Big-SeqDB-Gen: A Formal and Scalable Approach for Parallel Generation of Big Synthetic Sequence Databases

Rim Moussa(✉)

ENICarthage Engineering School of Carthage, Carthage, Tunisia
rim.moussa@esti.rnu.tn

Abstract. The recognition that data is of big economic value and the significant hardware achievements in low cost data storage, high-speed networks and high performance parallel computing, foster new research directions on large-scale knowledge discovery from *big sequence databases*. There are many applications involving *sequence databases*, such as customer shopping sequences, web clickstreams, and biological sequences. All these applications are concerned by the big data problem. There is no doubt that fast mining of billions of sequences is a challenge. However, due to the non availability of big data sets, it is not possible to assess knowledge discovery algorithms over big sequence databases. For both privacy and security concerns, Companies do not disclose their data. In the other hand, existing synthetic sequence generators are not up to the big data challenge.

In this paper, *first* we propose a formal and scalable approach for *Parallel Generation of Big Synthetic Sequence Databases*. Based on *Whitney numbers*, the underlying *Parallel Sequence Generator* (*i*) creates billions of distinct sequences in parallel and (*ii*) ensures that *injected sequential patterns* satisfy user-specified *sequences' characteristics*. *Second*, we report a scalability and scale-out performance study of the *Parallel Sequence Generator*, for various sequence databases' sizes and various number of *Sequence Generators* in a shared-nothing cluster of nodes.

Keywords: Big synthetic data · Sequence database · Sequential pattern · Parallel generator · Whitney numbers

1 Introduction

There are many applications involving *sequence databases*, namely customer shopping sequences, web clickstreams, biological sequences, and sequences of events in science and engineering. Jiawei Han, Micheline Kamber and Jian Pei define a *Sequence Database* as it *consists of sequences of ordered elements or events, recorded with or without a concrete notion of time* [1]. Problems addressed within sequence databases, include mining the frequently occurring patterns [2–6], mining for outliers patterns [7,8], building efficient sequence databases and indexes for sequence data [9,10], mining compressing sequential patterns [11,12] and comparing sequences for similarity [13]. Most published papers in the literature address

© Springer International Publishing Switzerland 2016
R. Nambiar and M. Poess (Eds.): TPCTC 2015, LNCS 9508, pp. 61–76, 2016.
DOI: 10.1007/978-3-319-31409-9_5

the *Frequent Sequential Pattern Mining problem.* The latter was introduced by Agrawal and Srikant in 1995 [2] and is defined as follows: *Given a database of sequences, where each sequence consists of a list of transactions ordered by transaction time and each transaction is a set of items, sequential pattern mining is to discover all sequential patterns with a user-specified minimum support.* An example of a sequential pattern is that *customers typically rent video Star Wars, then Empire Strickes Back, then Return of the Jedi.* Elements of a sequential pattern might be sets of items (i.e., itemsets), with a sequential pattern which looks as *customers typically rent video Star Wars, then the triplet Return of the Jedi, Lord of Ring and Alien movies.*

Experiences with mining big data ascertain that *more data usually beats better algorithms* [14]. All pattern mining algorithms over sequence databases are concerned by the big data challenges. Big data adds a further level of complexity to any knowledge discovery algorithm. However, due to the non availability of big real data sets, it is not possible to assess sequential patterns' mining algorithms over big sequence databases. For both privacy and security concerns, companies do not disclose and share their data. It is also complex to encode real data sets, while preserving their characteristics. On the other hand, available synthetic sequence generators such as *IBM Quest Synthetic Data Generator* [15] are not up to the big data challenge. Hence, in this paper, we propose a formal and scalable approach based on *Whitney numbers* for *Parallel Generation of Big Synthetic Sequence Databases* satisfying both user-specified *sequences' characteristics* and *velocity* requirements.

In this paper, we make the following contributions,

- We propose a new efficient and fast approach based on *Whitney numbers* for a parallel generation of big sequence databases,
- We assess by performance measurements the scalability and the scale-out of the proposed *Parallel Sequence Generator* on a GRID5000 cluster of shared-nothing nodes [16]. Performance measurements report the throughput in terms of MBps and in terms of number of sequences created and stored per second for various number of *sequence generators* (termed *workers* in distributed computing) and various *number of injected sequential patterns.* The latter grows linearly with the sequence database size.

The paper is organized as follows, Sect. 2 overviews existing sequence generators. Section 3 presents basic concepts of *sequence databases.* Section 4 details our proposed *Parallel Sequence Generator* (for short *PSG*), precisely the requirements it fulfills and its computational model. Section 5 presents a thorough performance study of *PSG.* Finally, Sect. 6 concludes the paper and presents future research.

2 Related Work

The most known generator of sequential patterns is the *IBM Quest Synthetic Data Generator* [15,17,18]. A second testbed for patterns' mining is described in [19], although the testbed is not available for download. After a performance study of distributed implementations [18] of *GSP* [3] and *PrefixSpan* [6] algorithms, we investigated the source code of the *IBM Quest Synthetic Data Generator*. The generator reveals the shortcomings enumerated below,

1. First issue is related to the fact that the benchmark is not documented. The original source code is no longer available through IBM web site[1]. Available implementations address portability and compatibility issues.
2. Second issue is related to sequences' generation. Indeed, regards generated sequences, no evident correlation could be drawn from input parameters, and particularly how do they should scale with the sequence database size. A random process is used for generating sequences and corrupting base sequential patterns used for populating the sequence database [15,17,18]. This process does not guarantee that a sequential pattern repeats a number of times proportional to the database size.
3. Third issue is related to capacity and velocity requirements, the *IBM Quest Synthetic Data Generator* was not designed for *fast* generation of *big* sequence databases.

Most data mining benchmarks relate to small test datasets. Many big data benchmarks exist, but have different objectives. For instance, the *TeraSort* benchmark [20] measures the time to sort 1 TB (10 billions of 100 Bytes records) of randomly generated data. The *Parallel Data Generator Framework* (PDGF) [21,22] allows parallel generation of big relational databases. The *BigDataBench* [23] proposes several benchmarks specifications to model five important application domains, including search engine, social networks, e-commerce, multimedia data analytics and bioinformatics.

To the best of our knowledge, the *Parallel Sequence Generator* is the first synthetic sequence generator addressing big data and velocity requirements. Our contribution is then three fold (*i*) a computational approach based on *Whitney numbers* allowing the generation of billions of data sequences, (*ii*) an efficient implementation and an experimental assessment of the scalability and the scale-out of the proposed *Parallel Sequence Generator*, finally (*iii*) an open-source code, available for download in order to help researchers in benchmarking knowledge discovery algorithms over big sequence databases [18].

3 Sequence Databases: Concepts and Primitives

Given a database of customer purchase histories, one would like to mine and predict the behaviors of customers. A customer buying A and then B is likely to buy C, D and E. A marketing manager can then send advertisements of products C, D and E to clients who have bought A and then B. $\langle\{A\}\{B\}\{C, D, E\}\rangle$ is termed a *sequential pattern*.

[1] URL: http://www.research.ibm.com/labs/almaden/index.shtml#assocSynData does not point to the benchmark homepage.

Sequence ID	*Sequence*
s_1	$\langle\{1\}\{1,2,3\}\{1,3\}\{1,4\}\{3,6\}\rangle$
s_2	$\langle\{1,4\}\{3\}\{2,3\}\{1,5\}\rangle$
s_3	$\langle\{5,6\}\{1,2\}\{4,6\}\{3\}\{2\}\rangle$
s_4	$\langle\{5\}\{7\}\{1,6\}\{3\}\{2\}\{3\}\rangle$

Fig. 1. Example of $\mathcal{S}$-a database of sequences.

Figure 1 illustrates a *sequence database* $\mathcal{S}$ composed of four sequences, which abstract customer-shopping sequences. The set of items in $\mathcal{S}$ is $\{1,2,3,4,5,6,7\}$. The *count* of a sequence s, denoted by $count(s)$, is defined as the number of sequences that contain s. For instance for $s = \langle\{1\}\{3\}\{3\}\rangle$, $count(s) = 2$. Indeed, s is a *subsequence* of both s_1 and s_2, denoted as $s \sqsubseteq s_1$ and $s \sqsubseteq s_2$. Inversely, s_1 and s_2 are *supersequences* of s. A sequence contributes only one to the count of a sequential pattern, for instance $count(\langle\{1\}\{1\}\rangle) = 2$. The support of a sequence s, denoted by $support(s)$, is defined as $count(s)$ divided by the total number of sequences seen. If $support(s) \geq \tau$, where τ is a user-supplied minimum support threshold, then we say that s is a *frequent sequential pattern*. For $\tau = 0.75$, $s' = \langle\{1\}\{3\}\{2\}\rangle$ is a frequent sequential pattern. Indeed, s' is a subsequence of all of s_2, s_3 and s_4. Finally, the *length* of a sequence s, denoted by $|s|$ is the sum all its itemsets' lengths, and a *k-sequence* is a sequence of length k. For instance, s_1 is a 9-sequence and $\langle\{1\}\{3\}\{3\}\rangle$ is a 3-sequence.

The major approaches for mining of sequential patterns [2–6] are based on the The *Apriori property*. The latter states that *all non empty subsets of a frequent itemset must also be frequent*, including frequent items. This property is also denoted *antimonotonicity*. If a sequence is infrequent, all of its supersequences must be infrequent, and if a sequence is frequent, all of its subsequences must be frequent. For instance for $\tau = 0.75$, all of $\langle\{1\}\{3\}\rangle$, $\langle\{1\}\{2\}\rangle$, $\langle\{3\}\{2\}\rangle$, are subsequences of $s' = \langle\{1\}\{3\}\{2\}\rangle$ and are frequent sequential patterns. For more details, readers are invited to check the seminal paper on *Sequential Patterns Mining* by Agrawal R. and Srikant R. [2].

4 Parallel Generation of a Sequence Database

Very early, the Database community proposed synthetic benchmarks, which handle big data and variety of workloads. Our work is mainly inspired by [24], the TPC benchmarks [25], and PDGF [21,22]. In the sequel, first, we define goals that the proposed *Parallel Sequence Generator* (for short *PSG*) fulfills. Second, we detail a formal method based on *Whitney enumerators* for the enumeration of *sequential patterns*, denoted as *source sequences* in this paper.

4.1 Requirements

The *Parallel Sequence Generator* is designed so that it fulfills well known requirements of benchmarking [25,26], namely,

- *Relevance*: *PSG* implements *Whitney Enumerators* a computational method which efficiently enumerates in parallel distinct *source sequences* to be injected in the *sequence database*,
- *Repeatability*: for multiple runs with same input parameters, *PSG* outputs a sequence database with same characteristics, namely sequence database volume, sequence size, number of sequences, average number of items per sequence, average number of itemsets per sequence, and *source sequences* with lengths and quotas equal to input parameters,
- *Economy*: *PSG* is open-source and is hardware and platform independent,
- *Fairness*: the generator does not overfit a particular algorithm of sequential pattern mining, and provides directions to generate a sequence database for testing the mining capacity of algorithms through variation of database size and sequential patterns size.
- *Performance*: *PSG* reports metrics demonstrating its velocity for synthetic sequence generation. Experiments are carried out in order to assess scalability and scale-out performance of *PSG*.

4.2 ***Whitney Enumerators*** for the Enumeration of ***Source Sequences***

Raissi and Pei used *Whitney numbers* in order to bound the number of frequent sequential patterns [27]. *PSG* implements *Whitney Enumerators* a computational method based on *Whitney numbers* which efficiently enumerates in parallel distinct *source sequences*. *PSG* is based on the *Apriori property*: given a finite set of items $\mathcal{I}$, which cardinality is n; *PSG* generates distinct *source sequences* of a given length k, to be injected in the *sequence database*. Next, we show how to enumerate *source sequences* using *Whitney enumerators*.

Enumerating the *k-sequences* is described in the recurrence relation introduced in Eq. 1. $\mathcal{WE}_k$ stands for *Whitney Enumerator* of *source sequences* of length k and $\mathcal{E}\binom{n}{i}$ stands for *Combination Enumerator*.

$$\mathcal{WE}_k = \bigcup_{i=0}^{k-1} \mathcal{E}\binom{n}{k-i} \times \mathcal{WE}_i \ \ with \begin{cases} n = |\mathcal{I}| \\ \mathcal{WE}_0 = \varnothing \\ \mathcal{WE}_1 = \mathcal{E}\binom{n}{1} \end{cases} \tag{1}$$

For instance, for $\mathcal{I} = \{1, 2\}$,

- $\mathcal{WE}_1 = \mathcal{E}\binom{2}{1} = \{1\}, \{2\}$
- $\mathcal{WE}_2 = \bigcup_{i=0}^{1} \mathcal{E}\binom{2}{2-i} \times \mathcal{WE}_i = \mathcal{E}\binom{2}{2} \times \mathcal{WE}_0 \cup \mathcal{E}\binom{2}{1} \times \mathcal{WE}_1 = \{1, 2\} \times \varnothing \cup \{1\}, \{2\} \times \{1\}, \{2\} = \{1, 2\}, \{1\}\{1\}, \{1\}\{2\}, \{2\}\{1\}, \{2\}\{2\}$.

Figure 2 illustrates compositions of *source sequences* obtained from $\mathcal{WE}_5$ and $\mathcal{I}$, such that $|\mathcal{I}| = 10$. Notice that each branch allows the enumeration of a number of *source sequences* presented in blue. For instance, the last branch

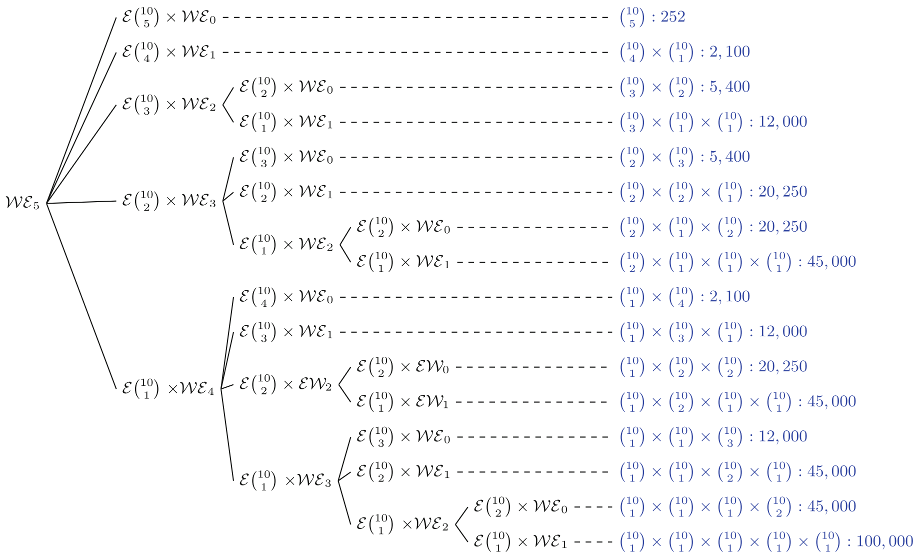

$\mathcal{WE}_5$
$\mathcal{E}\binom{10}{5} \times \mathcal{WE}_0$ — $\binom{10}{5}$: 252
$\mathcal{E}\binom{10}{4} \times \mathcal{WE}_1$ — $\binom{10}{4} \times \binom{10}{1}$: 2,100
$\mathcal{E}\binom{10}{3} \times \mathcal{WE}_2$
$\mathcal{E}\binom{10}{2} \times \mathcal{WE}_0$ — $\binom{10}{3} \times \binom{10}{2}$: 5,400
$\mathcal{E}\binom{10}{1} \times \mathcal{WE}_1$ — $\binom{10}{3} \times \binom{10}{1} \times \binom{10}{1}$: 12,000
$\mathcal{E}\binom{10}{2} \times \mathcal{WE}_3$
$\mathcal{E}\binom{10}{3} \times \mathcal{WE}_0$ — $\binom{10}{2} \times \binom{10}{3}$: 5,400
$\mathcal{E}\binom{10}{2} \times \mathcal{WE}_1$ — $\binom{10}{2} \times \binom{10}{2} \times \binom{10}{1}$: 20,250
$\mathcal{E}\binom{10}{1} \times \mathcal{WE}_2$
$\mathcal{E}\binom{10}{2} \times \mathcal{WE}_0$ — $\binom{10}{2} \times \binom{10}{1} \times \binom{10}{2}$: 20,250
$\mathcal{E}\binom{10}{1} \times \mathcal{WE}_1$ — $\binom{10}{2} \times \binom{10}{1} \times \binom{10}{1} \times \binom{10}{1}$: 45,000
$\mathcal{E}\binom{10}{1} \times \mathcal{WE}_4$
$\mathcal{E}\binom{10}{4} \times \mathcal{WE}_0$ — $\binom{10}{1} \times \binom{10}{4}$: 2,100
$\mathcal{E}\binom{10}{3} \times \mathcal{WE}_1$ — $\binom{10}{1} \times \binom{10}{3} \times \binom{10}{1}$: 12,000
$\mathcal{E}\binom{10}{2} \times \mathcal{EW}_2$
$\mathcal{E}\binom{10}{2} \times \mathcal{EW}_0$ — $\binom{10}{1} \times \binom{10}{2} \times \binom{10}{2}$: 20,250
$\mathcal{E}\binom{10}{1} \times \mathcal{EW}_1$ — $\binom{10}{1} \times \binom{10}{2} \times \binom{10}{1} \times \binom{10}{1}$: 45,000
$\mathcal{E}\binom{10}{1} \times \mathcal{WE}_3$
$\mathcal{E}\binom{10}{3} \times \mathcal{WE}_0$ — $\binom{10}{1} \times \binom{10}{1} \times \binom{10}{3}$: 12,000
$\mathcal{E}\binom{10}{2} \times \mathcal{WE}_1$ — $\binom{10}{1} \times \binom{10}{1} \times \binom{10}{2} \times \binom{10}{1}$: 45,000
$\mathcal{E}\binom{10}{1} \times \mathcal{WE}_2$
$\mathcal{E}\binom{10}{2} \times \mathcal{WE}_0$ — $\binom{10}{1} \times \binom{10}{1} \times \binom{10}{1} \times \binom{10}{2}$: 45,000
$\mathcal{E}\binom{10}{1} \times \mathcal{WE}_1$ — $\binom{10}{1} \times \binom{10}{1} \times \binom{10}{1} \times \binom{10}{1} \times \binom{10}{1}$: 100,000

Fig. 2. *Source sequence* enumeration and count for $\mathcal{WE}_5$ ($k = 5$) and $n = 10$ (Color figure online).

allows the enumeration of 10^5 *source sequences*, such that each is composed of five singletons, while the first branch's capacity is only 252 sequences, and each *source sequence* is a single itemset which contains five items. For small values: $k = 5$ and $n = 10$, one could enumerate $392,002$ *source sequences*.

Equation 2 introduced by Raissi and Pei [27] allows the count of each *Whitney number* in terms of number of *source sequences*. Table 1 presents capacities of *Whitney numbers* while varying k for $|\mathcal{I}| = 50$, as well as the count of single itemset sequences and k itemsets sequences. Notice that, for $|\mathcal{I}| = 50$, $\mathcal{WE}_5$ allows the enumeration of more than one billion of *source sequences*, and $\mathcal{WE}_{10}$ enumerates more than two and half trillions of *source sequences* (one trillion = 10^{18}). For higher values of k and $|\mathcal{I}|$, enumerating and storing all possible *source sequences* can turn into high storage costs and memory leaks. Next, we detail an efficient enumeration procedure.

$$\mathcal{W}_k = \sum_{i=0}^{k-1} \binom{n}{k-i} \times \mathcal{W}_i \ \ with \begin{cases} n = |\mathcal{I}| \\ \mathcal{W}_0 = 1 \\ \mathcal{W}_1 = n \end{cases} \tag{2}$$

4.3 Efficient Enumeration of *Source Sequences*

Hereafter, we describe how *Parallel Sequence Generator* enumerates in parallel variety of *source sequences* at less cost.

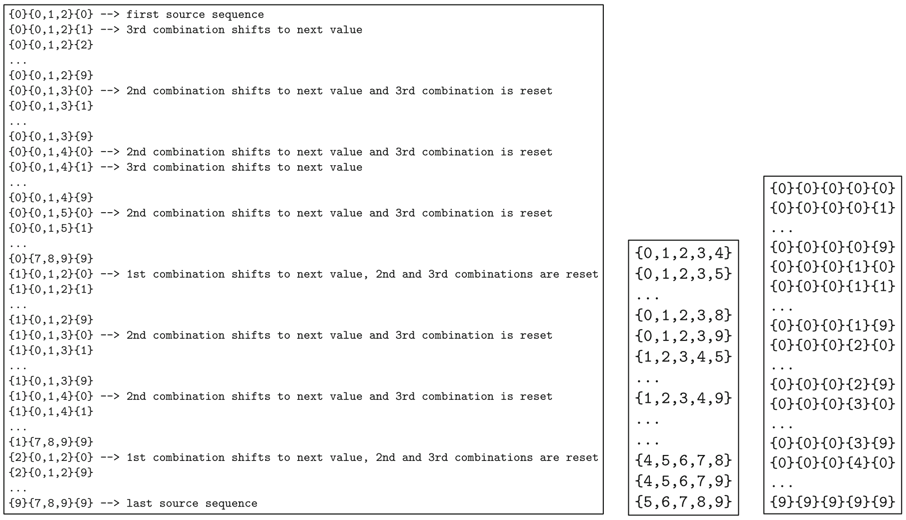
Table 1. Whitney numbers' capacities for $|\mathcal{I}| = 50$.

W_k	Nbr. of source sequences	Nbr. of k itemsets source sequences	Nbr. of single itemset source sequences
W_1	50	50	50
W_2	3,725	2,500	1,225
W_3	267,100	125,000	19,600
W_4	19,128,425	6,250,000	230,300
W_5	1,370,262,510	312,500,000	2,118,760
W_6	98,160,302,325	15,625,000,000	15,890,700
W_7	7,031,803,751,400	781,250,000,000	99,884,400
W_8	503,729,624,143,775	39,062,500,000,000	536,878,650
W_9	36,085,128,550,756,000	1,953,125,000,000,000	2,505,433,700
W_{10}	2,584,990,924,265,820,000	97,656,250,000,000,000	10,272,278,170

Enumerate *Source Sequences* at Less Cost. We propose algorithms for the enumeration of a *Combination* contents as well as for the *Cross product of Combinations*. Our algorithms save a *current context*, which is composed of a *current combination* and a *current cross of combinations*. The enumeration is then performed through successive calls of *next sequence method*. The source code of *Whitney numbers* and *Whitney enumerators* manipulations for *source sequences*' enumeration is available for download [18].

Figure 3 demonstrates the enumeration process. Starting with the first source sequence of the 10th branch of $\mathcal{WE}_5$, which is $\{0\}\{0,1,2\}\{0\}$, the next source sequence is obtained by shifting third combination to next value in order to obtain source sequence $\{0\}\{0,1,2\}\{1\}$. Successive calls of *next sequence method* continue so, until we reach source sequence $\{0\}\{0,1,2\}\{9\}$. The next source sequence is obtained by reset of third combination and shift of second combination to next value, in order to obtain source sequence $\{0\}\{0,1,3\}\{0\}$. The enumeration procedure is generalized to cross products of multiple combinations [18].

Enumerate Variety of *Source Sequences*. As illustrated in Fig. 2, *source sequences* of same length k have different number of itemsets. The first branch is composed of a single itemset, while the last branch is composed of k itemsets *source sequences*. A depth-first traversal of the tree will enumerate source sequences branch by branch. Within each branch, *source sequences* feature the same number of itemsets and the same number of items for each itemset. For the example illustrated in Fig. 2, the enumeration of the first 10,000 *source sequences* stops at the third branch, and does not include any *source sequence* beyond this branch. This might have an impact on the mining process. Thus, in order to variate generated *source sequences*, we preponderate the number of source sequences to be generated along each branch capacity of the tree. Likewise, the



```
{0}{0,1,2}{0} --> first source sequence
{0}{0,1,2}{1} --> 3rd combination shifts to next value
{0}{0,1,2}{2}
...
{0}{0,1,2}{9}
{0}{0,1,3}{0} --> 2nd combination shifts to next value and 3rd combination is reset
{0}{0,1,3}{1}
...
{0}{0,1,3}{9}
{0}{0,1,4}{0} --> 2nd combination shifts to next value and 3rd combination is reset
{0}{0,1,4}{1} --> 3rd combination shifts to next value
...
{0}{0,1,4}{9}
{0}{0,1,5}{0} --> 2nd combination shifts to next value and 3rd combination is reset
{0}{0,1,5}{1}
...
{0}{7,8,9}{9}
{1}{0,1,2}{0} --> 1st combination shifts to next value, 2nd and 3rd combinations are reset
{1}{0,1,2}{1}
...
{1}{0,1,2}{9}
{1}{0,1,3}{0} --> 2nd combination shifts to next value and 3rd combination is reset
{1}{0,1,3}{1}
...
{1}{0,1,3}{9}
{1}{0,1,4}{0} --> 2nd combination shifts to next value and 3rd combination is reset
{1}{0,1,4}{1}
...
{1}{7,8,9}{9}
{2}{0,1,2}{0} --> 1st combination shifts to next value, 2nd and 3rd combinations are reset
{2}{0,1,2}{9}
...
{9}{7,8,9}{9} --> last source sequence
```

```
{0,1,2,3,4}
{0,1,2,3,5}
...
{0,1,2,3,8}
{0,1,2,3,9}
{1,2,3,4,5}
...
{1,2,3,4,9}
...
...
{4,5,6,7,8}
{4,5,6,7,9}
{5,6,7,8,9}
```

```
{0}{0}{0}{0}{0}
{0}{0}{0}{0}{1}
...
{0}{0}{0}{0}{9}
{0}{0}{0}{1}{0}
{0}{0}{0}{1}{1}
...
{0}{0}{0}{1}{9}
{0}{0}{0}{2}{0}
...
{0}{0}{0}{2}{9}
{0}{0}{0}{3}{0}
...
{0}{0}{0}{3}{9}
{0}{0}{0}{4}{0}
...
{9}{9}{9}{9}{9}
```

Fig. 3. Excerpt of enumerated source sequences in (a) 10^{th} branch: $\binom{10}{1} \times \binom{10}{3} \times \binom{10}{1}$, (b) 1^{st} branch: $\binom{10}{5}$, (c) *last* branch: $\binom{10}{1} \times \binom{10}{1} \times \binom{10}{1} \times \binom{10}{1} \times \binom{10}{1}$, for $\mathcal{WE}_5$ and $\mathcal{I} = \{0, 1, 2, 3, ..., 9\}$.

10,000 *source sequences* will be generated from each of the 16 branches with the following quotas, $[6, 54, 137, 307, 137, 517, 516, 1148, 53, 307, 516, 1148, 306, 1148, 1147, 2553]$.

PSG allows generation of other specific compositions of *source sequences*, namely,

- *Source sequences with a single itemset*, which are typical data sets for frequent itemsets mining algorithms (a.k.a. market basket analysis) (see 2nd box in Fig. 3),
- *Source sequences composed of singletons*, which are typical event type sequences (see 3rd box in Fig. 3),
- *Source sequences of different lengths* through the use of different *Whitney enumerators*. Each *Whitney enumerator* has its own source of items i.e. $\mathcal{I}$, so that source sequences generated using smaller Whitney enumerators are not subsequences of source sequences generated using bigger Whitney enumerators.

Emit Sequences. We vary sequences' contents as follows: initially each source sequence is composed of a number of itemsets in the range 1 to k itemsets and of exactly k frequent items. All frequent items are in $\mathcal{I}$. In order to mimic real datasets, we add more itemsets and we append to each sequence random items, which do not belong to $\mathcal{I}$. *Padded items* are distributed among all itemsets of the sequence. Each sequence s is finally emitted a number of times which depicts the *count*(s). All of the input parameters, *number of* *padded items*, *number of itemsets* and *sequence support* follow a Poisson distribution. For instance, $\langle\{0\}\{0,1,2\}\{0\}\rangle$ is a *source sequence* for both following sequences $\langle\{\mathbf{0},70,80\}\{180,200\}\{\mathbf{0,1,2},53,65,103\}\{\mathbf{0},1000\}\rangle$ and $\langle\{1003\}\{78,309\}\{\mathbf{0}\}\{407,509\}\{\mathbf{0,1,2},5000\}\{507,809\}\{\mathbf{0},3000\}\{67,89\}\rangle$.

Enumerate in Parallel. For parallel generation of distinct source sequences, *Whitney numbers* are communicated to a pool of *M Sequence Generators*. Each *Sequence Generator* has a logical identifier in the range: $0 \ldots M-1$. *Sequence Generators* generate simultaneously generate distinct source sequences using the same Whitney numbers. For so, for each new branch of a Whitney Enumerator, each *Sequence Generator* identified by sg_j skips j *source sequences*. Then, each time it processes a *source sequence*, it skips M sources sequences, simulating a round robin distribution scheme [18]. Notice that this way, sequences having the same source sequence are clustered. For declustering purpose, all *Sequence Generators* may emit the same *source sequence* with different padding patterns.

5 Implementation and Performance Measurements

We implemented the *Parallel Sequence Generator* (*PSG*) using *MapReduce framework* [28] of Apache Hadoop 2.4 YARN. The generation load is evenly distributed among all *Sequence Generators*. Each *Sequence Generator* (Mapper in MapReduce framework terminology) is responsible for the creation of sequences using x *source sequences*, such that x is equal to the *number of source sequences for injection* divided by the *number of Sequence Generators*. For so, it creates a single file and writes into generated sequences. Finally, the *Sequence Generator* emits the volume of data sequences as well as the number of generated sequences. A Reducer aggregates summaries of generation results, it calculates the total volume and the total number of sequences written into *Hadoop Distributed File System* (HDFS).

A performance study was conducted in a shared-nothing cluster of nodes to demonstrate the scalability of the proposed *Parallel Sequence Generator*. The hardware system configuration used for performance measurements are Suno nodes located at Sophia site of french HPC platform GRID5000 [16]. Each Suno node has 32 GB of memory, its CPUs are Intel Xeon E5520, 2.27 GHz, with 2 CPUs per node and 4 cores per CPU. All nodes are connected by a 10 Gbps Ethernet.

The primary goal of carried-out experiments is to assess the scalability and the scale out of *PSG*. We are interested in two metrics, namely (1) the *Throughput in terms of Mega Bytes per second* (MBps), and (2) *the Throughput in terms of sequences per second* (#Seqs/sec). We report these metrics for different experiment settings, namely,

- *Hadoop cluster size*: the hadoop cluster is composed of one *master* and 2, 5 or 10 *slave nodes*. The Hadoop block size is set to 256 MB and the replication factor is set to 1 in order to reduce data redundancy overhead, and determine the maximum allowed throughput rates.

- *Number of sequence generators*: each *slave node* sets up a number of *sequence generators*, which also corresponds to the number of output data files. This parameter denotes the degree of parallelism in sequence generation and writing to HDDs. *Sequence generators* run in parallel in order to increase write throughput performances.
- *Number of source sequences injected in the database*: the size of the sequence database grows linearly with the number of injected source sequences (see Fig. 12). For experiments, a sequence is 420 bytes. This size relates to 5-sequences type (i.e., $\mathcal{WE}_5$), with an average of 25 items padded to each source sequence distributed over an average of 15 itemsets. Each source sequence repeats in average 5 % of the number of source sequences injected.

Experiments compare *PSG* to *TestDFSIO*. The latter is a distributed I/O benchmark tool, part of the Hadoop distribution. Each mapper in *TestDFSIO-write workload* creates a file and a 1 MB buffer and repeatedly writes the buffer into the output file until the file size reaches a user-specified value. For instance, a workload example of *TestDFSIO* could be create 10 files, such that each file is 10 GB. *TestDFSIO* reports average throughput per node, to be multiplied by the cluster size in order to obtain the aggregated write throughput. We compare throughput performances of *PSG* to *TestDFSIO*, in order to highlight the sequence generation overhead.

Figure 4 presents performance measurements of *PSG* compared to *TestDFSIO* for a 3 nodes' cluster. The cluster is composed of one master and 2 slave nodes. It sets up 10 *Sequence Generators*, which create sequences independently from each other. *PSG* creates a sequence database of over 450 GB with more than 2 billions of sequences, it succeeds to write 1.2 millions of sequences per second at a throughput of 287 MBps. The throughput is measured for various *number of injected source sequences* in the range 1,000 .. 200,000. A maximum throughput of 315 MBps is recorded, which results from the injection of 90,000

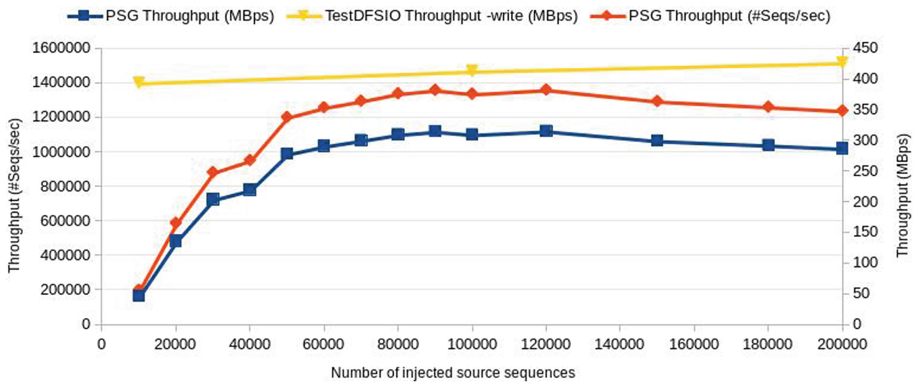


Fig. 4. *PSG* throughput performance results for a 3 nodes' cluster for 10 *sequence generators*, compared to *TestDFSIO* benchmark with 10 mappers.

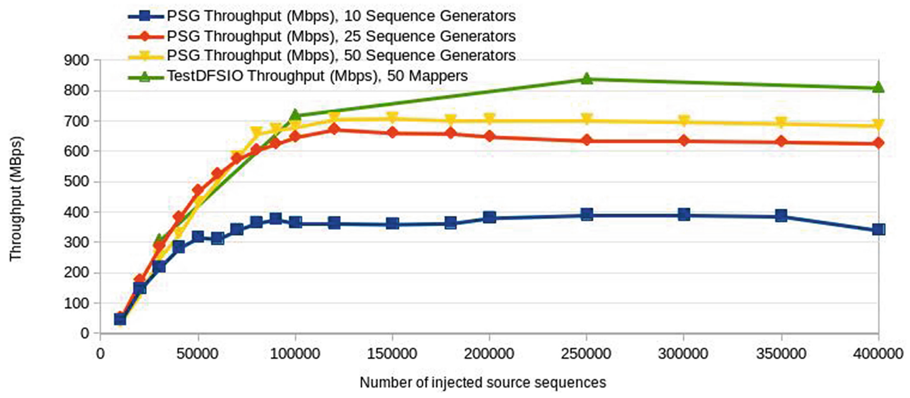


Fig. 5. *PSG* throughput performance results (MBps) for a 6 nodes' cluster and 10, 25, 50 *Sequence Generators*, compared to *TestDFSIO -write workload* benchmark with 50 Mappers.

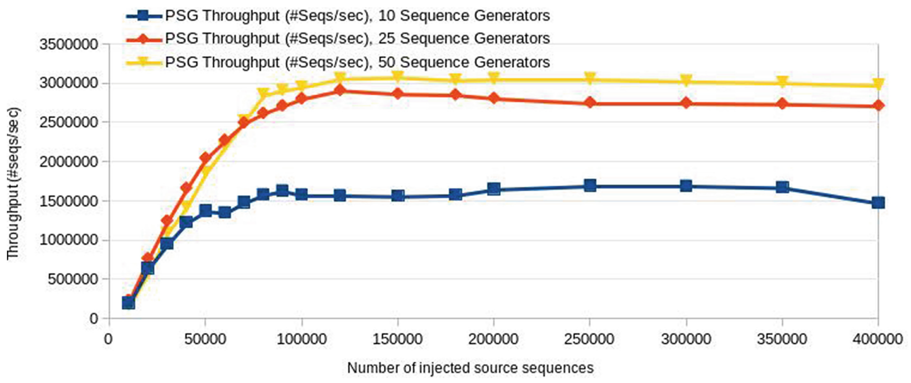


Fig. 6. *PSG* throughput performance results in terms of sequences per second for a 6 nodes' cluster and various number of *Sequence Generators*.

source sequences. This corresponds to a 91 GB Sequence Database, composed of more 400 millions of sequences.

Figures 5 and 6 present throughput performance measurements of *PSG* respectively in terms of MBps and #Seqs/sec for a 6 nodes' cluster. The cluster is composed of one master and 5 slave nodes. It sets up various number of *Sequence Generators*, which create sequences in parallel independently from each other. *PSG* creates a sequence database of over 1.8TB with more than 8 billions of sequences, it succeeds to write 3 millions of sequences per second at a throughput of 694 MBps. The throughput is measured for various number of *source sequences* in the range 10,000 .. 400,000. For each experiment, whether for 10, 25 or 50 *Sequence Generators*, the throughput increases for a number of *source sequences* less than 100,000, then it is invariant, and finally slightly

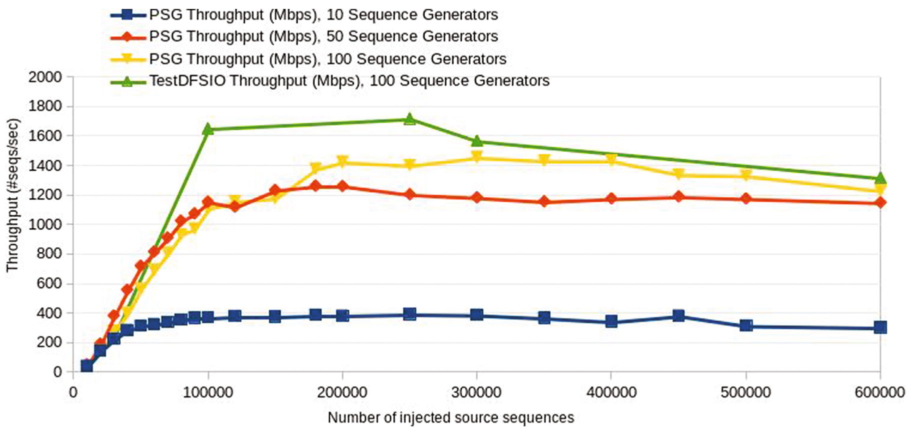


Fig. 7. PSG throughput performance results in terms of MBps for 11 nodes' cluster and 10, 25, 50 *Sequence Generators*, compared to *TestDFSIO -write workload* benchmark with 100 Mappers.

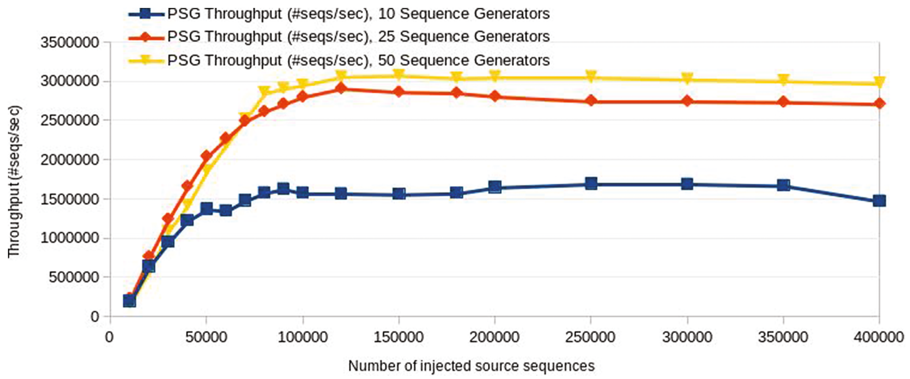


Fig. 8. *PSG* throughput performance results in terms of sequences per second for a 11 nodes' cluster and various number of *Sequence Generators.*

decreases due to the saturation of HDDs of slave nodes. It reaches a maximum value of 741.61 MBps for 50 *Sequence generators* and 180,000 source sequences. This corresponds to a 365 GB Sequence Database composed of more than one billion and half of sequences.

Figures 7 and 8 present respectively throughput performance measurements of *PSG* respectively in terms of MBps and #Seqs/sec for an 11 nodes' cluster. The cluster is composed of one master and 10 slave nodes. It sets up various numbers of *Sequence Generators*, which create sequences in parallel independently from each other. *PSG* creates a sequence database of over 4TB with more than 18 billions of sequences, it succeeds to write 5.3 millions of sequences per second at a throughput of 1.2 GBps(1230 MBps). The throughput is measured for various number of *injected source sequences* in the range 10,000 .. 600,000. The throughput increases for a number of *source sequences* less than 100,000, then it is almost invariant, and finally slightly decreases due to the saturation of HDDs of slave nodes. It reaches a maximum value of 1.45 GBps (1481.51 MBps) for 100 *Sequence Generators* and 300,000 *injected source sequences.*

Notice that we could not create bigger databases for HDDs' space constraints. Indeed, for an 11 nodes' cluster (one master and 10 slave nodes), the exception message when creating a sequence database with 700,000 *source sequences* is *Error: org.apache.hadoop.ipc.RemoteException (java.io.IOException): File/sequences/sequences_97.seq could only be replicated to 0 nodes instead of minReplication (=1). There are 10 datanode(s) running and no node(s) are excluded in this operation.*

In conclusion, the sequence generation is proved efficient, especially for big Sequence databases. Comparisons with *TestDFSIO* shows that for big sequence databases, HDFS IO operations which consist in appends to data files are much more expensive than enumeration costs of *source sequences.* Figures 9 and 10

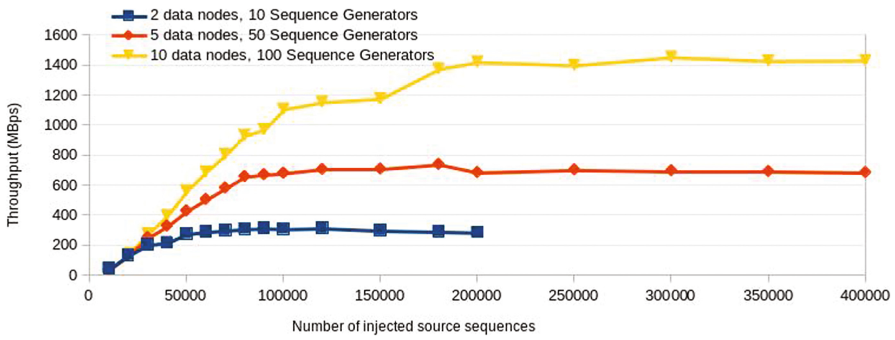


Fig. 9. Comparison of *PSG* Throughput (MBps) performance evaluation for various number of hadoop data nodes.

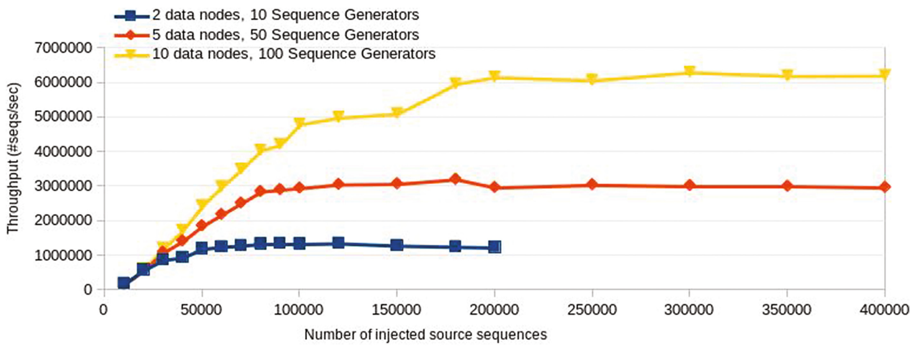


Fig. 10. Comparison of *PSG* Throughput (#Seqs/sec) performance evaluation for various number of hadoop data nodes.

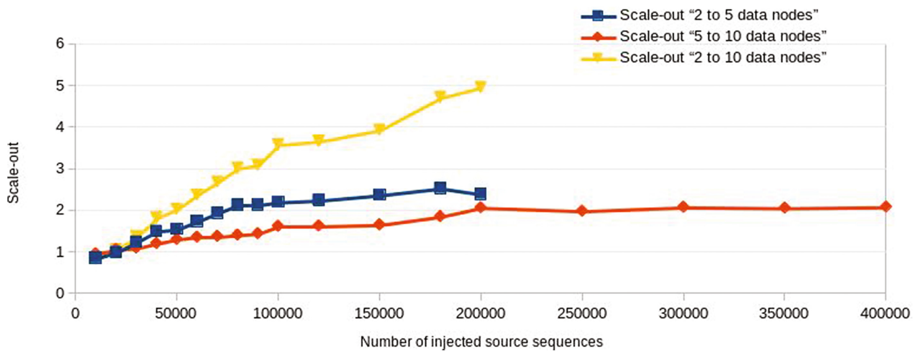


Fig. 11. *PSG* Scale-out Tests.

# source sequences	*# sequences* ($\times 10^6$)	*Sequence DB* (GB)
10,000	5	1.12
20,000	20	4.50
30,000	45	10.10
40,000	80	17.97
50,000	125	28.18
60,000	180	40.56
70,000	245	55.21
80,000	320	72.13
90,000	405	91.34
100,000	500	112.71
120,000	720	162.62
150,000	1,125	253.54
180,000	1,620	364.76
200,000	2,000	450.69
250,000	3,125	704.10
300,000	5,000	1014.20
350,000	6,125	1379.91
400,000	8,000	1802.52
450,000	10,125	2280.85
500,000	12,500	2816.97
600,000	180,000	4056.40

Fig. 12. Average *number of sequences* (millions) and *volume* (Giga Bytes) of generated *Sequence DBs*.

illustrate best performance measurements obtained for each cluster size. Figure 11 calculates the *scale-out* factor for the three cluster size settings, for a *number of injected source sequences* limited by the generation capacity of each cluster. Comparisons to a 3 nodes' cluster holds up to 200,000 *injected source sequences*, and comparisons to a 6 nodes' cluster holds up to 400,000 *injected source sequences*. Pairwise comparisons of the three cluster sizes shows that the *scale out* is almost ideal for big sequence databases. Indeed, *n times* the number of data nodes results in *n times* the write throughput.

6 Conclusions and Future Work

Starting from unavailability of synthetic big sequence databases for mining sequential patterns. *First*, this paper proposes a scalable and formal approach for *Parallel Generation of Big Synthetic Sequence Databases* satisfying both user-specified *sequences' characteristics* and *velocity* requirements. Experiments prove that the underlying *Parallel Sequence Generator* (*i*) creates billions of different sequences in parallel, (*ii*) ensures that injected source sequences satisfy the user requirements especially sequential pattern length characteristic. *Second*, the paper reports a scalability and scale-out performance study of the *Parallel Sequence Generator*, for various sequence databases' sizes and various number of *Sequence Generators* in a shared-nothing cluster of nodes.

Future work is mainly oriented towards three different directions. *First*, we aim to conduct thorough performance study of *GSP** and *PrefixSpan**: our proposed parallel implementations of *GSP* [3] and *PrefixSpan* [6] algorithms, using big sequence databases generated using *PSG*. *Second*, we aim to propose sophisticated algorithms with lessons learned from the performance studies of *GSP** and *PrefixSpan**. *Third*, we aim to customize *Parallel Sequence Generator* in order to generate datasets close to real data sets particularly for event sequences of computer logs, where large clusters emit millions of log entries per second.

Acknowledgements. We acknowledge with thanks a VLDB travel fellowship.

References

1. Han, P.J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques. The Morgan Kaufmann Series in Data Management Systems, 3rd edn. Morgan Kaufmann, Burlington (2011)
2. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proceedings of the 11th International Conference on Data Engineering (ICDE), pp. 3–14 (1995)
3. Srikant, R., Agrawal, R.: Mining sequential patterns: generalizations and performance improvements. In: 5th International Conference on Extending Database Technology Proceedings (EDBT), pp. 3–17 (1996)
4. Zaki, M.J.: Efficient enumeration of frequent sequences. In: Proceedings of ACM CIKM International Conference on Information and Knowledge Management, pp. 68–75 (1998)
5. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1–12 (2000)
6. Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Prefixspan: mining sequential patterns by prefix-projected growth. In: Proceedings of the 17th International Conference on Data Engineering, pp. 215–224 (2001)
7. Sun, P., Chawla, S., Arunasalam, B.: Mining for outliers in sequential databases. In: Proceedings of the 6th SIAM International Conference on Data Mining, pp. 94–105 (2006)
8. Hemalatha, C.S., Vaidehi, V., Lakshmi, R.: Minimal infrequent pattern based approach for mining outliers in data streams. Expert Syst. Appl. **42**, 1998–2012 (2015)

9. Cheng, H., Yan, X., Han, J.: Seqindex: indexing sequences by sequential pattern analysis. In: Proceedings of SIAM International Conference on Data Mining. SDM, pp. 601–605 (2005)
10. Lin, M.Y., Lee, S.Y.: Fast discovery of sequential patterns through memory indexing and database partitioning. J. Inf. Sci. Eng. **21**, 109–128 (2005)
11. Xin, D., Han, J., Yan, X., Cheng, H.: Mining compressed frequent-pattern sets. In: Proceedings of the 31st International Conference on Very Large DataBases, pp. 709–720 (2005)
12. Lam, H.T., Mörchen, F., Fradkin, D., Calders, T.: Mining compressing sequential patterns. Stat. Anal. Data Min. **7**, 34–52 (2014)
13. Li, H., Homer, N.: A survey of sequence alignment algorithms for next-generation sequencing. Briefings Bioinform. **11**, 473–483 (2010)
14. Rajaraman, A.: More data usually beats better algorithms (2008). http://anand.typepad.com/datawocky/2008/04/data-versus-alg.html
15. Srikant, R.: IBM quest synthetic data generator (1999). http://sourceforge.net/projects/ibmquestdatagen/files/
16. Grid5000: Large-scale and versatile testbed for experiment-driven research: distributed computing-HPC and big data (2015). https://www.grid5000.fr/
17. Kum, H.C., Chang, J.H., Wang, W.: Benchmarking the effectiveness of sequential pattern mining methods. Data Knowl. Eng. **60**, 30–50 (2007)
18. Moussa, R.: Mining big sequence databases (2015). https://sites.google.com/site/rimmoussa/miningbigseqdb
19. Pei, J., Mao, R., Hu, K., Zhu, H.: Towards data mining benchmarking: a testbedfor performance study of frequent pattern mining. In: Proceedings of ACM SIGMOD International Conference on Management of Data, p. 592 (2000)
20. Gray, J.: Sort benchmark home page (2008). http://research.microsoft.com/barc/SortBenchmark/
21. Tilmann, R., Meikel, P.: Parallel data generation for performance analysis of large, complex RDBMS. In: Proceedings of the 4th International Workshop on Testing Database Systems, pp. 5:1–5:6 (2011)
22. Poess, M., Rabl, T., Frank, M., Danisch, M.: A PDGF implementation for TPC-H. In: Nambiar, R., Poess, M. (eds.) TPCTC 2011. LNCS, vol. 7144, pp. 196–212. Springer, Heidelberg (2012)
23. Luo, C., Gao, W., Jia, Z., Han, R., Li, J., Lin, X., Wang, L., Zhu, Y., Zhan, J.: Handbook of BigDataBench: A Big Data Benchmark Suite (2015). http://prof.ict.ac.cn/BigDataBench
24. Jim, G., Prakash, S., Susanne, E., Ken, B., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 243–252 (1994)
25. Transaction Processing Council: TPC benchmarks (2015). http://www.tpc.org/
26. Karl, H.: The art of building a good benchmark. In: Proceedings of TPC-TC, pp. 18–30 (2009)
27. Raïssi, C., Pei, J.: Towards bounding sequential patterns. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1379–1387 (2011)
28. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: 6th Symposium on Operating System Design and Implementation (OSDI), pp. 137–150 (2004)