG’s main functionality focuses on the computation of the highest possible throughput, i.e. the Social Action Rating (SoAR), of the evaluating data store and the data store’s Socialites Rating based on a service level agreement (SLA). The SLA parameter needs to be bounded by the developer according to the desired response time and the freshness interval \([1]\). The Core operations currently supported by the BG Benchmark are the following \([2]\):

- View Profile
- List Friends
- View Friends Request
- Invite Friend
- Accept Friend
- Reject Friend
- Thaw FriendShip

2. PROJECT DESCRIPTION
One of the major contributions of the BG Benchmark is the computation of the total amount of unpredictable data that the data store under experimentation can produce. Unpredictable data can be either stale, inconsistent or simply invalid data. For instance, the design of a Cache Augmented SQL System \([3]\) may result in dirty reads \([4]\), or even suffer from race conditions that can leave the cache and the database in an inconsistent state \([5]\). BG handles this computational scenario through the built-in Validation Phase operation.

However, the bottleneck of the current implementation of the Validation Phase relates to the total execution time that BG requires in order to complete the operation. The work we present in this report is centered on approaches which can improve the computation time of the Validation Phase and result in a more scalable performance of the BG Benchmark Framework. Following, we describe the validation phase of the BG Benchmark and our experimental solutions on how to efficiently tackle this limitation.

3. BG VALIDATION PHASE
BG Validation Phase is an offline procedure that processes log records produced during the benchmarking phase to quantify the amount of stale data generated by the data store. When benchmarking a data store, BG generates log records for every read and write action referencing a specific data-item inside the database upon which the corresponding operation was executed. Every log record contains the start and end timestamp of the action’s lifespan along with other state instructions such as the observed or changed value of the referenced data-item. For each read action, the validation phase identifies the overlapping concurrent write actions, enumerates all the possible serial schedules and computes the range of acceptable values for that particular read
action. If the value observed by the read action is not in the range of acceptable values, then the data store has produced stale data.

4. BG CURRENT APPROACH
The Validation Phase currently supported by the BG Benchmark stages all the log records of the write actions in a main memory interval tree [6]. Even though this approach is very well suited when processing a small number of log records, it has proven to be inefficient when dealing with much larger volumes. The rational behind this claim is that once the size of the log records exhausts the available physical memory of the BG Client, the operating system will exhibit a thrashing behaviour and thus resulting in a long-lived validation process.

5. RECOMMENDED APPROACH
The proposed implementation of the Validation Phase is based on a distributed computation of the stale data by leveraging the MapReduce [7] programming model through the Apache Spark [8] framework. In order the proposed technique to be extendible and compatible (JavaVM) with the existing source code of the BG Benchmark and leverage the full capabilities of the Spark engine we deployed the suggested technique in the Scala programming language [9].

5.1 MOTIVATION
Given that large sizes of log records cannot fit entirely inside memory in order to be processed sequentially we needed a different approach that could distribute the offline computation in a more performant manner. Based on the characteristics of the read and the update log records we identified key-value correlations which can be distributed across multiple machines and perform the computation locally in each machine without overloading the available memory.

5.2 TECHNIQUE
The idea behind our solution is to partition the log records into chunks and process them in parallel instead of importing all of them into the memory at once. In principle, we achieve to validate each partition of the log records independently of the rest and hence drastically reduce the overall execution time of the existing BG implementation. There are three possible partitioning techniques that can be applied based on the attributes of the log records:

- Partition by data-item
- Partition by time
- Hybrid partition by data-item and time.

In this report we present the implementation and evaluation of the partitioning by data-item technique, as described below:

1. Partition both the write and read log records by creating a key for each partition which is the combination of the referenced data-item identifier and the applied BG operation (e.g. Accept Friend, Reject Friend) on that particular item. The corresponding value for
each partition is the value presented in the log record for the respective read/write action.

2. Group (i.e. groupBy or reduceBy) the partitions based on their associated keys.

3. For each partition:
   A. Read the database initial state
   B. Stage all write records referencing the data-item and update the database state.
   C. Create an interval tree for the partition.
   D. Store each write and related state inside the tree using the start and end timestamps.
   E. Validate all the read records referencing that data-item.

**Complete Validation Algorithm**

The validation algorithm evaluates each read record in turn and relies on two essential components. The value of the read operation and the state of the database at the time of the operation. If the read-value does not match with the database state, then the data store has generated stale data.

For each read operation inside the partitions we create a range of acceptable values, i.e. \(DataRange\), in order to evaluate the correctness of the read value. Following we present the actual test cases we consider for counting the total amount of stale data:

a. There are no updates for the specific partition:
   a.1 \(DataRange = \{\text{database initial state}\}\)
   a.2 continue to step f

b. The read operation is after all the write operations (Fig.1). Then we take the last completed write operation (highest end time) and we check against any overlapping intervals inside the interval tree.
   b.1 \(DataRange = \{\text{values of the overlapping intervals}\}\)
   b.2 continue to step f

c. Check If the read operation has started before the very first write operation of the interval tree (Fig. 2). Start time of read operation is lower than the start time of the first update. Extend the DataRange to contain the initial state:
c.1 \( DataRange = DataRange + \{\text{database initial state}\} \)

c.2 continue to step d

d. Check if the read operation overlaps (Fig.3) with intervals inside the interval tree.

d.1 \( DataRange = \{\text{values of the overlapping intervals}\} \)

d.2 continue to step e

e. Find the last completed write operation before the read operation. In essence, the write with the closest end time to the start time of the read. If such an interval exists, then use it to find the overlapping intervals inside the interval tree and extend the DataRange:

e.1 \( DataRange = DataRange + \{\text{values of the overlapping intervals}\} \)

e.2 continue to step f

Else if no overlapping interval exists (Fig.4) then we extend the DataRange to contain the initial state of the database:

e.3 \( DataRange = DataRange + \{\text{database initial state}\} \)

e.4 continue to step f

f. Evaluate the read operation’s value against the \( DataRange \). If it does not fall inside the range, then increase stale counter by 1.

For counting, i.e. stale counter, the overall stale data that occurred during the evaluation of the underlying database in parallel, we utilize the concept of Spark Accumulators. These are shared-variables that can only be used as add-ons through an associative operation and can therefore be efficiently supported in parallelized operations.

1. \( DR = {} \)
2. if (no updates)
3.   \( DR = \{\text{initial state}\} \)
4. For each read do:
5.   [validate read value with DR]
6.   end_validation for this partition
7. else
8. create interval tree
9. For each read do:
10. if (read.low >= latest_update.high)
11.   \( DR = \{\text{values of latest_update overlapping intervals}\} \)
12.   go to 21
13. if (read.low <= first_update.low)
14.   \( DR += \{\text{initial_state}\} \)
15. \( DR += \{\text{values of read overlapping intervals}\} \)
16. closest_update = {find closest update to read_operation}
17. if (no closest_update)
18.   \( DR += \{\text{initial_state}\} \)
19. else
20.   \( DR += \{\text{values of closest_update overlapping intervals}\} \)
21. [validate read value with DR]
6. EVALUATIONS

6.1 STALENESS
Following, we present the stale-data computation error between our application and the existing implementation in BG. We found that for the MySQL system and OrientDB both implementations provide same results while for the HBase implementation there is a very tiny differentiation which we need to investigate further.

<table>
<thead>
<tr>
<th>TestCases</th>
<th>Total-Reads</th>
<th>Total-Updates</th>
<th>BG-Stale-Reads</th>
<th>DIP-Stale-Reads</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>883460</td>
<td>2673</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>1713172</td>
<td>5476</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>848495</td>
<td>2630</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>HBase</td>
<td>3764203</td>
<td>667571</td>
<td>0</td>
<td>23</td>
<td>6.00E-06</td>
</tr>
<tr>
<td>HBase</td>
<td>4509136</td>
<td>617521</td>
<td>584094</td>
<td>580208</td>
<td>6.60E-03</td>
</tr>
<tr>
<td>OrientDB</td>
<td>339125</td>
<td>30834</td>
<td>123</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

| **Total-Reads**: number of total read operations  
| **Total-Updates**: number of total update operations  
| **BG-Stale-Reads**: number of stale read operations returned by current implementation in BG  
| **DIP-Stale-Reads**: number of stale read operations returned by Data-Item based Partition  
| **Error**: percentage difference between BG-Stale-Reads and DIP-Stale-Reads |

6.2 Scala-Spark vs PySpark
Moreover, we present a comparison (Fig.5) for the above datasets between Scala-Spark and PySpark in terms of execution time. The observed times show that Python requires almost double-time the time of Scala. The main reason behind this differentiation is that Python does not solely call the Spark Libraries but needs to perform a significant amount of processing like creation of interval trees and identification of stale data. Therefore, the translation overhead of the same procedure for both Scala and Python is inherently much higher for Python.

![Performance - Scala-Spark vs Py-Spark](image)

**Figure 5**
*Experiment Specifics:*
- CPU cores: 4
- CPU type: 2.7GHz Intel Core i7
- spark.driver.memory: 1024m
- spark.executor.memory: 2048m
7. FUTURE WORK
The current work is not completely computationally efficient. To be more specific, if the amount of log records for any partition does not fit in the main memory of a server, the server may still show memory thrashing behaviour. One possible approach to solve this problem is to further divide the log records for each partition inside in each server based on their relative timestamps. This will significantly enhance the performance of the Validation Phase since the read and write logs will be processed by sized-measured chunks which will fit entirely in server’s memory. This is the hybrid technique where we initially distribute the log records based on their (user_id, bg_operation) key combination and then we compute the actual stale reads by processing groups of write and read operations within each server that satisfy specific time constraints.

8. REFERENCES