Analysis of SSD Utilization by Graph Processing Systems

Haider Qutbuddin[1], Dr. Saif-ur-Rahman[2]

[1,2]*Shaheed Zulfikar Ali Bhutto Institute of Science and Technology (SZABIST) Karachi, Pakistan*

[1]qutbuddin.haider@yahoo.com

[2]saif.rahman@szabist.edu.pk

Abstract—Graph Processing Systems are highly productive when it comes to graph data. While using data parallel approach, it could not exploit common characteristics of a graph computation workload. To address all these challenges, distributed graph processing frameworks were introduced which inherited both the properties of graph parallel systems and data parallel system. Usually, the standard operators which were being used by data parallel systems were filter, join, reduce and etc. while graph parallel system introduced operators such as sub-graph, mrTriplets and etc. In comparison with graph framework operators, the standard relational operators were too slow. Traditionally, all the frameworks and their benchmarks were executed over hard disk drive but modern storage technology has evolved which lead us to use Solid State Drives. Solid state drives are known for their lightning speed as it manages to retrieve and populate data using pulse. This paper presents an analysis of SSD by utilizing graph processing systems. It also discuss the pros and cons faced by the Graph Processing Frameworks and by using TRIM support how the issue of wear leveling can be resolved.

I. INTRODUCTION

A graph database is basically an accumulation of vertices and edges where every vertex speaks to a substance, for example, an individual or business considered to be an edge and each edge speaks to an association or relationship between two vertexes. Each vertex in a graph database is characterized by an interesting identifier, a set of friendly edges and/or approaching edges and a set of properties communicated as key/worth sets. Each one edge is characterized by an exceptional identifier, a beginning spot and/or closure place vertices and a set of properties. Graph databases are appropriate for dissecting interconnections, which is the reason there has been a ton of enthusiasm toward utilizing graph databases to mine information from online networking. Graph databases are additionally helpful for working with information in business trains that include complex connections and element pattern for example, store network administration, distinguishing the wellspring or source of an IP telephony issue.

There is a need of better graph processing frameworks as enormous amount of data is obtained in the form of graph which is highly related and carries complex connections in between. Building frameworks that processes inconceivable measures of information have been made straighter forward by the presentation of the Map reduce, and its open-source usage Hadoop. These frameworks offer programmed adaptability to great volumes of information, programmed flaw tolerance and a basic programming interface based on actualizing. It has been perceived that these frameworks are not generally suitable when preparing information as a substantial graph.

In this research, the authors have targeted Apache Spark using GraphX API which is a popular graph processing framework which manipulates data using data parallel and graph parallel approaches. One of the frameworks includes graph parallel operators which joins vertex, edge accumulations, apply changes on the properties and structure and move information along edges in the graph.

All these frameworks are either calculating edge or vertex in memory fashion or keeping it on Disk while executing algorithms on large datasets. It is necessary for the disk to play its role and store chunk of data in it while the memory is busy manipulating other data. As soon as some space is available, this data is then provided to memory for further calculation. In this research, the disk role is being replaced with emerging technology known as solid state drive from further calculation. Solid state drive is known for its speed.

II. SOLID STATE DRIVES

Solid State Drive (SSD) has lifted the performance to the next level. Its design and architecture is pretty much similar to a memory stick which makes it different from the conventional hard disk drive which is operated by a read write head. The SSD architecture consist a flash memory and a controller. Basically, flash memory does not have a mechanical arm; it relies on a controller which is just like a brain to the SSD. The controller decides how to store the data and how to retrieve the data. Furthermore, the memory comprises of cells which stores the data. The cells which are

the core part of SSD consist of a single transistor and a floating gate which are used to store electron in it. [1]

A. Single Level Cell

Single Level Cell (SLC) is cell type in solid state storage which tends to store a single bit in an each cell. Single level cell is constantly in one of two states as elaborated in table 1.

Table 1. SLC Levels [1]

Value	State
0	Programmed
1	Erased

The state is dictated by the level of charge that is connected to the cell. Since there are just two decisions, zero or one, the condition of the cell can be deciphered rapidly and the shots of bit mistakes is diminished. Individual Single level cell memory can manage roughly 100,000 compose operations before disappointment. When a cell is composed to its capacity, the cell begins to overlook what is put away and information debasement can occur.

Single level cell flash is by and large utilized as a part of business and mechanical applications and installed frameworks that oblige superior and long haul unwavering quality.

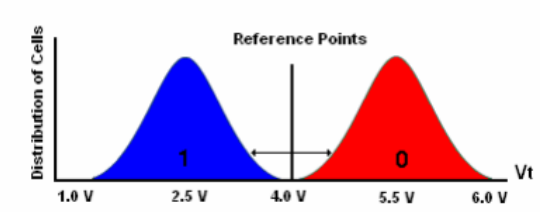

Fig. (1). Voltage Reference for SLC [1]

Figure 1 explains that (0) or (1) which is controlled from limit voltage ”Vt” of a cell, where else edge voltage is controlled from measure of charge which is then diverted to skimming door of a flash cell, setting charge on the skimming entryway will build the limit voltage of a cell, at the point when the edge voltage is sufficiently high around 4 volts these cells will be perused as Programmed, no charge, or edge voltage greater than 4 volts, will affect the cell to be sensed as eradicated.

B. Multi-Level Cell

Multi-Level Cell (MLC) is a cell type in solid state storage which tends to store a two bit in an each cell as mentioned in table 2.

Table 2. MLC Levels [1]

Value	State
00	Fully Programmed
01	Partially Programmed
10	Partially Erased
11	Fully Erased

Multi-Level Cell has a higher bit lapse rate than Single Level Cell. In fact, there are more open doors for confounding the cell’s state just opposite to single-level cell which just stores 1 bit for every cell and is constantly in one of two states, modified (0) or deleted (1). Multi-Level Cell have more than two states on the grounds that every bit in the cell is either customized or eradicated. Multi-Level Cell-2, for instance, has four states. Multi-Level Cell-3 has eight states and Multi Level Cell-4 has sixteen; as with Single Level Cell, each one state is controlled by the level of electrical charge that is connected to the cell.

As a rule, the more bits the cell has, the less compose cycles it will have. Case in point, a 2-bit Multi Level cell is useful for around 3,000 to 10,000 compose operations before it starts to fall flat, while a 3-bit Multi Level Cell would just have 300 to 3,000 compose cycles.

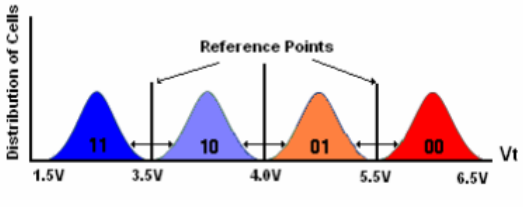

Fig. (2). Voltage Reference for MLC [1]

Flash cell’s capacity to store charge is the reason behind Multi Level Cell working. Since, the delta between each one level has diminished, the affectability between each one level expanded. In this way, more inflexibly controlled writing computer programs is required to control a more exact measure of charge put away on the drifting entryway.

In this way, Multi-Level Cell works in a similar fashion like Single Level Cell flash memory. Edge voltage “VT” is utilized to control condition of a flash memory and at the end measure of charge on the skimming entryway is the thing that decides the limit voltage. As shown in the above figure 2, currently Multi Level Cell uses two bits, or 4 levels. In any case, it is conceivable to hold more bits.

C. SSD Wear Leveling

All SSD memory experiences wear, which happens in light of the fact that eradicating or programming a cell subjects it to wear because of the voltage connected. Every time a charge is caught in the transistor’s entryway dielectric

and reasons a changeless move in the cell's attributes, which after various cycles, shows as a fizzled or failed cell. Basically, Wear leveling is a kind of process which is intended to augment the life of Solid State Drive [2]. Solid State Drive is buildup of microchips that store information only in blocks. Every one of these block can endure a limited number of system/eradicate cycles before getting to be questionable. For instance, SLC NAND glimmer is ordinarily appraised at around 100,000 project/eradicate cycles. Wear leveling masterminds information with the goal that compose/eradicate cycles are appropriated equitably among the majority of the pieces in the device.

Wear leveling is regularly overseen by the SSD flash memory controller which utilizes a wear leveling calculation to figure out which physical piece to utilize every time information is modified [2]. There are two sorts of SSD Wear Leveling algorithm: dynamic and static. In Dynamic wear leveling, pools delete blocks and choose the block with the most reduced eradicate mean the following compose. Where as in Static wear leveling, then again, chooses the target obstruct with the most reduced general delete tally, deletes the piece if fundamental, composes new information to the square and guarantees that blocks of static information are moved when their blocks are eradicated. It is then check underneath a certain limit.

Wear leveling is also overcome by Trim Support. A TRIM order empowers working framework to discover the checked pages before they are required and wipe them clean [3]. Cleaning these information pages already spares time when composition on the information pages is required again. To work accurately, TRIM must be upheld by both the solid state drive and the working framework that are being utilized. At the point when both the OS and the SSD help TRIM, individual pages can be cleaned and solid state drive will be educated that the pages are presently clear and can be composed on. This sort of cleaning and correspondence is the key way in keeping the drive performing to the best of its capacities.

III. APACHE SPARK

GraphX is actualized on top of Spark, a broadly utilized data parallel system, Like Hadoop Map reduce. A Spark group comprises of a solitary driver hub and numerous laborer hubs. The driver hub is in charge of assignment booking and dispatching while the laborer hubs are in charge of the genuine reckoning and physical information stockpiling. Spark gives the Resilient Distributed Dataset (RDD) in memory stockpiling deliberation. Resilient Distributed Dataset is accumulations of protests that are divided over a cluster [4]. Rather than the two-stage Map reduces topology, Spark backs general calculation (DAGs) by making numerous information parallel administrators on Resilient Distributed Dataset making it more suitable for communicating complex information streams [4].

A. GraphX

The adaptability and execution of GraphX is taken from plain choices and improvements made in the physical execution layer. Configuration of the physical representation as an arrangement of joins and collections, keeping up the legitimate records can considerably accelerate nearby join and collection execution. Secondly, it minimize correspondence in graphs by utilizing vertex-cut apportioning in which edges are parceled equitably over a group and vertices are imitated to machines with contiguous edges [4].In the end, chart processing is ordinarily iterative and subsequently, a records can be built.

1) Graph Parallel Systems: The expanding scale and need of graph organized information has prompted the rise of a scope of graph parallel frameworks or graph parallel systems. Every framework is fabricated around a variety of the graph parallel deliberation which comprises of a property diagram and a vertex-program that runs on every vertex in the chart and can interface with nearby vertex-programs through messages. Each instance of the vertex-system can read and adjust its vertex property and the properties on adjoining edges and in some cases, even the properties on nearby vertices. Generally, frameworks embrace the mass synchronous execution model in which all vertex-projects run simultaneously in a grouping of super steps working on the neighboring vertex-system state or on messages from the past super-step. [5] Since it guarantees deterministic execution, disentangles debugging, what's more that empowers deficiency tolerance. [6]

2) Graph Parallel Operator: The edge between graph parallel and data parallel lies between its operators unlike traditional operators. Graph parallel system consists of such operators which specialize in utilizing and exploiting a graph. The GraphX framework uncovers the standard information parallel administrators which are found in contemporary data stream frameworks. The unary administrators channel, outline, reduce by key. Furthermore, creates another gathering with the records uprooted, changed and then again totaled. The paired administrator left join performs a standard left external equijoin by key. Two of the operators which are map and filter are totally data parallel without obliging any information development on the other hand correspond [4]. Then again reduce by key and left join administrators may require generous information development relying upon how the data is distributed. The Graph administrator builds a property diagram from vertex and edge accumulations. In numerous applications, the vertex gathering may contain copy vertex properties or may not contain properties for vertices in the edge gathering, case in point when working with web information; web-connections may indicate missing pages or pages may have been viewed various times [4]. While the Graph operator creates a chart, arranged perspective of accumulations, the vertices, edges, and triplets produce gathering focused perspectives of a chart, the vertices and edges graph

deconstruct the property chart into the comparing vertex and edge accumulations. The gathering perspectives are utilized when registering totals breaking down the aftereffects of chart reckoning or when sparing diagrams to outside data stores. By combining properties along edges, the triplet's administrator empowers an extensive variety of graph reckoning. For instance, the organization of the triplets and data parallel channel administrators can be utilized to concentrate edges that compass two spaces or join clients with diverse investments. Moreover, the triplet's administrator is utilized to develop the other graph parallel administrators which are sub graphs and mrTriplets, mapv and mape administrators change the vertex and edge properties individually and furnish a proportional payback graph. The guide UDF gave to mapv and mape can just give quality esteem and cannot alter the structure.

B. Resilient Distributed Dataset

Spark is implemented utilizing Scala around the idea of Resilient Distributed Datasets (RDD) and gives activities/ changes on top of RDD; formally a RDD was perused, divide accumulation of records. RDD must be made through deterministic operations on either information in steady storage or different RDDs. This is the sacred vessel of what a RDD is. RDD are a changeless versatile dispersed accumulation of records which can be put away in the unstable memory or in a persevering storage HDFS, Hbase and can be changed over into an alternate RDD through a percentage of the changes, an activity like tally can likewise be connected on a RDD. [6] At the point when the memory is not sufficient enough for the information to fit in, it can be either spilled to the drive or is simply left to be reproduced upon appeal for the same; likewise a RDD can be stored in memory for as often as possible reserved information. Let's say distinctive questions are run on the same set of information repeatedly, this specific information can be kept in memory for better execution times.

1) Partitioning: Graph parallel calculation requires every vertex or edge to be prepared in the connection of its neighborhood. In addition, each change relies upon the after effect of conveyed joins between vertices and edges [7]. As an outcome, indexing and information format are paramount steps in attaining a productive dispersed execution. Since, the graph structure portrays data development; dispersed graph processing frameworks depend on graph parceling and proficient graph storage to minimize correspondence and capacity overhead. It also guarantees adjusted computation. Usually graph is partitioned into two segments which either edge cut or vertex cut.

2) Edge Cut: An edge-slice interestingly assigns vertex to machines while permitting edges to compass crosswise over machines. Below figure 3 illustrates edge cut.

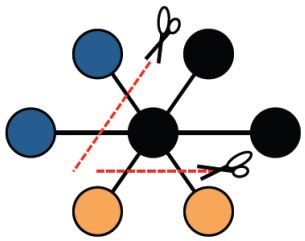

Fig. (3). Edge Cut [7]

The correspondence and capacity overhead of an edge-cut is specifically corresponding to the quantity of edges that are cut [7]. Subsequently, correspondence overhead can be decreased and guarantee adjusted computation by minimizing both the quantity of cut edges and the number of vertices assigned out to the most stacked machine, then again for most expensive scale genuine graph, building an ideal edge- cut can be costly [7]. As an outcome, numerous graph reckoning frameworks have embraced the procedure of randomly disjointed vertices over the cluster.

3) Vertex Cut: Vertex Cut is based on distributing nodes over the machines. It uniformly assigns out edges to machines and permit vertices to compass over numerous machines. Below figure 4 illustrates edge cut.

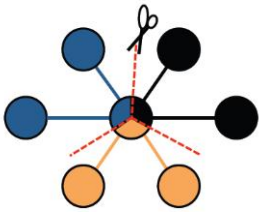

Fig .(4). Vertex Cut [7]

The correspondence and capacity overhead of a vertex-slice is straightforwardly corresponding to the whole of the quantity of machines crossed by every vertex. In this way, correspondence overhead can be decreased and guarantee adjusted calculation by uniformly allocating edges to machines in way that minimizes the quantity of machines crossed by every vertex [7]. Developing ideal vertex-cuts is likewise restrictively lavish on large scale graphs. The least difficult methodology is to utilize a hash capacity to arbitrarily appoint edges to machines through a straightforward examination. It can be demonstrated that for the force law degree appropriations found in real graphs, arbitrary vertex-cuts can be requested to size more productive than arbitrary edge-cuts.

C. Representing Vertex Cut in Tabular Format

Vertex cut representation in GraphX resilient scattered graph data structure is achieved using three unordered evenly parceled tables actualized as Spark Resilient Distributed

Dataset. Below is the figure 5 in which there is a representation of tabular format for vertex cut.

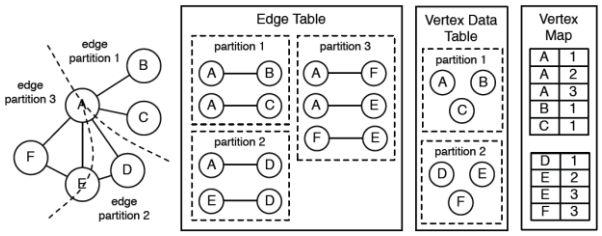

Fig. (5). Tabular representation of vertex cut [7]

1) Edge table: Edge table parameters are PID, src, dst, and information. It stores the adjacency structure and edge information. Every edge is spoken to as a tuple comprising of the source vertex id, terminus vertex id, client characterized information and a virtual part identifier PID [7]. It must be noted that the edge table contains just the vertex ids and not the vertex information. The edge table is mapped by the PID.

2) Vertex Data Table: Edge table parameters are id and data. It stores the vertex information as a vertex (id, information) sets. The vertex information table is ordered and divided by the vertex id.

3) Vertex Map: Edge table parameters are (id, pid). It gives a mapping from the id of a vertex to the ids of the virtual segments that contain contiguous edges. Case in point, on the grounds that vertex (A) is connected with edges in all parts, there are three tuples identified with (A) in the vertex map table [6]. The vertex guide table is apportioned by the vertex id.

IV. EXPERIMENT

A. Experimental Setup

The method that has been proposed in this research is tested by performing experiments. A setup is created for testing proposed method. The experiment is conducted on Dell latitude E6510 machine with a core i5 Intel Processor, 4GB of RAM and SSD Liteon Model LCT-256M3S capacity 256 GB and hard disk drive Samsung 500 GB.

B. Data Set

Live Journal dataset is an online community which is free with a member count of more than 10 million members. Live journal helps member to maintain their journals and individual blogs. It also allows people to define which member in this community is their friend. Below are the details of the dataset mentioned in figure 6.

The twitter dataset consists of circles which includes profile that can be termed as feature. This dataset was basically crawled out from a public source. It consist of 81306 Nodes and 1768149 edges

Dataset statistics	
Nodes	4847571
Edges	68993773
Nodes in largest WCC	4843953 (0.999)
Edges in largest WCC	68983820 (1.000)
Nodes in largest SCC	3828682 (0.790)
Edges in largest SCC	65825429 (0.954)
Average clustering coefficient	0.2742
Number of triangles	285730264
Fraction of closed triangles	0.04266
Diameter (longest shortest path)	16
90-percentile effective diameter	6.5
Dataset statistics	
Nodes	81306
Edges	1768149
Nodes in largest WCC	81306 (1.000)
Edges in largest WCC	1768149 (1.000)
Nodes in largest SCC	68413 (0.841)
Edges in largest SCC	1685163 (0.953)
Average clustering coefficient	0.5653
Number of triangles	13082506
Fraction of closed triangles	0.06415
Diameter (longest shortest path)	7
90-percentile effective diameter	4.5

Fig. (6). Details of Selected Datasets

C. Hard Disk Drive Specs

- Hard drive interface: Serial ATA II
- Hard drive capacity: 500 GB
- Hard drive speed: 5400 RPM
- Drive device, buffer size: 8 MB
- Hard disk average seek time: 12 ms
- Average latency: 5.6 ms
- Drive ready time: 4 s

D. Solid State Drive Specs

- Solid State drive interface: Serial ATA 6 GB/s compatible with SATA 3 GB/s and SATA 1.5G/s
- Solid State drive capacity: 256 GB
- Solid State drive Controller: Marvel 88SS9174 Flash controller
- Flash Memory: MLC
- Seq. Read: 520 MB/s
- Seq. Write: 430 MB/s
- TRIM Command: Supports

E. Hard Disk Drive Benchmark

For Bench marking, a Read Write test for Hard Disk Drive of 500MB was conducted; a test data was used on which below result was obtained.

While bench marking, hard disk drive on write operation, it initiated at around 150MB per second and then settled at 84MB per second as seen in figure 7, while on read operation it started from 35MB per second and then stabilized at 84MB per second. The reason behind slow read initiation is when the read operation starts, the head of drive needed to move towards the starting block which takes time but, when performing write operation, it instantly started to write where the head was because the partition was kept empty.

F. Solid State Drive Benchmark

For Bench marking, a Read Write test for Solid State Drive of 500MB was conducted; a test data was used on which the below result was obtained.

While bench marking, solid state drive on write operation, it initiated at around 250MB per second. In figure 8, it can be noticed that on write operation, SSD is not working on constant speed because SSD operates on block writing and there can be cells between this process which consist of some other data, so shifting those cells and continuing the operation might cause variation in speed. While on read operation, it started from 210MB per second and then stabilized at 265MB per second. The reason why it reads so fast is because of the nature of SSD. SSD read and write data using electronic signal which will always move fast as compared to the read write head.

G. Page Rank Algorithm

Page rank is a numeric esteem that elaborates how essential a page is on the web. It is assumed that when one page connects to an alternate page, it is successfully making a choice for the other page. The more votes that are thrown for a page, the more imperative the page must be; likewise' the significance of the page that is making the choice decides how essential the vote itself is. Page rank computes a page's imperativeness from the votes cast for it. How imperative each vote is, considered when a page's Page rank is computed. In terms of graph, it calculates ranking of each node present in the graph. The initial values for all vertexes are set the same and then it repeatedly updates the formula to get the vertex rank converge. Since, it is required to execute the entire algorithm on Disk rather than doing it in a traditional way of going on memory, the storage levels were customized within the code so that the edge and vertex are kept on Disk. Furthermore, the recent vertex cut were cached to give better performance while execution. The cache part was removed so that on every turn, Disk is consumed and a clearer picture of disk consumption can be obtained. Thenceforth, for every resilient distributed dataset, the storage level was changed to disk so that after each vertex and edge resilient distributed dataset is calculated. It kept on disk and then called from disk. Below is the customized code for page rank algorithm which is kept in Analytics file for Scala.

```
    val partitionStrategy: Option[PartitionStrategy] = options.remove("partStrategy")

      .map(PartitionStrategy.fromString(_))
    val edgeStorageLevel = options.remove("edgeStorageLevel")
.map(StorageLevel.fromString(_))
      .getOrElse(StorageLevel.DISK_ONLY)
    val vertexStorageLevel = options.remove("vertexStorageLevel")
      .map(StorageLevel.fromString(_))
      .getOrElse(StorageLevel.DISK_ONLY)
    taskType match {
      case "pagerank" =>
        val tol = options.remove("tol")
        .map(_.toFloat).getOrElse(0.001F)
        val outFname = options.remove("output").getOrElse("")
        val numIterOpt = options.remove("numIter").map(_.toInt)
        options.foreach {
          case (opt, _) => throw new IllegalArgumentException("Invalid option: " + opt)
        }
        println("=======================")
        println("|      PageRankMod      |")
        println("=======================") val sc = new SparkContext(conf.setAppName("PageRank("
        + fname + ")"))
        val graph = GraphLoader.edgeListFile(sc, fname,
          numEdgePartitions = numEPart,
        edgeStorageLevel = edgeStorageLevel,
        vertexStorageLevel = vertexStorageLevel)
        .partitionBy(EdgePartition2D)
       println("GRAPHX: Number of vertices " + graph.vertices.count)
       println("GRAPHX: Number of edges " + graph.edges.count)
       val pr = (numIterOpt match {
         case Some(numIter) => PageRank.run(graph, numIter)
```

```
    case None => PageRank.runUntilConvergence(graph, tol)
  }).vertices.cache()
  println("GRAPHX: Total rank: " + pr.map(_._2).reduce(_ + _))
  if (!outFname.isEmpty) {
    logWarning("Saving pageranks of pages to " + outFname)
    pr.map { case(id, r) => id + "\t" + r }.saveAsTextFile(outFname)
  }
  sc.stop()
```

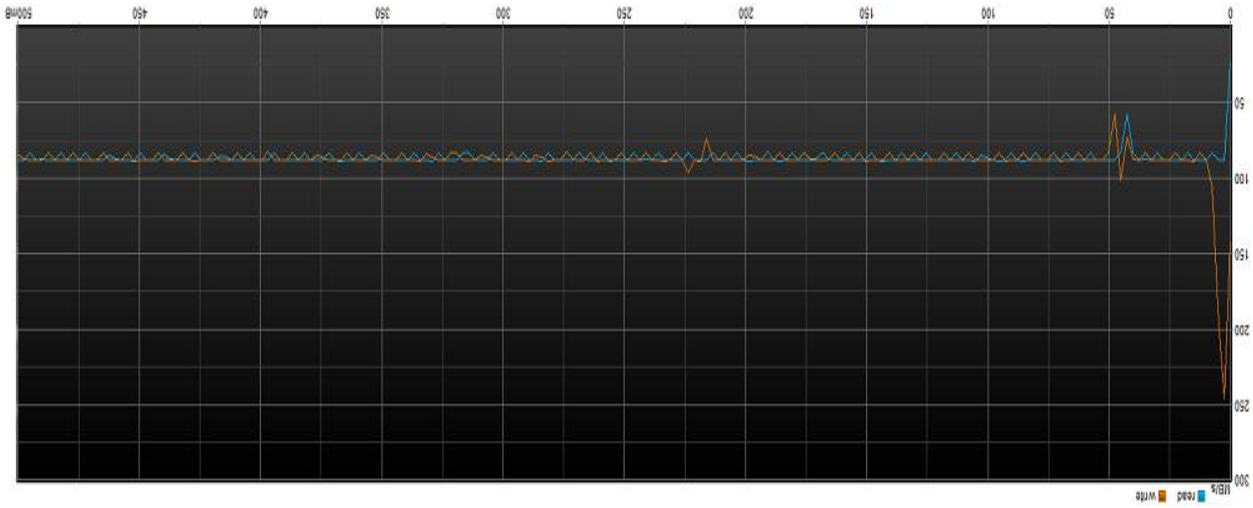

Fig. (7). Hard Disk Benchmark

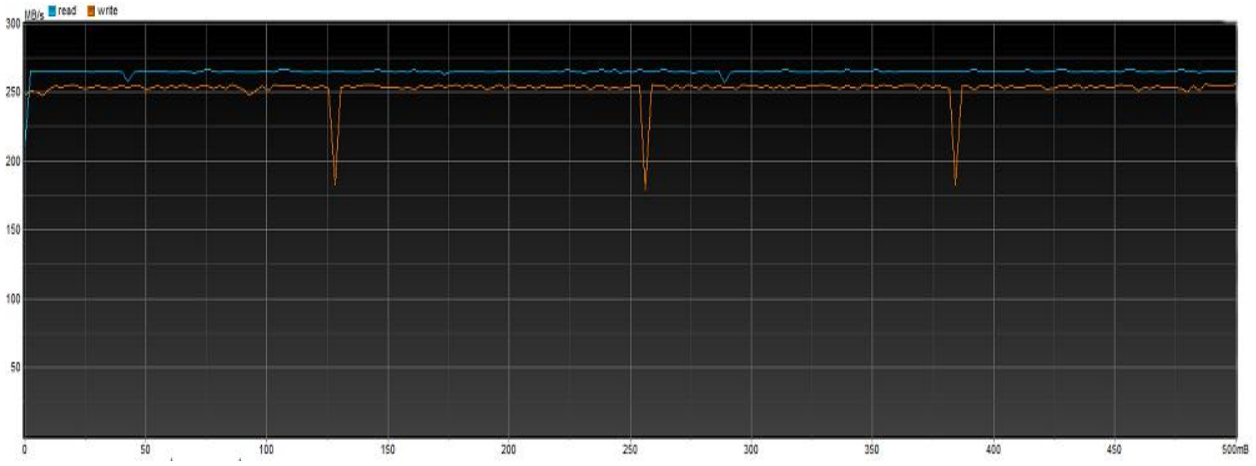

Fig. (8). SSD Benchmark

H. GraphX Shuffle Write Benchmark

Unlike Hadoop, Sparke Map Tasks write the output specifically to disk drive. There is no utilization of an in memory storage. Each one Map Task composes the same number of mix records as the quantity of Reduce Task. It writes one shuffle file for one task. Livejournal Dataset was used which was partitioned from 1GB to 100MB on which this operation is performed, the dataset was stored on distributed file system Hadoop and the results were carried out on Spark using GraphX api, on page rank algorithm the dataset was executed.

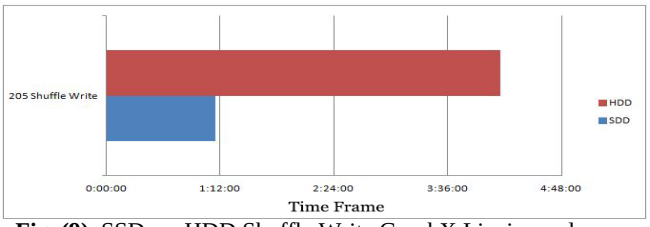

Fig. (9). SSD vs. HDD Shuffle Write GraphX Livejournal

In Figure 9, it can be observed how solid state drive has outclassed hard disk drive. On 205MB per second shuffle write, it just took SSD around 61 minutes while performing the same operation over HDD it took 4 hours and 9 minutes. From the above results, it can be noted that the potential of graph processing framework and it can exploit the nature of solid state drive.

Twitter Dataset was used on which this operation is performed. The dataset was stored on distributed file system Hadoop and the results were carried out on Spark using GraphX api, on page rank algorithm the dataset was executed.

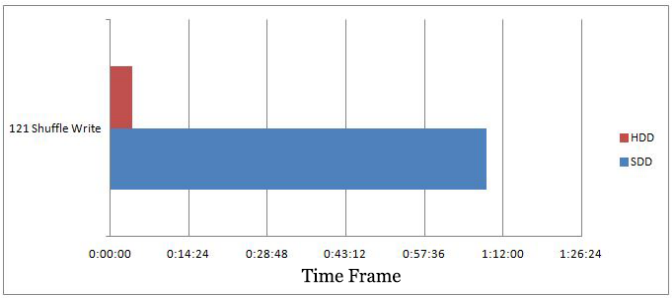

Fig. (10). SSD vs. HDD Shuffle Write GraphX Twitter

In Figure 10, it can be observe how hard disk drive has outclassed solid state drive. On 121MB per second shuffle write, it just took SSD 61 minutes while performing the same operation over HDD it took 4 hours and 9 minutes. From the above results, the potential of graph processing framework can be commemorated and it can exploit the nature of solid state drive.

At long last, a SSD will utilize one of two interface advances: SATA 3 GB/s (likewise promoted as SATA 2 or SATA II), or the fresher and quicker SATA 6 GB/s (otherwise known as SATA 3 or SATA III). Drives with the more advanced interface are perfect with machines equipped with more established engineering and the other way around. Yet a SATA 6gb/s drive will convey its best execution just on the off chance that its joined with a SATA 6gb/s interface.

V. CONCLUSION AND FUTURE WORK

Although SSD has its own advantages over hard disk drive but still there are area's which needs to be improved while working with solid state drives. If benchmarks are taken into consideration, no doubt SSD is incomparable with hard disk drive but if write operations of SSD is observe closely, it still requires optimization. While observing Graph Processing framework on these storage mediums, the results were not as expected on all datasets. When page rank algorithm was executed on live journal dataset, SSD outclassed HDD but on twitter dataset results came out to be opposite. At this level it can be concluded that solid state drives still lacks consistency while performing write operations, getting specific to graph frameworks, SSD does not outclassed HDD on all datasets.

This section focuses on related work in SSD optimization. There still exist problems in SSD while performing write operation as shown in Fig 9. The write operation performed by SSD still lacks consistency; however, read operations are consistent. Furthermore, SSD does not perform consistently on all datasets. It can be clearly observed in Fig 10 where hard disk drive has outclassed SSD.

REFERENCES

[1] *Slc vs. mlc: An analysis of flash memory*. Super Talent Technology, Inc. White Paper, San Jose, CA, USA.

[2] *Flash solid-state disk reliability*. Texas Memory Systems White Paper, 2008.

[3] C. King and T. Vidas. "Empirical analysis of solid state disk data retention when used with contemporary operating systems," *Digital Investigation*, vol. 8, pp: S111–S117, 2011.

[4] D. Crankshaw, A. Dave, R. S. Xin, J. E. Gonzalez, M. J. Franklin and I. Stoica. "The graphx graph processing system." UC Berkeley AMPLab.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson and C. Guestrin. "Powergraph: Distributed graph-parallel computation on natural graphs." In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12*), 2012, vol. 12, pp: 17-30.

[6] J. Yan, G. Tan and N. Sun. "Gre: A graph runtime engine for large-scale distributed graph-parallel applications," CoRR, abs/1310.5603, 2013.

[7] R. S. Xin, J. E. Gonzalez, M. J. Franklin and I. Stoica, "Graphx: A resilient distributed graph system on spark," In *First International Workshop on Graph Data Management Experiences and Systems, (GRADES '13)*, 2013, Art. 2.