國立臺灣大學電機資訊學院電機工程學研究所

碩士論文

Graduate Institute of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

Eclat Grouping: 基於分群策略之頻繁項目集探勘演算法加速

Eclat Grouping: An Efficient Frequent Itemset Mining Algorithm Based on Grouping Strategy

溫明浩

Ming-Hao Wen

指導教授: 陳銘憲 博士

Advisor: Ming-Syan Chen, Ph.D.

中華民國 109 年 7 月

July, 2020

doi:10.6342/NTU202001665

國立臺灣大學碩士學位論文
口試委員會審定書

Eclat Grouping: 基於分群策略之頻繁項目集探勘演算法加速

Eclat Grouping: An Efficient Frequent Itemset Mining Algorithm Based on Grouping Strategy

本論文係溫明浩君（R07921060）在國立臺灣大學電機工程研究所完成之碩士學位論文，於民國 109 年 7 月 24 日承下列考試委員審查通過及口試及格，特此證明

口試委員： 陳銘憲

（指導教授）

楊得年　葉彌妍

帥宏翰　[illegible]

所　　長： 吳忠幟

doi:10.6342/NTU202001665

Acknowledgements

首先，感謝陳銘憲教授在這兩年間指導我如何就問題進行分析與研究，並且時常與我討論研究的進行與結果。每當論文的研究與實驗上遇到瓶頸時，教授都會耐心的引導我找出問題的癥結點，並且帶出可能的解決辦法，使得在研究的路途上順遂不少。也很感謝教授時常會對於不同領域的研究提出觀點與見解，讓我得以快速了解平時鮮少觸碰的領域。

另外，我想感謝口試委員們：陳銘憲教授、帥宏翰教授、楊得年教授、王釧茹教授以及葉彌妍教授的指導，願意撥空來聆聽我的碩士論文報告，並且以不同的角度與想法提供了許多的建議與改進方法，讓我得以再次琢磨我的研究，使之更加成熟。

再來，我想感謝同居的庭安、郁棋、志恩、俊賢、威斌、睿修以及奕君，大家都很好相處，並且互相支持與勉勵，玩耍時一起玩耍，趕工時一起在實驗室待到天亮。儘管大家都忙於各自的研究進度，也能夠排出時間來互相討論各自的研究並提出建議，真的是猶如一個溫馨的大家庭。在此也要特別感謝庭安，感謝她不辭辛勞的修改我的論文，不厭其煩地糾正錯誤。

當然，還有實驗室的學長姊、學弟妹以及助理，不管是生活上或是研究上的大小事，大家都隨叫隨到，讓我在這兩年過得非常充實愉快。尤其感謝均筑，陪著我們熬夜準備口試，以及給予建議。

doi:10.6342/NTU202001665

摘要

頻繁項目集探勘是一資料探勘的分支，是一基礎且重要的技術，其目的在於找出物品之間的關聯性。頻繁項目集探勘的發展歷史悠久，直至今日依然有不少關於此技術的論文發表，試圖加速或優化其探勘的過程。根據資料的緊湊程度，可以分成密集、中等及稀疏，每種都有著不同的性質，而大多數的演算法都只對於特定的性質進行算法上的加速。然而，隨著數位化的普及，各式資料皆以數位的形式儲存，使得資料量迅速成長，許多演算法的短處也在資料量的成長下逐漸顯露出來。為了使演算法更具通用性，本篇論文提出基於分群策略之頻繁項目集探勘演算法加速 (Eclat Grouping)，在二元陣列的表示法下，試圖利用分群的方式降低資料維度以加速頻繁項目集探勘的過程。Eclat Grouping 利用二元陣列稀疏的特性，將資料分群合併，藉此減少空值的情形與降低運算量，同時利用二階段驗證來確保結果的正確性。此外，我們也藉由中央處理器的特定指令對陣列做平行化處理，進一步加速運算流程。實驗結果顯示 Eclat Grouping 在稀疏資料集下優越於以往的演算法，同時在密集資料集下保持著近似的運算速度。

關鍵字：頻繁項目集挖掘、關聯規則、垂直挖掘、Eclat、AVX

doi:10.6342/NTU202001665

Abstract

Frequent itemset mining (F.I.M) is a branch of data mining. It is a fundamental and important technique, which aims to find out the relation between items. The concept of F.I.M was proposed many years ago and there are still have many works that try to optimize and accelerate the mining process. According to the density of the database, it can be divided into three types, including dense, mid-dense, and sparse database. Each type of database has its characteristics and most of the algorithms are only specific to one type of density. Due to the popularization of digitization, the size of databases grows rapidly. However, the growing data also enlarges the shortcomings in most algorithms. To make the algorithm more general in different density of databases, we proposed an efficient frequent itemset mining algorithm based on the grouping strategy called *Eclat Grouping*. We use the grouping method to reduce the data dimension based on binary array representation and accelerate the process of F.I.M. Moreover, we use two-stage acceleration to filter non-frequent itemsets in the first stage and ensure the correctness of the result in

doi:10.6342/NTU202001665

the second stage. In addition, we use certain instructions to parallelize and accelerate the computing process. The experiment shows that Eclat Grouping is faster than others in sparse databases with negligible change for the running time in dense databases.

Keywords: Frequent itemset mining, association rule, vertical mining, Eclat, AVX

doi:10.6342/NTU202001665

Contents

doi:10.6342/NTU202001665

doi:10.6342/NTU202001665

List of Figures

doi:10.6342/NTU202001665

List of Tables

doi:10.6342/NTU202001665

Chapter 1

Introduction

Data mining is an automatic process to discover some hidden knowledge or useful information from a large amount of data. The concept of data mining has been proposed in decades and has become popular in recent years. The reason that data mining becomes popular these years is due to the popularization of the electronic products, many information starts to record in digit format and becomes a huge data. Fayyad et al. [7] has classified six classes of tasks that data mining involved, which are Anomaly detection, Association rule learning, Clustering, Classification, Regression, and Summarization. Frequent itemset mining (F.I.M) is a branch of association rule mining, which aims to discover the relations among things. F.I.M is a fundamental and important technique in data exploration that can be applied in various domains. For instance, F.I.M can be used to find the commodities that are frequently bought together. In biology domain, it can help to find out the relations among DNA sequences. With the information service becomes popular, how to extract and arrange useful information becomes an important issue. Usually, the database for a data mining quest consists of lots of transactions, each transaction contains numbers of unique items. According to the number of items in every transaction, a database can

doi:10.6342/NTU202001665

be divided into three types of densities: dense, mid-dense, and sparse database. Density is one of the most important characteristics, which may influence the way to design an data mining algorithm. Lots of algorithms have been proposed and tried to accelerate the mining process. However, most of the algorithms can be only accelerated on a particular type of density database, and cannot handle with others. To deal with this problem, we proposed an efficient frequent itemset mining algorithm based on the grouping strategy called Eclat Grouping, which is scalable on different types of densities. Based on the Eclat algorithm [19], Eclat Grouping adopts the grouping strategy to combine transactions, which reduces the size of the array in the binary representation. Since the grouping method modifies transactions in the database, we proposed a two-stage acceleration to ensure the correctness of the results. Moreover, because of the parallelizable on the binary array, we applied AVX2 (Advanced Vector Extensions 2) instruction to accelerate the counting process in Eclat Grouping. The measurements show that Eclat Grouping is faster than other methods in sparse databases and has comparable performance as other dense-based algorithms in dense databases.

doi:10.6342/NTU202001665

Chapter 2

Related Work

There are three well-known algorithms in frequent itemset mining, which are Apriori [3], Eclat [19], and Frequent Pattern tree (FP-growth) [10].

Apriori is a bottom-up, breadth-first search algorithm. It obtains itemset candidates with length $(k+1)$ from frequent itemsets with length k, then scans the database to calculate their frequency. Apriori uses the downward closure property to prune the search space. However, it scans the database multiple times and enumerates all possible candidates, which leads to high memory consumption and scanning time. To address these issue, many Apriori based approach have been proposed [15] [5] [12] [17].

FP-growth is a trie based algorithm, which only scans the database twice. In the first scan, it obtains all frequent itemsets with length 1 and rearranges the order of items in the database. Next, it builds up a prefix tree with only singleton frequent itemsets, which means it compresses the database into a frequent pattern tree. Then, it recursively retrieves the tree to obtain frequent itemsets. FP-growth has been proved that it is more efficient than Apriori [10]. However, the memory space required and the retrieving mechanism

doi:10.6342/NTU202001665

make it difficult to be applied in a dense database. Based on FP-growth, many improvement work have been proposed [6] [16] [9] [8].

Eclat is a depth-first search algorithm. It scans the database once and transforms the data representation into a vertical format, which makes the searching process easy. Besides, it generates new itemsets from equivalent class which can reduce the candidates based on the previous result. The details of Eclat is introduced in Section 3.2.

doi:10.6342/NTU202001665

Chapter 3

Preliminaries

In this chapter, we define the problem that we are going to solve and introduce the Eclat algorithm and the improved algorithms based on Eclat.

3.1 Problem statement

For the formal definition of the problem, Agrawal et al. [2] state as follows: Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items. Let D denote as a database of transactions, where each transaction has a unique identifier (tid). Every transaction contains a set of items, such that $tid \subseteq I$. A set of items called an *itemset* is denoted as X. X with k items called a k-itemset. The **support** of X means the fraction of transactions in D that can be found X as a subset. An itemset is **frequent** if its support is greater than user-specified threshold, *minimum support* (*minSup*). The frequent itemset mining task is to generate all **frequent itemsets** in D. Given m items, there have 2^m candidates that might be frequent itemsets. Each candidate needs to be verified by searching all transaction which requires lots of computation. Thus, an efficient method that can fast traverse the searching space

doi:10.6342/NTU202001665

Horizontal data format	
Tid	**Itemset**
1	i_1
2	i_1, i_2
3	i_1, i_2, i_3

Vertical data foramt	
Itemset	**TIDset**
$\{i_1\}$	1 , 2, 3
$\{i_2\}$	2, 3
$\{i_3\}$	3

Table 3.1: The database in the left is in horizontal format, the other one in the right is in vertical format.

and find out frequent itemsets is needed.

3.2 Eclat algorithm

In general, the database is organized in horizontal data format, each row is a transaction that contains different numbers of items. For the Eclat algorithm, it transforms the data into the vertical format. Each row represents a set of tid, which also called **TIDset**. Table 3.1 shows the difference between horizontal format and vertical format.

First, Eclat scans the database row by row, if an item exists in the transaction, Eclat adds its tid into corresponding item's TIDset. After that, we get all items and their corresponding TIDset. Next, it checks their length of TIDsets which equals to their supports and discards items if its support is lower than the minimum support. Thus, we get all frequent 1-itemsets. To obtain other frequent k-itemsets efficiently, a concept called **equivalent class** (eClass) is introduced. It is defined as a set of k-itemsets that share (k-1) items. For example, $\{i_1, i_2, i_3\}$ and $\{i_1, i_2, i_4\}$ are 3-itemsets which share the prefix $\{i_1, i_2\}$. Thus, they belong to the $\{i_1, i_2\}$-eClass. For itemsets $\{i_1, i_2, i_3\}$ and $\{i_1, i_4, i_5\}$, since they only share one item i_1, they do not belong to the same equivalent class. By using this concept, we can generate a (k+1)-itemset by combining two k-itemsets which both are in the same equivalent class. Thus, it can reduces the numbers of itemsets that are not possible

doi:10.6342/NTU202001665

frequent. After generating a (k+1)-itemset, we obtain its TIDset by intersecting two TID-sets from the corresponding k-itemsets and thus obtain its support by getting the length of TIDset. An example is shown in Figure 3.1. After getting frequent 1-itemsets, we pick up itemset $\{A\}$ and its TIDset, generate new itemsets and corresponding TIDsets by combining and intersecting with other frequent 1-itemsets. Since the supports of $\{A, B\}$ and $\{A, E\}$ are not over the minimum support, we discard them. For $\{B\}$-eClass, since $\{B, C\}$ and $\{B, E\}$ are frequent itemsets, we can generate $\{B, C, E\}$ from them. Its support is larger than minimum support, hence it is also a frequent itemset. At last, there is only one frequent itemset in $\{C\}$-eClass, thus the mining stops. Overall, the all frequent itemsets are $\{A\}$, $\{B\}$, $\{C\}$, $\{E\}$, $\{A, C\}$, $\{B, C\}$, $\{B, E\}$, $\{B, C, E\}$ and $\{C, E\}$.

Since tid is used to identify the transaction, we set tid of each transaction equals to the row number in the database. Also, since we scan the database row by row, we can guarantee that the TIDset is sorted. The sorted TIDsets make the intersection can be done efficiently. The process of fasting intersection can be followed as below: Let pointer P_1 and P_2 point to the first tid in TIDsets $Tidset\_1$, $Tidset\_2$ respectively, since TIDset is an ordinal sequence, if the tid pointed by P_1 equals to the tid pointed by P_2, this tid will be added into the new TIDset, and then P_1 and P_2 move to the next tid. If the tid pointed by P_1 is larger than the tid pointed by P_2, P_2 moves to the next tid, and vice versa. Figure 3.2 shows an example of how it works. First, 1 is smaller than 3, P_1 points to the next tid. In step 2, the tid pointed by P_1 equals to the tid pointed by P_2, so the counter of support s becomes 1 and tid 3 is added into the new TIDset, and then P_1 and P_2 point to the next element. In step 3, 5 is larger than 4, thus P_2 move forward. Step 4 and 5 are in the same situation that both tids pointed by P_1 and P_2 are the same, hence these tids are added into the new TIDset and s increases. Finally, both P_1 and P_2 point to the end. The process then

doi:10.6342/NTU202001665

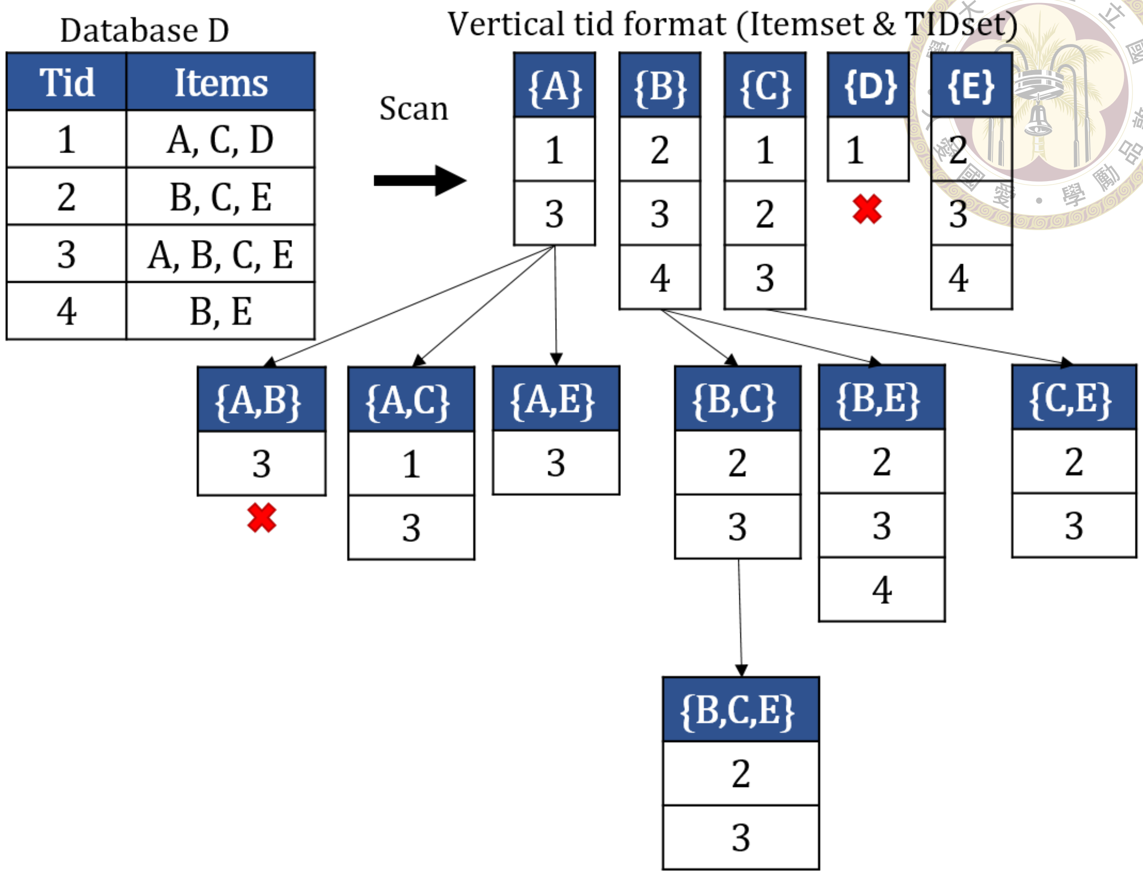


Figure 3.1: Example of mining process using Eclat algorithm

ends.

3.3 Existing improvements on Eclat algorithm

After the Eclat was proposed, many methods have come out to improve the computation efficiency. Here we introduce three significant and inspiring works below.

dEclat proposed by Zaki et al. [20] presents a new vertical data representation called *diffset*. TIDset belonging to the itemset only keeps track on the **tid** that do not have this itemset, which is totally opposite to Eclat. When the database is in high density, diffset can store fewer data compared with TIDset, which can lower down the memory usage and reduce the intersection complexity greatly. Also, the equivalent class not only share the

doi:10.6342/NTU202001665

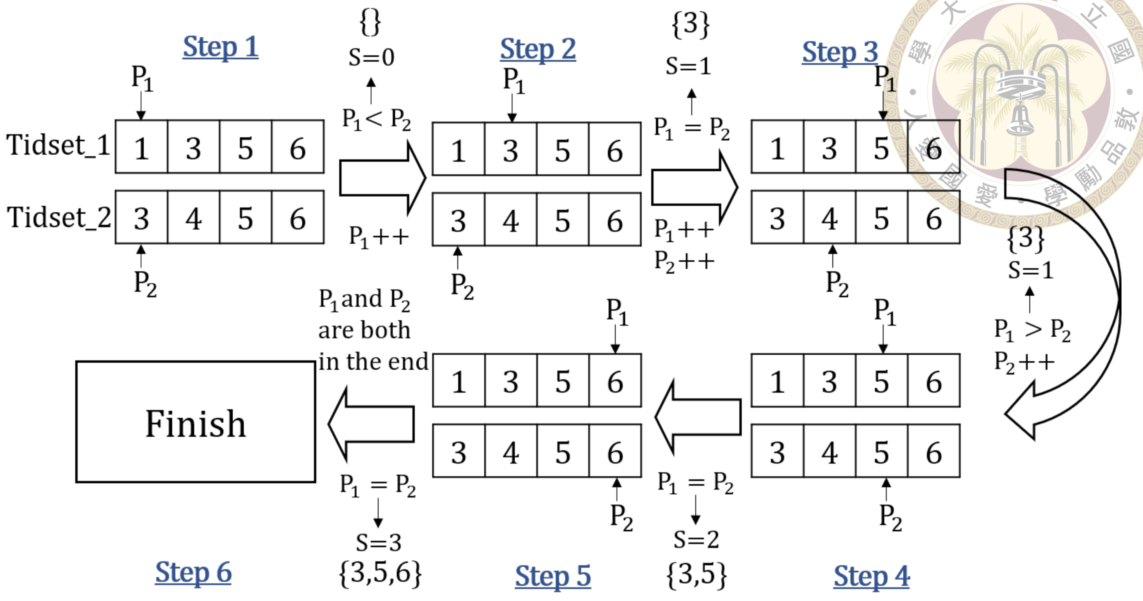


Figure 3.2: Example of intersection on Eclat

common itemset, but also share the common diffset, which makes the intersection more efficient.

hEclat proposed by Xiong et al. [18] uses boolean matrix, which size equals the number of transactions in the database, to replace TIDset. It reduces the complexity of intersection by simply using bitwise **AND** operation. It is efficient when using a dense database or with few transactions, but it requires higher memory space and processing time compared to Eclat when the database is large and sparse.

Eclat_growth proposed by Ma et al. [13] presents a new search strategy. First, it scans the database, stores the data in vertical format, and clips non-frequent 1-itemsets. Second, it builds a two-dimensional tree, each row is for different lengths of itemsets. Every time it picks one 1-itemset and combines with the itemset on the tree orderly to generate new frequent itemsets. If a new itemset is frequent, it will store into the corresponding location on the tree, otherwise it traverses all children and marks them to avoid unnecessary computation. To improve the generating and support counting efficiency, Eclat_growth

doi:10.6342/NTU202001665

proposed a method called BSRI (Boolean array setting and retrieval by indexes of transactions) which turns frequent 1-itemsets into a boolean array where the length equals to the number of transactions and sets value to 1 if it contains the corresponding tid. Therefore, the intersection only needs to check the corresponding value. Thus, the BSRI method decreases the computational complexity.

doi:10.6342/NTU202001665

Chapter 4

Algorithm

To further improve the computation efficiency of dense data, we proposed a new algorithm called Eclat Grouping. In this algorithm, we apply the grouping method on vertical bitset which compresses the length of the bitset. For a grouping factor g, it groups each g tids into one new grouping-tid. To generate frequent k-itemsets, it uses two-stage acceleration to early clip non-frequent itemsets to save computation time. Furthermore, we apply the AVX2-based Harley-Seal approach to accelerate the support counting in vertical bitset format.

The main process of Eclat Grouping algorithm is shown in Algorithm 1 and Algorithm 2. First, the process of scanning database is the same as Eclat, which scans row by row and transforms into a vertical tid format. After discarding non-frequent 1-itemsets, we generate bitsets and grouping bitsets for every frequent 1-itemset. Next, we pick every two frequent 1-itemsets to generate a new 2-itemset and then move it into a two-stage acceleration process. Inside this process, we generate grouping bitset and bitset for the itemset and check whether it is a frequent itemset. If the itemset is frequent, we add it into the corresponding equivalent class, otherwise we clip it. Eclat Grouping iterates through

doi:10.6342/NTU202001665

all equivalent classes recursively until there is no more new equivalent class generated.

Algorithm 1: Eclat Grouping

```
    Input: Database $D$, minimum support $Sup_{min}$
    Output: Frequent itemsets FIs
 1  FreqItemset_1 = ScanDatabase($D$);
 2  Transaction_grouping(FreqItemset_1);
 3  FIs = EquivalentClassMining(FreqItemset_1, $Sup_{min}$);
 4  return FIs;
```

Algorithm 2: EquivalentClassMining

```
    Input: Equivalent class $EClass$, minimum support $Sup_{min}$
    Output: Frequent itemset list $FIs$
 1  $FIs$ = Create empty list
 2  for i = 0; i < length(EClass); i++ do
 3      NewEClass = CreateNullEClass
 4      for j = i+1; j < length(EClass; j++) do
 5          TwoStageAcceleration($EClass[i]$, $EClass[j]$, NewEClass, $Sup_{min}$)
 6      end
 7      if NewEClass not Null then
 8          ChildFIs = EquivalentClassMining(NewEClass)
 9          $FIs$.AddFIs(ChildFIs)
10  end
11  $FIs$.AddFIs($EClass$)
12  return $FIs$
```

4.1 Vertical bitset representation

There are two types of vertical format representation, which is TIDset and bitset. TIDset only keeps track of tid that contains the corresponding itemset, so it is suitable for the sparse database. However, the process of comparison during intersection is unfriendly in the dense database. As the process shown in Figure 3.2, it needs to go through all elements. On the other hand, although bitset representation stores all of tids, the intersection

doi:10.6342/NTU202001665

Non-Grouping bitset

Itemset / Tid	{A}	{B}	{C}
1	1	0	1
2	0	1	1
3	0	1	1
4	0	1	0
5	1	0	1

$\xrightarrow{g=2}$

Grouping bitset

Itemset / Gtid	{A}	{B}	{C}
G_1	1	1	1
G_2	0	1	1
G_3	1	0	1

Table 4.1: Example of Transaction Grouping

of bitset does not check the bits one by one. In fact, it checks n bits in one **AND** operation, where n depends on the processor (e.g., n=64 in x64 processor). Therefore, bitset representation is efficient in the dense database while keeps the processing speed in the sparse database. Hence, we choose bitset to be our data representation format.

4.2 Transaction grouping

Since bitset is a boolean array, there are lots of zero value inside, especially in the sparse database. Although we have accelerated the process of intersection by partial paralleling, it still wastes lots of computation resources on those redundant values. To deal with the problem, we proposed a method called **transaction grouping**. Assume we have n transactions in D, for a grouping factor g, it takes the **union** of each g transactions into grouping-transaction, thus the number of transactions becomes to $\left\lceil \frac{n}{g} \right\rceil$. Table 4.1 shows the example of transaction grouping. For g = 2, it merges tid 1 and 2 into G_1, 3 and 4 into G_2, and G_3 equals to tid 5 since there's no more tid can be merged. Transaction Grouping enlarges the importance of itemset in every $gtid$ since the number of transactions decreases. The strategy of taking union rather than intersect ensures that no frequent itemset will be dropped since the occurrence of an itemset is potentially raised g times against non-grouping bitset.

doi:10.6342/NTU202001665

4.3 Two-stage Acceleration

The strategy of taking union when grouping is to ensure that no frequent itemset will be missing. However, it also causes some non-frequent itemsets to be misjudged. To solve this problem, we propose a method called **two-stage acceleration**, the algorithm is shown on Algorithm 3. It can be divided into two-stages: **filter stage** and **verification stage**. At the filter stage, each grouping bitset of the candidate can be generated by intersecting two grouping bitsets. If its support is lower than the minimum support, it will be clipped. In contrast, if the support is higher or equals to the minimum support. Since we can't guarantee the correctness, it moves to the verification stage. In the verification stage, the bitset of the candidate is generated by using the non-grouping bitsets. After checking its support, we can verify whether this itemset is frequent or not. two-stage acceleration can filter out non-frequent itemsets earlier with lower computation. However, it is obvious that when the error rate caused by transaction grouping is high, more candidates will move into the verification stage. Thus, choosing grouping factor g carefully to avoid error is needed.

Algorithm 3: TwoStageAcceleration

```
   Input: $Sup_{min}$, frequent itemset $FI_1$ and $FI_2$, Equivalent class $EClass$
   /* Filter stage */
1  Candidate = CreateCandidate()
2  Candidate.bitset_group = Intersect($FI_1$.bitset_group, $FI_2$.bitset_group)
3  support_group = SupportCounting(Candidate.bitset_group)
4  if support_group $> Sup_{min}$ then
       /* Verification stage */
5      Candidate.bitset = Intersect($Candidate_1$.bitset, $Candidate_2$.bitset)
6      support = SupportCounting(Candidate.bitset)
7      if support $> Sup_{min}$ then
8          $EClass$.Add(Candidate)
9  return
```

doi:10.6342/NTU202001665

4.4 Support counting acceleration

A naive way to compute the support of bitset is to check the value of each bit individually. For an n-bit bitset, it takes n cycles to complete. While this approach is simple, it takes a longer time than TIDset format either in the dense or sparse database. As the size of data grows rapidly in recent years, the concept of the bitset has been used widely in different domains to deal with memory storage and computation efficiency. Thus, most processors have dedicated instructions to count the number of ones, like **popcnt** on x64 processors and **vcnt** on ARM processors [1]. Since support counting is a work to accumulate all bits, it is suitable to use the concept of divide and conquer. Moreover, it can be done in parallel. Warren [11] had presented an approach called Harley-Seal function which aggregates bitwise parallel carry-save adder (CSA). Based on this algorithm, Mula et al. [14] adapted the Harley-Seal function into AVX2 instructions, which is capable of dealing with 256 bits in one instruction. By applying this method, we can speed up at least 25 times faster than the naive method.

[1] **popcnt** instruction is include in instruction set **SSE4.2**, which was first available in Intel Nehalem microarchitecture and AMD Barcelona microarchitecture. **vcnt** instruction is included in instruction set **NEON** released by ARM

doi:10.6342/NTU202001665

Chapter 5

Experiment and Analysis

In this chapter, we compare Eclat Grouping with Eclat, dEclat, Eclat_growth, hEclat[1] and measure them in different datasets. The runtime environment is described in Table 5.1.

The dataset we used can be found on Frequent Itemset Mining Dataset Repository [1]. It contains different types of real datasets and synthetic datasets generated by IBM Quest Market-Basket Synthetic Data Generator [4]. Since density is an important factor in frequent itemset mining, we divided these datasets into three groups: sparse, mid-dense, and dense. Table 5.2 shows the information about databases. To reduce the measurement error caused by the runtime environment, we measure every experiment five times and take the average as the result.

5.1 Performance Analysis

For the first experiment, we show the processing time in sparse databases. Figure 5.1, 5.2 and 5.3 show the performance on T10I4D100K, retail and kosarak, respec-

[1]The paper of hEclat didn't mention the detail of support counting, hence we implement it in the same way as in Eclat Grouping.

doi:10.6342/NTU202001665

Type	**model**
CPU	Intel i9-9820X
Memory	128GB
Operating System	Ubuntu 18.04.3 LTS
Programming language	C++14
Compiler	GCC-7.5.0

Table 5.1: Experimental environment

Dataset	Number of transactions	Numbers of item	Average length	Type
T10I4D100K	100000	870	10	Sparse
retail	88162	16470	10	Sparse
kosarak	990002	41270	8	Sparse
T40I10D100K	100000	942	40	Mid-dense
pumsb_star	49046	2088	50	Mid-dense
connect	67557	129	43	Dense
chess	3196	75	37	Dense
mushroom	8124	119	23	Dense
accidents	332899	464	34	Dense

Table 5.2: The Information of Datasets

tively. Since dEclat is opposite to Eclat, which is suitable in the dense database, we don't consider it in this part of the experiment. The results of these datasets show a similar trend. Eclat takes the longest processing among these algorithms and Eclat_growth takes longer processing time than hEclat and Eclat Grouping. However, the gap is getting close and even less than hEclat and Eclat Grouping when minimum support becomes small. When minimum support decreases, the number of items in tidset is getting small, which facilitates the comparison in Eclat_growth. On the other hand, Eclat Grouping spends the least processing time in most of the minimum support settings, this credit gives to the grouping method and two-stage validation by decreasing the computation cycles and filtering out candidates in the filter stage.

Figure 5.4 and 5.5 are the performance of mid-dense database T40I10D100K and pumsb_star. As the density increased, Eclat_growth and Eclat become strenuous due to the comparison mechanism. However, bitset representation is not affected by density.

doi:10.6342/NTU202001665

Thus, hEclat and Eclat Grouping outperform other algorithms. On the other hand, we can observe that the relation between hEclat and Eclat Grouping is different from the relation in sparse databases, the curves become close and sometimes even crossover each other. The reason is that the growing density leads to a higher error rate and makes more candidates need to be checked in the verification stage. Thus, the grouping method in the mid-dense database becomes less efficient than in the sparse database but still works.

Finally, we investigate processing time on dense database chess, accidents, connect and mushroom, the result are on Figure 5.7, 5.9, 5.6 and 5.8. We do not contain Eclat since dEclat is more suitable in the dense database. Based on the result, although dEclat is designed for dense database, Eclat Grouping and hEclat still outperform it because of the bitset representation. It makes the process of generating new itemset can be done in parallel, thus faster than tidset in the dense situation. For hEclat and Eclat Ggrouping, the relation between both is opposite to the relation in the mid-dense database. The dense database makes lots of 0 to be combined with 1 in the grouping process and leads to misjudging in the filter stage and makes many non-frequent itemsets should be verified in the verification stage. However, there are still some candidates that are filtered out by the filter stage and able to save computation time to cover the extra processing time. Therefore, the result of Eclat Grouping and hEclat are still close.

5.2 Grouping factor analysis

In this section, we discuss the grouping factor of g. We investigate the relation between the value of the grouping factor and the processing time. Figure 5.10, 5.11 and 5.12 show the result in the database T10I4D100K, T40I10D100K, and mushroom, respectively.

doi:10.6342/NTU202001665

The blue curve in the chart means the processing time in different grouping factors, while the red curve represents how many candidates are filtered out in the filter stage.

First, we analyze g in the sparse database T10I4D100K, the result is in Figure 5.10. We see that the curve of pruned candidates is approximate to exponential decrease, which makes the processing time increase steeply. Larger g indicates more transactions are grouped, which increases the possibility of combining 1 and 0 into the same group. Low minimum support results in low fault tolerance. A little error can cause support over the minimum support and forces the candidate to move to the verification stage. Thus, the processing time in Figure 5.10a increases more rapidly than in Figure 5.10b.

The result in the mid-dense database T40I10D100K is similar to T10I4D100K, but with sharper curves. As we can see, when the number of pruned candidates drops to nearly zero, the processing time starts to decrease. Since the filtered stage cannot filter most of the candidates, the computation in this stage becomes redundant. Hence, increase the grouping factor can reduce the computation cycles and thus reduce the processing time.

The behavior in the dense database mushroom, which shows in Figure 5.12, is much different from the sparse and mid-dense database. The reason for this phenomenon is the same as the reason for decreasing processing time in the mid-dense database. Hence, for dense databases, a large grouping factor can accelerate the processing time.

doi:10.6342/NTU202001665

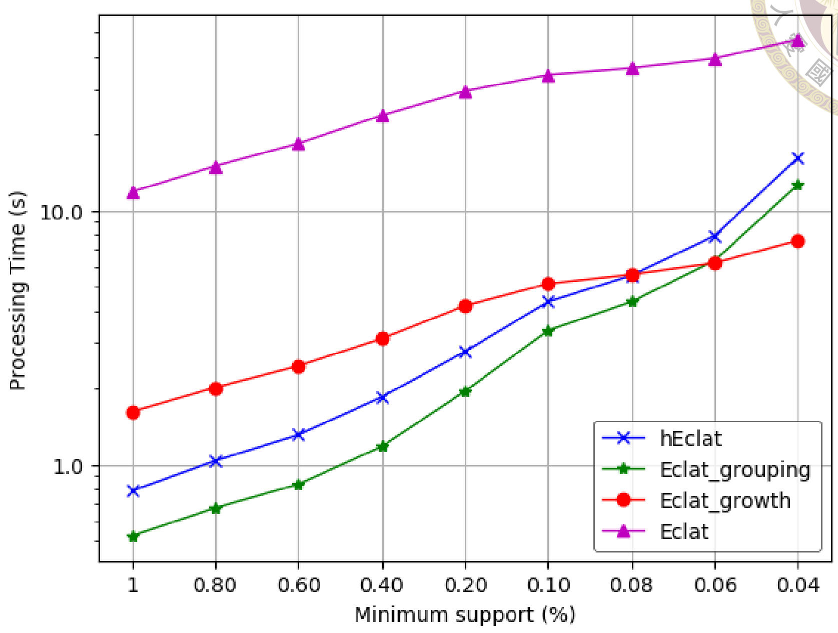


Figure 5.1: Processing time in **T10I4D100K** in log scale

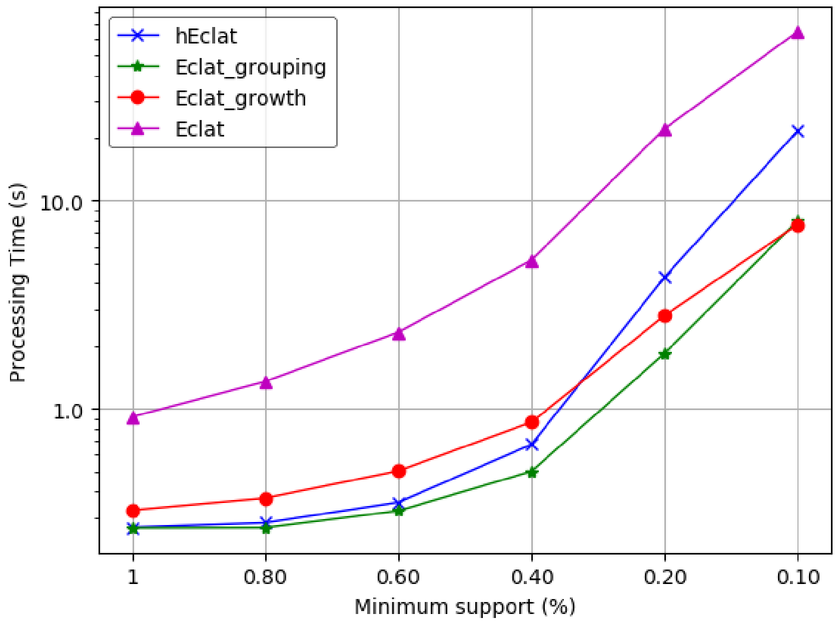


Figure 5.2: Processing time in **retail** in log scale

doi:10.6342/NTU202001665

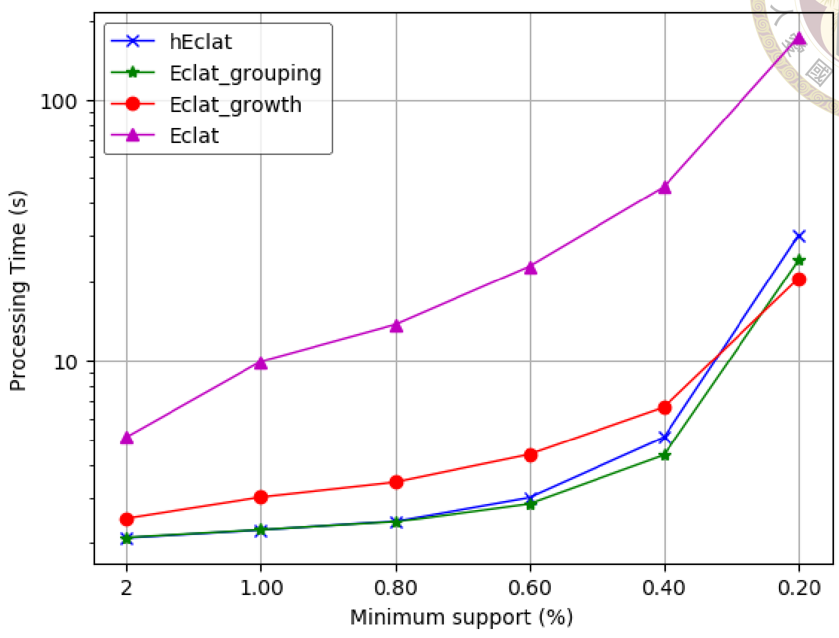


Figure 5.3: Processing time in **korasak** in log scale

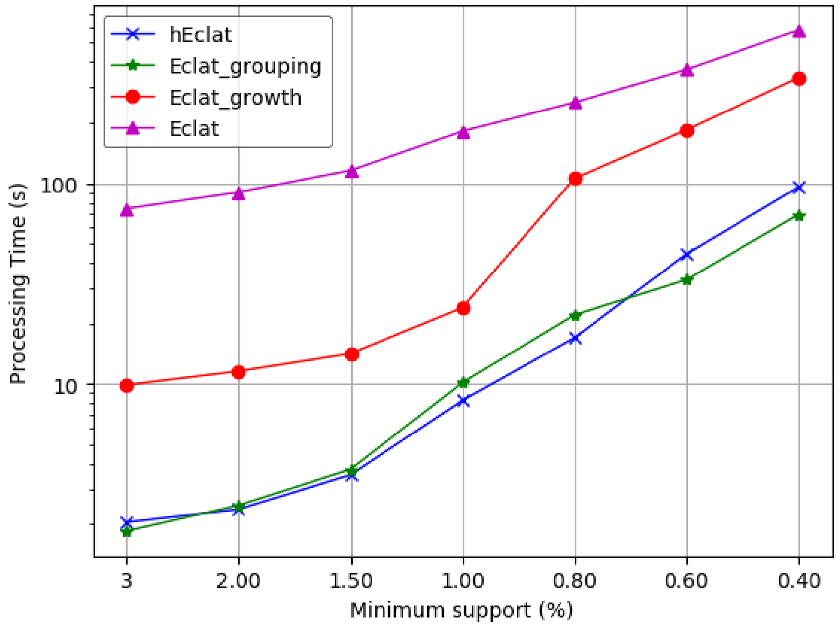


Figure 5.4: Processing time in **T40I10D100K** in log scale

doi:10.6342/NTU202001665

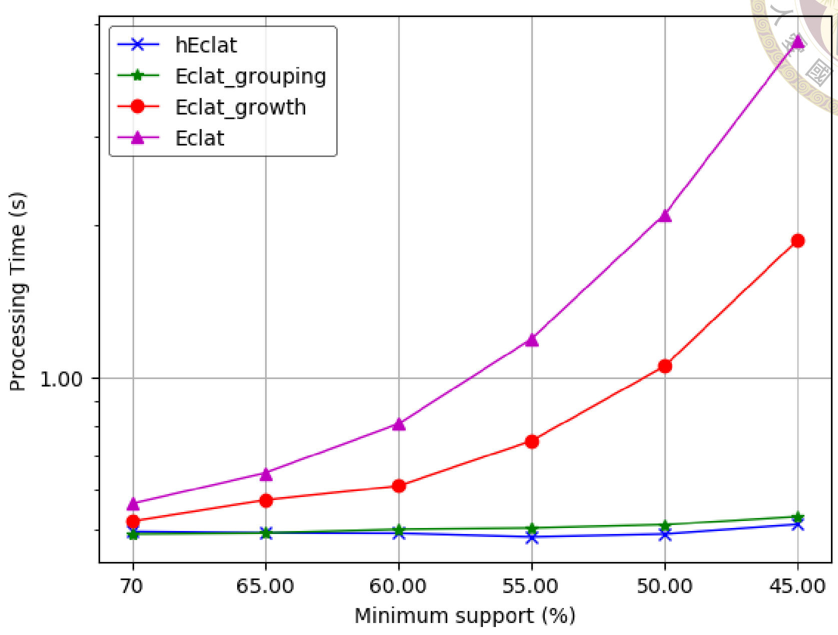


Figure 5.5: Processing time in **pumsb_star** in log scale

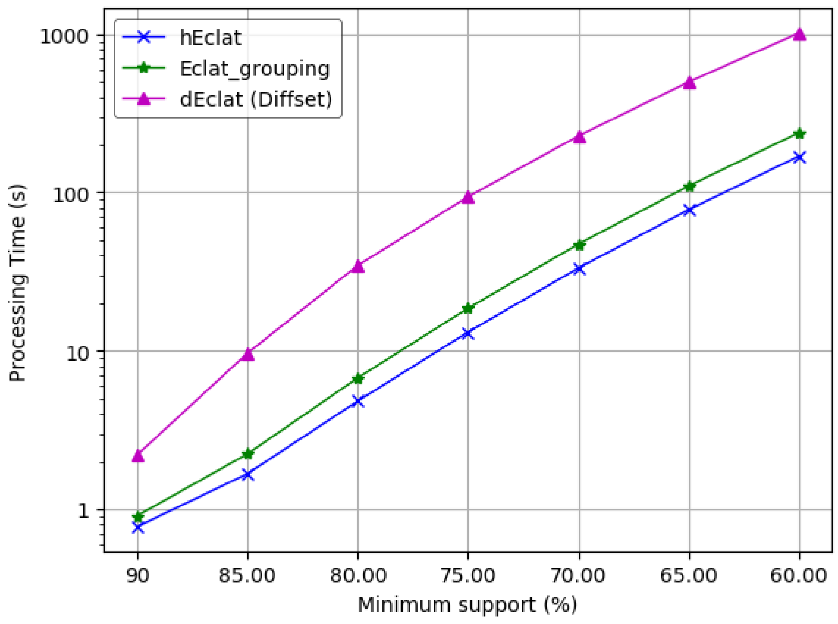


Figure 5.6: Processing time in **connect** in log scale

doi:10.6342/NTU202001665

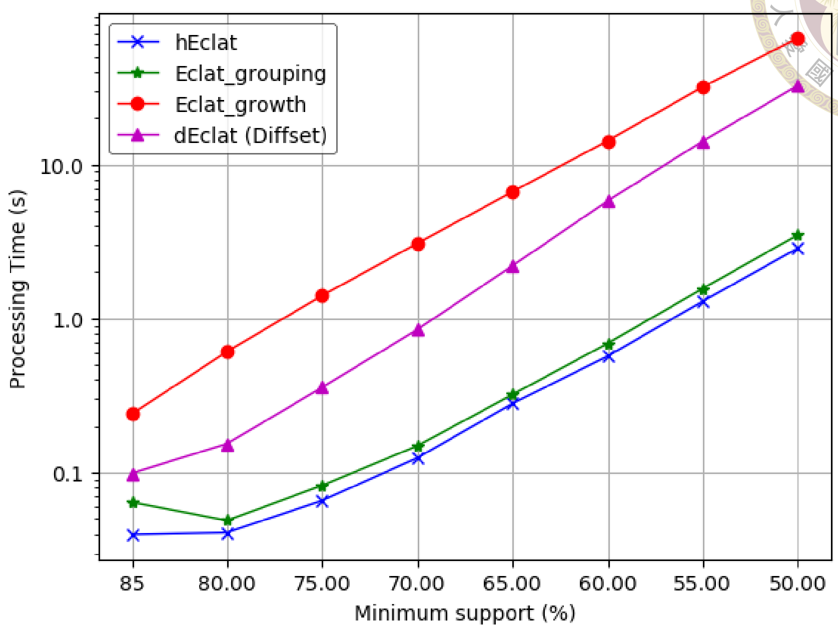


Figure 5.7: Processing time in **chess** in log scale

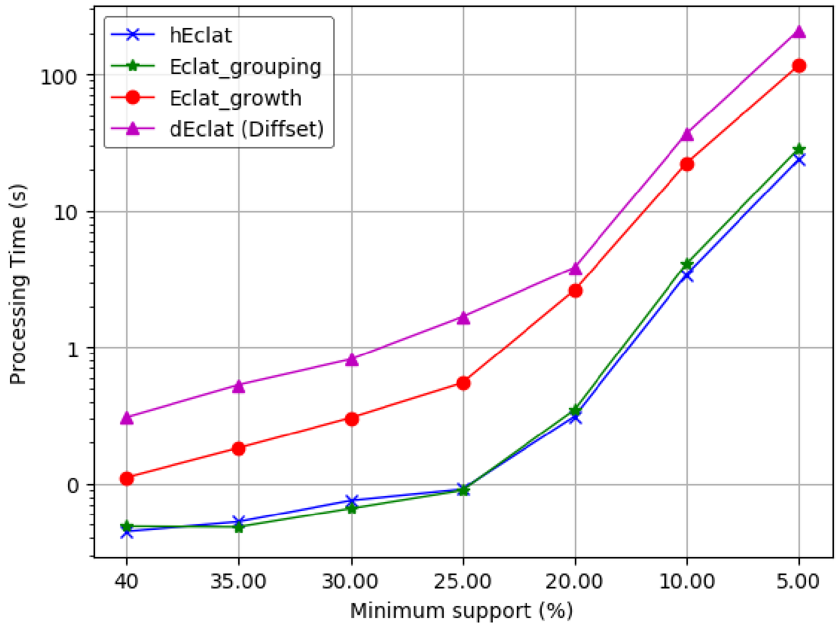


Figure 5.8: Processing time in **mushroom** in log scale

doi:10.6342/NTU202001665

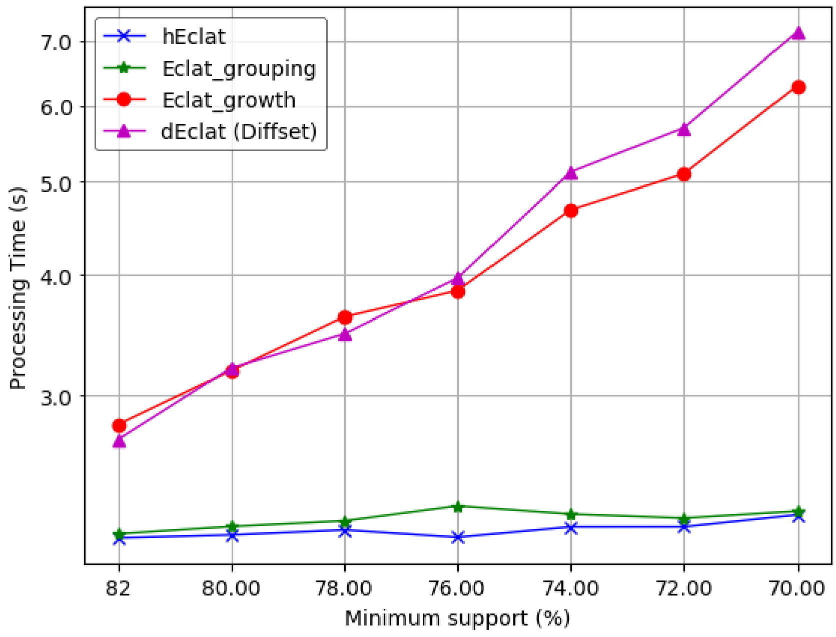
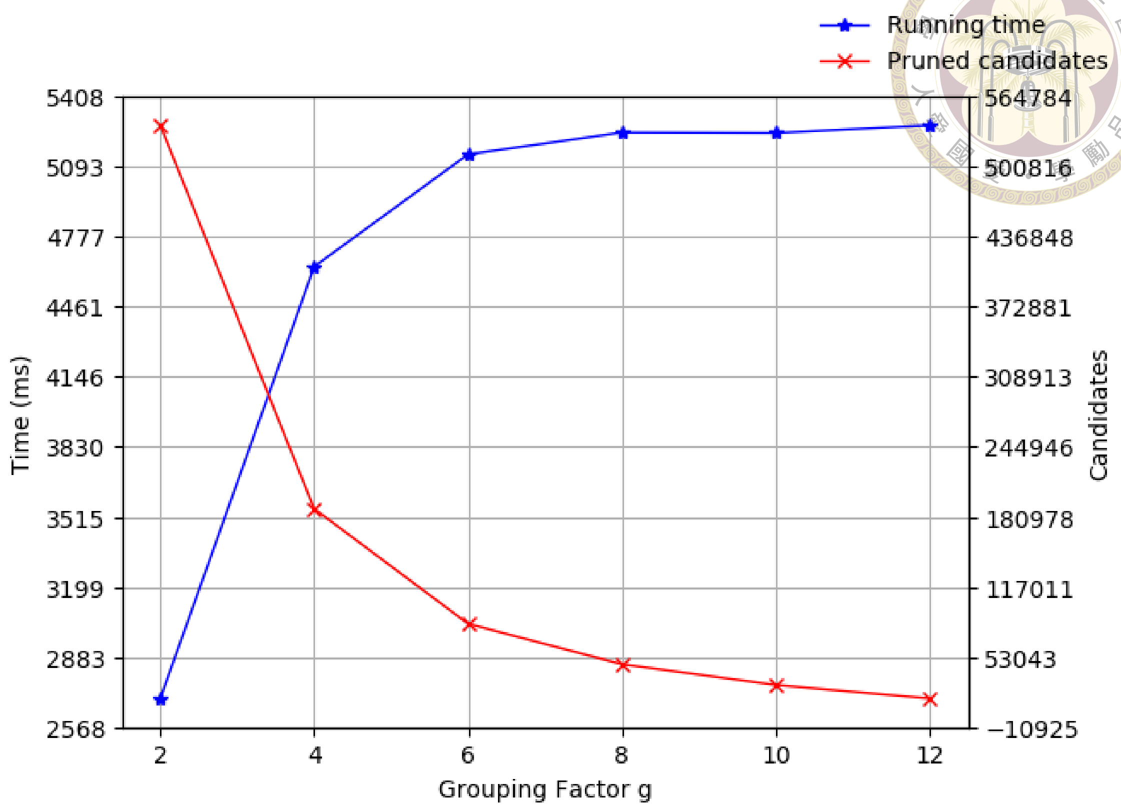
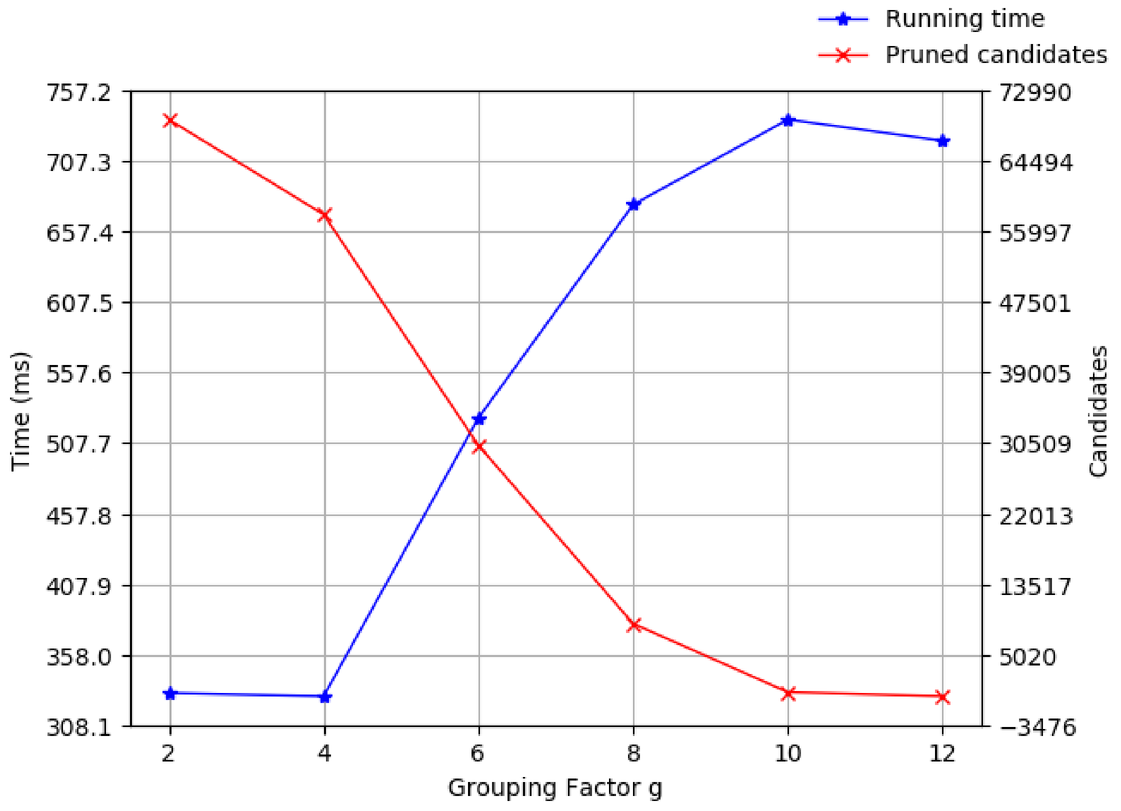


Figure 5.9: Processing time in **accidents** in log scale

doi:10.6342/NTU202001665



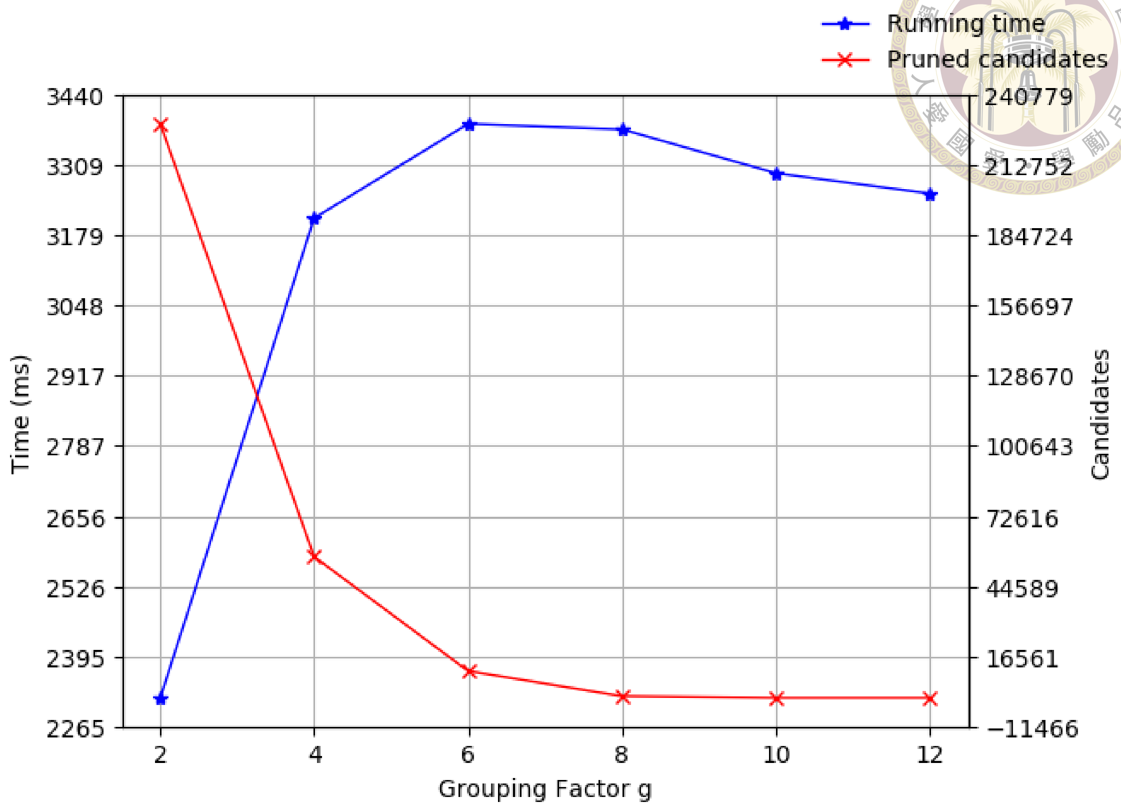
(a) Minimum support = 0.08%



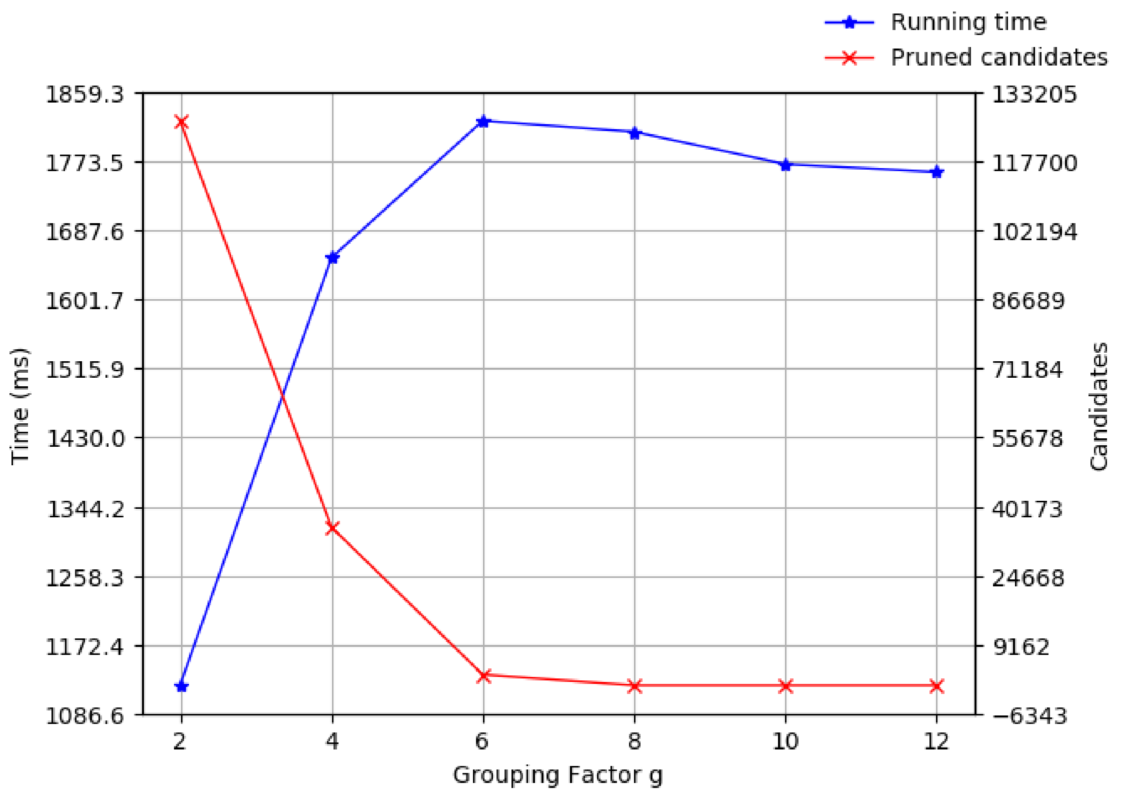
(b) Minimum support = 1.00%

Figure 5.10: Grouping factor analysis in **T10I4D100K**.

doi:10.6342/NTU202001665



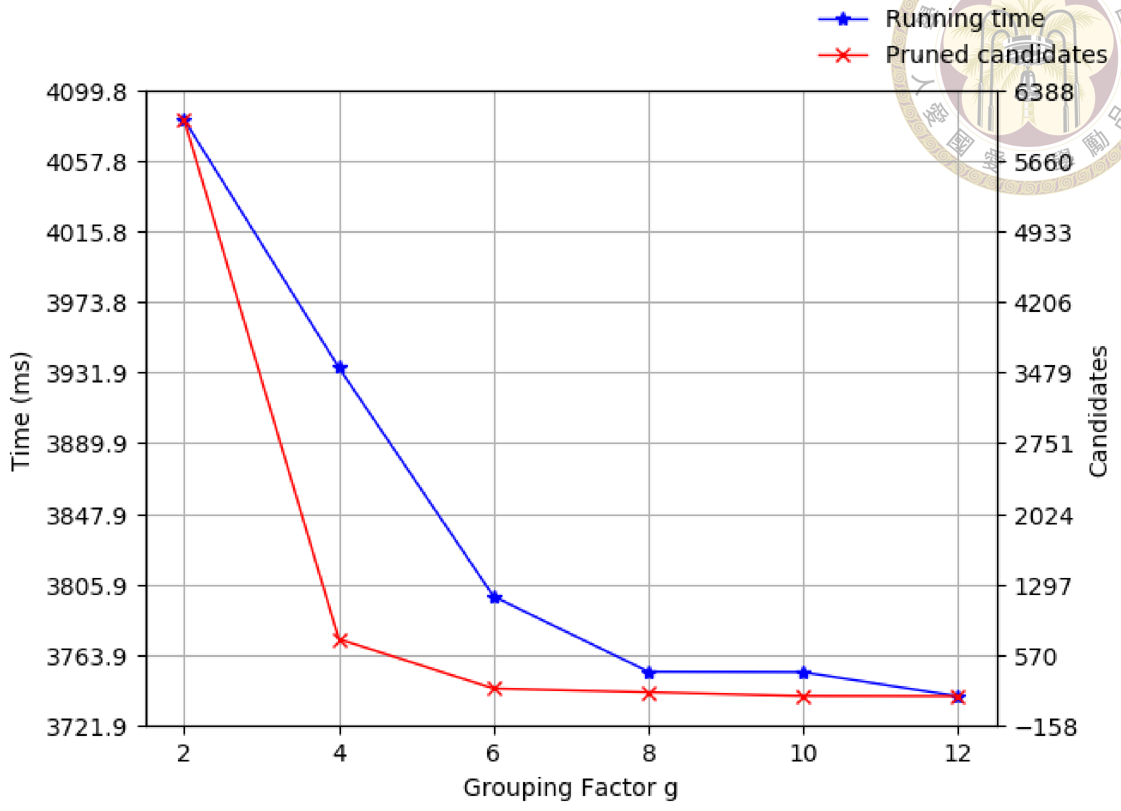
(a) Minimum support = 1.50%



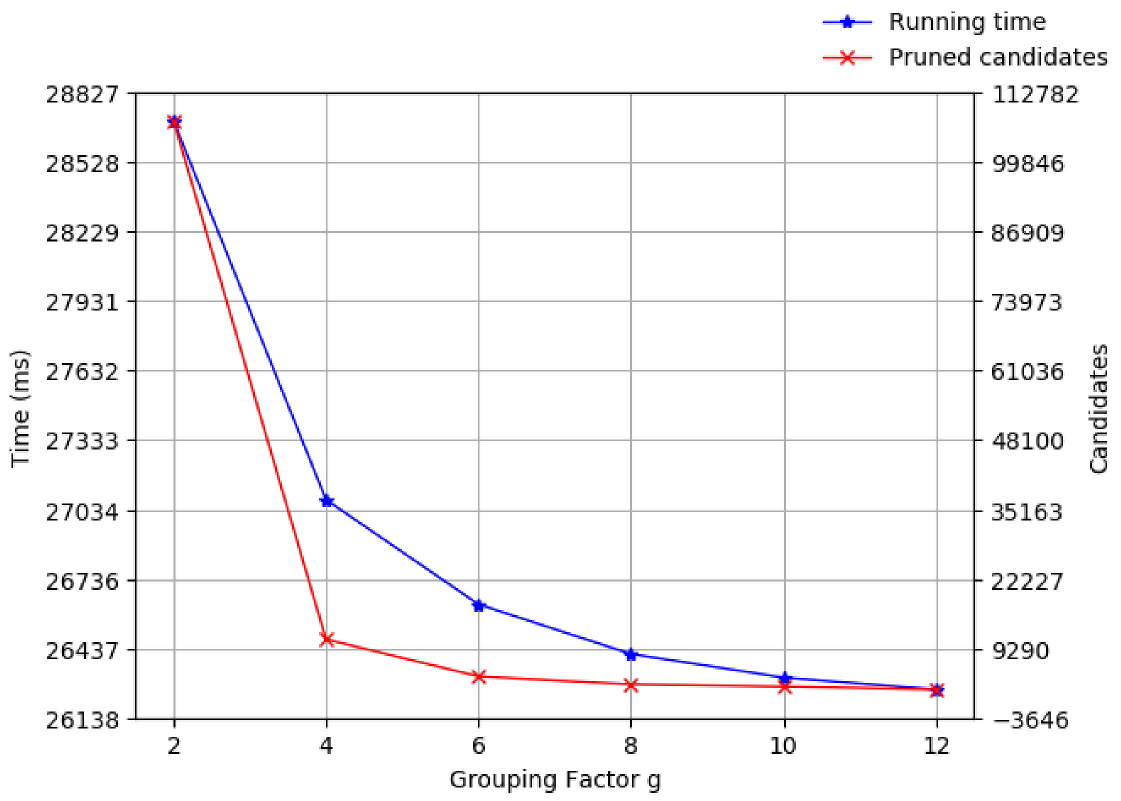
(b) Minimum support = 2.50%

Figure 5.11: Grouping factor analysis in **T40I10D100K**.

doi:10.6342/NTU202001665



(a) Minimum support = 10.00%



(b) Minimum support = 5.00%

Figure 5.12: Grouping factor analysis in **mushroom**.

doi:10.6342/NTU202001665

Chapter 6

Conclusion

Frequent itemset mining is one of the most important techniques in data exploration, many algorithms were proposed to accelerate the mining process. Eclat and other Eclat-based algorithms are efficient methods to deal with the problem, but most of them are not adaptable in both sparse and dense databases. Therefore, we proposed an algorithm based on the Eclat algorithm, call Eclat Grouping. It uses the grouping method to reduce the length of bitset while using two-stage acceleration to accelerate the process and ensures accuracy. We parallelize the support counting process using AVX2 instruction, which is generally supported in recent processors. The result shows that Eclat Grouping accelerates the processing time in sparse databases with negligible change for the running time in dense databases.

doi:10.6342/NTU202001665

References

[1] Frequent itemset mining dataset repository. http://fimi.uantwerpen.be/data/. Accessed: 2020-06-01.

[2] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, June 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB ’94, page 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[4] D. Bhalodiya. Ibm quest market-basket synthetic data generator, 05 2014.

[5] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. *ACM SIGMOD Record*, 26, 12 2001.

[6] C.-K. Chiou and J. Tseng. An incremental mining algorithm for association rules based on minimal perfect hashing and pruning. volume 7234, pages 106–113, 04 2012.

[7] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37, Mar. 1996.

doi:10.6342/NTU202001665

[8] P. Gera and S. Jyothi. Tree-based incremental association rule mining without candidate itemset generation. 12 2010.

[9] G. Grahne and J. Zhu. Fast algorithms for frequent itemset mining using fp-trees. *IEEE Trans. on Knowl. and Data Eng.*, 17(10):1347–1362, Oct. 2005.

[10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000.

[11] J. Henry S. Warren. The quest for an accelerated population count. In *Beautiful Code: Leading Programmers Explain How They Think*, chapter 10, pages 147–158. O’ Reilly Media, 2007.

[12] Jun-Lin Lin and M. H. Dunham. Mining association rules: anti-skew algorithms. In *Proceedings 14th International Conference on Data Engineering*, pages 486–493, 1998.

[13] Z. Ma, J. Yang, T. Zhang, and F. Liu. An improved eclat algorithm for mining association rules based on increased search strategy. *International Journal of Database Theory and Application*, 9:251–266, 05 2016.

[14] W. Muła, N. Kurz, and D. Lemire. Faster population counts using avx2 instructions. *Computer Journal*, 61, 11 2016.

[15] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. *SIGMOD Rec.*, 24(2):175–186, May 1995.

[16] B. Rácz. nonordfp: An fp-growth variation without rebuilding the fp-tree. In *FIMI*, 2004.

doi:10.6342/NTU202001665

[17] V. Vaithiyanathan, K. Rajeswari, R. Phalnikar, and S. Tonge. Improved apriori algorithm based on selection criterion. In *2012 IEEE International Conference on Computational Intelligence and Computing Research*, pages 1–4, 2012.

[18] Z. Xiong, P. Chen, and Y. Zhang. Improvement of eclat algorithm for association rules based on hash boolean matrix. *Application Research of Computers*, 27(4): 1323–1325, Apr 2010.

[19] M. J. Zaki. Scalable algorithms for association mining. *IEEE Trans. on Knowl. and Data Eng.*, 12(3):372–390, May 2000.

[20] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’ 03, page 326 – 335, New York, NY, USA, 2003. Association for Computing Machinery.

doi:10.6342/NTU202001665