Mining of Frequent Item with BSW Chunking

Pratik S. Chopade
ME Scholar
Department of Computer Engineering
G.S.M.C.O.E Balewadi, Pune.
e-mail: Pratik.chopade87@gmail.com

Prof. Priyanka More
Assistant Professor
Department of Computer Engineering
G.S.M.C.O.E Balewadi, Pune.

Abstract— Apriori is an algorithm for finding the frequent patterns in transactional databases is considered as one of the most important data mining problems. Apriori algorithm is a masterpiece algorithm of association rule mining. This algorithm somehow has constraint and thus, giving the opportunity to do this research. Increased availability of the Multicore processors is forcing us to re-design algorithms and applications so as to accomplishment the computational power from multiple cores finding frequent item sets is more expensive in terms of computing resources utilization and CPU power. Thus superiority of parallel apriori algorithms effect on parallelizing the process of frequent item set find. The parallel frequent item sets mining algorithms gives the direction to solve the issue of distributing the candidates among processors. Efficient algorithm to discover frequent patterns is important in data mining research Lots of algorithms for mining association rules and their mutations are proposed on basis of Apriori algorithm, but traditional algorithms are not efficient. The objective of the Apriori Algorithm is to find associations between different sets of data. It is occasionally referred to as "Market Basket Analysis". Every several set of data has a number of items and is called a transaction. The achievement of Apriori is sets of rules that tell us how often items are contained in sets of data. In order to find more valuable rules, our basic aim is to implement apriori algorithm using multithreading approach which can utilization our system hardware power to improved algorithm is reasonable and effective, can extract more value information.

Keywords- *Apriori algorithm, Association rules, data mining, parallel apriori algorithm, parallel implementation.*

*****

I. Introduction

The transactions carried out in retail sector have the huge amount of data. This data require data mining tool to extract hidden patterns which may organization to predict future trends and behaviors. Data Mining or Knowledge Discovery in Databases is an advanced approach which refers to the extraction of previously unknown and useful information from large databases. Aggregation of big data from different origin of the society but a little knowledge situation has led to knowledge find from databases which is called data mining. The data sources can consist of data warehouses, databases, and other information repositories that are rushed into the system dynamically [1].

Association Rule Mining is a significant technique of data mining. This technique has more attention on finding interesting relationships. For understanding these relationships, a technique called Market Basket Analysis has been popularized in Data Mining. This helps in understanding the business organizations. This paper shows that how addition of new parameters improves the efficiency of Apriori algorithm by comparing the results of improved algorithm with the results of serially implemented algorithm. The improved algorithm will utilize the multiple core of processor for finding the association among the item sets.

II. Literature Survey

The conventional methods spend lot of time to resolve the problems or decision making for profitable business. Data mining formulate databases for finding hidden patterns, finding anticipating information that experts may miss. Hence, this paper reviews the various trends of data mining and its relative applications from past to present and discusses how adequately can be used for targeting profitable customers in campaigns and utilize the multiple cores of the processor for faster execution [4].

A. Usage of Apriori Algorithm in retail sector

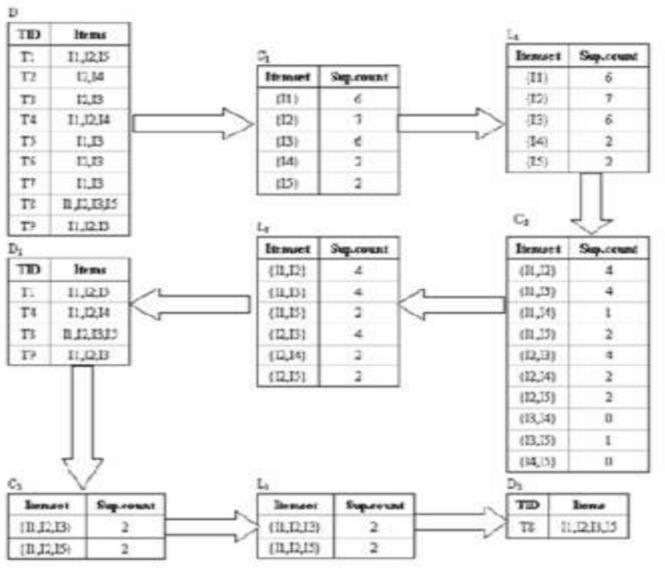
This paper proposes the role of Apriori Algorithm for Finding the Association Rules in Data Mining. Association rule mining is interested in finding frequent rules that describe association between unrelated frequent items in databases, and it has two main measurements: support and confidence values. The frequent item sets is defined as the item set that have support value greater than or equal to a minimum threshold support value, and frequent rules as the rules that have confidence value greater than or equal to minimum threshold confidence value. These threshold values are generally assumed to be feasible for mining frequent item sets [1]. Association Rule Mining is all about finding all rules whose support and confidence outstrip the threshold, minimum support and minimum confidence values. Association rule mining advance on two main steps. The first step is to find all item sets with adequate supports and the second step is to generate association rules by combining these frequent or large item-sets.

Support and Confidence: Any given association rule has a support level and a confidence level. Support is the percentage of the population which fascinates the rule or in the other words the support for a rule R is the ratio of the number of occurrence of R, given all occurrences of all rules [5]. The support of an association pattern is the percentage of task-relevant data transactions for which the pattern is true.

$$support(A \rightarrow B) = \frac{number\ of\ tuples\ containing\ both\ A\ and\ B}{total\ number\ of\ tuples}$$

If the percentage of the population in which the antecedent is satisfied is s, then the confidence is that percentage in which the consequent is also satisfied. The confidence of a rule A→B, is the ratio of the number of occurrences of B given A, among all other occurrences given A. Confidence is defined as the measure of certainty or trustworthiness associated with each discovered pattern A →B. Confidence (A →B) = P (B|A) means the probability of B that all know A.

$$confidence(A \rightarrow B) = \frac{number\ of\ tuples\ containing\ both\ A\ and\ B}{number\ of\ tuples\ containing\ \ A}$$



D

TID	Items
T1	I1,I2,I5
T2	I2,I4
T3	I2,I3
T4	I1,I2,I4
T5	I1,I3
T6	I2,I3
T7	I1,I3
T8	I1,I2,I3,I5
T9	I1,I2,I3

C1

Itemset	Sup.count
{I1}	6
{I2}	7
{I3}	6
{I4}	2
{I5}	2

L1

Itemset	Sup.count
{I1}	6
{I2}	7
{I3}	6
{I4}	2
{I5}	2

C2

Itemset	Sup.count
{I1,I2}	4
{I1,I3}	4
{I1,I4}	1
{I1,I5}	2
{I2,I3}	4
{I2,I4}	2
{I2,I5}	2
{I3,I4}	0
{I3,I5}	1
{I4,I5}	0

L2

Itemset	Sup.count
{I1,I2}	4
{I1,I3}	4
{I1,I5}	2
{I2,I3}	4
{I2,I4}	2
{I2,I5}	2

D1

TID	Items
T1	I1,I2,I3
T4	I1,I2,I4
T8	I1,I2,I3,I5
T9	I1,I2,I3

C3

Itemset	Sup.count
{I1,I2,I3}	2
{I1,I2,I5}	2

L3

Itemset	Sup.count
{I1,I2,I3}	2
{I1,I2,I5}	2

D3

TID	Items
T8	I1,I2,I3,I5

Figure 1 Generation of candidate itemsets and frequent itemsets

B. *Multithreading in java*

The Java run-time system [2] depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java adopts threads to enable the entire environment to be nonsynchronous. This helps to lower inefficiency by preventing the waste of CPU cycles. Java's multithreading system is assembled upon the Thread class, its methods, and its companion interface Runnable. The Thread class specifies several methods that help on operate threads. After a computational job is designed and realized as set of tasks, an optimal assignment of these tasks to the processing elements in a given architecture needs to be determined. This problem is called the scheduling problem [6] and is known to be one of the most challenging problems in parallel and distributed computing. a Please do not revise any of the current designations. Java support flexible and easy use of threads; yet, java does not contain methods for thread affinity to the processors. Setting an affinity thread to multiprocessor is not new to research, since it was already sustained by other multiprogramming languages for example C in UNIX platform and C# in Windows platform.

III. METHODOLOGY USED FOR IMPLEMENTATION

A. *Multi core*

Multi core indicate two or more processors. But they differ from separate parallel processors as they are combined on the same chip circuit. A multi core processor developed message passing or shared memory inter core communication methods for multiprocessing. If the number of threads are less than or equal to the number of cores, separate core is allot to each thread and threads run independently on multiple cores. (Figure 2) If the number of threads is more than the number of cores, the cores are shared among the threads.

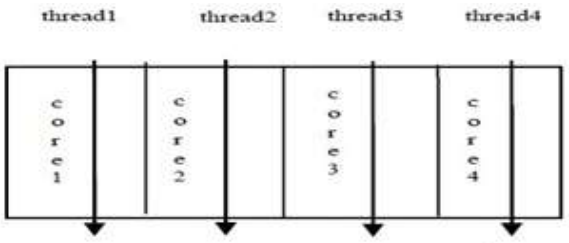


Figure 2: Independent threads on the cores

The idea of multiple cores may seem trivial at first instance. First of all we need to recognize whether the processor should be homogeneous or expose some heterogeneity. Most current general-purpose multi-core processors are homogeneous both in instruction set architecture and performance. This means Multi-core and Many-core Processor Architectures that the cores can execute the same binaries and that it does not really matter, from functional point of view, on which core a program runs. Recent multi-core architectures, allow the system software to regulate the clock frequency for each core individually in order to either save power or to temporarily boost single-thread performance. Most of these homogeneous architectures also create a shared global address space with full cache coherency, so that from a software perspective, one cannot distinguish one core from the other even if the process (or thread) migrates during run-time. By contrast, a heterogeneous architecture features at least two different kinds of cores that may differ in both the instruction set architecture (ISA) and functionality and performance. The most widespread example of a heterogeneous multi-core architecture is the Cell BE architecture, jointly developed by IBM, Sony and Toshiba [2] and used in areas such as gaming devices and computers targeting high performance computing.

B. *Multithreaded Java Applications on Manycore Systems*

Developers of Java applications take up multithreading to carry out higher performance by utilizing multiple processing cores. This performance gain continues to occur until the time spent on subsequent portion of the program outweighs the enhancement conclude through parallelism. Because Java is a supervise programming language, the administration of a Java application is resolve by two work factors: the time used in application execution (mutator time) and the time used in execution of runtime systems such as garbage collection (GC time). These two aspects can also disturb the scalability of a Java application.
In this paper, they have conducted an investigation to reveal factors that can affect scalability of Java applications. Our study considers both mutator and GC times to acknowledge insights on how each can share to the overall scalability of an application. First, we measured the application level lock contention with various numbers of threads. This implies that performance improvement achieved through parallelism in scalable applications outweighs the overhead due to higher instances of lock contention. Second, we characterized object lifespans and solve that higher execution parallelism can root the objects to live for long time. Longer object life spans degrade GC performance because most of the ongoing GC techniques, including those based on the notion of generations,

are most effective when objects die young. Execution parallelism can recover performance by having more threads jointly performing work. In scalable applications, workload can be separated evenly among threads, and therefore, it is important to employ more threads. However, doing so requires precise synchronization of proportion resources that can lead to more lock acquisitions and instances of contention. In addition, we also display that object lifetime is also distressed as the heap usage is an aggregation of objects needed by both operating and suspended threads. Based on the results described in this work, we make two approaches to upgrade scalability of Java applications by focusing on JVM and OS implementation.

C. Load Balancing

The influence of load balance in parallel systems and applications is extensively recognized. Contemporary multiprocessor operating systems, such as Windows Server, Solaris 10, Linux 2.6 and FreeBSD 7.2, use a two-level scheduling path way to enable efficient resource allocation. The first level uses a distributed run queue model with per core queues and fair scheduling policies to manage each core. The second level is load balancing that fix up tasks across cores. The first level scheduling is about time, the second scheduling about space. The implementations in use share a similar design philosophy:

1) Threads are expected to be independent;
2) Load is related to queue length and
3) Locality is significant.

The ongoing scheme of load balancing mechanisms incorporates expectation about the workload management. Interactive workloads are summarizing by independent tasks that are quiescent for long periods (relative to CPU-intensive applications) of time. Server workloads accommodate a large number of threads that are mostly independent and use synchronization for mutual exclusion on limited percentage of data items and not for enforcing data or control dependence [6]. To accommodate these workloads, the load balancing implementations in use do not start threads on new cores based on comprehensive system information. Another implicit expectation is that applications are either single threaded or, when multi-threaded, they run in a devoted environment.

The classic characteristics of existing load balancing designs can be summarized as follows:

1) They are architect to perform best in the cases where cores are periodically idle; and
2) Balancing adopt a core-grained global optimality criterion (equal queue length using integer arithmetic).

D. File Chunking Mechanism

Policroniades and Pratt examined the effectiveness of variable size chunking and fixed size chunking using website data, different data profiles in academic data, source codes, compressed data, and packed files[1].

Fixed Size Chunking: There are two approaches in partitioning a file into chunks: fixed size chunking and variable size chunking. In fixed size chunking, a file is partitioned into fixed size units, e.g., 8 Kbyte blocks. It is smooth, quick, and computationally very economical. A number of proceeding works have adopted fixed-size chunking for backup applications and for large-scale file systems. However, when a small amount of content is inserted to or deleted from the original file, the fixed size chunking may generate a set of chunks that are entirely different from the original ones even though most of the file contents remain intact.

Variable Size Chunking: Variable size chunking partitions a file based on the content of the file, not the offset. Variable size chunking is relatively prosperous against the insertion/deletion of the file. The Basic Sliding Window (BSW) algorithm is widely used in variable size chunking. Fig. 3 explains the BSW algorithm. The BSW algorithm establishes a window of byte stream starting from the beginning of a file. It computes a signature, which is a hash value of byte stream in the window region. If the signature matches the predefined bit pattern, the algorithm sets the chunk boundary at the end of the window. After each comparison, the window slides one byte position and computes hash function again.

In this paper, a innovative multicore chunking algorithm, MUCH, this parallelizes the variable size chunking. To date, most of the existing works on de-duplication focus on expediting the redundancy detection process, while less attention has been paid on how to make the file chunking faster. We found that variable size chunking is computationally very expensive and is a significant bottleneck in the overall de-duplication process. The performance gap between the CPU chunking speed and the I/O bandwidth is expected to become wider with the recent introduction of faster I/O interconnections, and faster storage devices. Incorporating this technology improvement, i.e., increase in the number of CPU cores and emergence of faster storage devices; we developed a parallel chunking algorithm, which aims at making the variable chunking speed on par with the storage I/O bandwidths. We found that the legacy variable size chunking algorithm yields a different set of chunks if the parallelism degree changes, a phenomenon referred to as Multithreaded Chunking Anomaly. We propose a multicore chunking algorithm, MUCH, which guarantees Chunking Invariability [3].

E. Basic Sliding Window (BSW)

Step1:
First interpretation is to endorse quick sort, when pivot is at Kth position, all elements on the right side are greater than pivot, hence, all elements on the left side automatically become K smallest elements of given array.

Step2:
Put an array of K elements, Fill it with first K elements of given input array.

Now from K+1 element, check if the present or ongoing element is less than the maximum element in the auxiliary array, if yes, add this element into array.

Only complication with above description is that we need to keep record of maximum element still workable. How can we keep record of maximum element in set of integer?

Step 3:
Great! In O (1) we would get the max element among K

elements already chose as smallest K elements. If max in present set is greater than newly considered element, we need to wipe out max and suggest new element in set of K smallest element. Now we can quickly obtain K minimum elements in array of N.

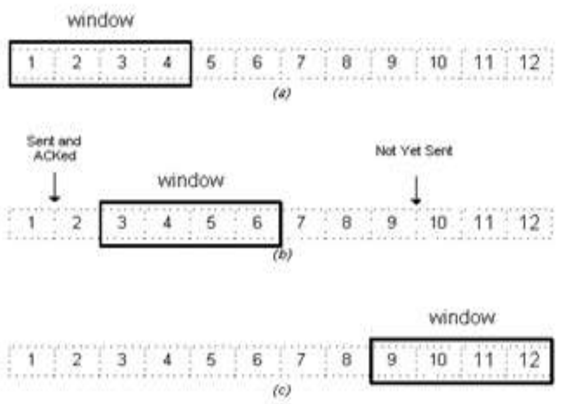


Fig 3 - BSW Mechanism

IV. Proposed System

In this paper, we have introduce the parallel implementation of Apriori Algorithm which can be used by the retailer to extract knowledge for the taking the better business decision.
The transactional database is split into chunks depending on the cores present on the processor. The chunks are processed by the processor parallel to lower the execution time.

V. Conclusion

In summary, the parallel execution can raise the performance by having more threads. The workload is split among the multiple cores of the processor thus decreasing the execution time.

VI. Refernces

[1] JugendraDongre, S. V. Tokekar, and GendLalPrajapati. The Role of Apriori Algorithm for Finding the Association Rules in Data Mining International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), 2014.

[2] JunjieQian, Du Li, WitawasSrisa-an, Hong Jiang and Sharad Seth. Factors Affecting Scalability of Multithreaded Java Applications on Manycore Systems, School of Computer Science, Carnegie Mellon University, 2015.I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[3] Youjip Won, Kyeongyeol Lim, and Jaehong Min. MUCH: Multithreaded Content-Based File Chunking, IEEE Transaction On Computers, VOL. 64, NO. 5, May 2015.

[4] Agrawal R, Srikant R - Fast algorithms for mining association rules‖ In: Proceedings of the 1994 international conference on very large data bases (VLDB‟94), 1994 Santiago, Chile, and pp 487–499

[5] MamtaDhanda,” An Approach To Extract Efficient Frequent Patterns From Transactional Database”,In: International Journal of Engineering Science and Technology (IJEST), Vol.3 No.7 July 2011, ISSN:0975-546. M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.

[6] Zaki MJ (1999) Parallel and distributed association mining: a survey. IEEE Concurr7(4):4–25, *Special issue on Parallel Mechanisms for Data Mining.*