Generating Trigger-Action Program from Traces Based on Association Rule Mining: A Survey

Yicheng Wang
Institute of Software
Chinese Academy of Sciences
Haidian Qu, Beijing, China
wangyicheng215@mails.ucas.ac.cn

Abstract

Trigger-action programming (TAP) is a novel approch to define connections between different Internet of Things (IoT) devices and services, which could translate human's intent of IoT devices into desired automation. User-written TAP rules have been widely used as a programming interface in popular IoT systems. As users may experience difficulties in discovering related devices functionality and designing rules, recent works try to automatically generate TAP rules from past user event traces (time-stamped logs of sensor readings and manual actuations of devices). which turns problem into association rule mining of user's trace sequences.

This survey presents a taxonomy of classic association rule mining (ARM) algorithms, indicates the basic workflow and the important key features of these representative algorithms. We also investigate the meritorious applications in Ambient Intelligence (AmI) area, which utilize ARM algorithms to mining the frequent pattern in user event traces, and classify the applications by the algorithms they use. This survey serves as an introduction for IoT related researchers who are intend to realize smart device automation by past behavior mining, and we identifies notable opportunities for further work.

CCS Concepts: • **Human-centered computing → Ambient intelligence**; **Ubiquitous and mobile computing**; • **Software and its engineering → General programming languages**; • **Information systems → Data mining**; • **Information Storage and Retrieval → Retrieval models**.

Advanced Software Engineering'21, July 2021, Beijing, China

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords: Trigger-action programming, Internet of Things, Event prediction, Association Mining, Sequential Pattern Mining

ACM Reference Format:
Yicheng Wang. 2021. Generating Trigger-Action Program from Traces Based on Association Rule Mining: A Survey. In *Proceedings of TCSE Conference (Advanced Software Engineering'21).* ISCAS, Beijing, BJ, China, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The goal of Ambient Intelligence (AmI) is to realize smart interactive environments which could assist us in our everyday tasks, as reacting to our commands and predicting our behavior. The achievement of this goal is advancing with the developement of artificial intelligence techniques and the popularization of Internet of Things (IoT). Plenty of machine learning algorithms have been used in **behaviour modelling and prediction**, including probabilistic graphical models[44, 47, 64], imitation learning[42, 43],neural network[60, 66]. This survey mainly focus on rule-based AmI implementation[28, 53], which determining what tasks are likely to be done next and doing them according to context-dependent rulesets, translate user's intent of IoT devices into desired automation.

Trigger-action programming (TAP)[21, 26, 61] is a novel rule-based approch that provide interfaces for users to create personalized rulesets. TAP is in form of if-this-then-that (e.g., "IF trigger occurs WHILE conditions are true, THEN take some action"), which is available on most popular IoT systems including IFTTT[1], Microsoft Flow[32], Samsung SmartThings[45], etc. These interface offer non-technical users the opportunity to define the connection between different IoT devices and to automate smart spaces in simple scenarios.

However, there are limitations having users write rules. Firstly, users writing rules may contain bugs or otherwise fail to match their intent[5, 41]. Furthermore, users often find it hard to reason about how sensors (e.g., motion sensors) in smart homes work. Tools have been developed to assist users writing TAP rules and detecting bugs in TAP programs[34, 36], while it is difficult in the general case to solve the problems above. Therefore, recent works try to generate TAP rules or predict user activities based on traces automatically.

Normally, traces of IoT devices are time-stamped logs of sensor readings and manual actuations of devices. The basic idea of smart space automation is taking traces as input, mining the frequent and valuable patterns in the trace, recording these patterns as event-driven rules, regarding the proper features in the pattern as *Trigger, Condition* and *Action* of rules. When the *Trigger* and *Condition* of IoT devices is satisfied in the future use, take the *Action* foreseeingly. In this way, the process of generating of TAP rules is turning into the mining of frequent pattern, i.e., **Sequential Pattern Mining** or **Association Rule Mining (ARM)**.

ARM is the task of finding correlations between items in a dataset, as sequential trace log is our main concern, the task can also be considered as finding a collection of events that occur relatively close to each other in a given partial order. This article presents a survey of association mining fundamentals, detailing the classic ARM algorithms, stating the key features of each algorithm, showing how these algorithms are applied in the AmI related works. The contributions of this survey inluding:

- We summarize the most representative ARM algorithms, covers a variety of general types, inluding *Apriori*, *SPADE*, *PrefixSpan*, *FP-Growth*, *CLOSET*, as well as the algorithm derived from them. We describe the workflow the each algorithm, display their most important characteristics.
- We investigate the ponderable works in the smart space research area, which utilize the ARM algorithm introduced in this survey to find the frequent pattern in the trace and speculate the future actions.
- We discuss the current development tendency of TAP rules generating and ARM algorithm optimization, and point out notable opportunities for further work.

The organization of this survey is as following. From section 2 to section 6, each section will describe a representative ARM algorithm, introduce the algorithms generalized from the orginal algorithm, present serveral smart space applications of corresponding algorithm. section 7 will make a taxonomy and conclusion of current algorithms, and make a vision for future development.

2 Apriori

Apriori[2, 4] is the core algorithm of **candidate generation algorithm**. Candidate generation algorithms identify candidate itemsets before validating them with respect to incorporated constraints, where the generation of candidates is based upon previously identified valid itemsets.

All the candidate generation algorithm including Apriori work on the **apriori property**, which states that *"All nonempty subsets of a frequent itemset must also be frequent"*. As a consequence, if a sequence can not pass the minimum support test, all of its supersequences will also fail the test. Algorithm 1 presents the iterative Apriori algorithm that requires k dataset traversals. Apriori introduce constraint inclusion *support* to reduce exploration. The algorithm derives candidate itemsets C_k from $V_{k-1}|k > 1$, incorporating support to reduce $|C_k|$. From this, $|V_k|$ is derived through a scan of the dataset, accruing counts for each $c \in |C_k|$. Thus given a support threshold minsup, $V_k = C_k : \sigma(C_k) \geq minsup$. The algorithm has two main parts, candidate generation and validation. The set of candidates is formed by the set of items, E, given $k = 1$, otherwise it is based on a merge function involving members of L_{k-1}. Subsequent accrual determines the candidate itemset support, and those meeting the nominated threshold minsup are appended to the valid set L_k.

Algorithm 1 Apriori algorithm

begin
  $L_1 \leftarrow Frequent1 - itemset$
  $k \leftarrow 2$
  while $L_{k-1} \neq \phi$ **do**
    $Temp \leftarrow candidateItemSet(L_{k-1})$
    $C_k \leftarrow frequencyOfItemSet(Temp)$
    $L_k \leftarrow compareWithMinSupport(C_k, minsup)$
    $k \leftarrow k + 1$
  end while
  return L
end

As for candidate itemset storage structure, Apriori introduce hash-trees, a combination of b-tree and hashtable structures. The structure is effectively a b-tree for which every internal node is a hashtable, and every leaf node or bucket contains a set of itemsets. When a bucket reaches its quota of itemsets, the hash-tree extends by replacing the bucket with a new hashtable into whose buckets the itemsets are placed.

2.1 Apriori-Based Algorithms

To deal with the methods of counting the sequences produced, AprioriAll algorithm counts all of the sequences whereas AprioriSome and DynamicSome[3] are designed to only produce maximal sequences and therefore can take advantage of this by first counting longer sequences and only counting shorter ones that are not contained in longer ones. This is done by utilizing a forward phase that finds all sequences of a certain length and a backward phase that finds the remaining long sequences not discovered during the forward phase.

In order to address time issue, Generalized Sequential Patterns (GSP) algorithm[58] included time constraints (minimum and maximum gap between transactions), sliding windows, and taxonomies. The minimum and/or maximum gap between adjacent elements was included to reduce the number of "trivial" rules that may be produced. The sliding window enhances the timing constraints by allowing elements of a sequential pattern to be present in a set of transactions that occur within the user-specified time window. Finally, the user-defined taxonomy (is-a hierarchy) which is present in many datasets allows sequential patterns to include elements from any level in the taxonomy.

The PSP algorithm[39] was inspired by GSP but has improvements that make it possible to perform retrieval optimizations. The process uses transactional databases as its source of data and a candidate generation and scan approach for the discovery of frequent sequences. The difference lies in the way that the candidate sequences are organized. GSP and its predecessors use hash tables at each internal node of the candidate tree, whereas the PSP approach organizes the candidates in a prefix-tree according to their common elements which results in lower memory overhead and faster retrievals.

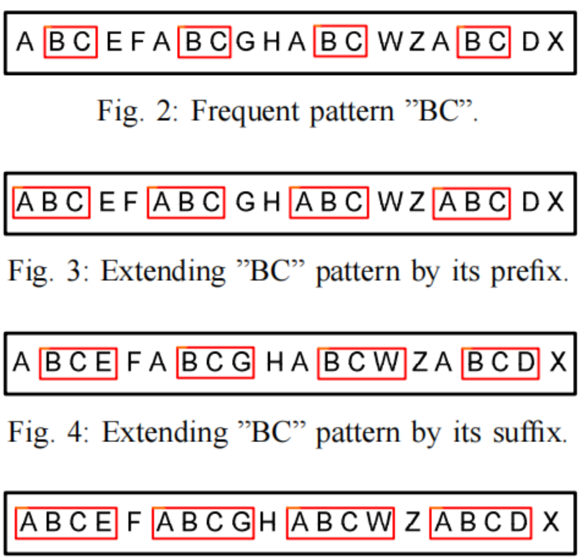


Fig. 2: Frequent pattern "BC".

Fig. 3: Extending "BC" pattern by its prefix.

Fig. 4: Extending "BC" pattern by its suffix.

Figure 1. Using Apriori-type algorithm to find rules[55]

2.2 Apriori-Based Applications

The Weka[63] implementation of an Apriori-type algorithm is used, which iteratively reduces the minimum support until it finds the required number of rules within a given minimum confidence. Jakkula[27, 28] using Weka to identify the frequent activities, or events, which occur during the day and establishing temporal relations among them, describes a method of discovering temporal relations in smart home datasets and applying them to perform activity prediction on the frequently-occurring events. It tests various configurations in Weka to find the best rules which can aid the prediction process.

CASAS[55] introduce an adaptive smart home system that utilizes Apriori-Based techniques to discover patterns in resident's daily activities and to generate automation polices that mimic these patterns. Its core algorithm FPAM takes a bottom-up approach just like Apriori, However, unlike the Apriori algorithm, not only does it discover frequent sequences, but it also tries to find periodic sequences and their periodicity. As Figure 1 shows, FPAM extends sequences that made the cutoff in the previous iteration by the two events that occur before and after the current sequence instances in the data. FPAM incrementally increases the window size until no frequent or periodic sequences within the new window size are found or a user-defined limit on the window size is reached.

PUBS[7] discovers frequent patterns from users' traces and turns them into event-driven rules that represent a subset of TAP rules. The ARM algorithm used by PUBS is similar to the Apriori with only two differences: 1) Limit possible associations with the object under analyzing. 2) The result does not consider a pair as a pattern, but only as sensors that can be potentially related in a meaningful way. To implement, PUBS modify Apriori by adding the aforementioned constraints has been used in this step. As in every association mining process, minimum coverage, support and window size values must be provided.

3 SPADE

The SPADE (Sequential PAttern Discovery using Equivalence classes) algorithm[72] is the most typical ARM algorithm for **vertical database**. vertical organization refers to the representation of objects as columns and with each row within the dataset representing an item. There are four advantages of the vertical organization over the horizontal organization[56]:

- Itemset validation is computationally faster in a vertical layout due to its item focus.
- Given the nonmonotonic constraints, items can be easily removed from the dataset, reducing its size.
- Vector compression is greater in larger datasets and hence better in a vertical layout as typically the number of objects exceeds the number of items.
- Given nonmonotonic constraint inclusion, vertical organization allows the computation of an itemset once its subsets have been deemed valid.

SPADE use combinatorial properties and lattice-based search techniques, allow constraints to be placed on the mined sequences. The key features of SPADE include the layout of the database in a vertical id-list database format with the search space decomposed into sub-lattices that can be processed independently in main memory thus enabling the database to be scanned only three times or just once on some preprocessed data. Using the vertical id-list database in Figure 2, all frequent 1-sequences can be computed in one

(a) Input-Sequence Database

Sequence Id	Time	Items
1	10	C D
1	15	A B C
1	20	A B F
1	25	A C D F
2	15	A B F
2	20	E
3	10	A B F
4	10	D G H
4	20	B F
4	25	A G H

(b) Id-Lists for the Items

A		B		D		F	
SID	EID	SID	EID	SID	EID	SID	EID
1	15	1	15	1	10	1	20
1	20	1	20	1	25	1	25
1	25	2	15	4	10	2	15
2	15	3	10			3	10
3	10	4	20			4	20
4	25						20

SID: Sequence Id
EID: Time

Figure 2. Vertical Formatting Data Layout[72]

database scan. Computing the F_2 can be achieved in one of two ways: by preprocessing and collecting all 2-sequences above a user-specified lower bound, or by performing a vertical to horizontal transformation dynamically. Once this has been completed the process continues by decomposing the 2-sequences into prefix-based parent equivalence classes followed by the enumeration of all other frequent sequences via either breadth-first or depth-first searches within each equivalence class. The enumeration of the frequent sequences can be performed by joining the id-lists in one of three ways:

- Itemset and Itemset: joining AS and BS results in a new itemset ABS.
- Itemset and Sequence: joining AS with $B \rightarrow S$ results in a new sequence $B \rightarrow AS$.
- Sequence and Sequence: joining $A \rightarrow S$ with $B \rightarrow S$ gives rise to three possible results: a new itemset $AB \rightarrow S$, and two new sequences $A \rightarrow B \rightarrow S$and $B \rightarrow A \rightarrow S$.

3.1 SPADE-like Algorithm

The cSPADE algorithm[71] is the same as SPADE except that it incorporates one or more of the syntactic constraints as checks during the mining process, including length or width limitations on the sequences allows for highly structured data; minimum or maximum gap constraints on consecutive sequence; applying a time window on allowable sequences, requires the entire sequence to occur within the window; incorporating item constraints.

SPAM (Sequential PAttern Mining using a bitmap representation) [6] uses a novel depth-first traversal of the search space with various pruning mechanisms and a vertical bitmap representation of the database, which enables efficient support counting. A vertical bitmap for each item in the database is constructed while scanning the database for the first time with each bitmap having a bit corresponding to each element of the sequence in the database. One potential limiting factor on its usefulness is its requirement that all of the data fit into main memory.

The CCSM (Cache-based Constrained Sequence Miner) algorithm[52] uses a level-wise approach initially but overcomes many problems associated with this type of algorithm. This is achieved by using k-way intersections of id-lists to compute the support of candidates When a new sequence is generated, and if a common prefix is contained in the cache, then the associated id-list is reused and subsequent lines of the cache are rewritten. This enables only a single equality join to be performed between the common prefix and the new item, after which the result of the join is added to cache.

3.2 SPADE-base Applications

EMS (energy management system)[15] provides a non-intrusive and low-cost solution to recognize the states of appliances and to disaggregate the energy consumption of appliances in a house/building. It use SPADE to detect users' behaviors based on the usage patterns of appliances. EMS find that users' behaviors strongly correlate with a sequence of appliance usages and appliance states. Therefore, it can utilize the data including status of appliance and the time that the appliance state changes to detect users' behaviors. user behavior detection is mining the sequences from a number of cases. Each case represents an activity and its possible correlated events. SPADE algorithm is applied to learn the sequential patterns for each activity.

LFE (Load Forecasting Enhancement)[19] gives a set of insights into household-specific activity sequences influencing power consumption derived from a sequence mining algorithm, and a load forecasting study using identified frequent activity sequences as an enhancement. The implementation utilize the SPADE algorithm to understand a household's daily activity sequences, a sequence is given by all the activities performed by the household throughout a day and ordered by the start time of activities.

SHGuard[38] is an anomaly detection approach based on power usage data exposed from wireless communications in the smart home system. SHGuard extracts and builds the normal behavior pattern during the initialization stage. It continuously infers the smart devices' states by monitoring the electricity usage data and updates the user behavior patterns. It choose the SPADE algorithm to perform sequential pattern mining on power-usage behavior sequence dataset, preprocess the power-usage behaviors dataset and convert it into the standard input data format of the SPADE algorithm, which is $s = (sequenceID, eventID, item)$, and eventually obtain a frequent sequence pattern set.

4 PrefixSpan

PrefixSpan (Prefix-projected Sequential Pattern mining)[23] is the core algorithm of **pattern growth algorithm**. The frequent pattern growth paradigm removes the need for the candidate generation and prune steps that occur in the Apriori-type algorithms and does so by compressing the database representing the frequent sequences into a frequent pattern tree and then dividing this tree into a set of projected databases, which are mined separately[24].

Modern big data processing frameworks, including Hadoop, Spark[70] and Flink[14], were all implemented in object-oriented languages such as Scala and Java, due to theirs applicability across heterogeneous distributed clusters, convenience on memory management, and fertility of community resources. However, big data applications suffer from striking high GC cost under JVM. As reported by users and researchers, GC time can take even more than 50% of the overall application execution time in some cases[13, 54]. Some concurrent GC algorithms, e.g., G1 GC[17], are able to limit GC pause times to a certain extent, while mutator threads would be inevitably affected[65, 67].

The GC inefficiency problem under big data processing frameworks has been widely studied[10, 22, 67], and results from a combination of factors. First, big data applications are both *data-intensive* and *memory-intensive*, a large amount of data is loaded in the memory as objects at the runtime, putting servere overhead on the GC colloctor. Big data processing frameworks that caching intermediate computing results in the memory make thins even worse. Second, the memory usage pattern of big data applications is totally different from traditional Java applications, data objects of big data applications tend to stay in memory for a long period of time, which is opposite to the*weak generational hypothesis hypothesis*[51] that the classical GC algorithms based on.

The weak generational hypothesis states that most objects survive for a short time and only a few objects live long. For this reason, popular GC colloctors are implemented generational, i.e., divide the heap into generations, typically young generation for keeping short-lived objects, and old generation for keeping long-lived objects. Thus by collecting the young generation more frequently than old generation, GC colloctors reduce the average number of objects processed per GC cycle.

Under generational GC algorithms, long-lived data obejcts are destined to survive in *minor GC* cycles, which is target at young generation, and certain to be promoted to the old generation eventually when their survival times reach a certain threshold. In each GC cycle, all the long-lived data obejcts would be marked as "live" objects by the *accessibility analysis* across the object reference graph, and be compacted (copied) to another memory space. Even after being promoted, these long-lived obejcts would still not be "dead" in a short time, while their big volume makes it very likely to trigger *major GC* when the old generation is full, which introduces more unnecessary scan and copy of them. As objects marking and moving are considered to be the most time-consuming part of the GC cycle[59, 69], the long-lived data objects is regarded as the main reason to the degradation of GC efficiency and the main direction of optimization.

Moving back to unmanaged languages and leave memory management to developers is a tiring but possible solution[18, 37]. However, unmanaged languages are error-prone, espicially to big data applications whose memory usgae is stressful and complicated and whose running time is quite long. Furthermore, since a great number of existing big data processing frameworks are already developed in a managed language, it is unrealistic to re-implement once again[10, 50]. Some other works attempt to use the APIs from package *sun.misc.Unsafe* to handle the off-heap memory explicitly like unmanaged languages[40, 46, 50]. The problem is that, as the warning of its name, *Unsafe* package is unsafe. It has the drawbacks same as unmanaged languages, and it may introduce serialization/deserialization jobs in orginal big data environment. Therefore, more works still settle the long-lived objects in heap.Two main optimization techniques used in these work are lifetime-based memory management and region-based memory management. The former obtains the lifetime of data objects by static analysis of source code and users' annotation, reclaim the long-lived objects at their "dead" point in the code. The latter leverages the lifetime information to divide the data obejcts by their live time into separate memory spaces, facilitates one-time reclamation[16, 31, 35, 49, 62].

Inspired by the techniques above, we pick out the long-lived objects according to the lifetime information, and pretenure them in the older generation. As G1 GC are naturely region-based, we directly create a new generation aiming at the storage of long-lived data objects on the top of G1, whose regions would neither be involved in minor GC, nor be in the collection sets of major GC, unless it get the signal that the lifetime of the objects in current region is end. Thanks to Spark Tungsten[57] project and Flink who utlizing primitive arraies as memory pages to form unmanaged-languages-like condition on heap, we could readily pretune most long-lived data by pretuning memory page objects(primitive arraies). We evalute our idea by the time costed to creat memory pages. Result shows that, compared with orginal G1 GC, we eliminated 90% of minor GC pause time.

5 Motivation

The main challenge of pretenuring in JVM is determining the long-lived objects. Previous pretenuring works using sampling[25, 30], profiling[8, 9, 12] and annotation[11] to speculate the lifetime of all the objects. These approches inevitably introduce repeat executions, CPU competition or heavy users' efforts, many of works are highly dependent on the users' developement experience and knowledge on GC algorithm. Fortunately, long-lived objects determination is simpler under big data processing framworks for the characteristics below:

5.1 Data Path

More than 95% of runtime objects are created and used by a rather small and simple code referred as *data path*, that primarily conducts data manipulation like map, reduce, and relational operations. The objects created by *data path* are *data objects*, which are the main component of a long-lived objects. Only less than 5% of runtime objects are created by a large and complex code referred as *control path*.The *control objects* created by *control path* are principally for cluster management, scheduling, communication, and others, which are basiclly short-lived and negligible[13, 46, 49, 50]. This insight allow us focus on a small amount of code when handling long-lived data objects in big data applications, rather than pay attention to massive code in ordinary Java applications.

5.2 Memory Pages

As storing the data as objects requires additional memory footprint, some works use primitive array as memory pages to store the data value only. The structure of a data object in the JVM contains object headers and references to other objects, and the data value itself usually takes up no more than half of the space in the object[13, 46]. For the reason that the methods of the data object are seldom used, the "shells" of objects is not only meaningless, but also harmful as it introduces serialization/deserialization work when the data is transformed across the cluster through the network, which may accounte for 30% of the total execution time[46, 48] Therefore, these works allocate primitive array as memory page or memory container, gather the data with similar lifetime in the same memory page, and manage the long-lived data as a whole by managing the memory page objects[13, 35, 46, 50], reclaim the memory pages at the end of iterations, stages and computing operators.

Modern big data processing framworks utilized the idea of memory page, store the serialized data value in the memory page. As shown in Figure 1(a), Tungsten project of Spark uniform the in-heap and off-heap memory by memory blocks, Tungsten manages memory blocks as page table. The memory location of a data value is determined by the page number and the offset in page. Off-heap page number is represented by an absolute address, and the page offset is the current address. On-heap page number is represented by the reference to the corresponding array object, and the offset in page is the relative offset within the array object. When the data in the page are freed explicitly by Spark, the memory page object would be added to a memory pool made up by weak reference, which means the memory page either be reused before it is scanned, or be relcaimed in next GC cycle.

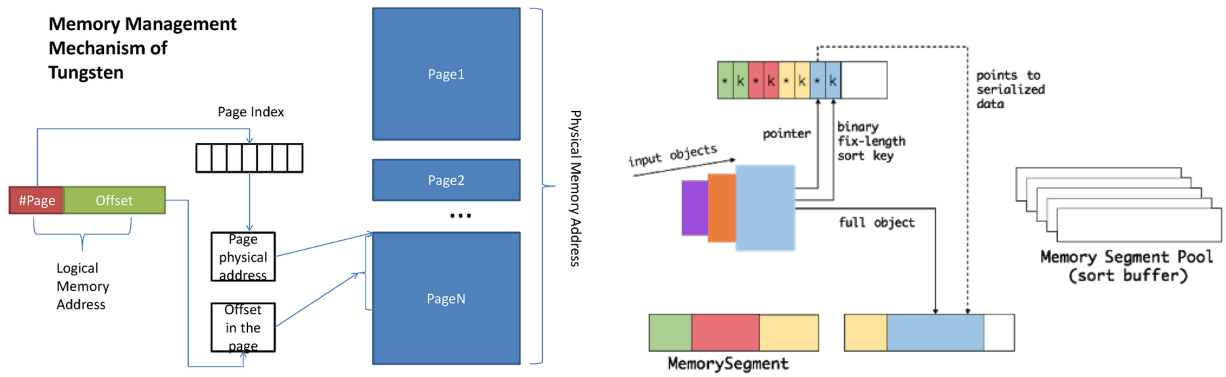
Similar to Spark Tungsten, Flink serializes objects into a fixed number of memory segments. As shown in Figure 1(b), memory manager of Flink holds a huge collection of memory segments in the memory pool. Different from memory blocks of Spark, memory segment is pre-allocated and fixed-sized, whose default size is 32KB. In addition, the lifetime of memory segments in the memory pool is same as computing task, it would only be released when the task end.

Design of memory page brings remarkable performance improvement to the big data processing frameworks, the reasons are as follows. (1) The binary form of data in memory page increases the storage density of memory, which allows more data cached in the memory. The employment of memory page decreases the number of objects, i.e., reduces the pressure of GC. (2) The frameworks implements specialized serializer and computing operators aiming at binary data speed up the execution and data transmission. (3) The continuous arrangement of data in memory page improves the spatial locality of memory accessment, which is a great concern of current GC algorithm for it improves L1/L2/L3 cache hit ratio of CPU[33, 68].

The contradiction is the choice between on-heap and off-heap, which are both available in Flink and Spark. The benefits of using off-heap are attractive, nevertheless the usage of off-heap is not safe and secure under big data frameworks as discussed in section 1, and Java Unsafe package is not available since JDK 11[20]. Therefore, allocating memory pages on heap is the most likely direction for the future. The left problem is, the cost of promoting memory pages and major GC is still noteworthy in a large memory environment[29], which can be solved exactly by pretenuring. The clear type and size of memory page makes this work achievable.

6 Desgin and Implementation

We implement our idea in the OpenJDK 8 HotSpot JVM, one of the most widely used industrial JVMs. Since HotSpot is a highly optimized production JVM, our algorithms were implemented carefully to prevent the other function and the overall performance of JVM, some analysis and choices are made during the implementation procedure. Essential details are as follows.



(a) Memory Block of Spark

(b) Memory Segment of Flink

Figure 3. Memory Page in Modern Big Data Processing Framworks.

6.1 Pick Out Long-Lived Objects

Cached records and accumulated shuffled records are the main source of long-lived data objects in big data processing framworks[67]. Cached records stand for reusable data which are retained in memory by developers, in order to reduce disk I/O. Developer explicitly caching data using method *cache()* or *persist()*, and release them from memory using method *unpersist()*. Cached *Resilient Distributed Dataset* (RDD) in Spark is a typical example that we focus on. The lifetime of shuffled records is more complicated, nevertheless, they are handled in memory pages. Therefore, we can only focus on memory block and memory segment.

As discussed in section 2.1, the manipulation of data objects is accomplished by a few code in data path. There is no exception on cached RDD, memory block and memory segement. The building function of these three kinds of objects are respectively:

- **Memory segement**: *byte* type array of fixed size 32K;
- **Memory block**: *long* type array of required data size;
- **Cached RDD**: Array of user-defined type with required length, each element is of user-defined type.

The next job is letting JVM figure out thses long-lived objects at runtime. Dispose of memory segement and memory block is rather straightforward, as they are primitive types array, the creation of them are done by corresponding initialed *TypeArrayKlass* in JVM, respectively *byteArrayKlassObj* and *longArrayKlassObj*. For the memory segement is fixed-sized, we identify it by the size when allocating. Though the length of memory block varies, it is generally longer than the length of ordinary long type array, we identify a long type array memory block if its length exceeds a threshold, the rest memory blocks whose length smaller than threshold are rather negligible. Dispose of cached RDD is rather difficult, for the element object of cached RDD may contain variety types of objects, we only handle the RDD array. Even so, we still can not identify it without the help from code of big data processing frameworks. We pass the user-defined type T to the JVM, and identify T type array cached RDD if its length exceeds threshold, which is a common approch to identify RDD array[62].

6.2 GC Colloctor Choice

After figured in the JVM, the allocation request of long-lived object is passed to the allocator, which varies from GC algorithm. There are two reasonable choice: Parallel scavenge GC and G1 GC, two most widely used GC collectors in HotSpot. Parallel is the default collector of JDK 8, which is the most popular version of JDK. G1 is the default collector since JDK 9, and is explicitly set as the default collector in Flink.

As shown in Figure 2, the distinct difference between Parallel and G1 is that, the young and old generations are both contiguous space with an explicit boundary in Parallel, and major GC would handle the whole heap space, while G1 logically separates young and old generations in a non-contiguous way, by dividing heap space into a large number of equal-sized regions, each region can be young or old space, only some garbage-filled heap regions picked in the collection set would be handled during most major GC. Specially, Parallel store humongous objects directly into old generation, and G1 stores humongous objects into humongous regions, which span multiple regions in old generation. Pretuning is easy to implement in Parallel GC, as the method of humongous objects allocation is relatively exposed, using this method could directly settle the long-lived memory pages in the old generation without the modification of heap lock in JVM. However, memory pages pretuned in Parallel are mixed with other ordinary objects promoted in old generation, still need to be scanned and moved during major GC, for major GC of Parallel target at whole heap space. Also the locality of memory pages may be damaged by compact phase of major GC, since the compact job is done by multiple GC threads, the new order of memory pages is uncertain.

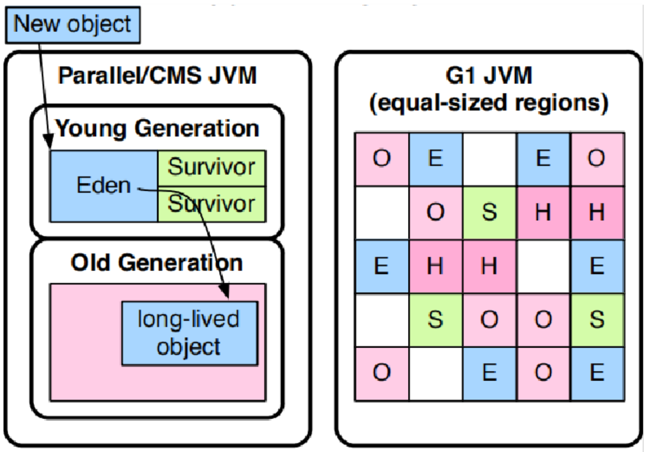


Figure 4. Heap distribution under Parallel scavenge GC and G1 GC algorithm

which makes the memory access sequence unfriendly to data sequence, degrades the L1/L2/L3 cache hit ratio of CPU. Besides, evaluation of garbage collectors on big data applications shows that Parallel always introduce long pause time and tail latency[33, 59, 67, 69], much inferior to G1 GC. For these reasons, Parallel is not our preference choice.

Modify on G1 GC could avoid the problems above. As G1 divides the heap space into regions, we use some of these regions to store long-lived memory pages to seperate them from ordinary objects and protect the locality of memory pages. To address unnecessary scan and movement during major GC, we let the regions that store memory pages skip away from collection set of G1 GC unless JVM get the release signal from big data processing frameworks. During major GC, G1 would only handle the regions in the collection set, thus memory pages would not be moved. In addition, memory page have no reference to other objects for the reason that it is only a carrier of data, which reduces the overhead of scan and footprint of remember set, which record the reference across the regions in G1 GC.

In implementation, we name these regions *Keep* regions in JVM, distinguish them from the normal old regions. Memory space occupied by *Keep* regions still count in old generation space, in order to keep the overall memory apportion of young and old generation.

6.3 Memory Allocation Buffer

Using humongous objects allocation method in G1 GC to allocate long-lived memory pages is unbearable. Because every humongous object allocation would occupy at least one heap region, whose size is much bigger than a memory page object. More importantly, humongous object allocation is in the slow path of objects allocation, i.e., required memory is obtained by heap lock competition, which is inefficient.

Allocation memory competition between different threads is solved by Thread Local Allocation Buffer (TLAB) in young generation and Promotion Local Allocation Buffer (PLAB) in old generation. As the example of TLAB in Figure 3, JVM divides the free memory space into allocation buffers, each buffer dedicated to a particular thread. since every thread allocate objects in the allocation buffer belong to it, there is no need for synchronization. Though we count keep regions in old generation, while PLAB is only used during GC for GC thread, we use TLAB in keep regions for mumator thread. For mumator thread already have a TLAB for young generation, we add another TLAB for keep regions and change the way mutator thread get TLAB, using pointer instead of offset to the base address of thread.

The size of TLAB is initialed when JVM start on the basis of the total heap size, and update at the end of each GC cycle according to the *Adaptive Policy*. Since memory page is larger than ordinary object, while TLAB size update is globally, we adjust the size of TLAB for keep regions independently. As the size of memory page is known, we set this size to a reasonable value, which can reduce TLAB waste, increases the number of memory pages a TLAB could continuously allocate without heap lock, consequently reduce the TLAB allocation throguh slow allocation path.

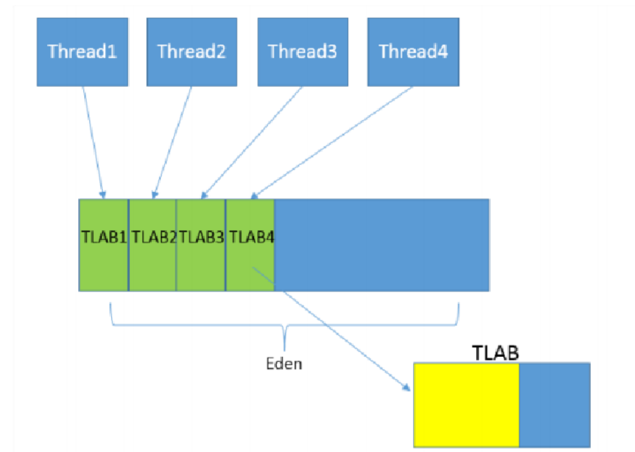


Figure 5. Thread Local Allocation Buffer

7 Evaluation

We evaluate our desgin by allocating memory pages, and we use representative big data processing frameworks Flink to verify the feasibility of our implementation. The result shows that, compared with orginal G1 GC, our implementation could achieve shorter execution time and GC pause time, less object movement and remember set footprint without negative effect on other functions.

7.1 Evaluation Setup

7.2 TLAB Size

7.3 GC Pause Time

References

[1] [n.d.]. IFTTT. [EB/OL]. https://ifttt.com.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 207–216.

[3] Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining sequential patterns. In *Proceedings of the eleventh international conference on data engineering*. IEEE, 3–14.

[4] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, Vol. 1215. Citeseer, 487–499.

[5] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the mirai botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*. 1093–1110.

[6] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. 2002. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 429–435.

[7] Asier Aztiria, Juan Carlos Augusto, Rosa Basagoiti, Alberto Izaguirre, and Diane J Cook. 2012. Discovering frequent user–environment interactions in intelligent environments. *Personal and Ubiquitous Computing* 16, 1 (2012), 91–103.

[8] Stephen M Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S McKinely, and J Eliot B Moss. 2001. Pretenuring for java. *ACM SIGPLAN Notices* 36, 11 (2001), 342–352.

[9] Rodrigo Bruno and Paulo Ferreira. 2017. POLM2: automatic profiling for object lifetime-aware memory management for hotspot big data applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 147–160.

[10] Rodrigo Bruno and Paulo Ferreira. 2018. A study on garbage collection algorithms for big data environments. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–35.

[11] 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*. 2–13.

[12] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. 2019. Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[13] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J Carey. 2013. A bloat-aware design for big data applications. In *Proceedings of the 2013 international symposium on memory management*. 119–130.

[14] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[15] Yung-Chi Chen, Chun-Mei Chu, Shiao-Li Tsao, and Tzung-Cheng Tsai. 2013. Detecting users' behaviors based on nonintrusive load monitoring technologies. In *2013 10th IEEE International Conference on Networking, Sensing and Control (ICNSC)*. IEEE, 804–809.

[16] Nachshon Cohen and Erez Petrank. 2015. Data structure aware garbage collector. In *Proceedings of the 2015 International Symposium on Memory Management*. 28–40.

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

[18] Mengsu Ding and Shimin , Chen. 2017. Helius: A Lightweight Big Data Processing System. *Journal of Computer Application* 37, 2 (2017), 305–310.

[19] Yong Ding, Julio Borges, Martin A Neumann, and Michael Beigl. 2015. Sequential pattern mining—A study to understand daily activity patterns for load forecasting enhancement. In *2015 IEEE First International Smart Cities Conference (ISC2)*. IEEE, 1–6.

[20] FLINK-2485. [n.d.]. Handle removal of Java Unsafe. [EB/OL]. https://issues.apache.org/jira/browse/FLINK-2485.

[21] Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. 2017. Personalization of context-dependent applications through trigger-action rules. *ACM Transactions on Computer-Human Interaction (TOCHI)* 24, 2 (2017), 1–33.

[22] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2013. A study of the scalability of stop-the-world garbage collectors on multicores. *ACM SIGPLAN Notices* 48, 4 (2013), 229–240.

[23] Jiawei Han. [n.d.]. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. Citeseer.

[24] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. *ACM sigmod record* 29, 2 (2000), 1–12.

[25] Timothy L Harris. 2000. Dynamic adaptive pre-tenuring. In *Proceedings of the 2nd international Symposium on Memory Management*. 127–136.

[26] Justin Huang and Maya Cakmak. 2015. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 215–225.

[27] Vikramaditya Jakkula and Diane J Cook. 2007. Learning temporal relations in smart home data. In *Proceedings of the Second International Conference on Technology and Aging, Canada*, Vol. 33. Citeseer.

[28] Vikramaditya Jakkula and Diane J Cook. 2007. Mining sensor data in smart environment for temporal activity prediction. *Poster session at the ACM SIGKDD, San Jose, CA* (2007).

[29] Wang JR. [n.d.]. Flink Memory Management. [EB/OL]. https://blog.jrwang.me/2019/flink-source-code-memory-management/.

[30] Maria Jump, Stephen M Blackburn, and Kathryn S McKinley. 2004. Dynamic object sampling for pretenuring. In *Proceedings of the 4th international symposium on Memory management*. 152–162.

[31] Iacovos G Kolokasis, Anastasios Papagiannis, Polyvios Pratikakis, Angelos Bilas, and Foivos Zakkak. 2020. Say Goodbye to Off-heap Caches! On-heap Caches Using Memory-Mapped I/O. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.

[32] Nat Levy. 2017. Microsoft updates IFTTT competitor Flow and custom app building tool PowerApps.

[33] Haoyu Li, Mingyu Wu, Binyu Zang, and Haibo Chen. 2019. ScissorGC: scalable and efficient compaction for Java full garbage collection. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 108–121.

[34] Jaime Lien, Nicholas Gillian, M Emre Karagozler, Patrick Amihood, Carsten Schwesig, Erik Olson, Hakim Raja, and Ivan Poupyrev. 2016. Soli: Ubiquitous gesture sensing with millimeter wave radar. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–19.

[35] 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. *Proceedings of the VLDB Endowment* 9, 12 (2016), 936–947.

[36] Yu Luo, Jianbo Ye, Reginald B Adams, Jia Li, Michelle G Newman, and James Z Wang. 2020. ARBEE: Towards automated recognition of bodily expression of emotion in the wild. *International journal of computer vision* 128, 1 (2020), 1–25.

[37] Martin Maas, Krste Asanović, and John Kubiatowicz. 2017. Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 138–143.

[38] Jian Mao, Shishi Zhu, Jingdong Bian, Qixiao Lin, and Jianwei Liu. 2019. Anomalous power-usage behavior detection from smart home wireless communications. *Journal of Communications and Information Networks* 4, 1 (2019), 13–23.

[39] Florent Masseglia, Fabienne Cathala, and Pascal Poncelet. 1998. The PSP approach for mining sequential patterns. In *European Symposium on Principles of Data Mining and Knowledge Discovery*. Springer, 176–184.

[40] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at your own risk: the Java unsafe API in the wild. *ACM Sigplan Notices* 50, 10 (2015), 695–710.

[41] Tomas Mikolov, Armand Joulin, and Marco Baroni. 2016. A roadmap towards machine intelligence. In *International Conference on Intelligent Text Processing and Computational Linguistics*. Springer, 29–61.

[42] Bryan Minor, Janardhan Rao Doppa, and Diane J Cook. 2015. Data-driven activity prediction: Algorithms, evaluation methodology, and applications. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 805–814.

[43] Bryan David Minor, Janardhan Rao Doppa, and Diane J Cook. 2017. Learning activity predictors from sensor data: Algorithms, evaluation, and applications. *IEEE transactions on knowledge and data engineering* 29, 12 (2017), 2744–2757.

[44] Gadelhag Mohmed, Ahmad Lotfi, and Amir Pourabdollah. 2020. Enhanced fuzzy finite state machine for human activity modelling and recognition. *Journal of Ambient Intelligence and Humanized Computing* 11, 12 (2020), 6077–6091.

[45] Walt Mossberg. 2014. SmartThings automates your house via sensors, app. *Recode. net* (2014).

[46] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. 2019. Gerenuk: Thin computation over big native data using speculative program transformation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 538–553.

[47] Ehsan Nazerfard and Diane J Cook. 2012. Bayesian networks structure learning for activity prediction in smart homes. In *2012 Eighth International Conference on Intelligent Environments*. IEEE, 50–56.

[48] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting managed heaps in distributed big data systems. *ACM SIGPLAN Notices* 53, 2 (2018), 56–69.

[49] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A high-performance big-data-friendly garbage collector. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 349–365.

[50] 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. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 675–690.

[51] Oracle. [n.d.]. JVM Generations. [EB/OL]. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html.

[52] Salvatore Orlando, Raffaele Perego, and Claudio Silvestri. 2004. A new algorithm for gap constrained sequence mining. In *Proceedings of the 2004 ACM symposium on Applied computing*. 540–547.

[53] Alex Pentland and Andrew Liu. 1999. Modeling and prediction of human behavior. *Neural computation* 11, 1 (1999), 229–242.

[54] Kehinde Philip. [n.d.]. Spark executor GC taking long. [EB/OL]. https://stackoverflow.com/questions/38965787/spark-executor-gc-taking-long.

[55] Parisa Rashidi and Diane J Cook. 2009. Keeping the resident in the loop: Adapting the smart home to the user. *IEEE Transactions on systems, man, and cybernetics-part A: systems and humans* 39, 5 (2009), 949–959.

[56] Pradeep Shenoy, Jayant R Haritsa, S Sudarshan, Gaurav Bhalotia, Mayank Bawa, and Devavrat Shah. 2000. Turbo-charging vertical mining of large databases. *ACM Sigmod Record* 29, 2 (2000), 22–33.

[57] Apache Spark. [n.d.]. Project Tungsten: Bringing Apache Spark Closer to Bare Metal. [EB/OL]. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[58] Ramakrishnan Srikant and Rakesh Agrawal. 1996. Mining quantitative association rules in large relational tables. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 1–12.

[59] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. 2018. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[60] Niek Tax. 2018. Human activity prediction in smart home environments with LSTM neural networks. In *2018 14th International Conference on Intelligent Environments (IE)*. IEEE, 40–47.

[61] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. 2014. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 803–812.

[62] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 347–362.

[63] Ian H Witten, Eibe Frank, Mark A Hall, CJ Pal, and MINING DATA. 2005. Practical machine learning tools and techniques. In *DATA MINING*, Vol. 2. 4.

[64] Eying Wu, Peng Zhang, Tun Lu, Hansu Gu, and Ning Gu. 2016. Behavior prediction using an improved Hidden Markov Model to support people with disabilities in smart homes. In *2016 IEEE 20th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, 560–565.

[65] Mingyu Wu, Ziming Zhao, Yanfei Yang, Haoyu Li, Haibo Chen, Binyu Zang, Haibing Guan, Sanhong Li, Chuansheng Lu, and Tongbao Zhang. 2020. Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services. In *2020 {USENIX} Annual Technical Conference ({USENIX} {ATC} 20)*. 159–172.

[66] Gaowei Xu, Min Liu, Fei Li, Feng Zhang, and Weiming Shen. 2016. User behavior prediction model for smart home using parallelized neural network algorithm. In *2016 IEEE 20th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, 221–226.

[67] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. 2019. An experimental evaluation of garbage collectors on big data applications. In *The 45th International Conference on Very Large Data Bases (VLDB'19)*.

[68] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020. Improving program locality in the GC using hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 301–313.

[69] Yang Yu, Tianyang Lei, Weihua Zhang, Haibo Chen, and Binyu Zang. 2016. Performance analysis and optimization of full garbage collection in memory-hungry environments. *ACM SIGPLAN Notices* 51, 7 (2016), 123–130.

[70] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[71] Mohammed J Zaki. 2000. Sequence mining in categorical domains: incorporating constraints. In *Proceedings of the ninth international conference on Information and knowledge management*. 422–429.

[72] Mohammed J Zaki. 2001. SPADE: An efficient algorithm for mining frequent sequences. *Machine learning* 42, 1 (2001), 31–60.