A study on Garbage Collection Algorithms for Big Data Environments

Rodrigo Bruno, INESC-ID / Instituto Superior Técnico, University of Lisbon
Paulo Ferreira, INESC-ID / Instituto Superior Técnico, University of Lisbon

The need to process and store massive amounts of data (Big Data), is a reality. In areas such as scientific experiments, social networks management, credit card fraud detection, targeted advertisement, and financial analysis, massive amounts of information is generated and processed daily to extract valuable, summarized information. Due to its fast development cycle (i.e., less expensive to develop), mainly because of automatic memory management, and rich community resources, managed object-oriented programming languages (such as Java) are the first choice to develop Big Data platforms (e.g., Cassandra, Spark) on which such Big Data applications are executed.

However, automatic memory management comes at a cost. This cost is introduced by the Garbage Collector which is responsible for collecting objects that are no longer being used. Although current (classic) garbage collection algorithms may be applicable to small scale applications, these algorithms are not appropriate to for large scale Big Data environments as they do not scale in terms of throughput and pause times.

In this work, current Big Data platforms and their memory profiles are studied to understand why classic algorithms (which are still the most commonly used) are not appropriate, and also to analyze recently proposed and relevant memory management algorithms, targeted to Big Data environments. The scalability of recent memory management algorithms is characterized in terms of throughput (improves the throughput of the application) and pause time (reduces the latency of the application) when comparing to classic algorithms. The study is concluded by presenting a taxonomy of the described works and some open problems, with regards to Big Data memory management, that could be addressed in future works.

CCS Concepts: •**General and reference** → **Surveys and overviews;** •**Computer systems organization** → **Cloud computing;** •**Software and its engineering** → **Allocation / deallocation strategies; Garbage collection;**

Additional Key Words and Phrases: Garbage Collection, Big Data, Processing Platforms, Storage Platform, Java, Memory Managed Rutime, Scalability, Big Data Environment

ACM Reference Format:
Rodrigo Bruno and Paulo Ferreira, 2016. A study on Garbage Collection Algorithms for Big Data Environments. *ACM Comput. Surv.* V, N, Article A (January 2016), 35 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The need to handle, store and process large amounts of data, Big Data, to extract valuable information, is a reality [Akerkar 2013]. Scientific experiments (such as protein folding, physics simulators, signal processing, etc), social networks management, credit card fraud detection, targeted advertisement, and financial analysis are just a few examples of areas in which large amounts (thousands or even hundreds of thousands of GBs) of information are generated and handled/processed daily. Moreover, the importance of live strategic information has led the biggest companies in the world (Google, Facebook, Oracle, Yahoo, Tweeter, Amazon, and others) to build Big Data platforms.

Big Data platforms are designed to efficiently handle massive amounts of data. There are many examples of such platforms: Hadoop [White 2009] (a MapReduce implementation), Cassandra [Lakshman and Malik 2010] (a distributed Key-Value store), Neo4J [Robinson et al. 2013] (a graph database), Spark [Zaharia et al. 2010] (a cluster computing system for Big Data applications), Google File System [Ghemawat et al. 2003] (a distributed file system), Naiad [Murray et al. 2013] (a dataflow system), Dryad [Isard et al. 2007] (a scheduler for distributed Big Data applications), etc. A particularly challenging property of such systems is the need for scalability. In other words, Big Data platforms' performance is expected to increase proportionally to an increase in the amount of resources.

It is also a fact that many of these Big Data platforms are running on managed object-oriented programming languages (such as Java). Java if often the preferred choice for designing and implementing such platforms mainly because of its fast development cycles and rich community resource. However, despite making programming easier, these languages use automated memory management (Garbage Collection) that comes at a performance cost. This cost is significantly magnified when these managed runtimes implementations are used to run Big Data applications, which tend to harvest computational resources.

Moving back to unmanaged languages (such as C or C++) could be a possible solution. However, unmanaged languages are error-prone (debugging memory problems is known to be a long and painful task). Furthermore, since a great number of existing Big Data platforms are already developed in a managed language, it is unrealistic to re-implement them from scratch.

Therefore, with the advent of Big Data platforms, new scalability requirements are posed to actual managed runtimes such as the Java Virtual Machine (JVM). These runtimes (and the JVM in particular) must allow Big Data platforms to scale, i.e., as new Big Data platforms demand more computational resources (such as memory and processing power), current memory management strategies must provide solutions that scale with these new resource requirements. Current studies show that the overhead introduced by memory management can be up to one third of the total execution time [Gidra et al. 2013], and that this performance overhead cannot be solved by adding additional resources [Nguyen et al. 2015; Bruno et al. 2017].

Given i) the importance of Big Data platforms, ii) the relevance of managed runtimes to develop such platforms, and iii) the difficult and significant performance problems posed by automatic memory management on Big Data application performance [Nguyen et al. 2015; Bruno et al. 2017], this work studies recent Garbage Collection algorithms designed to cope with the scalability requirements posed by Big Data platforms. The goal of this work is to provide a clear view over the recent works on Garbage Collection, which aim to improve the performance of Big Data applications by reducing the negative performance impact of automatic memory management. It is true that there are also interesting contributions on automatic memory management contributions for runtime environments such as JavaScript engines [Clifford et al. 2015], which focus web applications. However note that such engines do not support the target environment and applications of this work: large-scale and long-running Big Data applications, which lead to long workloads, with high resource demands.

To the best of the authors' knowledge, the work presented in this paper is the first to explore the limitations of current Garbage Collection implementations in Big Data environments, and to present and compare several proposed works in the area. This is a very relevant topic as there is a growing need to improve current collectors to keep up with the demands posed by new Big Data environments.

The paper is organized as follows. Section 2 analyzes the types of Big Data platforms and how memory is handled by each type of platforms, what types of memory layouts are used and which scalability challenges they pose regarding memory management. Then, Section 3 provides an overview of the main concepts in memory management. These concepts are necessary to introduce and describe the classic Garbage Collection (GC) algorithms (see Section 4) which are the basis for more recent algorithms described in Section 5. Section 5 is divided in: i) throughout oriented algorithms, and ii) latency oriented algorithms because some recent algorithms are optimized either for throughout (i.e., these algorithms try to keep application throughput as high as possible) or for latency (i.e., these algorithms try to keep applications pauses as short as possible). The work concludes with open research directions for Big Data memory management (Section 6) and some conclusions on Section 7.

In sum, the main contributions are the following:

— a comprehensive description of Big Data environments' memory profiles, i.e., how objects created by Big Data applications are kept in memory, and which scalability challenges arise from these applications;
— an analysis of recent GC algorithms targeted to Big Data platforms which deal with the main scalability problems present in Big Data environments: throughput and latency.[1]
— a set of open problems that can lead to new research directions which could result in solutions for actual problems in today's design of memory management algorithms and strategies.

The authors hope that this work provides a better understanding on: i) how current Big Data platforms impact memory management in managed runtimes, and ii) the limitations of current GC algorithms that could be used for developing new, more capable, memory management algorithms for Big Data platforms.

2. BIG DATA ENVIRONMENTS

The term Big Data was used for the first time in an article by NASA researchers Michael Cox and David Ellsworth [Cox and Ellsworth 1997]. The pair claimed that the rise of data was becoming an issue for current computer systems, which they called the "problem of big data". In fact, in recent years, the amount of data handled by computing systems is growing. However, not only the amount of data is growing, but also the speed at which it grows is increasing.

Data can be big in many ways. Big Data can be applied in many areas and in each of which it may have slightly different meanings. Within this work, Big Data is used to represent high volumes of data that, because of its dimension, need specialized software tools to handle it (i.e., tools previously developed do not scale to large data sets, in different performance metrics). The typical motivation for storing and processing such volumes of data is to extract valuable/summarized information from large sets of data.

Throughout the paper, the following terms are used: Big Data environments, Big Data platforms, and Big Data applications (Figure 1 illustrates these concepts). The first (Big Data environments) refers to a group of one or more Big Data platforms that are used to complete a specific task. These platforms are frequently organized in a stack, i.e., each platform is given the output of the previous platforms and prepares the input of the next (Figure 2 shows an example of such stack: the Hadoop stack).

[1]In this work, two different terms are used: latency and pause time. Latency is used to describe the response time of an application to the user. Pause time is used to describe the amount of time that the application is stopped because of the GC (for example, to collect memory). The two terms are related given that application pauses will have a direct impact on application latency; the longer the pause, the higher the application latency.

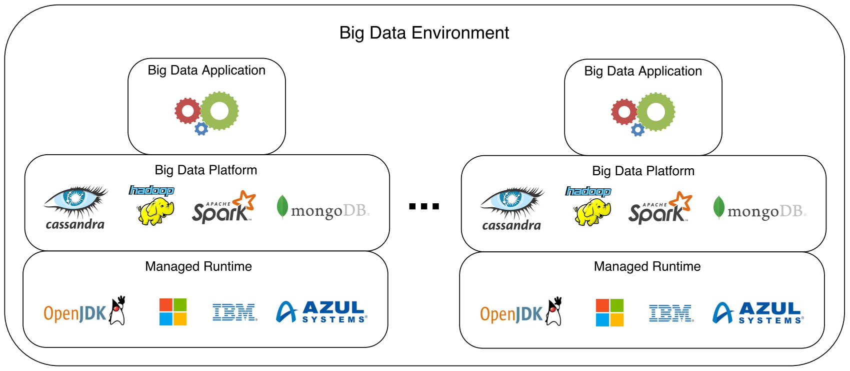

Fig. 1. Big Data Environment Taxonomy

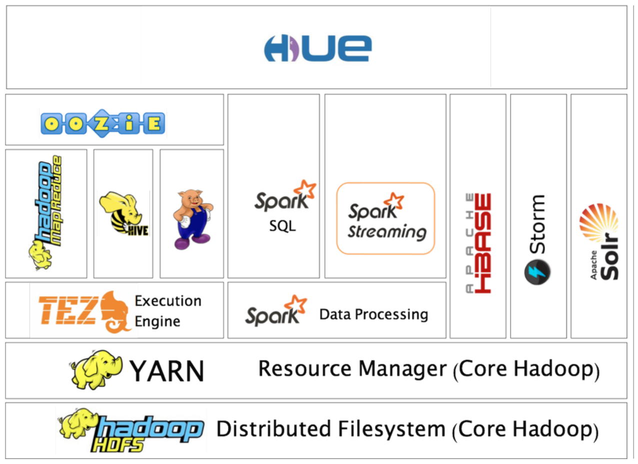

Fig. 2. Big Data Platform Stack Example: Hadoop Stack

Each Big Data platform represents a processing or storing engine running on top of a managed runtime environment (for example, a JVM). Finally, a Big Data application represents the user code executed or served by the engine (inside the Big Data platform).

The fact that many existing platforms do not apply to Big Data due to scalability problems is now widely accepted as more and more companies invest large amounts of money for creating new Big Data platforms capable of storing and processing their data [Bryant et al. 2008; Lynch 2008]. Among many possible real world Big Data use case scenarios, some of them may be the following:

— Trend Analysis. It is known that big companies often apply data mining techniques (machine learning, for example) to extract sales patterns associated with some product, advertising, or pricing. This is a clear example where large volumes of data (sales reports in this case) are stored and then processed to extract valuable information. Such information helps company owners adapt their offer to the available market;
— Behavioral Analytics. Similarly to sales report processing, user's information regarding, for example, purchasing or searching habits, can be used to improve the user experience by automatically suggesting similar results or for performing targeted ad-

vertising. This is only possible if each user's interactions are recorded and processed to extract the behavior pattern of each user within a short time frame;

— Internet Search Engines. The web crawling process (from which Internet search indexes are built) is another example of a Big Data use case. Companies such as Google, Yahoo!, and Microsoft, process, every day, large amounts of Internet web pages to feed rankings (and other kinds of metrics) to different search engines. In this scenario, data is not only analyzed for pattern extraction but also transformed into another representation, one that enables search engines to rank pages according to several desired metrics (keywords, popularity, date of creation, and more);

— Fraud Detection. By extracting user's behavioral information, companies can also detect potential fraudulent behaviors and take a preventive measures. Credit card fraud detection is a real example of this use case. Companies fighting fraud detect unlikely transactions (according to the users' behavior and historic) and stop them.

Scalability in Big Data environments is most often measured in terms of throughout and latency scalability. Within this work, consider that: i) being throughput scalable means that the throughout (number of operations per amount of time) should increase proportionally to the amount of resources added to the system, and ii) being latency scalable means that the latency (duration of a single request) should not increase when the throughput increases. For example, in fraud detection system, the number of credit card transactions verified per second (throughput) is as important as the duration (latency) of a single credit card transaction verification. Therefore, in the case of fraud detection, the ideal system is both throughput and latency scalable (i.e., the system should increase its throughput as more resources are used but the latency should not be affected by increasing the throughput). However, as discussed in Section 5, throughout and latency can be difficult to achieve at the same time.

The throughput and latency scalability problems are further aggravated if a stack of Big Data platforms (Big Data environment) is considered. In this scenario, the throughput of the whole environment is as high as the throughput of the system with lower throughput, and the latency is as low as the system with higher latency. In other words, a single platform can compromise the scalability of the whole environment.

The challenge of extracting valuable information from very large volumes of information can be decomposed in into two sub-problems: i) how to store massive amounts of data and provide scalable read and write performance, and ii) how to process massive amounts of data in a efficient and scalable way. Both sub-problems are usually handled by different types of platforms: storage and processing. For the remainder of this section, each type of platform is analyzed in separate (storage and processing platforms), identifying, for each one, their memory profiles and the resulting challenges, which memory management algorithms are faced with.

2.1. Processing Platforms

A Big Data processing platform, in the most simple and generic way, is a system which i) receives input data, ii) processes data, and iii) generates output data. The system can be composed by an arbitrary number of nodes, which can exchange information during the processing stage. Input data can, for example, be retrieved from: i) a storage platform, ii) other processing platforms, or iii) directly from sensors. Output data can, be sent to: i) a storage platform, ii) to other processing platforms, or iii) to the final user.

In the remainder of this section, representative real world Big Data processing platforms are analyzed. The goal is to understand how these platforms work and, most importantly, how memory is used by these platforms, or, in other words, their memory profile.

2.1.1. MapReduce-based Platforms. MapReduce [Dean and Ghemawat 2008] is a popular programming model nowadays [Herodotou and Babu 2011; Dittrich and Quiané-Ruiz 2012]. In a MapReduce application, computation is divided into two stages: map and reduce. First, input data is loaded and processed by mappers (nodes assigned with a map task). Mappers produce intermediate data which is then shuffled, i.e., sorted and split among reducers (nodes assigned with a reduce task). In the reduce stage, data is processed into the final output.

Several MapReduce implementations were produced but Apache's Hadoop [White 2009] soon become the de facto standard implementation for both industry and academia [Dittrich and Quiané-Ruiz 2012]. In fact, Hadoop is currently used by some of the worlds' largest Information Technology companies, Facebook [Borthakur et al. 2011], Twitter [Lin and Ryaboy 2013], LinkedIn [Sumbaly et al. 2013], and Yahoo [Shvachko et al. 2010].

The novelty behind recent MapReduce programming model implementations is that most distribution and fault tolerance details are hidden from the programmer. Thus, only two functions need to be defined: i) a map function which converts input data into intermediate data, and ii) a reduce function which aggregates intermediate data. All other steps regarding task distribution, intermediate data shuffling, reading and writing from and to the storage platform, and recovering failed nodes is handled automatically by the platform. Additionally, Hadoop comes with the Hadoop Distributed File System, HDFS, (addressed in Section 2.2) which was specially designed to handle large amounts of data. These two systems (Hadoop MapReduce and HDFS), while working together, form a Big Data environment with both processing and storage capabilities.

Another important factor about MapReduce and HDFS is that both platforms are basic building blocks for more complex Big Data platforms (these platforms represent stack layers on top of MapReduce and HDFS) such as Hive [Thusoo et al. 2009], Pig [Olston et al. 2008], and Spark [Zaharia et al. 2010]:

— Hive is a data management platform that works on top of Hadoop. Hive provides data summarization, query, and analysis, exporting an SQL-like language called HiveQL which automatically converts to MapReduce jobs to execute in Hadoop. All data is read from and written to HDFS;
— Pig is a workload designer that creates MapReduce workloads using a Hadoop MapReduce cycle as building block for more complex operations. Pig proposes a language called Pig Latin that abstracts the MapReduce programming model idiom into a notation that makes MapReduce programming high-level, very similar to SQL for RDBMS systems. Similarly to Hive, input and output data comes and goes to HDFS and Hadoop MapReduce is used to perform the MapReduce tasks;
— Spark is a MapReduce engine (among other capabilities) that enables efficient in-memory data processing. Spark aims at improving the performance of applications that reuse data between MapReduce cycles. In Hadoop, between each cycle, all data must be flushed to disk and retrieved in the next MapReduce cycle. To cope with this problem Spark provides the Resilient Distributed Dataset (RDD) which maintains a various sets of objects that can be used in subsequent MapReduce iterations. Spark is very popular, for example, in iterative machine learning algorithms.

2.1.2. Directed Graph Computing Platforms. Another relevant type of processing platforms to consider is the one based on directed graphs. This is a more general model than MapReduce model as it allows arbitrary flows of data among computations. The model uses directed graphs to express data processing tasks (vertexes), and data dependencies (edges). The developer is left with the job of building the computation graph and providing a function to execute on edges.

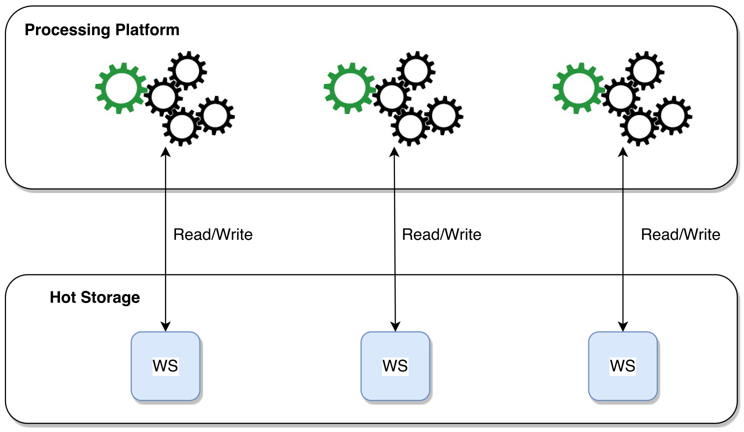

Fig. 3. Typical Processing Platform

Several Big Data platforms have been proposed using graphs to express computations and data flows. The goal for the rest of this section is to analyze some well-known platforms, namely Dryad [Yu et al. 2008], Naiad [Murray et al. 2013], Pregel [Malewicz et al. 2010], and MillWhell [Akidau et al. 2013], in order to understand their memory profiles. Although these platforms might not run on top of the same runtime environment (some platforms might run on top of a JVM, others might run on top of the .NET Common Language Runtime [Box and Pattison 2002]), all runtime environments must deal with automatic memory management, which is directly affected by the memory profile of each platform.

Dryad is a general-purpose (i.e., can produce any king of graph-based workload) execution engine. Dryad allows the definition of distributed applications that organize computation in edges and communication as data channels. The platform provides automatic application scheduling, handles faults, and automatically moves data along edges into vertexes. Dryad application developers can specify an any kind of computation graph (which must be directed and acyclic) to describe application's data flows, and the computation that takes place at vertexes using subroutines.

Naiad is, similarly to Dryad, an execution system for distributed data-parallel applications. However, as opposed to Dryad, it allows the definition of directed graphs with cycles. The main goal of Naiad is to provide a platform which processes a continuous incoming flow of data and to allow low-latency, on-the-fly queries over the processed data. This is specially important for a number of areas such as: data stream analysis, interactive machine learning, and interactive graph mining. Similarly to Dryad, communication between vertexes can be implemented automatically (this is typically left to the developer to decide) in multiple ways.

MillWhell is a similar approach for low-latency data streaming platform. It also provides developers with the abstraction of directed computing graphs which can be built using arbitrary and dynamic topologies. Data is delivered continuously along edges in the graph. MillWhell, similarly to the previous approaches, provides fault tolerance at the framework level (i.e., the programmer does not have to deal with faults, the framework automatically handles them).

2.1.3. Processing Platforms Memory Profile. The reader might notice that all processing platforms discussed so far can be reduced to a model where an arbitrary set of nodes perform some computation (task), and data flows (in and out) among computing nodes. Therefore, since most processing platforms can be reduced to a common representa-

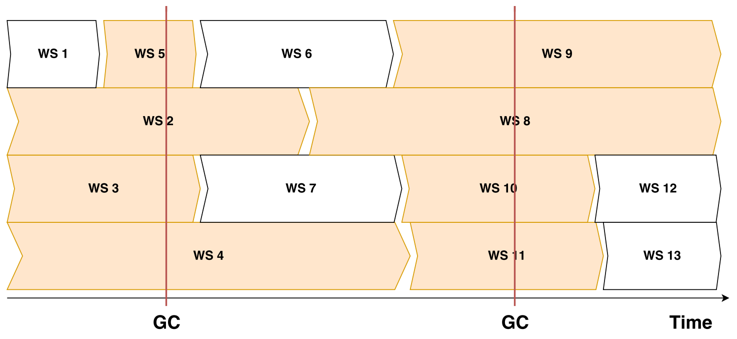

Fig. 4. Processing Platform Working Sets

tion, most of the problems of a specific platform will apply to the other processing platforms.

The memory profile for processing platforms is very characteristic. Each task usually has a Working Set (WS) which is loaded into memory (the WS can vary in size according to each platform and application), and is used to read and write throughout the execution of the task (see Figure 3). Each WS is specific to a single task and therefore, is considered garbage after the task is finished. If multiple tasks run in parallel, multiple WSs will be present in memory at the same time. Finally, each task can have different execution times, resulting in different WSs being present in memory for different amounts of time (Figure 4 illustrates this situation). If a GC is triggered, all WSs currently being used (represented as yellow boxes in Figure 4) will be handled by the collector, and, as discussed in Section 5, all data within is going to be moved to other memory location, producing a severe throughput degradation and high application latencies. White boxes represent WSs that are not currently being used, and therefore are ignored by the collector.

The practical effect of this problem is present in many platforms. These platforms suffer from high GC interference, hindering application throughput and/or latency. For example, in a platform with multiple tasks, each with dependencies from other tasks, GC can turn nodes into computation stragglers. Other consequence is the increased latency, result of long GC pauses. In Section 4, these problems are further discussed: why processing platforms' memory profile stresses memory management, leading to throughput and latency scalability problems.

Programmers try to reduce the GC interference by using a number of techniques such as: i) delaying collections as much as possible, ii) using pools of objects that can be reused multiple times (to reduce object allocations), and iii) serializing multiple objects into very large arrays of bytes. These solutions, as discussed in Section 4, have very limited success.

2.2. Storage Platforms

A storage platform, in the most generic way, is a system that provides read and write operations to some managed storage. The platform can orchestrate a number of nodes to store data. Each node provides volatile but fast storage, and persistent but slow storage. Read and write operations may obey a variety of consistency models [Lamport 1978] (this topic, however, is not in the scope of this work). In this section, the most relevant types of storage platforms are briefly analyzed. The goal is to understand how these platforms work and, most importantly, their memory profile.

2.2.1. Distributed File Systems. A Distributed File System (DFS) is a storage system in which files/objects are accessed using similar interfaces and semantics to local file systems. Therefore, DFSs normally provide an hierarchical file organization which can be accessed using basic file system primitives such as open, close, read, and write. It is often the case that DFSs can be mounted on the local file system.

The Hadoop Distributed File System (HDFS) is a very popular example of a Big Data DFS. Inspired by the Google File System [Ghemawat et al. 2003], HDFS aims at providing an efficient approach to access large-scale volumes of data. It uses a centralized entity which stores metadata and many data nodes to store all the files. HDFS, which integrates with Hadoop MapReduce, also employs several important performance optimizations that do not fall within the scope of this document.

2.2.2. Graph Databases. A graph database is a king of storage platform that provides a graph management interface for accessing graphs stored within it [Robinson et al. 2013]. With the evolution of data, many companies soon started to represent their application domains using graphs, which, for some applications such as social networks, gives a much more intuitive representation than other data models. Additionally, these systems provide efficient graph computing/search engines which enable applications to perform queries or even modify the graph in a very efficient and scalable way. Several graph databases have been developed, but only two representative examples are addressed in this section.

Titan[2] is a distributed graph database featuring scalable graph storing and querying over multi-node clusters. Titan is a very versatile solution as it can use several storage back-ends, for example Cassandra (see Section 2.2.3), and exports several high level APIs. Titan is often used with Gremlin[3], a graph traversal language.

Another example of a graph database is Neo4J [Van Bruggen 2014]. As opposed to Titan, Neo4J is a centralized graph databases that offers applications a set of primitives to build and manage graphs.

2.2.3. Key-Value and Table Stores. The last two types of Big Data storage platform to consider are key-value stores and table stores. For the sake of simplicity, within this section, it is assumed that the only difference between both types of storage platform is the way data is presented to the application (i.e., the interface): as a distributed key-value store, (similar to a Distributed Hash Table), or as a table store (in which information is formatted in rows and columns). Additionally, only two representative platforms are discussed: Cassandra [Lakshman and Malik 2010], and HBase [George 2011] (based Google's BigTable [Chang et al. 2008]). There are many other platforms (such as Dynamo [DeCandia et al. 2007] , OracleDB[4], MongoDB [Chodorow 2013], etc.) but those are not addressed since the principles behind all these solutions are very similar.

HBase is a distributed table-oriented database. It is inspired by Google's BigTable and runs on top of Hadoop MapReduce and HDFS. HBase provides strictly consistent data access and automatic sharding of data. HBase uses tables to store objects in rows and columns. To be more precise, applications store data into tables which consist of rows and column families containing columns. Each row can then include different sets of columns and each column is indexed with a user-provided key and is grouped into column families. Also, all table cells are versioned and their content is stored as byte arrays.

[2]Titan's web page can be accessed at http://thinkaurelius.github.io/titan/.
[3]Gremlin's web page can be accessed at https://github.com/tinkerpop/gremlin/wiki.
[4]OracleDB's web page can be found at https://www.oracle.com/database/index.html

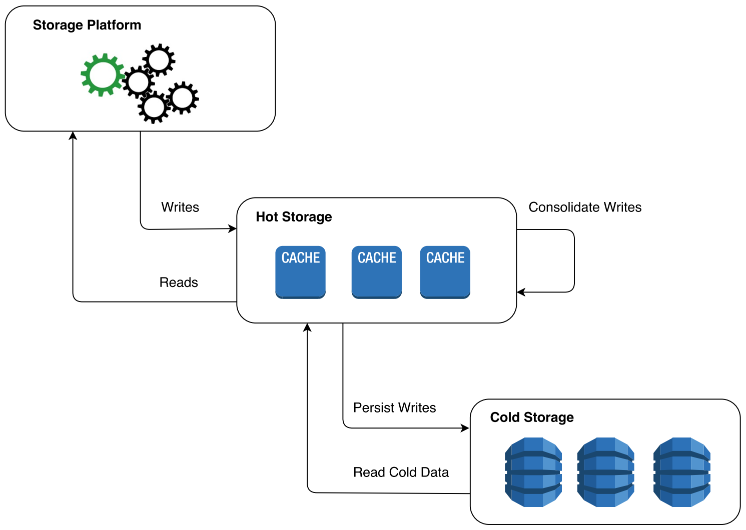

Fig. 5. Typical Storage Platform

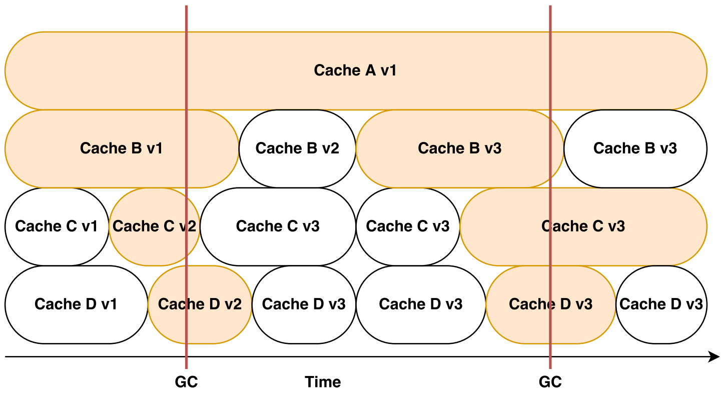

Fig. 6. Storage Platform Caches

Apache's Cassandra is a distributed key-value store. It is designed scale to very large amounts of data, distributed across many nodes. while providing a highly available service with no central point of failure (as opposed to HBase, Cassandra has no centralized master entity). The major difference between Cassandra and HBase lies on the data model provided by both solutions. Cassandra provides a distributed key-value store in which where columns (or values) can associated to specific keys. In Cassandra, one cannot nest column families but can specify consistency requirements per query (which is not possible in HBase). Moreover, Cassandra is write-oriented (i.e., the platform is optimized for write intensive workloads) whereas HBase is designed for read intensive workloads.

2.2.4. Storage Platforms Memory Profile. In general, storage platforms take advantage of fast/hot storage to keep caches of recently read or written objects while all remaining objects are stored in slow/cold storage (see Figure 5).

Similarly to processing platforms, storage platforms have a very specific memory profile. These platforms tend to cache as much objects in hot storage (usually DRAM) as possible in order to provide fast data access, and to consolidate writes. For example, in Cassandra, the result of write operations is cached in large tables in memory in the hope that future read or write operations will use/overwrite the same result thus avoiding a slower access to disk (this is commonly known as write consolidation). Multiple caches can coexist in memory at the same time, and may have different eviction policies (usually limited by the available memory). According to the authors' experience, caching data in memory to avoid slow disk accesses and to consolidate writes is a frequent technique across many storage platforms.

By aggressively caching data, storage platforms keep many live (reachable) objects in memory, leading to severe GC effort to keep all objects in memory (this problem is further discussed in Section 5). This produces the same problem discussed in Section 2.1.3, i.e., during a collection, all objects belonging to all active caches will be handled by the collector (they are moved to other memory location). Figure 6 illustrates this problem; active caches upon collection (represented in yellow) will be moved to other memory location (caches represented in white are not being used anymore and therefore are not considered by the collector). In this scenario, GC will lead to long applications pauses, directly increasing the platform latency (for example, read or write operation latency in Cassandra) and reducing throughput. In Section 4, these problems are further discussed: why storage platforms' memory profile stresses memory management, leading to latency and throughput scalability problems.

Naive solutions such as: i) severely limiting the size of the memory heap, and ii) reducing the number of requests to handle per second will not only reduce the throughput but will not solve the problem (i.e., it will only soften its effects).

3. MEMORY MANAGEMENT BACKGROUND

Memory management (both automatic and manual) is the process of managing memory. In particular, there are two special concerns regarding memory management: i) provide memory when requested (memory allocation), and ii) free unused memory for future reuse (memory deallocation). This is a classical problem in every Operating System (OS) [Tanenbaum 2007]. Nevertheless, memory management is also a fundamental problem for the JVM since it has to automatically manage memory, which is previously allocated by some underlying OS, and is necessary for the end-user application to run (it acts as an intermediary management system between the underlying OS and the application).

3.1. Background Concepts

Before delving further on how memory is managed by the JVM, it is important to provide some background concepts and explain how memory is structured in a Java application. The first important concept to introduce is the heap. From the JVM point of view, a heap is a contiguous array (or set of arrays) of memory positions which may be occupied or free. These memory positions are used to store objects. An object is a contiguous set of memory positions allocated for the end-user application to use. An object is divided into fields (or slots) which contains a reference or some other scalar non-reference type (an integer, for example). A reference is either a pointer to a heap object or the distinguished value, *null*.

In most applications, many objects populate the heap. Therefore, the heap is often characterized as a directed graph where nodes are application objects and edges are references from other objects (or from a root). A root is a reference held by the JVM that points to an object inside the object graph. There are several root references and the objects pointed by these references are named root objects. Examples of root objects include global variables, and variables held in CPU registers.

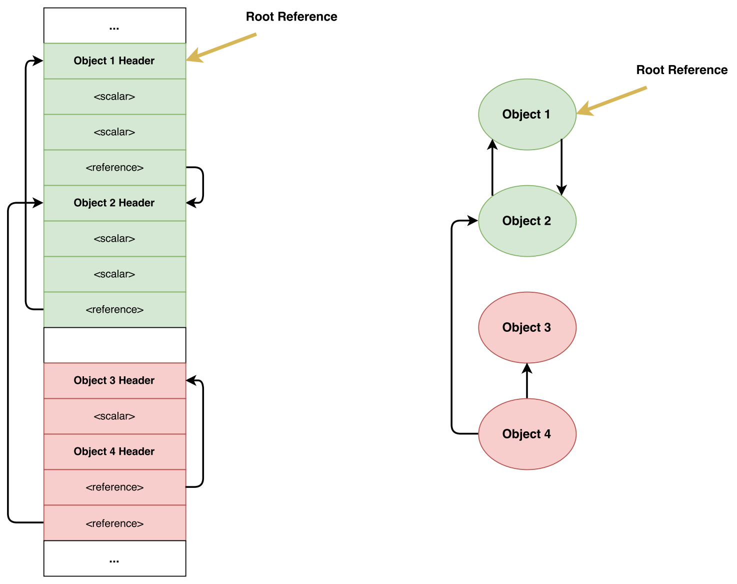

Fig. 7. Java Memory Heap (left) and the corresponding Java Object Graph (right)

Objects in the object graph can be identified as reachable (or live) or unreachable (or dead). An object is said to be live if there is a path of objects (such that a reference from one to the other exists) starting from any root object that reaches the object. On the other hand, if there is no path from any root object to the object, the object is considered dead and its memory should be collected for future reuse.

Figure 7 presents the concepts introduced in this section. On the left, a Java heap is presented as a continuous array of memory positions with objects. Each objects contains several fields with scalar or reference types. The corresponding object graph is presented on the right. Reachable (live) objects are represented in green while unreachable (dead) objects are represented in red.

Following the terminology introduced by [Dijkstra et al. 1978], a garbage-collected Java program is composed by two elements: i) a mutator, and ii) a collector. The mutator represents the user application which mutates the heap by allocating objects and mutating these objects (changing references and fields). On the other hand, the collector represents the garbage collector code which manages memory.

3.2. Garbage Collection in the JVM

The task of managing memory in a JVM is handled by the garbage collector (GC), which is responsible for several tasks: i) allocate memory for new objects, ii) ensure that all live objects are kept in memory, and iii) collect memory used by objects that are no longer alive (garbage). The GC is therefore a set of algorithms that hide most memory management issues from higher level languages that run on top of the JVM (e.g. Java, Groovy, Scala, etc.). The use of automatic memory management via GC was a choice taken by Java creators [Gosling 2000] right from the beginning and it is not in the scope of this work neither to motivate the use of GC, nor to present its advantages or disadvantages regarding explicit memory management (for this discussion, please refer to [Jones et al. 2011]).

It is important to notice, however, that the GC does not solve all memory problems. For example, nothing prevents the application from: i) keeping references to data that is not going to be used ever again, or ii) allocating objects indefinitely (and keeping references to all of them) until the JVM runs out of memory. In such scenarios, there are no unreachable objects and therefore, the GC cannot free any memory, resulting in an out of memory error that eventually shuts down the application.

3.3. GC Properties

There are many GC algorithms, with different implementations, many of which using different approaches to collect dead objects. To better understand the desirable properties and trade offs of each GC algorithm, it is important to point out the most critical properties/factors that can be considered when comparing the performance of different GC algorithms:

— Safety. A safe collector never reclaims the storage of live objects;
— Throughput. Throughput is a performance measure for user applications. The goal for every GC is not to decrease the application throughput but to increase it if possible (compared to not using automatic memory management);
— Completeness. A complete collector is one such that all garbage is eventually reclaimed. This is a special concern for reference counting approaches (described in 4.2), which, by design, are not complete;
— Promptness. High promptness means that the collector takes little time to reclaim garbage (once such garbage is created). On the other hand, low promptness characterizes GCs that take a long time to reclaim garbage (after such garbage is created);
— Pause time. How much time the application must stop to let the GC execute. The time that the application is stopped is called pause time. For the duration of the pause time, no application task can be running;
— Space overhead. The amount of space required to perform GC. For example, copying collectors usually need more memory (to perform a clean copy) than compacting collectors (which only moves the object to the beginning of the heap);
— Scalability. A GC is considered scalable if an increase in the number of objects in memory does not compromise or lead to a significant performance decrease in any of the previous metrics.

4. CLASSIC GARBAGE COLLECTION ALGORITHMS

Having described the basic concepts concerning general GC, this section is dedicated to study the local GC problem, i.e., memory management performed on a single process' address space. The study of distributed memory management, distributed garbage collection, is out of the scope of this paper.

The study starts by presenting the most commonly used algorithms for allocating memory. Then the three main families of collectors (reference counting, tracing, and partitioned/hybrid) are presented, followed by a discussion of typical GC design choices.

4.1. Memory Allocation

The two main aspects of memory management, memory allocation and memory reclamation, are tightly linked in a sense that the way memory is reclaimed places constraints on how it is allocated and vice-versa. While a memory allocator needs a collector to identify free memory positions, the collector may also need the allocator to provide free memory to enable some GC operation (see Section 4.5 for more details).

Sequential allocation is the first and most simple allocation algorithm. It uses a large free chunk of memory from the heap. Only two pointers need to be maintained between allocations: a free pointer (which limits the last allocated fraction of the heap), and a limit pointer (which points to the last usable memory position). When an allocation takes place, the free pointer is incremented by the number of requested blocks. If there are not enough blocks between the free pointer and the limit pointer, an error is reported and the allocation fails. This algorithm is also called bump pointer allocation because of the way it ”bumps” the free pointer.

Despite its simplicity, the algorithm is efficient and provides good locality [Blackburn et al. 2004]. However, as time goes by, some objects become unreachable while others are still reachable. This results in many small allocated blocks interleaved with many unallocated blocks, i.e., high fragmentation.

The alternative to sequential allocation is free-list allocation. In a basic free-list allocation algorithm, a single data structure (a list) holds the size and location of all the free memory blocks. When some memory is requested, the allocator goes through the list, searching for a block of memory that fits the requested size, and respecting an allocation policy. Typical allocation policies are one of the following:

— first-fit, the simplest approach. The allocator stops searching the list when it finds a block with at least the required number of memory blocks. The allocator might split the free chunk in two if the chunk is larger than required;
— next-fit, a variant of the first-fit algorithm. It starts searching for a block of suitable size where the last search ended. If the allocator reaches the end of the list, it restarts the search from the beginning of the list.
— best-fit finds the free block whose size if the closest to the request. The idea is to minimize memory waste and avoid splitting large memory blocks unnecessarily. This is the policy behind the well know Buddy [Knowlton 1965] allocation algorithm.

Even with free-list allocation, the user might notice that the time it takes to allocate memory is linear with the size of the memory (heap). If the size of the memory grows significantly, the time needed to allocate some blocks will become prohibitive. To cope with this problem, there are some optimizations. The first optimization consists on using balanced binary trees to improve worst-case behaviour from linear to logarithmic in the size of the heap. Hence, instead of going through all elements of a list, the allocator can traverse the tree searching for the block with the requested size. This technique is also known as fast fit allocation [Stephenson 1983].

The second optimization comes from the fact that much of the time consumed by a free-list allocator is still spent searching for free blocks of appropriate size. Therefore, using multiple free-lists, whose members are grouped by size, can speed allocation. By using enough lists with appropriate block ranges it is possible to achieve allocation in almost constant time.

So far, the described techniques and algorithms manage the whole heap, i.e., if some memory needs to be allocated, the allocator must preserve the integrity of all allocation data structures by using atomic operations or locks. In a highly threaded environment, this is a serious bottleneck. The common solution to cope with this problem is to give each thread is own allocation area, the thread local allocation buffer (TLAB) [Jones and King 2005; Gay and Steensgaard 2000]. This way, each thread can allocate memory from its TLAB independently. Threads may always request new TLABs from the heap if the current TLAB runs out of memory. Only interactions with the global memory pool (heap) are synchronized (e.g., if some object does not fit the TLAB, it must be allocated directly on the heap). Dimpsey [Dimpsey et al. 2000] measured substantial performance improvements in a multi-threaded Java system using a TLAB for each thread. It was also possible to conclude that TLABs tend to absorb almost all allocation of small objects. Most accesses to the heap turned out to be requests for new TLABs. TLABs can be used for sequential allocation or free-list allocation.

Having described how current GC implementations handle memory allocation, the next sections are focused on how memory is reclaimed.

4.2. Reference Counting Algorithms

As the name suggests, reference counting algorithms (first introduced by [Collins 1960]) literally count references to objects. Such algorithms are based on the following invariant: an object is considered alive if and only if the number of references to the object is greater than zero (note that these algorithms erroneously consider objects included in cycles of garbage as live objects). Therefore, to be able to know if an object is alive or not, reference counting algorithms keep a reference counter for each object. Reference counting is considered direct GC (as opposed to indirect GC which is discussed next) since it identifies garbage, i.e., objects with no incoming references.

ALGORITHM 1: Reference Counting

```
Procedure Allocate($objType$)
    $object$ = allocateObject($objType$)
    resetCounter($object$)
    return $object$

(atomic) Procedure Mutate($parent, slot, newChild$)
    AddReference ($newChild$)
    DelReference ($parent.slot$)
    $parent.slot = newChild$

Procedure AddReference($object$)
    incrementCounter($object$)

Procedure DelReference($object$)
    decrementCounter($object$)
    if getCounter($object$) == 0 then
        for each child in childRefs($object$) do
            DelReference ($child$)
        end
        free($object$)
    end
```

Algorithm 1 presents a simple implementation of a reference counting algorithm. It is important to note that the `Mutate` operation must be atomic, i.e., there should not be multiple interspersed calls to this operation, otherwise race conditions could occur, resulting in erroneous updates of reference counters.

Contrary to reference tracing algorithms (see Section 4.3), reference counting algorithms provide some interesting properties: i) the GC overhead is distributed throughout the computation, i.e., is does not depend on the size of the heap, but, instead, on the amount of work done by the mutator; ii) garbage can be collected almost instantaneously (as the collector knows instantly when the number of incoming references reaches zero); and iii) it preservers cache locality (by not traversing the object graph and therefore destroying the application working set cache locality).

These advantages come with two problems: i) high overhead of maintaining track of a counter for each object (which incur into synchronized operations whenever it needs to be updated); and ii) reference counting is not complete, i.e., not all garbage is collected (particularly, cyclic garbage).

To cope with the first problem, [Blackburn and McKinley 2003] propose a useful taxonomy of solutions:

- deferred reference counting, delay the identification of garbage to specific periodic checkpoints. This way, some synchronization steps are avoided;
- coalescing, a technique based on the hint that many reference count adjustments are temporary and therefore, can be ignored (for example, GC operations on local variables). With coalescing, only the first and the last state of an object filed should be considered. Reference counting increments or decrements should only be considered at specific checkpoints, thus safely discarding many other intermediary states;
- buffered reference counting, in which all reference count increments and decrements are buffered for later processing.

All these three approaches try to reduce some of the synchronization overhead inherent to updating global reference counters. To deal with the second problem (completeness), the most widely used solution is to perform trial deletion. Trial deletion is a technique that requires a backtracking algorithm to visit objects that are suspected to contain cyclic garbage. The main idea behind the algorithm (described in [Paz et al. 2007]) is to check if cyclic garbage is uncovered when some reference is deleted. If the reference count of the object whose reference is deleted reaches zero, it exposes the existence of cyclic garbage.

4.3. Reference Tracing Algorithms

Reference tracing algorithms rely on traversing the object graph and marking reachable objects. Reference tracing is quite straightforward; objects that are marked during reference tracing are considered alive. All memory positions that are not marked, are considered to be garbage and will be freed. Hence reference tracing is considered indirect GC, i.e., it does not detect garbage but live objects instead. Typical implementations of reference tracing collectors are also known as mark-and-sweep collectors [McCarthy 1960].

Algorithm 2 present a basic implementation of a mark-and-sweep collector.

This mark-and-sweep algorithm, despite its simplicity, has some problems regarding the need to stop the mutator from changing the object graph during GC (this is discussed in more detail on Section 4.5). To cope with this problem, a second mark-and-sweep implementation, which uses the tricolour abstraction [Dijkstra et al. 1978], is used. This approach, also called tri-color marking, provides a state for each object in the object graph. Hence, each object can be in one of the following states:

- white, object not reached, the initial state for all objects;
- black, object that has no outgoing references to white objects. Objects in this state are not candidates for collection;
- gray, object that still has references to white objects. Gray objects are not considered for collection (eventually, they will turn black).

Tri-color marking starts by placing all root objects in the gray set (set of gray objects) and all remaining objects in the white set (set of white objects). The algorithm then proceeds as follows: while there are objects in the gray set, pick one object (from the gray set), move it to the black set (turning it into a black object), and place all objects that it references in the gray set (turning objects into gray objects). In the end, objects in the black set are considered alive. All other objects (white objects) can be garbage-collected. Using the aforementioned steps, the algorithm keeps the following invariant: no black object points directly to a white object.

By using a state for each object, the collector can remember which objects were already verified and therefore, it can run incrementally or even concurrently with the mutator. However, care must be taken to track situations where the mutator writes

ALGORITHM 2: Mark and Sweep

```
Procedure MarkRoots()
    for each object in roots do
        if ! isMarked(object) then
            setMarked(object)
            push(objStack)
            Mark ()
        end
    end

Procedure Mark()
    while ! isEmpty(objStack) do
        object = pop(objStack)
        for each child in childRefs(object) do
            if ! isMarked(object) then
                setMarked(child)
                push(objStack, child)
            end
        end
    end

Procedure Sweep(heapStart, heapEnd)
    curr = nextObject(heapStart)
    while curr < heapEnd do
        if isMarked(curr) then
            unsetMarked(curr)
        else
            free(curr)
        curr = nextObject(curr)
    end
```

a reference coming from a black object towards a white object (this would break the algorithm invariant).

A final remark about this algorithm is that, although sweeping needs to search the whole heap (for collecting white objects), this task can be delayed and performed by the allocator [Hughes 1982] (this technique is called lazy-sweeping).

4.4. Computational Complexity of Memory Collection Algorithms

After analyzing the two classic algorithms to collect memory, it is important to analyze how each algorithm behaves in large scale environments. In particular, two dimensions should be considered: i) number of live objects, and ii) mutator speed (e.g., the speed at which the mutator dirties memory).

4.4.1. Number of Live Objects. From Algorithms 1 and 2, it can be concluded that reference counting algorithms do not depend on the number of live objects while tracing algorithms do. In other words, the complexity of reference counting algorithms on the the number of live objects is constant while the complexity of tracing algorithms on the number of live objects is linear. This simply comes from the design of each algorithm. While one needs to traverse through the object graph, the other does not (reference counting).

4.4.2. Mutator Speed. From the algorithms' description, it can be concluded that reference counting algorithms do depend on the mutator speed while tracing algorithms do not. Hence, the complexity of reference counting algorithms on the mutator speed is linear while the complexity of tracing algorithms is constant on the mutator speed.

In sum, both algorithms depend linearly on some important metric (either mutator speed, or size of the number of live objects). This leads to a clear trade-off where there is no algorithm that is superior on both metrics; therefore, the appropriate algorithm must be selected according to the target application/workload.

Since memory management algorithms always resort to one of the two presented algorithms (reference tracing or reference counting), recent memory management algorithms' complexity can be obtained by identifying the classic algorithm running underneath.

4.5. Design Choices

Both approaches for GC, tracing and reference counting, can be designed and optimized for different situations. For example, in a multi-core architecture, one would want to take advantage of multiple cores to split the GC task among several cores (to achieve higher performance). Another interesting and challenging scenario is to run the GC concurrently with the application (mutator). In a multi-core architecture, mutator threads can run concurrently with collector threads, therefore increasing the responsiveness and decreasing the application pause times. Yet another possible optimization is to periodically clean (by copying or compacting) areas of memory with low live data or high fragmentation.

To sum up, each of the previously presented approaches to GC (tracing or counting) can be customized according to several design choices:

— Serial versus Parallel — The collection task can be executed by one or several threads. For example, in reference tracing, traversing an object be can done in a serial mode (single thread) or in a parallel mode (multiple threads). It is clear that a parallel implementation of either reference tracing or reference counting can harness multiple execution flows on available CPU cores but it also requires a more careful implementation due to complex concurrency issues;

— Concurrent versus Incremental versus Stop-the-World — Stop-the-World GC means that most of the GC work is done when no mutator task is running. This means that all application threads are stopped periodically to enable GC to run. To minimize the time the application is stopped (pause time), one could implement an incremental GC, in which the collection is done in steps, per memory page, per sub-heap, per sub-graph. If the goal is to mitigate application pauses, it is possible to implement a concurrent GC, where both mutator and collector run at the same time.
It is important to notice some trade-offs regarding GC implementations. Stop-the-World implementations are the simplest because there is no need to synchronize mutator and collector threads. Yet, it is the best option for a throughput oriented applications because it does the collection in only one step and lets the application run at full speed the rest of the time. The same is not true for incremental or concurrent GCs. These are targeted to applications with low latency requirements. As the collection is done in steps, it might require more time to collect all garbage. The necessary synchronization between mutator and collector threads is also a source of overhead compared to Stop-the-World implementations. The use of read barriers [Baker 1978] and/or write barriers [Nettles et al. 1992] are common approaches to synchronize mutator access to objects being collected. In both approaches, some mutator's reads and/or writes are checked for conflicts before the operation takes effect.

— Compaction versus Copying versus Non-Moving — The last design decision is about whether or not to move live objects in order to reduce fragmentation. Fragmentation occurs when objects die and free space appears between live objects. The problem is that, with time, most free memory is split into very small fragments. This leads to three serious problems: i) locality is reduced, i.e., objects used by the application are scattered through all the heap; ii) objects, which cannot fit inside memory fragments cannot be allocated unless more memory is requested by the application; iii) the total amount of memory used by the application is high (since fragments between live objects force the application to allocate more memory to keep creating objects).
To solve the fragmentation problem, two typical solutions can be employed: i) compaction, and ii) copying. Both techniques require live objects to be moved and grouped to reduce fragmentation. Compaction is frequently used to move all live objects to the start of some memory segment (for example, a memory page); copying, on the other hand moves live objects from one memory segment to another. Although requiring more memory, copying allows the application to group objects from multiple memory pages (with few live objects) into a single page. Pages from where objects were copied can be freed. The same does not occur with compaction, where multiple pages with few live objects can still coexist.
The decision of when to apply compaction or copying is also an interesting research problem (that falls outside the scope of this work). Typical solutions involve measuring the percentage of: i) fragmentation, ii) live objects, and iii) memory usage for each memory segment. Only if there is few live objects or high fragmentation, the cost of copying or compacting will compensate the overhead of moving live objects [Soman et al. 2008].

4.6. Partitioned/Hybrid Algorithms

So far, only monolithic approaches to GC have been described, i.e., the whole heap is collected using one GC algorithm only. However, nothing prevents heap partitioning into multiple partitions/sub-heaps and apply different GC approaches. The motivation behind these hybrid algorithms resides in the fact that, different objects might have different properties that could be explored using different GC approaches.

The idea of heap partitioning was first explored by [Bishop 1977]. With time, several partitioning models have been proposed:

— partitioning by mobility, where objects are distinguished based on their mobility, i.e., objects that can be moved and objects that can not be moved or are very costly to move;
— partitioning by size, where objects of certain dimensions are placed in a separate object space, to prevent or minimize fragmentation;
— partitioning for space, where objects are placed in different memory spaces so that the overhead applying GC techniques such as copying can be reduced. To this end, each memory space can be processed separately;
— partitioning by kind, where objects can be segregated by some property, such as type. This can offer some benefits as properties can be assessed using the object's memory address (thus avoiding loading the object's header from memory);
— partitioning for yield, the most well-known and widely used partitioning technique, where objects are segregated to exploit their life cycles (i.e., group objects by their estimated life time). Studies have confirmed that Java object's lifetime follows a bi-modal distribution [Jones and Ryder 2006; 2008] and that most objects die young [Ungar 1984];
— partitioning by thread, where objects are allocated in thread-local heaps, similar to a TLAB [Jones and King 2005; Gay and Steensgaard 2000]. Such object place-

ment leads to high concurrency improvements since only one mutator thread must be stopped at each time to collect garbage;

For the remainder of this section, a deeper look is taken at the most used type of heap partitioning, generational GC [Lieberman and Hewitt 1983], where partitioning considers the age of the object. As already discussed before, Java objects' life time tends to be split between long lived objects and short lived objects. Using this property, it is possible to split objects according to their life cycle and use different sub-heaps (or generations) for long and short lived objects. The age of an object can be simply the number of collections the object has survived.

Considering that short lived objects turn into garbage very soon, the young generation (sub-heap where short-lived objects are allocated) will most likely be occupied with very few live objects very quickly. On the other hand, the old generation will take much longer to accumulate garbage. Using this knowledge, generational GCs are able to reduce the application's pause time by collecting more often the young generation (which is usually small) and less often collecting the old generation (which is usually large).

Generational collection can also improve throughput by avoiding processing long-lived objects too often. However, there are costs to pay. First, any old generation garbage will take longer to be reclaimed. Second, cycles comprising objects in multiple generations might not be reclaimed directly (as each GC cannot determine if references going to other generations are part of a cycle). Third, generational GCs impose a bookkeeping overhead on mutators in order to track references that span generations, an overhead hoped to be small compared to the benefits. For example, in a scenario with only two generations (young and old), these references are typically coming from old to young generation and therefore are part of the young generation root set (also called remember set), necessary to allow the young generation to be collected. These references can be maintained by using a write barrier [Ungar and Jackson 1988; Moon 1984; Appel 1989] or indirect pointers [Lieberman and Hewitt 1983].

To deal with the possible high pause time incoming from collecting old generations, which might be large, [Hudson and Moss 1992] propose a new approach, the train algorithm. In this algorithm the mature object space is divided into cars (memory segments) of fixed memory size. GC collects at most one car each time it runs. Additionally, objects are moved (from one car to another) in order to cluster related objects (i.e., with similar lifetimes). When some car is empty, it can be recycled. This way, using the train algorithm, the application pause time drops significantly because only a fraction of the old generation is reclaimed at a time. Splitting objects into cars, however, introduces some complexity to track inter-car references.

5. GARBAGE COLLECTION ALGORITHMS FOR BIG DATA ENVIRONMENTS

So far, the two classic approaches to collect garbage (reference counting and tracing) have been addressed. These algorithms, however, show several problems which limit the scalability of today's Big Data platforms.

Starting with reference counting algorithms, there are two main problems. First, these algorithms are not complete and therefore need extra techniques to collect cycles of garbage (such as trial deletion). Trial deletion, comes at a very high cost in terms of computational cost (reducing application's throughput) since it has to simulate the deletion of a possibly large number of objects. The larger the object graph is, the longer trial deletion can take. Second, there must be write barriers on all reference modification instruction (to account for new and deleted references). This obviously incurs into major application throughput penalties since, after each write, the application is

Table I. Object Copying in Copying Collectors

Number of Cores	Total Working Set Size (GB)	Object Copy Time (ms)
4	2	300
8	4	700
16	8	1500
32	16	3100
64	32	6300
128	64	12700

stopped and the GC steps in to fix reference counters. Even if optimization techniques are utilized, throughput is still largely penalized [Jones et al. 2011].

For these two reasons (which impose a severe impact on application throughput), reference counting algorithms are not used in most production JVMs such as the OpenJDK HotSpot JVM (the most widely used JVM implementation).

Tracing algorithms, on the other hand, are the most used type of GC algorithms nowadays. Implementations such as the Concurrent Mark Sweep (CMS), a widely used production GC available in HotSpot, combine a set of techniques to optimize the GC process. CMS, in particular, is a generational collector (i.e., the heap is partitioned for yield). There are two generations: young and old. The fist (young generation) is where all new objects are allocated. A parallel copy collector periodically traverses the young generation and copies objects of a certain age (implementation specific) to the old generation. The old generation uses a parallel, concurrent, and non-moving Mark Sweep collector to reclaim objects residing in the old generation. This collector has been shown to offer acceptable performance (throughput and pause time) for many applications with not very demanding throughput and pause time requirements.

However, when it comes to most Big Data platforms, with massive amounts of objects in memory and with high throughput and pause time requirements to cope with, CMS can be a limiting factor mainly because of three major problems.

First, tracing algorithms (and therefore CMS as well), have to traverse the whole heap to identify garbage. This becomes a problem if the number of live objects grows to very large numbers. In such scenarios, the process of tracing the heap (which is concurrent with application threads) can take so long that eventually memory is exhausted and therefore a full collection (that collects the whole heap) is triggered. These collection cycles can take dozens or even hundreds of seconds to collect all objects in memory [Gidra et al. 2013]. This obviously has a severe impact on throughput and pause time for the running applications.

Secondly, the other problem is directly related to the memory profiles analyzed in Sections 2.1.3 and 2.2.4. This problem is not directly linked to CMS, but instead, is present in any copy collector. As previously stated, both processing and storage platforms can keep many live objects in memory: working sets in the case of processing platforms, and database table caches in the case of storage platforms. Also remember that all these objects (belonging to caches and working sets) are allocated in the young generation which, when full, is collected. During this process, all live objects are copied to the old generation. This copying process is slow and is limited to the memory bandwidth available on the hardware. Therefore, and since processing platforms keep their active working sets alive (usually one per task/core) and storage platforms keep their caches alive (usually one per table/database), many live objects will be copied within the heap, leading to frequent and length full collections (reducing the application throughput and increasing the application pause time). In other words, the well established assumption that most objects die young is not true for a wide range of Big Data platforms [Bruno et al. 2017; Nguyen et al. 2015; Cohen and Petrank 2015].

Consider the following instantiation of this problem. Assume that a particular processing or storage platform uses a number of cores/threads to handle working sets of

500 MB in size. Also, imagine that the young generation is sized to be able to contain all the working sets (so as to minimize the number of young generation collections). Finally assume that each working set might take different amounts of time to process and that DDR 3 memory is used (which has 10 GB/s of memory bandwidth). Table I presents the estimated copy times (i.e., the amount of time during a GC cycle that is spent copying objects in memory). As the number of cores increases, the total size of the working set increases, leading to increased object copy times (since every time a GC cycle occurs, all the working set will be copied around in memory).

As the results suggest, object copying time escalates very quickly, leading to unaffordable application pause times. This is a direct consequence the lack of scalability in memory bandwidth compared to the scalability seen in the number of cores and size of the memory. In other words, the number of cores and size of the memory grows faster than the memory bandwidth. This is a serious problem since almost all industrial collectors rely on object copying (which is limited to the amount of available network bandwidth).

To further aggravate this problem, there is no simple solution:

— Heap Resizing - does not solve the problem because, eventually, the heap will become full, and therefore, the collector will stop the application and copy all live objects. In other words, increasing the size of the heap only pushes the problem further away but does not solve it;
— Reduce Working Set Size - the size of the working set can be reduced down in order to make objects die faster (i.e., threads process and discard tasks faster). This can effectively reduce the amount of copying in the heap but it will bring overheads related to synchronization and coordination to manage massive amounts of fine-grained tasks/working sets.
— Reuse Data Objects - it is also possible to reuse data objects to avoid creating new ones to hold new data. This dramatically reduces the speed at which the application allocates objects and therefore, less collections are triggered. The main problem with this approach is that it forces programmers to use a very unnatural programming style, opposed to the normal object oriented programming style of allocating new objects whenever necessary;
— Using Off-Heap Memory - using Off-Heap memory is a way to allocate objects outside the GC-managed heap (only some runtime environments support this feature). This effectively reduce the GC-overhead but forces programmers to employ manual memory management, which is error prone and completely obviates the benefits of running on top of a managed runtime environment.

The third problem is fragmentation in the old generation. As objects with longer life-cycles[5] live, objects with shorter life-cycles (but already in the old generation) will become unreachable. This results in a highly fragmented old generation which leads to decreased locality and can even lead to situations where no more memory can be allocated (although there is enough free space) because of fragmentation.

To conclude, current collectors provided by production JVMs still present scalability challenges that need to be addressed. For this reason, several relevant solutions have been published to try to alleviate these problems. In the next sections, several of such solutions are presented and analyzed in detail. The analysis is divided into throughput oriented, and pause time oriented solutions, since most of these solutions are focused on improving or have the largest impact on one of these metrics.

[5]An object's life-cycle is a term used hereafter to refer to the moment of creation and collection of a particular object or set of objects. Thus, objects with similar life-cycles are created and collected approximately at the same time.

5.1. Throughput Oriented Memory Management

Several improvements have been proposed for reducing the negative impact that GC has on applications' throughput. This section studies some of the most recent and relevant GC solutions that try to increase the application throughput by removing the overhead introduced by automatic memory management (i.e., GC).

5.1.1. Broom. Broom [Gog et al. 2015] proposes the use of region-based memory management as a way to reduce the cost of managing massive amounts of objects usually created by Big Data processing platforms.

The authors want to take advantage of the fact that many objects created by processing platforms (Naiad, for this specific work) have very similar life-cycles. By knowing this, Broom enables platform developers to group all these objects whose life-cycles are similar in separate regions. These regions could be easily collected (including all the objects within) whenever the objects within these regions are not necessary anymore. In other words, and relating to the concepts introduced in Section 2.1, Broom stores objects of different working sets in different regions; after a task is complete, the working set is discarded and the region is freed (knowing that all objects within will not be used again).

Three types of regions are proposed: transferable regions, task-scoped regions, and temporary regions. Transferable regions are used to store objects that persist across tasks and can be used by different tasks across time. Task-scoped regions are meant to store objects belonging to a single task. Finally, temporary regions are used to store temporary objects; these objects cannot persist across method boundaries.

To avoid complex reference management between regions, Broom does not allow references from: i) objects inside temporary regions to objects inside task-scoped, ii) objects inside temporary regions to objects inside transferable regions, and iii) objects inside task-scoped regions to transferable regions. This way, objects that live for longer periods of time never reference objects with smaller life-times and therefore, no region is kept alive because of other region.

Using Broom prototype implemented for the Mono (a Common Language Runtime for Linux), the authors were able to reduce the task runtime of Naiad for up to 34%.

Despite the positive resuts, Broom presents some limitations: i) the programmer must have a very clear understanding of the objects' lifecycles in order to be able to group them properly into regions; ii) this is even aggravated by the fact that inter-object references are limited (objects from temporary regions cannot reference task-scoped regions, for example); iii) Broom is only a prototype used for Naiad, i.e., it only works with Naiad, meaning that it cannot be used with other Big Data platforms.

5.1.2. FACADE. FACADE [Nguyen et al. 2015] is a compiler framework for Big Data platforms. The proposed system takes as input any Big Data platform byte code and modifies the code to use native memory (off-heap) instead of the GC-managed memory (on-heap). Native memory or off-heap is a way to access memory that is not managed by the GC. When using native memory, the programmer is responsible for allocating and deallocating memory (much like in a C/C++ application). The idea behind FACADE is that all the native memory code (potentially hard to code and to debug) is automatically generated and replaces regular Java code.

Using the transformed byte code, the platform is able to reduce the number of objects in the GC-managed heap memory, thus reducing the GC effort to keep these objects in the managed heap, leading to an increase in the application throughput. Relating to the concepts explained in Section 2.1, FACADE is pushing objects belonging to working sets to native memory (i.e., out of the reach of the GC).

The problem of avoiding GC by pushing objects into off-heap is that the programmer must explicitly collect all memory. In other words, FACADE must be able to collect all objects that are allocated in native memory. In order to solve this problem, FACADE requires the programmer to specify when a new working set must be created and when a working set can be collected (note that FACADE does not allow the existence of multiple separate working sets at a time). Therefore, this system is mostly appropriate for iteration-based processing platforms, whose working sets are discarded by the end of each task/iteration.

The authors successfully used FACADE to transform the byte code of seven Big Data applications across three Big Data platforms: GraphChi [Kyrola et al. 2012], Hyracks [Borkar et al. 2011], and GPS [Salihoglu and Widom 2013]. Results showed that the execution time can be reduced by up to 48 times.

The main drawback presented by this solution is its limitation regarding the range of workloads that can be used. Since FACADE only allows one working set (per-thread) at a time, it does not support non-iterative workloads such as the ones typically associated with storage platforms. In storage platforms, working sets (caches) are not bound to a single thread (while on processing platforms, processing tasks usually are) thus making it very difficult to use FACADE. Another related problem is the way FACADE requires programmers to identify when working sets start and finish. Between these two code locations, FACADE intercepts all allocations and places them in off-heap, meaning that programmers must remove all non-data objects from within these boundaries (working sets' start and finish). A final comment on FACADE's evaluation is that it is done using the Parallel Scavenge GC, an obsolete and unrealistic GC for Big Data platforms. Current GCs used in realistic OpenJDK production settings are usually CMS or G1. If evaluated against these collectors, FACADE would show smaller (but realistic) throughput improvements.

5.1.3. Deca. Deca [Lu et al. 2016] is an extended/modified version of Spark which tries to reduce the GC overhead present in Spark because of its massive creation of objects with very different lifetimes (i.e., some groups of objects may have a very short lifetime while other groups of objects might live for a long period of time). The authors propose a lifetime-based memory management so that objects are grouped according to their estimated lifetime.

Using this approach, objects created by the platform (which will potentially live for a long period of time) are serialized into large buffers thus avoiding continuous GC marking and tracing. By keeping the bulk of the data maintained in memory (by Spark) inside large buffers, Deca is able to reduce the GC marking and tracing overhead and therefore it is able to increase the platform throughput.

As with previous systems (such as FACADE), one problem of maintaining serialized versions of objects is how to keep their memory consistent while efficiently reading and writing to it. Deca solves this problem by pre-allocating large arrays where objects will fit into. To determine the size of these arrays, Deca estimates the size of each data object (actually it uses an upper bound of the size).

In practice, Deca is in a way similar to FACADE. Despite the fact that the first only works for Spark and the second works for any iterative workflow platform, both of them try to hide massive amounts of data objects from the GC in a way to avoid the GC overhead associated with keeping these objects in memory (namely the tracing overhead). Relating to the concepts introduced in Sections 2.1.3 and 2.2.4, Deca is pushing the working sets and intermediate data (similar to the caches present in storage platforms) into large buffers, away from the collector.

The authors were able to improve Spark throughput by reducing its execution time by up to 42 times, using workloads such as Connected Components, Page Rank, Word Count, among others.

Deca is, however, specific to a single processing system, Spark. In other words Deca cannot be used in other platforms. In addition, the technique used to modify Spark (allocating objects in large arrays) is often unpractical as object allocations happen in so many code locations (making it harder to change from heap allocations into array allocations), and therefore requiring a major rewriting the platform.

5.1.4. NumaGiC. NumaGiC [Gidra et al. 2015] presents several developments to improve GC performance in cache-coherent Non-Uniform Memory Access (NUMA) environments. The authors propose several mechanisms to reduce the amount of inter NUMA node reference tracing performed by GC threads. By improving reference tracing locality (i.e., only trace references local to the current NUMA node where the GC thread runs), NumaGiC is able to improve applications' throughput.

With this collector, objects are placed in specific NUMA nodes not only upon allocation but also upon copying (after a collection). The most appropriate NUMA node to place the object is determined using several policies:

— new objects are placed in the same NUMA node where the mutator thread that creates the object is running;
— the roots of a GC thread are chosen to be located mostly on the NUMA node where the GC thread is running;
— objects that survive a young collection are copied to the same NUMA node where the GC thread (that handles the object copying) is running;
— upon heap compaction, NumaGiC tries to maintain objects in the same NUMA node.

With these policies, it is still possible to end up with an unbalanced distribution of objects, i.e., some NUMA nodes can end up having most objects allocated in it. To solve this problem, GC threads running on different NUMA nodes steal work from other GC threads. If a GC thread finds a reference to an object residing in a remote NUMA node, it notifies the remote GC thread (running on the corresponding NUMA node) to process that object.

NumaGiC is implemented on the OpenJDK HotSpot 7 JVM as an extension of the existing Parallel Scavenge GC (wihch is similar to CMS but is not concurrent). The authors compared their new collector with NAPS [Gidra et al. 2013] (NUMA-aware Parallel Scavenge) using platforms such as Spark and Neo4J, and improved the throughput of those platforms by up to 45%. Nevertheless, it would be very interesting to confirm that the benefits obtained with Parallel Scavenge can also be obtained with concurrent GC (which is the most realistic setup nowadays) such as CMS or G1.

5.1.5. Data Structure Aware Garbage Collector. DSA [Cohen and Petrank 2015] is a collector which tries to benefit from the fact that particular objects are inside a data structure to improve collector's performance and therefore, alleviate the GC overhead on the platform's throughput. The motivation behind DSA is that: i) there are many Big Data platforms which are data structure oriented (mainly storage platforms), and ii) a collector able to distinguish objects that are inside a data structure (and therefore are alive) would avoid handling (tracing for example) these objects in the hope that a large portion of the overhead caused by the collector would be eliminated.

The programmer is required to explicitly tell DSA: i) which classes are part of a data structure, and ii) when objects belonging to a data structure should be collected. If the programmer fails to report the deletion of an object or reports false information, the correctness of the collector is not compromised, the only consequence is a degradation in the performance of DSA to collect such objects.

Objects belonging to a data structure are allocated in a separate heap area, away from other regular objects. According to the authors, this also provides locality for the objects inside the data structure (that are collocated in the same heap area). From the collector point of view, objects belonging to a data structure are considered as root objects. Tracing is improved by having most data structure objects in the same heap area (benefits from locality).

Relating this work with concepts introduced in Section 2.2.4, DSA is pushing caches into a separate heap to improve locality and therefore improve the performance of the collector and platform.

DSA is implemented on JikesRVM [Alpern et al. 2000] (a research JVM) and it was tested with KittyCache, SPECjbb2005, and HSQLDB. DSA improved throughput up to 20% using KittyCache, 6% using SPECjbb2005 and 32% using HSQLDB.

However, DSA is implemented in a research JVM and not a production JVM, which difficults comparing its results with other approaches available. Furthermore, DSA requires the programmer to inform the JVM: i) of all the classes that should go into a separate space (data structure space), and ii) whenever an object inside a data structure is removed. This requires a lot of effort from the programmer. An additional problem is that some objects belonging to a data structure class might never go into a data structure (i.e., can be temporary objects). This breaks the main goal of the paper and DSA has no apparent way of preventing this.

5.2. Pause Time Oriented Memory Management

Having discussed several throughput oriented systems, GC solutions whose main goal is to reduce the application latency introduced by automatic memory management (i.e., GC) are now presented.

5.2.1. Taurus. Taurus [Maas et al. 2016] presents a holistic language runtime system for coordinating Big Data platforms running across multiple physical nodes. The authors point out that these platforms run distributed on multiple physical nodes and that each node uses a managed runtime environment such as a JVM to run a Big Data platform on top of it. However, each runtime is not aware of the existence of others working for the same platform (and possibly running the same application).

This lack of communication between runtime environments leads to individual runtime-level decisions that optimize the performance of the local runtime but that are not coordinated, leading to undesirable scenarios if the whole platform performance is not considered. For example, if a JVM starts a new collection while other JVMs are already running collections, although it might be beneficial for the performance of the local node to start a collection, it might lead to significant platform level latencies because many JVMs are paused for collection at the same time.

Taurus solves this problem by presenting an Holistic Runtime System. This system makes platform-level decisions for the entire/global (cluster-wise) platform. Therefore, and reusing the previous example, using Taurus, JVMs are periodically requested to start a collection at different times therefore minimizing the number of JVM pauses for collection at any time.

Using Taurus, application developers can supply policies (written in a special DSL) to guide Taurus on how to perform runtime-level actions such as garbage collection. This solution is based on the OpenJDK HotSpot JVM and represents a JVM drop-in replacement. The authors were able to reduce Cassandra read latency by around 50% and write latency by 75% (both results for the 99.99 percentile).

The obvious limitation of Taurus is the assumption that there are always enough spare resources to replace the nodes that need to go under maintenance. This is not obvious if multiple nodes require maintenance at the same time or if maintenance takes too long. In such situations, for example, during fast workload changes, the number of nodes that need to go under maintenance can easily go over the number of spare resources, resulting in high application latencies.

5.2.2. Garbage First. Garbage First [Detlefs et al. 2004] (G1) is the most recent collector available in the OpenJDK HotSpot JVM, being the next default collector in OpenJDK 9. G1 represents an evolution regarding CMS with the goal of being able to reduce the application pause time while keeping an acceptable throughput. Its main idea is to divide the heap into small regions that can be collected (if needed) in order to maximize the amount of collected garbage while staying below the max acceptable pause time. By doing so, G1 also eliminates the need for full collections (which were known to lead to unacceptably long pause times in CMS).

As with CMS, G1 is generational (i.e., the heap is divided into young and old generations) and, therefore, each heap region can be either in the young generation or in the old generation. There are also three types of collections:

— minor collections, only regions belonging to the young generation can be collected;
— mixed collections, all regions belonging to young generation are collected and some regions from the old generations are also collected (this process is described further below);
— full collections, all regions belonging to both generations are collected;

One of the main benefits of having the heap divided into regions is the possibility to perform mixed collections, where the collector selectively collects regions from the old generation and collects them. This keeps the heap from being fragmented and without free space.

Regions belonging to the old generation are selected for a mixed collection according to their amount of live data. Regions with less live data will be the first to be selected to be included in a mixed collection (hence the name Garbage First). This serves two purposes: i) collecting regions with less live data is faster than collecting regions with more live data (since less objects need to be copied to other regions), thus improving the performance of the collector; ii) since the collector has a maximum acceptable pause time for a collection (which is a user-defined constant), regions which are faster to collect are also easier to collect while still being able to respect the limit pause time.

G1 relies on periodic concurrent marking cycles (a concurrent marking cycle traverses the heap marking each live object) to estimate the amount of live objects in each region. This information, together with other statistics build over time regarding, for example, previous collections, is used to estimate the time needed to collect each region. With this information, the collector constructs a set of regions to collect that maximizes the amount of garbage collected but still does not exceed the maximum desirable pause time.

The authors show that G1 is able to respect a soft real-time pause time goal and that it was not possible to obtain such pause times with CMS. Two well known benchmarks were used: telco[6] and SPECjbb.[7]

Although representing a major improvement regarding previous production GCs (Parallel Scavenge and CMS), G1 suffers from very long GC pauses when Big Data platforms create large portions of objects with long lifecycles; for example: i) the creation of a working set (when a task starts) in a processing platform, and ii) the creation of caches in a storage platform. In both situations, GC pauses are very long due to ob-

[6]The telco benchmark can be found at http://speleotrove.com/decimal/telco.html.
[7]SPECjbb benchmark suite can be found at https://www.spec.org/jbb2015/.

ject copying between heap spaces and go over the maximum desirable pause time set by the user.

5.2.3. Continuously Concurrent Compacting Collector. Continuously Concurrent Compacting Collector [Tene et al. 2011] (C4) is a collector developed by Azul Systems. This is a tracing and generational collector such as G1 and CMS are, but it is also distinct from previous collectors by supporting concurrent compaction (which is not supported neither by G1 nor CMS). In other words, C4 does not require stop-the-world pauses to collect garbage (note that G1 and CMS do require stop-the world pauses, during which all application threads are stopped, to collect garbage).

The C4 garbage collection algorithm is both concurrent (GC threads work while application threads are still active and changing the object graph) and collaborative (application threads can help GC threads doing some work, if needed). The GC algorithm relies on three phases:

— Marking, during this phase, GC threads traverse the object graph, marking each live object. This phase is very similar to concurrent tracing already present in G1 and CMS;
— Relocation, where live objects are moved to a free space (also known as compaction). During this phase, all live objects, marked in the marking phase, are relocated. This process is concurrent (GC threads work concurrently with application threads) and collaborative (in the sense that application threads help moving an object if they try to access it before the object is in its new location).
— Remapping is the final phase where references still pointing to the old location, where an object was moved from, are updated. This phase is also concurrent and collaborative (application threads trying to access an object that was moved out will get the new address of the object and automatically update the reference to point to its new address).

By relying on these three phases, which are concurrent and mostly collaborative, application threads are stopped for a very short period of time to correct some reference or help moving an object, but there will never be a stop-the-world pause that stops all application threads for a long period of time. This, however, comes at the price of heavily relying on barrier/trap handling that can reduce the overall application throughput.

To evaluate C4, the authors used several benchmarks from the SPECjbb2005 benchmark suite. Using transactional oriented workloads (from SPECjbb2005 benchmarks) C4 showed to reduce the worst case pause time by up to two orders of magnitude when compared to CMS.

C4's latency benefits eventually come at the cost of reduced overall throughput or increased resource utilization due to the extreme use of barrier/trap handling. Furthermore, long GC pauses can still occur if the memory allocation rate is above the speed at which the concurrent collector can free memory (for example, during workload shifts).

5.2.4. N-Generational Garbage Collector. N-Generational Garbage Collector [Bruno et al. 2017] (NG2C) presents the idea of extending the well established 2-generational heap layout (with young and old generation) into an N-generational heap layout, i.e., going from using only two generations into using N generations. By using an arbitrary number of generations and by being able to allocate objects directly into any generation, objects with different life-cycles can live into separate generations. This eliminates the need for object promotion (copying objects with longer life-cycles to another generation, the old generation) and compactions (copying objects with long life-cycles within the old generation to free space), the two sources of pause time in current generational collectors.

The motivation behind this work comes from the observation of the following problem: in all previous GCs, objects with different life-cycles are allocated in the same generation (young generation). With time, some objects become unreachable while others are still alive. Upon collection, reachable objects must be copied into the old generation. This copy process results in long pauses time as it depends on the memory bandwidth available in the hardware. This problem is magnified in Big Data platforms, where massive amounts of objects live for quite some time (from the GC perspective). In other words (and relating to the concepts introduced in Sections 2.1.3 and 2.2.4), most objects belonging to working sets and caches (that have long life-cycles) will be copied into the old generation, producing unacceptable application pause times. A similar problem occurs inside the old generation when some of the objects have longer life-cycles than others that have already been promoted into the old generation. These objects with longer life-cycles are copied within the old generation to compact the heap.

To solve these problems, the authors of NG2C propose the extension of the 2-generational heap layout, which is not fit for Big Data platforms, into an N-generational heap layout. By using an arbitrary number of generations, objects with different life-cycles (for example, all objects included in a working set or cache) can be allocated and live in a separate generation. Thus, once the working set or the cache is no longer needed, all objects within it will no longer be reachable and therefore that particular generation (and all the objects within it) can be collected with almost no pause time.

In order to identify objects with different life-cycles (objects included in working sets and caches), programmers' help is required. Thus, programmers must annotate the object allocation specifying in which generation the object should live. With this information, NG2C is able to allocate objects in different generations, thus avoiding the pause times resulting from promotion (copying objects from the young to the old generation) and from compaction (copying objects within the old generation). Note that non-annotated object allocations are still allocated in the young generation and are promoted to the old generation upon collection.

NG2C is implemented on the OpenJDK 8 HotSpot JVM as an extension of the G1 collector. This means that, by default, all objects are allocated in the young generation and that, if they have a long life-cycle, will be promoted to the old generation. Annotated objects will be automatically handled by the new functionality introduced by NG2C.

The authors evaluated NG2C using three Big Data platforms: Cassandra, GraphChi, and Lucene (an in-memory text search index). For each platform, data based on real workloads was used. The results show that NG2C is able to reduce the worst observable pause time by up to 95% for Cassandra, 85% for Lucene, and 96% for GraphChi.

Despite the good results, NG2C forces programmers to annotate object allocations and to keep track of which data structures are kept in which generation. Similarly to FACADE, authors argue that Big Data platform developers are well aware of the platforms' objects' lifecycles but it can still be considered a limiting factor to new programmers. Nevertheless, the authors developed a tool that helps programmers preparing the code to use the new collector by providing hints of where and how to change the code.

5.3. Memory Management Algorithm Comparison

To conclude this study, Table II summarizes all the presented solutions. The table is divided into several columns, each of which concerning a different feature of each solution:

Table II. Taxonomy of Big Data Memory Management Algorithms

Algorithm	Black/White Box	Developer Effort	Target Platform	Main Goal
Broom [Gog et al. 2015]	white	high	processing (Naiad)	throughput
FACADE [Nguyen et al. 2015]	black	low	iterative processing	throughput
Deca [Lu et al. 2016]	black	high	processing (Spark)	throughput
NumaGiC [Gidra et al. 2015]	white	none	processing,storage	throughput
DSA [Cohen and Petrank 2015]	white	medium	processing,storage	throughput
Taurus [Maas et al. 2016]	black	low	processing,storage	latency
G1 [Detlefs et al. 2004]	white	none	processing,storage	latency
C4 [Tene et al. 2011]	white	none	processing,storage	latency
NG2C [Bruno et al. 2017]	white	low	processing,storage	latency

— (black/white) **Box**, a black box algorithm is one that does not interfere with the GC algorithm itself. In other words, this algorithm does not change the GC although it might produce effects it (such as alleviate its work). A white box solution is one that changes the GC implementation for improving it;
— (none/low/medium/high) **Developer Effort**, measures the effort needed to apply the algorithm to existing an Big Data platform. If no effort is required, for example in G1, it means that the developer does not have to change the platform to take advantage of the benefits offered by, for example, G1;
— (processing/storage) **Target Platform**, the platform type where this algorithm is designed to run into;
— (throughput/latency) **Main Goal**, if the algorithm's main goal is to improve throughput or latency.

From Table II it is possible to observe that FACADE, Deca, and Taurus provide a block box solution, i.e., these algorithms do not change or replace any GC algorithm, they only alleviate the amount of work given to the GC.

Regarding the developer effort, most algorithms require some developer effort (only NumaGiC, G1, and C4 do not need platform modifications). These modifications can be seen as a serious drawback since it requires specialized knowledge that only platform developers might have. Algorithms which require high developer effort (Broom and Deca) can be very difficult to use due to high implementation costs.

Broom, FACADE, and Deca, are optimized only for a specific processing platform or subset of platforms while all other platforms are designed to work with both processing and storage platforms.

As described in the previous sections, each algorithm has most impact on either throughput or latency. Most of these algorithms have, nevertheless, a positive impact (although lower) on the other metric.

In order to use any of these algorithms in a real world scenario, two main considerations must be taken into account. First, black box solutions means that there will be an additional component that works between the application and the runtime environment (JVM), for example to instrument code, or to intercept code calls. On the other hand, white box solutions mean that the runtime environment will be changed, i.e., a new specific runtime version must be used. The second consideration is the amount of developer effort required. Solutions such as C4 and G1 are simply drop-ins, meaning that no extra effort is required (except for some configuration parameters). Then, for each other solution, some degree of programmer effort is required to change the target application.

A comparative performance analysis can not be easily done mainly because of two problems. First, different algorithms might work on different platforms or even different runtime environments, making it impossible to test both algorithms in the same environment. Second, some algorithms do not have an open source implementation, meaning that one would have to implement the algorithm from scratch in order to evaluate it. Nevertheless, this study describes these algorithms and present the results claimed by the authors. Moreover, the goal of this work is not to give precise measurements for each algorithm but to describe and point out existing design problems.

To conclude, the authors consider both the developer effort and the target platform as the two most important factors when considering an algorithm to improve either throughput or latency. Algorithms should have low or no associated developer effort to facilitate its introduction into Big Data platforms and should be as generic as possible so that the algorithm can be applied to a wide range of platforms and workloads. In particular, Broom, FACADE, and Deca are only applicable to a sub-set of processing platforms. NimaGiC is applicable to both processing and storage platforms and with no developer effort but only solves problems related to memory accesses in NUMA processors. DSA requires a considerable amount of developer effort. With regards to latency oriented solutions, all four solutions are applicable to both processing and storage platforms and require low (Taurus and NG2C) or no (G1, C4) user effort.

6. OPEN RESEARCH PROBLEMS ON BIG DATA MEMORY MANAGEMENT

Throughout this work, many memory management scalability issues were presented. Some of these issues/challenges, however, remain untackled, possibly leading to limitations on application performance and scalability. This section provides insights into possible new research opportunities to solve existing problems.

6.1. Tracing Millions of Live Objects

As discussed in Section 4.3, tracing collectors need to trace through the application graph in order to mark live objects. Although this is a good idea for applications that generate a lot of objects that become unreachable very soon, it is usually a severe performance issue for large-scale systems such as databases or memory caches that hold massive amounts of objects in memory. For these applications, the collector would have to trace through enormous amounts of objects, which would take a very long time.

This long trace cycle leads to several problems. First, resources such as memory bandwidth and processing power are used to trace objects periodically, even if these objects did not change over time. Second, by taking a long time to finish and therefore to identify dead objects, the application might not have free memory to work even if there are dead objects that, if reclaimed, would free enough memory for the application to work.

In sum, research on how to efficiently trace objects is required. In particular, the amount of tracing should not depend on the number of live objects because applications might cache massive amounts of objects in memory.

6.2. Application Cache-friendly Memory Management

Concurrent tracing is now present in almost all recent collectors but it presents problems in cache-locality. Since tracing threads run concurrently with application threads, application cache locality is ruined every time a collector thread starts to trace objects. When application threads take over, the cache contains no longer the records of its previous iteration, leading to small bumps in performance every time threads are rescheduled. To handle this problem, it is necessary a collector that does not affect application cache.

6.3. High-density Object Graphs

As discussed in Section 4.6, recent collectors copy objects between generations in order to promote objects that survive multiple collections and, therefore, are expected to live longer than the majority of objects. However, collectors do not take into consideration which objects contain reference to each other when promoting them. For example, a data structure such as a list of objects results in a set of objects where each has a reference to the another one (the next one in the list). If these objects are very close in memory, traversing the list is greatly improved since the number of pages that need to be loaded is reduced. On the other hand, if each object resides in a separate page, the number of pages to load is proportional to the number of objects in the list.

The problem with current GC implementations is that when moving objects, the collector does not try to improve the density of the graph (i.e., reduce the memory distance between objects that point to each other), being this process completely independent of the object graph links. This results in object graphs with very low-density, where objects that point to each other can be very far apart within the heap, with an obvious negative performance impact.

6.4. Application-aware Memory Management

By design, the collector and the application should never interact. The application allocates and mutates objects and the collector allocates and deallocates (collects) unreachable objects. This is great because the programmer does not need to deal with low-level garbage collection concepts but, for very complex applications, which require very predictable and precise behavior, this is a clear drawback.

In other words, advanced developers might benefit from improved APIs that interact with the GC. This could be used, for example, to control when to collect garbage, which type of GC to run next, or even which parts of the object graph to collect. Obviously, this improved interface into the collector would only be used by advanced programmers. By leveraging hints/instructions given by the programmer, the GC impact on application performance could be reduced since it would become much more predictable and controllable.

7. CONCLUSIONS

In sum, this work presented a study of current Big Data environments and platforms, focusing on how memory is used by these environments, i.e., their memory profile. With this information, current (classic) garbage collection algorithms were analyzed and some scalability problems were uncovered. Several relevant and recent systems which try to solve some of the scalability problems presented by classic garbage collection algorithms are presented and analyzed in detail. This study on current solutions closes by presenting a taxonomy of all the analyzed algorithms.

This work provides a powerful insight into the state-of-art of memory management for memory managed runtime environments, which are very popular and support many programming languages (Go, Java, C#, Python, Closure, Scala, Ruby, and others) and can also inspire new works to improve current memory management scalability challenges.

ACKNOWLEDGMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 and through the FCT scholarship SFRH/BD/103745/2014.

REFERENCES

Rajendra Akerkar. 2013. *Big data computing*. CRC Press.

Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

Bowen Alpern, C Richard Attanasio, John J Barton, Michael G Burke, Perry Cheng, J-D Choi, Anthony Cocchi, Stephen J Fink, David Grove, Michael Hind, and others. 2000. The Jalapeno virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238.

Andrew W Appel. 1989. Simple generational garbage collection and fast allocation. *Software: Practice and Experience* 19, 2 (1989), 171–183.

Henry G. Baker, Jr. 1978. List Processing in Real Time on a Serial Computer. *Commun. ACM* 21, 4 (April 1978), 280–294. DOI:http://dx.doi.org/10.1145/359460.359470

Peter B Bishop. 1977. *Computer Systems with a Very Large Address Space and Garbage Collection.* Technical Report. DTIC Document.

S. Blackburn, P. Cheng, and K. McKinley. 2004. Oil and water? High performance garbage collection in Java with MMTk. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. 137–146. DOI:http://dx.doi.org/10.1109/ICSE.2004.1317436

Stephen M. Blackburn and Kathryn S. McKinley. 2003. Ulterior Reference Counting: Fast Garbage Collection Without a Long Wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA '03)*. ACM, New York, NY, USA, 344–358. DOI:http://dx.doi.org/10.1145/949305.949336

Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1151–1162.

Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, and others. 2011. Apache Hadoop goes realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 1071–1080.

Don Box and Ted Pattison. 2002. *Essential. Net: the common language runtime*. Addison-Wesley Longman Publishing Co., Inc.

Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. 2017. NG2C: Pretenuring Garbage Collection with Dynamic Generations for HotSpot Big Data Applications. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM 2017)*. ACM, New York, NY, USA, 2–13. DOI:http://dx.doi.org/10.1145/3092255.3092272

Randal Bryant, Randy H Katz, and Edward D Lazowska. 2008. Big-Data Computing: Creating Revolutionary Breakthroughs in Commerce, Science and Society. (2008).

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

Kristina Chodorow. 2013. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.".

Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 105–117. DOI:http://dx.doi.org/10.1145/2754169.2754181

Nachshon Cohen and Erez Petrank. 2015. Data structure aware garbage collector. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 28–40.

George E. Collins. 1960. A Method for Overlapping and Erasure of Lists. *Commun. ACM* 3, 12 (Dec. 1960), 655–657. DOI:http://dx.doi.org/10.1145/367487.367501

Michael Cox and David Ellsworth. 1997. Application-controlled Demand Paging for Out-of-core Visualization. In *Proceedings of the 8th Conference on Visualization '97 (VIS '97)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 235–ff. http://dl.acm.org/citation.cfm?id=266989.267068

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 205–220.

David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*. ACM, 37–48.

Edsger W Dijkstra, Leslie Lamport, Alain J Martin, Carel S Scholten, and Elisabeth FM Steffens. 1978. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* 21, 11 (1978), 966–975.

R. Dimpsey, R. Arora, and K. Kuiper. 2000. Java server performance: A case study of building efficient, scalable Jvms. *IBM Systems Journal* 39, 1 (2000), 151–174. DOI:http://dx.doi.org/10.1147/sj.391.0151

Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. 2012. Efficient big data processing in Hadoop MapReduce. *Proceedings of the VLDB Endowment* 5, 12 (2012), 2014–2015.

David Gay and Bjarne Steensgaard. 2000. Fast escape analysis and stack allocation for object-based programs. In *Compiler Construction*. Springer, 82–93.

Lars George. 2011. *HBase: the definitive guide*. " O'Reilly Media, Inc.".

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 29–43.

Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2013. A study of the scalability of stop-the-world garbage collectors on multicores. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 229–240.

Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. Numagic: A garbage collector for big data on big numa machines. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 661–673.

Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

James Gosling. 2000. *The Java language specification*. Addison-Wesley Professional.

Herodotos Herodotou and Shivnath Babu. 2011. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *Proc. of the VLDB Endowment* 4, 11 (2011), 1111–1122.

Richard L Hudson and J Eliot B Moss. 1992. Incremental collection of mature objects. In *Memory Management*. Springer, 388–403.

R John M Hughes. 1982. A semi-incremental garbage collection algorithm. *Software: Practice and Experience* 12, 11 (1982), 1081–1082.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 59–72.

Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC.

Richard Jones and Andy C King. 2005. A fast analysis for thread-local garbage collection with dynamic class loading. In *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*. IEEE, 129–138.

Richard Jones and Chris Ryder. 2006. Garbage collection should be lifetime aware. *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)* (2006).

Richard E Jones and Chris Ryder. 2008. A study of Java object demographics. In *Proceedings of the 7th international symposium on Memory management*. ACM, 121–130.

Kenneth C. Knowlton. 1965. A Fast Storage Allocator. *Commun. ACM* 8, 10 (Oct. 1965), 623–624. DOI: http://dx.doi.org/10.1145/365628.365655

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 31–46.

Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

Henry Lieberman and Carl Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM* 26, 6 (1983), 419–429.

Jimmy Lin and Dmitriy Ryaboy. 2013. Scaling big data mining infrastructure: the twitter experience. *ACM SIGKDD Explorations Newsletter* 14, 2 (2013), 6–19.

Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. 2016. Lifetime-based Memory Management for Distributed Data Processing Systems. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 936–947. DOI: http://dx.doi.org/10.14778/2994509.2994513

Clifford Lynch. 2008. Big data: How do your data grow? *Nature* 455, 7209 (2008), 28–29.

Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 457–471.

Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. DOI:http://dx.doi.org/10.1145/367177.367199

David A Moon. 1984. Garbage collection in a large LISP system. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM, 235–246.

Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.

Scott Nettles, James O'Toole, David Pierce, and Nicholas Haines. 1992. *Replication-based incremental copying collection*. Springer.

Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ACM Sigplan Notices*, Vol. 50. ACM, 675–690.

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 1099–1110.

Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. 2007. An Efficient On-the-fly Cycle Collection. *ACM Trans. Program. Lang. Syst.* 29, 4, Article 20 (Aug. 2007). DOI:http://dx.doi.org/10.1145/1255450.1255453

Ian Robinson, Jim Webber, and Emil Eifrem. 2013. *Graph databases*. " O'Reilly Media, Inc.".

Semih Salihoglu and Jennifer Widom. 2013. GPS: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 22.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 1–10.

Sunil Soman, Chandra Krintz, and Laurent Daynès. 2008. Mtm2: Scalable memory management for multitasking managed runtime environments. In *ECOOP 2008–Object-Oriented Programming*. Springer, 335–361.

C. J. Stephenson. 1983. New Methods for Dynamic Storage Allocation (Fast Fits). In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles (SOSP '83)*. ACM, New York, NY, USA, 30–32. DOI:http://dx.doi.org/10.1145/800217.806613

Roshan Sumbaly, Jay Kreps, and Sam Shah. 2013. The big data ecosystem at linkedin. In *Proceedings of the 2013 international conference on Management of data*. ACM, 1125–1134.

Andrew S Tanenbaum. 2007. *Modern operating systems*. Prentice Hall Press.

Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The continuously concurrent compacting collector. *ACM SIGPLAN Notices* 46, 11 (2011), 79–88.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

David Ungar. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM Sigplan Notices* 19, 5 (1984), 157–167.

David Ungar and Frank Jackson. 1988. Tenuring policies for generation-based storage reclamation. In *ACM SIGPLAN Notices*, Vol. 23. ACM, 1–17.

Rik Van Bruggen. 2014. *Learning Neo4j*. Packt Publishing Ltd.

Tom White. 2009. *Hadoop: the definitive guide: the definitive guide*. " O'Reilly Media, Inc.".

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.. In *OSDI*, Vol. 8. 1–14.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. 10–10.