



**HAL**  
open science

# Efficiently Mining Large Gradual Patterns Using Chunked Storage Layout

Dickson Odhiambo Owuor, Anne Laurent

► **To cite this version:**

Dickson Odhiambo Owuor, Anne Laurent. Efficiently Mining Large Gradual Patterns Using Chunked Storage Layout. ADBIS 2021 - 25th European Conference on Advances in Databases and Information Systems, Aug 2021, Tartu, Estonia. pp.30-42, 10.1007/978-3-030-82472-3\_4 . lirmm-03320961

**HAL Id: lirmm-03320961**



**<https://hal-lirmm.ccsd.cnrs.fr/lirmm-03320961>**

Submitted on 16 Aug 2021

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Efficiently mining large gradual patterns using chunked storage layout

Dickson Odhiambo Owuor<sup>1</sup>  and Anne Laurent<sup>2</sup> 

<sup>1</sup> SCES, Strathmore University, Nairobi, Kenya  
dowuor@strathmore.edu

<sup>2</sup> LIRMM Univ Montpellier, CNRS, Montpellier, France  
anne.laurent@umontpellier.fr

**Abstract.** Existing approaches for extracting gradual patterns become inefficient in terms of memory usage when applied on data sets with huge numbers of objects. This inefficiency is caused by the contiguous nature of loading binary matrices into main memory as single blocks when validating candidate gradual patterns. This paper proposes an efficient storage layout that allows these matrices to be split and loaded into/from memory in multiple smaller chunks. We show how HDF5 (Hierarchical Data Format version 5) may be used to implement this chunked layout and our experiments reveal a great improvement in memory usage efficiency especially on huge data sets.

**Keywords:** Binary Matrices · Gradual Patterns · HDF5 · Memory Chunk · Zarr.

## 1 Introduction

Gradual patterns may be described as linguistic rules that are applied on a data set to extract correlations among its attributes [7,10]. For instance, given a data set shown in Table 1 (which is a numeric data set with 3 attributes {age, games, goals}), a linguistic gradual correlation may take the form: “*the lower the age, the more the goals scored.*”

Table 1: Sample data set  $\mathcal{D}_1$ .

<b>id</b>	<b>age</b>	<b>games</b>	<b>goals</b>
r1	30	100	2
r2	28	400	4
r3	26	200	5
r4	26	500	8

One major step in mining gradual patterns involves ranking tuples in the order that fulfill a specific pattern. For example, in Table 1 the pattern “*the*

*lower the age, the more the goals scored.*” is fulfilled by at least 3 ordered tuples:  $\{r1 \rightarrow r2 \rightarrow r3\}$ . For the reason that computing processors are natively designed to operate on binary data, the approach of representing ordered rankings as binary matrices yields high computational efficiency for mining gradual patterns using the bitwise AND operator [2,7,10].

However, the same can not be said of these binary matrices in terms of main memory usage. For instance, given a data set with  $n$  tuples and  $m$  attributes:

- for every attribute  $a$  in  $m$ , there may exist at least 2 *frequent* gradual items -  $(a, \uparrow)$  and  $(a, \downarrow)$  and,
- for every gradual item, a binary matrix of size  $(n \times n)$  must be loaded into memory.

Consequently, a single bitwise AND operation loads and holds multiple  $n \times n$  binary matrices into memory. This problem becomes overpowering when dealing with data sets with huge number of tuples. Most often, algorithms implemented on this approach crash when applied on such data sets since they require to be assigned an overwhelming amount of main memory at once (when performing the bitwise AND operation).

In this paper, we propose an approach that advances the bitwise AND operation such that it operates on multiple smaller chunks of the binary matrices. This approach allows efficient use of main memory while performing this operation on huge binary matrices. In addition, we design GRAD-L algorithm that implements this proposed approach. Our experiment results show that our proposed approach by far outperforms existing approaches especially when dealing with huge data sets.

The remainder of this paper is organized as follows: we provide preliminary definitions in Section 2; we review related approaches in Section 3; in Section 4, we propose an approach that allows efficient use of main memory through chunking binary matrices for the bitwise AND operation; we analyze the performance of our proposed approach in Section 5; we conclude and give future directions regarding this work in Section 6.

## 2 Preliminary Definitions

For the purpose of putting forward our proposed approach for mining large gradual patterns; in this section, we recall some definitions about gradual patterns taken from existing literature [2,10].

**Definition 1.** Gradual Item. *A gradual item  $g$  is a pair  $(a, v)$  where  $a$  is an attribute of a data set and  $v$  is a variation such that:  $v \in \{\uparrow, \downarrow\}$ , where  $\uparrow$  denotes an increasing variation and,  $\downarrow$  denotes a decreasing variation.*

*Example 1.*  $(age, \downarrow)$  is a gradual item that may be interpreted as: “the lower the age.”

**Definition 2.** Gradual Pattern. *A gradual pattern  $GP$  is a set of gradual items i.e.  $GP = \{(a_1, v_1), \dots, (a_n, v_n)\}$ .*

*Example 2.*  $\{(age, \downarrow), (goals, \uparrow)\}$  is a gradual pattern that may be interpreted as: “the lower the age, the more the goals scored.”

The quality of a gradual pattern is measured by *frequency support* which may be described as: “the proportion of objects/tuples/rows in a data set that fulfill that pattern.” For example, given the data set in Table 1, the pattern  $GP = \{(age, \downarrow), (goals, \uparrow)\}$  is fulfilled by tuples  $\{r1, r2, r3\}$  (which is 3 out of 4 tuples). Therefore, the frequency support,  $sup(GP)$ , of this pattern is 0.75.

On that account, given a minimum support threshold  $\sigma$ , a gradual pattern ( $GP$ ) is said to be **frequent** only if:  $sup(GP) \geq \sigma$ .

In the case of designing algorithms for mining gradual patterns from data sets, many existing works apply 3 main steps [2,7,9,10]:

1. identify gradual item sets (or patterns) that become *frequent* if their frequency support exceed a user-defined threshold,
2. ranking tuple pairs that fulfill the individual gradual items (of a candidate item set) and representing the ranks as *binary matrices* and,
3. applying a *bitwise AND* operator on the binary matrices in order to identify which gradual items may be joined to form a *frequent* gradual pattern.

For instance, given the data set in Table 1, we may identify 2 gradual patterns:  $gp_4 = \{(age, \downarrow), (games, \uparrow)\}$  and  $gp_5 = \{(games, \uparrow), (goals, \uparrow)\}$ . These 2 patterns require 3 gradual items  $g_1 = (age, \downarrow)$ ,  $g_2 = (games, \uparrow)$ ,  $g_3 = (goals, \uparrow)$  whose binary matrices  $M_{G_1}$ ,  $M_{G_2}$  and  $M_{G_3}$  (after ranking tuples of corresponding columns in Table 1) are shown in Table 2.

Table 2: Binary matrices  $M_{G_1}$ ,  $M_{G_2}$  and  $M_{G_3}$  for gradual items: (a)  $g_1 = (age, \downarrow)$ , (b)  $g_2 = (games, \uparrow)$ , (c)  $g_3 = (goals, \uparrow)$ .

(a)	(b)	(c)																																																																											
<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr><th>r</th><th>r1</th><th>r2</th><th>r3</th><th>r4</th></tr> </thead> <tbody> <tr><td>r1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>r2</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>r3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>r4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </tbody> </table>	r	r1	r2	r3	r4	r1	0	1	1	1	r2	0	0	1	1	r3	0	0	0	0	r4	0	0	0	0	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr><th>r</th><th>r1</th><th>r2</th><th>r3</th><th>r4</th></tr> </thead> <tbody> <tr><td>r1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>r2</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>r3</td><td>0</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>r4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </tbody> </table>	r	r1	r2	r3	r4	r1	0	1	1	1	r2	0	0	0	1	r3	0	1	0	1	r4	0	0	0	0	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr><th>r</th><th>r1</th><th>r2</th><th>r3</th><th>r4</th></tr> </thead> <tbody> <tr><td>r1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>r2</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>r3</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>r4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </tbody> </table>	r	r1	r2	r3	r4	r1	0	1	1	1	r2	0	0	1	1	r3	0	0	0	1	r4	0	0	0	0
r	r1	r2	r3	r4																																																																									
r1	0	1	1	1																																																																									
r2	0	0	1	1																																																																									
r3	0	0	0	0																																																																									
r4	0	0	0	0																																																																									
r	r1	r2	r3	r4																																																																									
r1	0	1	1	1																																																																									
r2	0	0	0	1																																																																									
r3	0	1	0	1																																																																									
r4	0	0	0	0																																																																									
r	r1	r2	r3	r4																																																																									
r1	0	1	1	1																																																																									
r2	0	0	1	1																																																																									
r3	0	0	0	1																																																																									
r4	0	0	0	0																																																																									

[2,7] propose the theorem that follows in order to join gradual items to form gradual patterns:

“Let  $gp_{12}$  be a gradual pattern generated by joining two gradual items  $g_1$  and  $g_2$ . The following matrix relation holds:  $M_{GP_{12}} = M_{G_1} \text{ AND } M_{G_2}$ ”.

This theorem relies heavily on the bitwise AND operator which provides good computational performance. For instance, we can apply a bitwise AND operation on the binary matrices in Table 1 in order to find binary matrices  $M_{GP_{12}}$  and  $M_{GP_{23}}$  for patterns  $gp_{12}$  and  $gp_{23}$  as shown in Table 3.

Table 3: Binary matrices  $M_{GP_{12}}$  and  $M_{GP_{23}}$  for gradual patterns: (a)  $gp_{12} = \{(age, \downarrow), (games, \uparrow)\}$ , (b)  $gp_{23} = \{(games, \uparrow), (goals, \uparrow)\}$ .

r	r1	r2	r3	r4
r1	0	1	1	1
r2	0	0	0	1
r3	0	0	0	0
r4	0	0	0	0

(a)

r	r1	r2	r3	r4
r1	0	1	1	1
r2	0	0	0	1
r3	0	0	0	1
r4	0	0	0	0

(b)

As can be seen in Table 2 and Table 3, the total sum of ordered ranks in the binary matrices is given by  $s = n(n - 1)/2$  where  $n$  is the number of columns/attributes. Therefore, the support of a gradual pattern  $gp$  is the ratio of *concordant rank count in the binary matrix* to the sum  $s$  [7].

### 3 State of the Art

Scientific data is increasing rapidly every year, thanks to technological advances in computing and storage efficiency [11,12]. Technologies such as HDF5 (Hierarchical Data Format v5) and Zarr provide high performance software and file formats that efficiently manage these huge volumes of data. For instance, [6] and [14] describe two models whose efficiencies have been greatly improved by using the Zarr and HDF5 data formats respectively.

According to [4], HDF5<sup>3</sup> is a technology suite that comprises a model, a software library and a hierarchical file format for storing and managing data. This suite is designed: (1) to support a wide variety of datatypes, (2) for fast Input/Output processing and (3) for managing *BigData*. These similar features are offered by Zarr<sup>4</sup> technology suite.

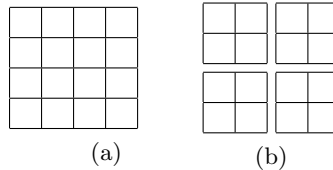


Fig. 1: (a) Contiguous storage layout and, (b) chunked storage layout.

One particular feature (provided by HDF5 and Zarr) that may be useful in mining gradual patterns from huge data sets is the *chunked storage layout*

<sup>3</sup> <https://portal.hdfgroup.org/display/HDF5/HDF5>

<sup>4</sup> <https://zarr.readthedocs.io/en/stable/index.html>

shown in Figure 1b. This feature allows for a huge data set to be split into multiple chunks which are stored separately in any order and any position within the HDF5/Zarr file. Additionally, chunks can be compressed, written and read individually, improving performance when dealing with such data sets [5].

Applying HDF5/Zarr chunked storage layout to binary matrices is one approach that may solve the problem (described in Section 1) of mining gradual patterns from huge data sets. The chunked storage layout may be exploited to allow the split of the bitwise AND operation (described in Section 1) on huge matrices (generated by reading and ranking all data set tuples in one attempt) into several repeated steps (where each step targets and loads manageable binary chunks into main memory).

However, using this approach implies chunking and storing binary matrices in secondary memory (i.e. HDF5/Zarr file) and, every repeated bitwise AND step includes the process of reading binary matrices from a secondary memory to main memory or/and writing updated binary matrices from the main memory to a secondary memory.

According to [4], chunked storage layout presents a higher overhead than contiguous storage layout when it comes to accessing and locating any element in the data set. The read/write overhead further increases when the chunked data set is compressed. Therefore, performance of the suggested approach of using a HDF5 chunked storage layout for gradual pattern mining may be greatly slowed down by the read/write overhead.

In the section that follows, we propose an approach that begins by chunking data set tuple reads in order to produce chunked binary matrices (getting rid of the need to store in HDF5/Zarr files).

## 4 Proposed Chunking Approach

In this section, we propose an approach for chunking binary matrices of gradual items into multiple small matrices that can be loaded and held into main memory piece-wisely in order to improve the memory usage efficiency. We modify the 3 main steps (described in Section 1) for mining gradual patterns as follows:

1. identify valid gradual patterns,
2. rank tuple pairs that fulfill the gradual items in the candidate gradual pattern in *chunks* and represent them in *multiple smaller* binary matrices and,
3. apply a bitwise AND operator on the *chunked* binary matrices in a *piecewise manner*.

### 4.1 Mapping Matrices into Chunked Layout

In the following, we use an example environment to expound on the steps of the proposed chunking approach. For the purpose of painting a clearer picture of this proposed approach, we use a sample data set (shown in Table 4a) to demonstrate the modified steps.

*Example 3.* Let  $gp = \{(age, \downarrow), (games, \uparrow)\}$  be a candidate gradual pattern. Using a user-defined chunk size (in this case we set the chunk size to 2) as shown in Table 4b.

Table 4: (a) Sample data set  $\mathcal{D}_2$ , (b) data set  $\mathcal{D}_2$  with its tuples chunked by a size of 2.

id	age	games	goals
r1	30	100	2
r2	28	400	4
r3	26	200	5
r4	25	500	8
r5	25	200	9
r6	24	500	1

(a)

id	age	games	goals
r1	30	100	2
r2	28	400	4
r3	26	200	5
r4	25	500	8
r5	25	200	9
r6	24	500	1

(b)

Firstly, we read and rank tuples fulfilling gradual items  $g_1 = (age, \downarrow)$  and  $g_2 = (games, \uparrow)$  using the chunks in a piecewise manner as shown in Table 5 and Table 6. Again in these two tables, we observe that the tuple rankings of gradual items  $g_1 = (age, \downarrow)$  and  $g_2 = (games, \uparrow)$  are represented by a total of 18 ( $2 \times 2$ ) binary matrices. In the classical approach, these rankings would be represented by 2 ( $6 \times 6$ ) binary matrices (see Table 1 and Table 2 in Section 1). Both approaches require the same size of memory to store all data in the binary matrices (which is 72 in total). However, the classical approach maps this data using a contiguous layout while, our proposed approach maps this data using a chunked layout.

Table 5: Chunked binary matrices for ranked tuples in Table 4b that fulfill gradual item  $g_1 = (age, \downarrow)$ .

r	r1	r2
r1	0	1
r2	0	0

(a)

r	r3	r4
r1	1	1
r2	1	1

(b)

r	r5	r6
r1	1	1
r2	1	1

(c)

r	r1	r2
r3	0	0
r4	0	0

(d)

r	r3	r4
r3	0	1
r4	0	0

(e)

r	r5	r6
r3	1	1
r4	0	1

(f)

r	r1	r2
r5	0	0
r6	0	0

(g)

r	r3	r4
r5	0	0
r6	0	0

(h)

r	r5	r6
r5	0	1
r6	0	0

(i)

Table 6: Chunked binary matrices for ranked tuples in Table 4b that fulfill gradual item  $g_2 = (games, \uparrow)$ .

<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r1</th><th>r2</th></tr> <tr><td>r1</td><td>0</td><td>1</td></tr> <tr><td>r2</td><td>0</td><td>0</td></tr> </table>	r̂	r1	r2	r1	0	1	r2	0	0	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r3</th><th>r4</th></tr> <tr><td>r1</td><td>1</td><td>1</td></tr> <tr><td>r2</td><td>0</td><td>1</td></tr> </table>	r̂	r3	r4	r1	1	1	r2	0	1	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r5</th><th>r6</th></tr> <tr><td>r1</td><td>1</td><td>1</td></tr> <tr><td>r2</td><td>0</td><td>1</td></tr> </table>	r̂	r5	r6	r1	1	1	r2	0	1	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r1</th><th>r2</th></tr> <tr><td>r3</td><td>0</td><td>1</td></tr> <tr><td>r4</td><td>0</td><td>0</td></tr> </table>	r̂	r1	r2	r3	0	1	r4	0	0	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r3</th><th>r4</th></tr> <tr><td>r3</td><td>0</td><td>1</td></tr> <tr><td>r4</td><td>0</td><td>0</td></tr> </table>	r̂	r3	r4	r3	0	1	r4	0	0
r̂	r1	r2																																															
r1	0	1																																															
r2	0	0																																															
r̂	r3	r4																																															
r1	1	1																																															
r2	0	1																																															
r̂	r5	r6																																															
r1	1	1																																															
r2	0	1																																															
r̂	r1	r2																																															
r3	0	1																																															
r4	0	0																																															
r̂	r3	r4																																															
r3	0	1																																															
r4	0	0																																															
(a)	(b)	(c)	(d)	(e)																																													
<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r5</th><th>r6</th></tr> <tr><td>r3</td><td>0</td><td>1</td></tr> <tr><td>r4</td><td>0</td><td>0</td></tr> </table>	r̂	r5	r6	r3	0	1	r4	0	0	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r1</th><th>r2</th></tr> <tr><td>r5</td><td>0</td><td>1</td></tr> <tr><td>r6</td><td>0</td><td>0</td></tr> </table>	r̂	r1	r2	r5	0	1	r6	0	0	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r3</th><th>r4</th></tr> <tr><td>r5</td><td>0</td><td>1</td></tr> <tr><td>r6</td><td>0</td><td>0</td></tr> </table>	r̂	r3	r4	r5	0	1	r6	0	0	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r5</th><th>r6</th></tr> <tr><td>r5</td><td>0</td><td>1</td></tr> <tr><td>r6</td><td>0</td><td>0</td></tr> </table>	r̂	r5	r6	r5	0	1	r6	0	0										
r̂	r5	r6																																															
r3	0	1																																															
r4	0	0																																															
r̂	r1	r2																																															
r5	0	1																																															
r6	0	0																																															
r̂	r3	r4																																															
r5	0	1																																															
r6	0	0																																															
r̂	r5	r6																																															
r5	0	1																																															
r6	0	0																																															
(f)	(g)	(h)	(i)																																														

Secondly, we perform a bitwise AND operation on the corresponding chunked matrices of gradual items  $g_1 = (age, \downarrow)$  and  $g_2 = (games, \uparrow)$  in order to determine if by joining them, the gradual pattern  $gp_{12} = \{(age, \downarrow), (games, \uparrow)\}$  is *frequent* (this is shown in Table 7). It should be underlined that gradual items (i.e.  $g_1$  and  $g_2$ ) should have binary matrices that match in number and size. Similarly, each matrix of one gradual item must be mapped to the corresponding matrix of the other gradual item during an AND operation. For instance, the matrix in Table 5(a) can only be mapped to the matrix in Table 6(a) during a bitwise AND operation to obtain the matrix in Table 7(a), and so on.

Table 7: Binary matrices for  $gp_{12} = \{(age, \downarrow), (games, \uparrow)\}$  after performing bitwise AND operation on chunked matrices of  $g_1$  and  $g_2$ .

<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r1</th><th>r2</th></tr> <tr><td>r1</td><td>0</td><td>1</td></tr> <tr><td>r2</td><td>0</td><td>0</td></tr> </table>	r̂	r1	r2	r1	0	1	r2	0	0	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r3</th><th>r4</th></tr> <tr><td>r1</td><td>1</td><td>1</td></tr> <tr><td>r2</td><td>0</td><td>1</td></tr> </table>	r̂	r3	r4	r1	1	1	r2	0	1	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r5</th><th>r6</th></tr> <tr><td>r1</td><td>1</td><td>1</td></tr> <tr><td>r2</td><td>0</td><td>1</td></tr> </table>	r̂	r5	r6	r1	1	1	r2	0	1	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r1</th><th>r2</th></tr> <tr><td>r3</td><td>0</td><td>0</td></tr> <tr><td>r4</td><td>0</td><td>0</td></tr> </table>	r̂	r1	r2	r3	0	0	r4	0	0	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r3</th><th>r4</th></tr> <tr><td>r3</td><td>0</td><td>1</td></tr> <tr><td>r4</td><td>0</td><td>0</td></tr> </table>	r̂	r3	r4	r3	0	1	r4	0	0
r̂	r1	r2																																															
r1	0	1																																															
r2	0	0																																															
r̂	r3	r4																																															
r1	1	1																																															
r2	0	1																																															
r̂	r5	r6																																															
r1	1	1																																															
r2	0	1																																															
r̂	r1	r2																																															
r3	0	0																																															
r4	0	0																																															
r̂	r3	r4																																															
r3	0	1																																															
r4	0	0																																															
(a)	(b)	(c)	(d)	(e)																																													
<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r5</th><th>r6</th></tr> <tr><td>r3</td><td>0</td><td>1</td></tr> <tr><td>r4</td><td>0</td><td>0</td></tr> </table>	r̂	r5	r6	r3	0	1	r4	0	0	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r1</th><th>r2</th></tr> <tr><td>r5</td><td>0</td><td>0</td></tr> <tr><td>r6</td><td>0</td><td>0</td></tr> </table>	r̂	r1	r2	r5	0	0	r6	0	0	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r3</th><th>r4</th></tr> <tr><td>r5</td><td>0</td><td>0</td></tr> <tr><td>r6</td><td>0</td><td>0</td></tr> </table>	r̂	r3	r4	r5	0	0	r6	0	0	<table border="1" style="border-collapse: collapse; text-align: left;"> <tr><th>r̂</th><th>r5</th><th>r6</th></tr> <tr><td>r5</td><td>0</td><td>1</td></tr> <tr><td>r6</td><td>0</td><td>0</td></tr> </table>	r̂	r5	r6	r5	0	1	r6	0	0										
r̂	r5	r6																																															
r3	0	1																																															
r4	0	0																																															
r̂	r1	r2																																															
r5	0	0																																															
r6	0	0																																															
r̂	r3	r4																																															
r5	0	0																																															
r6	0	0																																															
r̂	r5	r6																																															
r5	0	1																																															
r6	0	0																																															
(f)	(g)	(h)	(i)																																														

It is important to highlight that this chunked layout for binary matrices allows a bitwise AND operation to be broken down into multiple repetitions instead of a single operation as seen in the contiguous layout. This capability can be exploited to allow at least 2 chunked matching matrices to be loaded and held in main memory for every repeated AND operation. In this example, the bitwise AND operation is repeated at least 9 times for each twin of corresponding matrices.



Again in Table 7, we observe that binary matrices at (d), (g) and (h) sum up to 0; therefore, they are not significant in determining whether pattern  $gp_{12}$  is *frequent*. This phenomenon may be harnessed to increase the efficiency of this approach by skipping less significant binary matrices during the repetitive bitwise AND operation.

Lastly, let the user-defined support threshold be 0.5, then pattern  $gp_{12} = \{(age, \downarrow), (games, \uparrow)\}$  is *frequent* since its support is  $10/15$  or  $0.667$  (see derivation for frequency support in Example 2 - Section 1).

## 4.2 GRAD-L Algorithm

In the following, we present GRAD-L (Gradual-Large) shown in Algorithm 1 which implements the approach described in Section 4.1.

---

### Algorithm 1: GRAD-L (Gradual-Large)

---

**Input** : Data set  $\mathcal{D}$ , minimum support  $\sigma$ , chunk size  $C$   
**Output**: gradual patterns  $GP$

```

1  $GP \leftarrow \emptyset$ ;
2  $GP_c \leftarrow \text{gen\_gp\_candidates}()$ ;
3 for  $gp \in GP_c$  do
    ; /* gp - gradual pattern */
4      $M_{sum} \leftarrow 0$ ;
5     for  $gi \in gp$  do
        ; /* gi - gradual item */
6          $M_{bin} \leftarrow \text{chunk\_to\_matrix}(gi, \mathcal{D}, C)$ ;
7         if  $\text{calc\_sum}(M_{bin}) \leq 0$  then
8             Continue;
9         else
10            if  $gi$  is firstElement then
11                 $M_{bin1} \leftarrow M_{bin}$ ;
12                Break;
13            else
14                 $M_{bin2} \leftarrow M_{bin}$ ;
15                 $M_{bin} \leftarrow M_{bin1} \text{ AND } M_{bin2}$ ;
16                 $M_{sum} \leftarrow M_{sum} + \text{calc\_sum}(M_{bin})$ ;
17        end for
18         $sup \leftarrow \text{calc\_support}(M_{sum})$ ;
19        if  $sup \geq \sigma$  then
20             $GP.append(gp)$ ;
21 end for
22 return  $GP$ ;

```

---

In this algorithm, first we use existing techniques to identify gradual pattern candidates (*line 2*). Second, for each candidate we use its gradual items user-defined chunk-size to build chunked binary matrices and perform a bitwise

AND operation piece-wisely (*lines 3 – 13*). Third, we determine if the candidate pattern is *frequent* by comparing its support to the user-defined threshold.

### 4.3 Computational Complexity

In the following, we use the big-O notation [1,13] to analyze the computational complexity of GRAD-L algorithm. For every gradual pattern candidate that is generated: GRAD-L algorithm constructs multiple chunked binary matrices, performs a bitwise AND operation on the chunked binary matrices and calculates the frequency support of that candidate. We formulate the problem to and show the computational complexity of GRAD-L algorithm.

**Problem formulation.** Given a dataset  $\mathcal{D}$  with  $m$  attributes and  $n$  objects, we can generate numerous gradual pattern candidates each having  $k$  gradual items (where  $2 \geq k \leq m$ ). For each candidate, the classical GRAANK algorithm (proposed in [7]) builds binary matrices for every gradual item as shown in Table 2 (see Section 2). Next, a bitwise AND operation is performed on these matrices and frequency support of the resulting matrix computed as shown in Table 3 (see Section 2). Using the big-O notation, constructing the binary matrices through GRAANK algorithm results in a complexity of  $O(k \cdot n^2)$ . The bitwise AND operation and support computation have small complexities in comparison to that of constructing binary matrices.

For the case of GRAD-L algorithm, a user-defined chunk-size ( $q \times q$ ) (where  $q < n$ ) is used to construct  $y$  binary matrices for every gradual item. Therefore, the complexity of constructing binary matrices for every gradual pattern candidate is  $O(k \cdot \sum_1^y q^2)$ . Similarly, the bitwise AND operation and support computation have small and almost constant complexities.

**Search space size.** It is important to mention that for every generated candidate, the classical GRAANK algorithm and the proposed GRAD-L algorithm constructs binary matrices. Therefore, the complexity of  $x$  generated gradual pattern candidates is  $O(x \cdot k \cdot n^2)$  for GRAANK algorithm and  $O(x \cdot k \cdot \sum_1^y q^2)$  for GRAD-L algorithm.

## 5 Experiments

In this section, we present an experimental study of the computational and memory performance of our proposed algorithm. We implement the algorithm for GRAD-L approach described in Section 4 using Python Language. All the experiments were conducted on a High Performance Computing (HPC) **Meso@LR**<sup>5</sup> platform. We used one node comprising 14 cores of CPU and 128GB of RAM.

### 5.1 Source Code

The Python source code of the proposed algorithm is available at our GitHub repository: [https://github.com/owuordickson/large\\_gps.git](https://github.com/owuordickson/large_gps.git).

<sup>5</sup> <https://meso-lr.umontpellier.fr>

## 5.2 Data Set Description

Table 8: Experiment data sets.

Data set	#tuples	#attributes	Domain
Cargo 2000 (C2K)	3,942	98	Transport
Power Consump. (UCI)	2,075,259	9	Electrical

The ‘Cargo 2000’ data set, obtained from **UCI Machine Learning Repository** (UCI-MLR) [8], describes 98 tracking and tracing events that span 5 months of transport logistics execution. The ‘Power Consumption’ data set, obtained from **UCI-MLR** [3], describes the electric power consumption in one household (located in Sceaux, France) in terms of active power, voltage and global intensity with a one-minute sampling rate between 2006 and 2010.

## 5.3 Experiment Results

In the following, we present our experimental results which show the computational and memory usage of our proposed algorithm (GRAD-L), HDF5-based algorithm (GRAD-H5) and classical algorithm (GRAD) for mining gradual patterns. Using these 3 algorithms, we perform test runs on C2K and UCI data sets with minimum support threshold ( $\sigma$ ) set to 0.1.

We split the UCI data set into 5 data sets whose number of tuples range from 10,000 (10K), 116,203 (116K), 523,104 (523K), 1,000,000 (1M) and 2,075,259 (2M). All the test runs were repeated several times and the results are available at: [https://github.com/owuordickson/meso-hpc-lr/tree/master/results/large\\_gps/](https://github.com/owuordickson/meso-hpc-lr/tree/master/results/large_gps/).

## Computational Performance Results

Table 9 shows a result summary for computational run-time performance, number of extracted patterns and memory utilization of algorithms GRAD, GRAD-H5 and GRAD-L. It is important to highlight that algorithms GRAD and GRAD-H5 yield ‘*Memory Error*’ when executed on UCI data sets whose tuple size is greater than 100,000 and 500,000 respectively (represented as ‘NaN’ in Table 9). Figure 2 illustrates how run-time and memory usage breaks for GRAD (due to ‘*Memory Error*’) grows exponentially for GRAD-H5 (due to read/write overhead).

Computational run-time results show that GRAD-L (which implements our proposed chunked layout for loading binary matrices into memory) is the fastest of the 3 algorithms when executed in all the data sets. GRAD (which uses contiguous layout to load and hold binary matrices into memory) is relatively fast (compared to GRAD-H5) when executed on data set C2K and UCI 10K. However, it yields ‘*Memory Error*’ for UCI data sets greater than 100K since

Table 9: Summary of experiment results.

Data set	Size	Algorithm	Run-time (sec)		No. of patterns		Memory (KiB)	
			St.d.	Mean	St.d.	Mean	St.d.	Mean
C2K	3.9K	GRAD	24.125	702.536	0.000	2.000	2.044	<b>172.089</b>
		GRAD-H5	12.162	3786.30	0.000	2.000	4.313	497.450
		GRAD-L	0.653	<b>15.821</b>	0.894	1.400	38.643	501.200
UCI	10K	GRAD	1.448	51.682	0.408	1.833	0.564	<b>109.617</b>
		GRAD-H5	98.794	47.162	0.000	2.000	118.713	172.383
		GRAD-L	0.630	<b>5.017</b>	0.516	1.333	1.089	291.350
UCI	116K	GRAD	NaN	NaN	NaN	NaN	NaN	NaN
		GRAD-H5	143.543	33209.50	0.000	2.000	0.566	427.600
		GRAD-L	63.772	<b>524.787</b>	0.000	2.000	15.312	<b>276.367</b>
UCI	523K	GRAD	NaN	NaN	NaN	NaN	NaN	NaN
		GRAD-H5	NaN	NaN	NaN	NaN	NaN	NaN
		GRAD-L	1716.374	<b>10947.60</b>	1.000	<b>1.000</b>	22.228	<b>287.800</b>
UCI	1M	GRAD	NaN	NaN	NaN	NaN	NaN	NaN
		GRAD-H5	NaN	NaN	NaN	NaN	NaN	NaN
		GRAD-L	367.723	<b>39460.3</b>	0.577	<b>1.667</b>	1.386	<b>350.400</b>
UCI	2M	GRAD	NaN	NaN	NaN	NaN	NaN	NaN
		GRAD-H5	NaN	NaN	NaN	NaN	NaN	NaN
		GRAD-L	22113.287	<b>162616.7</b>	0.577	<b>1.333</b>	5.605	<b>367.333</b>

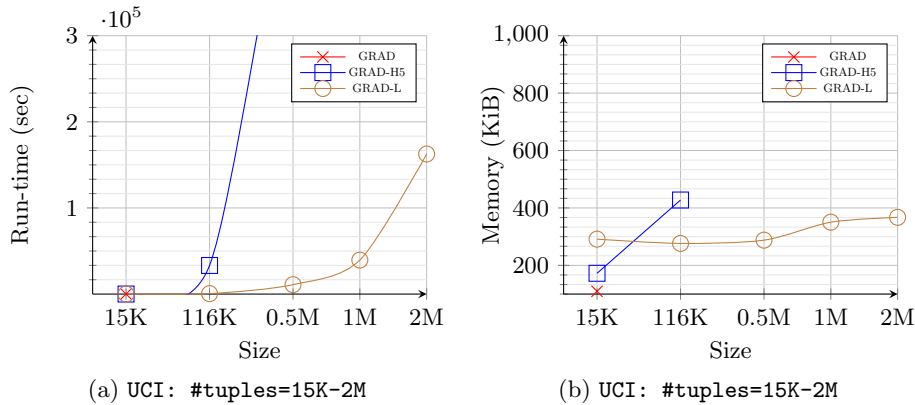


Fig. 2: Plot of run-time and memory usage against size of UCI data set.

sizes of binary matrices in main memory increase exponentially within a very short time and this exceeds the available memory. GRAD-H5 (which implements HDF5-based approach for dealing with huge binary matrices) has the slowest run-times of all the 3 algorithms. This may be attributed to read-write overhead that occurs in all bitwise AND operations.

Memory usage results show that GRAD has better memory utilization on data sets C2K and UCI 10K. However, GRAD-L has the best overall memory utilization since it does not yield ‘*Memory Error*’ on any of the 6 data sets. Number of patterns results show that almost all 3 algorithms extract similar number of gradual patterns.

### Consistent Gradual Patterns

This experiment reveals the consistent gradual patterns extracted by the 3 algorithms from data sets C2K and UCI when minimum support threshold ( $\sigma$ ) is set to 0.1. The results are shown in Table 10.

Table 10: Consistent gradual patterns.

Data set	Gradual patterns
C2K (3.9K)	$\{(i2\_rcs\_e, \downarrow), (o\_legid, \uparrow), (o\_dlv\_e, \downarrow)\}, sup = 0.23$
UCI (10K)	$\{(Sub\_metering\_3, \uparrow), (Global\_intensity, \downarrow)\}, sup = 0.172$
UCI (116K)	$\{(Voltage, \downarrow), (Sub\_metering\_1, \uparrow)\}, sup = 0.109$
UCI (523K)	$\{(Global\_intensity, \downarrow), (Sub\_metering\_2, \uparrow)\}, sup = 0.16$
UCI (1M)	$\{(Global\_reactive\_power, \downarrow), (Global\_intensity, \downarrow)\}, sup = 0.558$
UCI (2M)	$\{(Sub\_metering\_3, \uparrow), (Sub\_metering\_2, \downarrow)\}, sup = 0.159$

It is important to mention that for huge data sets, extracted gradual patterns are of relatively low quality. For this reason, we chose a low minimum support threshold ( $\sigma = 0.1$ ) in order to extract gradual patterns from all the data sets.

## 6 Conclusion and Future Works

In this paper, we explore two different approaches to solve the problem of mining gradual patterns from huge data sets (see Section 3 and Section 4). From the experiment results (presented in Section 5), we conclude that GRAD-L algorithm is the best performing algorithm (relative to GRAD and GRAD-H5 algorithms) both in terms of computational run-time and memory utilization. This proves that our proposed chunking approach (described in Section 4) utilizes main memory more efficiently than the classical approach (proposed in [7]) HDF5-based chunking approach (discussed in Section 3).

Future work may involve extensive experimentation on the GRAD-L approach with the aim of improving its memory usage efficiency even further. In addition to this, other future work may entail integrating the GRAD-L approach into data lake environments that hold numerous huge data sets. A good example of such an environment is OREME<sup>6</sup> which is a scientific research observatory that holds a huge collection of large scientific data sets.

## Acknowledgements

This work has been realized with the support of the High Performance Computing Platform: **MESO@LR**<sup>7</sup>, financed by the Occitanie / Pyrénées-Méditerranée Region, Montpellier Mediterranean Metropole and Montpellier University.

<sup>6</sup> <https://data.oreme.org/>

<sup>7</sup> <https://meso-lr.umontpellier.fr>

## References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)
2. Di-Jorio, L., Laurent, A., Teisseire, M.: Mining frequent gradual itemsets from large databases. In: *Advances in Intelligent Data Analysis VIII*. pp. 297–308. Springer-Verlag, Berlin, Heidelberg (2009). <https://doi.org/10/dvzk9c>
3. Dua, D., Graff, C.: UCI machine learning repository (2019), <http://archive.ics.uci.edu/ml>
4. Folk, M., Heber, G., Koziol, Q., Pourmal, E., Robinson, D.: An overview of the hdf5 technology suite and its applications. In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. p. 36–47. AD '11, Association for Computing Machinery, New York, NY, USA (2011). <https://doi.org/10.1145/1966895.1966900>
5. Howison, M.: Tuning hdf5 for lustre file systems (2010), <https://www.osti.gov/biblio/1050648>
6. Krijnen, T., Beetz, J.: An efficient binary storage format for ifc building models using hdf5 hierarchical data format. *Automation in Construction* **113**, 103134 (2020). <https://doi.org/https://doi.org/10.1016/j.autcon.2020.103134>
7. Laurent, A., Lesot, M.J., Rifqi, M.: Graank: Exploiting rank correlations for extracting gradual itemsets. In: *Proceedings of the 8th International Conference on Flexible Query Answering Systems*. pp. 382–393. FQAS '09, Springer-Verlag, Berlin, Heidelberg (2009). <https://doi.org/10/dn7jd7>
8. Metzger, A., Leitner, P., Ivanović, D., Schmieders, E., Franklin, R., Carro, M., Dustdar, S., Pohl, K.: Comparing and combining predictive business process monitoring techniques. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* **45**(2), 276–290 (2015)
9. Negrevergne, B., Termier, A., Rousset, M.C., Méhaut, J.F.: Paraminer: a generic pattern mining algorithm for multi-core architectures. *Data Mining and Knowledge Discovery* **28**(3), 593–633 (2014). <https://doi.org/10.1007/s10618-013-0313-2>
10. Owuor, D., Laurent, A., Orero, J.: Mining fuzzy-temporal gradual patterns. In: *2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. pp. 1–6. IEEE, New York, NY, USA (june 2019). <https://doi.org/10.1109/FUZZ-IEEE.2019.8858883>
11. Owuor, D., Laurent, A., Orero, J., Lobry, O.: Gradual pattern mining tool on cloud. *Extraction et Gestion des Connaissances: Actes EGC'2021* (2021)
12. Owuor, D.O., Laurent, A., Orero, J.O.: Exploiting IoT data crossings for gradual pattern mining through parallel processing. In: *ADBIS, TPD and EDA 2020 Common Workshops and Doctoral Consortium*. pp. 110–121. Springer International Publishing, Cham (2020). <https://doi.org/10/f3rg>
13. Vaz, R., Shah, V., Sawhney, A., Deolekar, R.: Automated big-o analysis of algorithms. In: *2017 International Conference on Nascent Technologies in Engineering (ICNTE)*. pp. 1–6 (Jan 2017). <https://doi.org/10.1109/ICNTE.2017.7947882>
14. Xu, H., Wei, W., Dennis, J., Paul, K.: Using cloud-friendly data format in earth system models. In: *AGU Fall Meeting Abstracts*. pp. IN13C–0728 (Dec 2019)