Distributed Frequent Hierarchical Pattern Mining for Robust and Efficient Large-Scale Association Discovery

---

a Dissertation

presented to

the Faculty of the Graduate School

at the University of Missouri-Columbia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

by

MICHAEL PHINNEY

Dr. Chi-Ren Shyu, Dissertation Supervisor

May 2017

The undersigned, appointed by the Associate Vice Chancellor of the Office of Research and Graduate Studies, have examined the dissertation entitled

DISTRIBUTED FREQUENT HIERARCHICAL PATTERN MINING FOR ROBUST AND EFFICIENT LARGE-SCALE ASSOCIATION DISCOVERY

presented by Michael Phinney,

a candidate for the degree of doctor of philosophy,

and hereby certify that, in their opinion, it is worthy of acceptance.

Professor Chi-Ren Shyu

Professor Jeffrey Uhlmann

Professor Prasad Calyam

Professor Guilherme DeSouza

Dedication

This dissertation is dedicated to my family and many friends.

Without their love and support, this work would not have been possible.

Acknowledgments

This dissertation would not have been possible without the guidance of my research advisor, Dr. Chi-Ren Shyu. In addition, my doctoral committee members, Dr. Jeffrey Uhlmann, Dr. Prasad Calyam, and Dr. Guilherme DeSouza, influenced the direction of this research. Furthermore, my collaborators were integral at each step of the research process: Sean Lander, Matt Spencer, Dr. Hongfei Cao, Yan Zhuang, Dr. Lincoln Sheets, Devin Petersohn, Kurt Bognar, Ali Raza, Blake Anderson, Andi Dhroso, Dr. Jerry Parker, and Dr. Philip S. Yu.

The research in this dissertation was funded by the US Deportment of Education Graduate Assistance in Areas of National Need (GAANN) Fellowship under grant number P200A100053, Paul K. and Dianne Shumaker Endowment for Biomedical Informatics, and National Science Foundation CNS-1429294.

Contents

Listing of figures

DISTRIBUTED FREQUENT HIERARCHICAL PATTERN MINING FOR ROBUST AND EFFICIENT LARGE-SCALE ASSOCIATION DISCOVERY

Michael Phinney

Dr. Chi-Ren Shyu, Dissertation Supervisor

ABSTRACT

Frequent pattern mining is a classic data mining technique, generally applicable to a wide range of application domains, and a mature area of research. The fundamental challenge arises from the combinatorial nature of frequent itemsets, scaling exponentially with respect to the number of unique items. Apriori-based and FPTree-based algorithms have dominated the space thus far. Initial phases of this research relied on the Apriori algorithm and utilized a distributed computing environment; we proposed the Cartesian Scheduler to manage Apriori's candidate generation process. To address the limitation of bottom-up frequent pattern mining algorithms such as Apriori and FPGrowth, we propose the Frequent Hierarchical Pattern Tree (FHPTree): a tree structure and new frequent pattern mining paradigm. The classic problem is redefined as frequent hierarchical pattern mining where the goal is to detect frequent maximal pattern covers. Under the proposed paradigm, compressed representations of maximal patterns are mined using a top-down FHPTree traversal, FHPGrowth, which detects large patterns before their subsets, thus yielding significant reductions in computation time. The FHPTree memory footprint is small; the number of nodes in the structure scales linearly with respect to the number of unique items. Additionally, the FHPTree serves as a persistent, dynamic data structure to index frequent patterns and enable efficient searches. When the search space is

exponential, efficient targeted mining capabilities are paramount; this is one of the key contributions of the FHPTree. This dissertation will demonstrate the performance of FHPGrowth, achieving a 300*x* speed up over state-of-the-art maximal pattern mining algorithms and approximately a 2400*x* speedup when utilizing FHPGrowth in a distributed computing environment. In addition, we allude to future research opportunities, and suggest various modifications to further optimize the FHPTree and FHPGrowth. Moreover, the methods we offer will have an impact on other data mining research areas including contrast set mining as well as spatial and temporal mining.

Chapter 1

Introduction

In this dissertation, we focus on a few classic problems in computer science: frequent pattern mining (FPM), association rule mining (ARM), and Cartesian products. Throughout the discussion, algorithms, optimizations and application are presented. With an algorithm design perspective, our focus is algorithm performance and scalability. All algorithms proposed in this dissertation were analyzed based on horizontal and vertical scalability and designed for distributed computing environments.

The general outline of this introduction is as follows. First, we introduce the general concepts and terminology of FPM and ARM. Next, we provide a survey of related work in FPM. A classic FPM algorithm, Apriori, relies on an iterative Cartesian Product at it's core; in the Cartesian Operations section, we briefly discuss the history of the Cartesian Product and highlight challenges faced in distributed computing environments. Then, we discuss several real-world applications of FPM and ARM. Finally, we provided a general overview of distributed computing and outline the remaining dissertation organization.

1.1 Background and General Terminology

ARM is a widely used and mature area of research in computer science and a key area of data mining [1]. The concept originated as part of market basket analysis. The high-level task was to identify cases where if a customer buys product i_1, i_2, and i_3; they will also buy product i_4 with probability p [2, 3]. The first step in ARM and identifying these *association rules* is FPM, finding all of the items that cooccur frequently. From this collection of frequent patterns, we may calculate conditional probabilities between itemsets, yielding association rules. The general concept of ARM is to identify patterns that exist in arbitrary transaction datasets. These transaction data are composed of events containing specific items or metrics involved in each event. These events can be anything from a doctors visit, a trip to the grocery store, taking an exam, or playing chess, among others.

One example of an interesting use-case is working with clinicians and a database containing electronic medical records [4–6]. In this scenario, transactions could be individual patient visits. Within each transaction (hospital visit), many measurements may be collected, such as height, weight, temperature and blood pressure. In this context, these measurements would make up the items for each transaction. Patient demographics such as race, age, and gender could also be considered in addition to those collected metrics. Each are considered an item within the patient transactions as well. In addition to those measurements, each transaction might contain high-level information about patient health, such as whether they are healthy, sick, or severely sick. ARM could help identify which

patient attributes seem to correlate with specific outcomes [6, 7]. For example, if the majority of men over the age of fifty are likely to have high blood pressure, ARM would be able to programmatically identify that rule. In the medical domain, this sort of predictive power is invaluable; it may enable clinicians and physicians to implement preventive medical treatments to improve patient health before a traumatic event occurs [8].

One of the major challenges associated with ARM arises from the complexity of FPM. The worst-case scenario, a dataset containing k items has $O(2^k)$ subsets. Suppose all of those subsets are frequent patterns. Regardless of which algorithm is selected to tackle the problem, the results will yield $O(2^k)$ patterns, as shown in Figure 1.1.

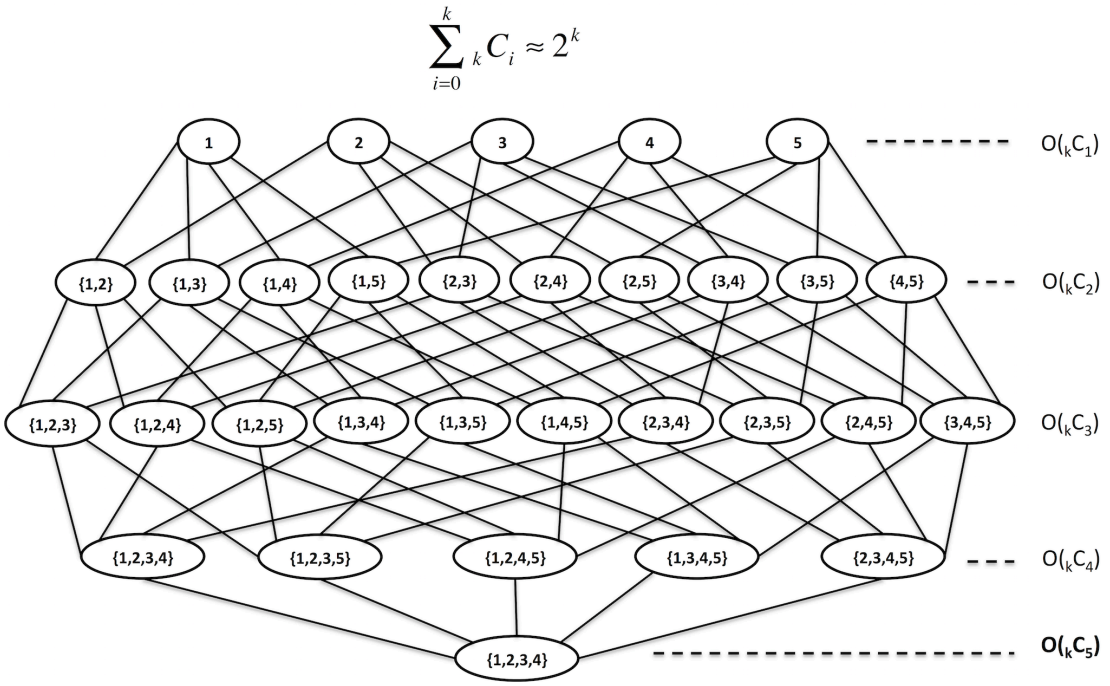

Figure 1.1: Visualization of the exponential nature of frequent itemsets.

Efficient algorithms began to gain popularity with Agrwawal in 1993 [9]. This area of research has been around for several decades; the traditional terminology used is defined as follows: Let $I = i_1, i_2, .., i_m$ be the set of all items. D be

a database of transactions where each transaction $T \subset I$. A collection of items $X \subset I$ is referred to as an itemset. X is contained in T if, and only if, $X \subset T$. An association rule is a direct implication following the form of $R_a => R_c$, where $R_a \subset I$ is considered the rule antecedent, $R_c \subset I$ is the rule consequent, and $R_a \cap Rc = emptyset$. A rule is said to have support s if and only if $s\%$ of all transactions in D contain $R_a \cup Rc$. Similarly, a rule has confidence c if and only if $c\%$ of the transactions containing R_a also contain both R_a and R_c. Confidence is the conditional probability of R_c given R_a. We say an itemset is frequent if and only if its corresponding support is greater than or equal to some user defined minimum support threshold, *min_support*. An itemset of size k is referred to as a $k - itemset$. We say an association rule is confident if and only if its confidence is greater than or equal a user defined minimum confidence threshold, *min_confidence* [10].

1.2 Classic Algorithms

Since the introduction of ARM by Agrwawal in 1993 [9], dozens of follow-up methods and incremental improvements have been proposed. Of the many contributions made, most can be generalized as Apriori-based or growth-based [11]. The Apriori-based approaches all utilize a candidate generation step followed by a filtering step to remove candidates that do not satisfy the minimum support criteria [12]. The growth-based approaches do not generate candidates; they construct a graph structure, and frequent patterns are identified by traversing the graph [13]. Both classes of algorithms contain approaches for identifying maximal and closed frequent patterns.

1.2.1 Apriori

The general workflow of the Apriori algorithm is given in Figure 1.2. The Apriori algorithm begins by scanning a database of transactions. Each transaction is then split into individual items, which are then counted. The items and frequencies are piped into a pruning step, which filters the data using the user-defined minimum support criteria, removing those relatively-infrequent items. If the resulting set of frequent items is non-empty, the data is passed to the candidate generation phase. This step uses those results to generate itemsets one size larger. This is possible because of a theorem stating: "All subsets of a frequent itemsets, must also be frequent," a concept referred to as downward closure. This notion allows us to take a bottom up approach to identifying frequent itemsets.

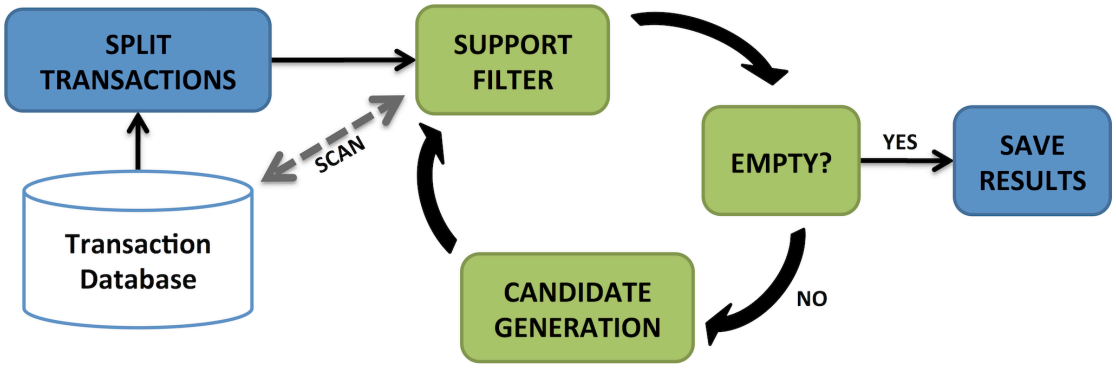

Figure 1.2: A high-level visual overview of the Apriori algorithm.

Once candidates have been generated, we screen the results through the support filter, which will remove all of the candidates that do not exceed the minimum support threshold. This process continues until the result of the support filter is empty at which point we will find no larger frequent itemsets. This general computing template has been studied, and a collect of efficient algorithms have been developed based upon it.

A variety of Apriori-based approaches have been proposed over the past decades, all of which are similar in that they iteratively utilize a candidate generation step followed by a filter based on the min_support threshold. The AIS algorithm iteratively scans a transaction database, generates candidates, and determines the support [9]. The candidate generation step is performed by augmenting frequent itemsets from the previous iteration, with items that occur in the same transaction. The disadvantage to this approach is that a large number of candidates are generated.

SETM differs from AIS in one significant way: During each pass, the list of transactions associated with each candidate itemset are stored, and the support for an itemset is determined by aggregating its corresponding transaction list [14]. The overhead from the transaction lists can be substantial and may become a bottleneck. The Apriori algorithm addresses several of the limitations of AIS and SETM by using only the frequent itemsets generated in the previous pass and does not reference the database of transactions when generating the candidates [12]. Rather, the candidate generation step is a self-join with the previous iteration's frequent itemsets. The database is then scanned to acquire the support counts for each candidate, and those that are not frequent are deleted. Similarly, Apriori-TID generates candidate using a self-join technique, but it provides an incremental improvement over Apriori by eliminating database scans from the support filter after the first pass [12]. As a result, it incurs more overhead and may perform worse than Apriori on the first few iterations. This fact inspired the Apriori-Hybrid algorithm, which utilizes Apriori for the initial passes and transitions into Apriori-TID when the estimated over-

head is manageable [12].

Hashing techniques have also been applied to offer substantial improvements over Apriori on the candidate generation phase. The DHP Algorithm can achieve improvements orders of magnitude over Apriori in the size of intermediate candidate generation [15]. CHARM focuses specifically on identifying closed frequent itemsets is able to prune more intermediate itemsets, which results in an improved runtime [16]. A-CLOSE is another Apriori-based algorithm designed to identify closed frequent itemsets [17]. Repetitive database scans are also removed in the BORDERS algorithm by keeping track of the list of transactions for each candidate itemset [18]. All subsequent support counts and candidate generation steps are performed without revisiting the transaction database. This frequent pattern mining approaches proposed in this research utilizes a similar technique. MAFIA prunes large amounts of intermediate itemsets by narrowing the search-space and focusing on maximal itemsets [19]. Additional approaches utilize concepts of computational optimization such as Apriori GA, which utilizes a genetic algorithm to extract frequent itemsets [20]. Another probabilistic method, Fuzzy Apriori, was proposed and implements fuzzy logic to determine frequent patterns [21].

Another alternative approach is Reduced Apriori Algorithm with Tag, which improves upon the efficiency of the pruning operation and reduces the burden of candidate generation [20]. Several other algorithms were proposed that focus on closed itemsets, and are able to prune and reduce the data a faster rate, yielding quicker runtimes [22–24]. A variety of flavors and combinations of tuning mechanisms provide incremental improvements to the underlying logic

workflow of candidate generation and support filtration.

In response to these readily available cluster-computing packages, a new suite of association rule mining algorithms are being produced. There is a push for existing algorithms to be rethought and slightly reformulated to work on distributed computing environments. A few years ago, shortly after Hadoop was released, a MapReduce implementation of Apriori was released [25, 26]. This approach enabled massive transaction databases to be processed, but suffered from the bottlenecks inherent of the MapReduce framework, disk IO. A few years later, soon after the introduction of Spark, an in-memory Apriori implementation was contributed [27]. It addressed the issues of the existing MapReduce implementation. As mentioned previously, this technique was not optimal; many improvements have been made to the basic Apriori algorithm. Another approach was proposed: Yet Another Frequent Itemset Mining (YAFIM), which takes advantage of Spark's broadcast variables feature [28]. The filter step was highly optimized by broadcasting the results of the previous iteration to each node in the cluster.

1.2.2 FPGrowth

Two of the main limitations of Apriori, candidate generation and repetitive database scans, can be addressed using an FP-Tree [13]. To alleviate the need to rescan the database and generate large intermediate collections of candidate itemsets, a tree structure is constructed and traversed in order to identify frequent itemsets. As a result, the approach requires exactly two database scans.

The first database scan counts the support for each item in the collection.

Then, the second database scan iterates over each transaction, and the frequent items are sorted in support-descending order. Next, we begin populating the FP-Tree. We first initialize a root node with the empty set as its label. Each transaction is then inserted into the FP-Tree. Each node in the FP-Tree has two labels: (1) the item it represents, and (2) a frequency counter. The first item i in the transaction will be the most frequent; if a child of the root node is labeled i, its corresponding frequency count is incremented. Otherwise a new node is created and given label i for the item and 1 for the initial frequency count. In addition to adding this new node to the tree, we add a link between i and the last node created with label i. This allows us to create linked lists between nodes with the same label and will help with calculating the item support. The process for inserting the second item from the transaction is very similar. However, the starting position for the graph traversal is the previous updated or inserted node. This process is typically recursively defined, and performed for all transactions within the database.

Once the tree has been populated with all of the transactions from the database, we can extract the frequent itemsets. This is accomplished by constructing conditional FPTrees. A conditional tree in constructed for each frequent item by isolating on those transactions that contain that specific item. This corresponds to a subtree of the FPTree, and after removing the item itself from this subtree, we refer to it as a conditional FPTree. From this point, the conditional FPTrees are traversed in a recursive manner to extract frequent patterns.

In many cases, FP-Growth is significantly faster than Apriori. In fact, the performance of FP-Growth and Apriori are largely dependent on the dataset and

TID	Items
1	{a,b}
2	{b,d,e}
3	{a,d}
4	{a,e,c}

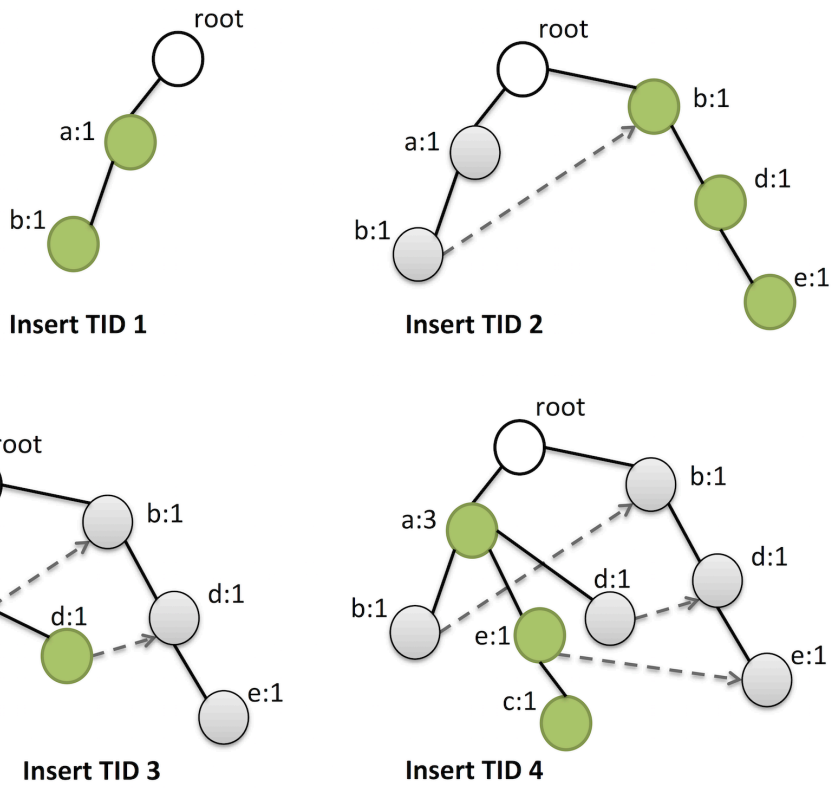

Figure 1.3: A high-level visual overview of the FPGrowth algorithm.

the support threshold set [11]. The major strong suit of the FP-Tree is its level of compression on the dataset. In a dense dataset, containing a small collection of items that occur over a massive set of transactions, FP-Growth has distinct advantages over Apriori. When the dataset is sparser, and the number of items increases, the FP-Tree can actually have an inflating effect on the data. Both of these approaches have been successful for a variety of reasons; however, both have limitations. Apriori is ideal for sparse datasets where the maximal frequent itemsets are not large. FPGrowth is great for dense datasets but suffers when the dataset becomes sparse and the number of items becomes large.

Fewer algorithms have been proposed on top of FP-Growth when compared to Apriori. A collection of algorithms were developed that focus on mining only maximal closed itemsets, including FPMax and CLOSET [29–31]. The FP-Tree

is not an ideal structure for handling dynamic transaction data. To address this issue a modified data structure called the CATs Tree was proposed [32]. To apply taxonomy to the itemsets, FP-tax was introduced [33]. The QFP-Growth algorithm was created to reduce the overhead associated with the FP-Tree as well as the number of conditional FP-Trees generated [34]. Pfp, a parallel implementation of FP-Growth was also proposed to allow the work in creating and populating the FP-Tree to be distributed on multicore environments [35].

In addition to FPM, sequential pattern mining (SPM) generalizations exist for the Apriori and FPGrowth algorithms discussed. The fundamental difference between FPM and SPM is item order. SPM considers the order of items important, while FPM focuses entirely on cooccurrence. The iterative discovery of increasingly long patterns is fundamentally unchanged.

1.3 Cartesian Operations

A naive Apriori implementation relies on a Cartesian product to be performed iteratively on increasingly large itemsets. This Cartesian product is the bottleneck for the Apriori algorithm; the candidate generation step scales proportionally to $O(n^2)$. As a result, we are interested in optimizing this operation to improve performance. By utilizing a distributed computing environment, we are able to extend the scalability by introducing additional hardware resources.

The Cartesian product (CP) was named after French philosopher, Rene Descartes (1596-1650) [36]. The CP $A \times B$ is defined as the set of all pairs (a, b) such that $a \in A$ and $b \in B$. This century-old concept has been referred to as all-against-all or pairwise comparisons.

CPs, also referred to as pairwise comparisons, are expensive operations. Having complexity $O(n^2)$, the class of algorithms that rely on a CP operations are not generally scalable. We are still interested in extending the reach of CP operations. As a result, we may turn to distributed computing. By increasing the amount of available computing resources, we may handle more data. Distributing a CP operation introduces new types of complexity. When performing a distributed Cartesian product, it is important to note that each individual comparison, (a, b) where $a \in A$ and $b \in B$, must take place on a single physical machine. This notion is what creates a challenge. At some point each pair of data points must exist on the same physical machine in our computing cluster in order for each comparison to be made. As a result, one of the main challenges of performing a CP in a distributed environment is facilitating the shuffle, coordinating where and when each individual comparison is performed. Apache Spark provides a built-in mechanism for facilitating this shuffle.

Apache Spark is an open source in-memory cluster computing framework [37]. It was introduced as an alternative or replacement to Hadoop MapReduce. A key component of Spark is the concept of resilient distributed datasets (RDDs), distributed, lazy execution-based, persistent data structures. Lazy execution implies that the evaluation of an expression is delayed until its value is needed. In effect, this strategy operates like the composition of functions and allows multiple expressions to be performed at once to avoid repeated evaluations. Also, since RDDs are persistent data structures, they can be cached which promotes efficient access to data that will be used repeatedly. The majority of our methodology is composed of and described as RDD transformations and actions [38].

A key challenge in distributed computing is ensuring a uniform distribution of data. Previous studies have been conducted on straggler detection, in particular, the causes of stragglers. In distributed computing, a straggler is a task or job that takes significantly longer than its concurrently running sibling tasks or jobs. For example, if a job is partitioned into 10 tasks to be processed in parallel, each of those 10 tasks should execute in a similar amount of time. If one task takes 10 times longer than the others, it would be considered a straggler. One of the main sources of stragglers is an imbalance in data that causes a single node to be burdened with the majority of work [39]. Randomization plays an important role in the Cartesian Scheduler and has been shown in the past to perform well under circumstances that require uniformly binning data points [40].

One of the most common use cases for CPs is search. Query languages like SQL, VSM, Cypher, SparQL, and XPath require pairwise combining and comparison operations [41–45]. It is most common to find these technologies in use on single machine environments.

In recent years, many query languages have gained support in distributed computing environments. For example, VSM acquired an extension into the distributed computing space [46]. Google announced F1, a scalable distributed relational database system [47]. In addition to database management systems, SQL-like language interfaces are continually becoming more common in distributed systems [48, 49].

Another search-based application for CPs is document similarity. Pairwise document similarity has been useful in information filtering, information retrieval, indexing and document ranking [50–52]. All-against-all comparisons

can be performed on large collections of documents using distance metrics such as cosine similarity and semantic similarity [53].

One of the most recent contributions to CPs in a distributed computing environment is Apache Spark's RDD method, Cartesian [37, 38]. The key contribution of Spark's CP is in the way they partition the problem, breaking the global CP into a collection of small, local CPs. After performing these local CPs, the results will be aggregated and are equivalent to the global CP. This approach is limited in the granularity and initial distribution of the partitions which becomes more noticeable as the data becomes larger and imbalanced. It can also lead to unsatisfactory performance for CP operations.

1.4 Real-World Applications

In this section, we provide a few examples of real world applications of frequent pattern mining and association rule mining. We provide a wide range of domains to express the generality of this data mining technique. In addition to FPM, we discuss applications of sequential pattern mining as well, since methodologically, both concepts are quite similar. All of the examples discussed are oversimplified and are merely used as conceptual guidelines.

1.4.1 Market Basket Analysis

Every time we checkout at the grocery store, barcodes are scanned and data is logged somewhere. This information is stored in a transactional format, and has a direct correlation to the consumer. When leveraged with data mining, this can be incredibly useful for marketing. For example, if the store knows that

most customers that buy bread also buy milk, the products could be physically relocated to be closer to reduce store congestion or further apart to increase the amount of time browsing in the store. This is one simple example for the usage of this associative product information.

1.4.2 Electronic Medical Records

With the technological shift in healthcare, options for data mining driven personalized medicine are becoming possible. Electronic medical records are tracking vast amounts of information from patient demographics to morbidities. This information can be used to learn common trends and associations between demographics, diseases, and health outcomes. For example, If you have high blood pressure and a genetic history of heart disease, what is the likelihood of cardiac arrest? Having the ability to answer questions such as this could dramatically improve health outcomes on a global level.

1.4.3 Bioinformatics and Genomics

For decades, biologists have been working to answer questions such as ”What are commonalities between the genetic information in humans, mice, or other animals” and ”Can this genetic correlation tell us more about the evolutionary timeline of Earth?” Information about phylogeny can be drawn from direct association between DNA sequence data. Stronger associations could suggest stronger evolutionary relationships. This is a simple example of how the concept of frequent pattern mining can be used to uncover information about our evolutionary history.

1.4.4 Stock Market

This technology could be used to analyze statistics in stock market trends as well. For example, correlations between companies might exist such that if the value of one stock shifts, the value of another could be affected. In this way, recommendations could be made to investors based on the status of 3rd party stocks.

1.4.5 Web Log Analysis

Recommendation systems are commonplace on the Internet today. Online giants like Netflix, Amazon, and Google are constantly analyzing user access patterns and making recommendations based on what others like you have done in the past. In the case of Netflix, after watching a movie, the system suggests other similar movies. How does Netflix know which movies are most similar? Movies are annotated with genres, casts, directors, release dates, and listings for users. This information can be used to identify correlations between movies and used as guidelines for recommendations.

1.5 Distributed Computing

Cluster computing is the concept of utilizing multiple computer machines together to solve a computational problem. Conceptually, a cluster can be interacted with as if it were a single machine with a single massive file system with a shared disk pool; the fundamental difference is that this cluster framework allows for the file system to scale horizontally by connecting more machines. In many scenarios, the cluster will have a master node and a collection of slave

nodes. All of these machines are connected and can communicate over a high-speed network. The master plays the role of the conductor and is responsible for task allocations and job scheduling. The master also usually acts as a gateway to the slave nodes. That is to say, all instructions are passed directly to the master node, and it will partition the work and distribute individual tasks to the slave nodes.

A distributed in-memory computing framework is designed to provide not only a shared disk pool, but a distributed memory space as well. Since disk operations are significantly slower than those that are performed in-memory, bringing this sort of concept to large-scale cluster computing frameworks can really a huge effect on performance and overall throughput. Systems like this are designed to allow data to persist in memory and to be shared between various computational workloads. The jobs need not be running simultaneously, or even in tandem. A wide array of applications have been built on top of Apache Spark, including Tachyon, a reliable memory centric distributed storage system. Figure 1.4 demonstrates the high-level conceptual layout of an in-memory cluster computing framework.

The figure above shows data being loaded onto the cluster, passing through the master node to be evenly distributed across the slave nodes where it will be either written to disk or persisted in memory. It is important to mention that these systems are designed to be fault tolerant. Each piece of data is replicated multiple times and stored on multiple machines to ensure that if a machine goes down data will not be lost.

One of the fundamental challenges of cluster computing relates to data local-

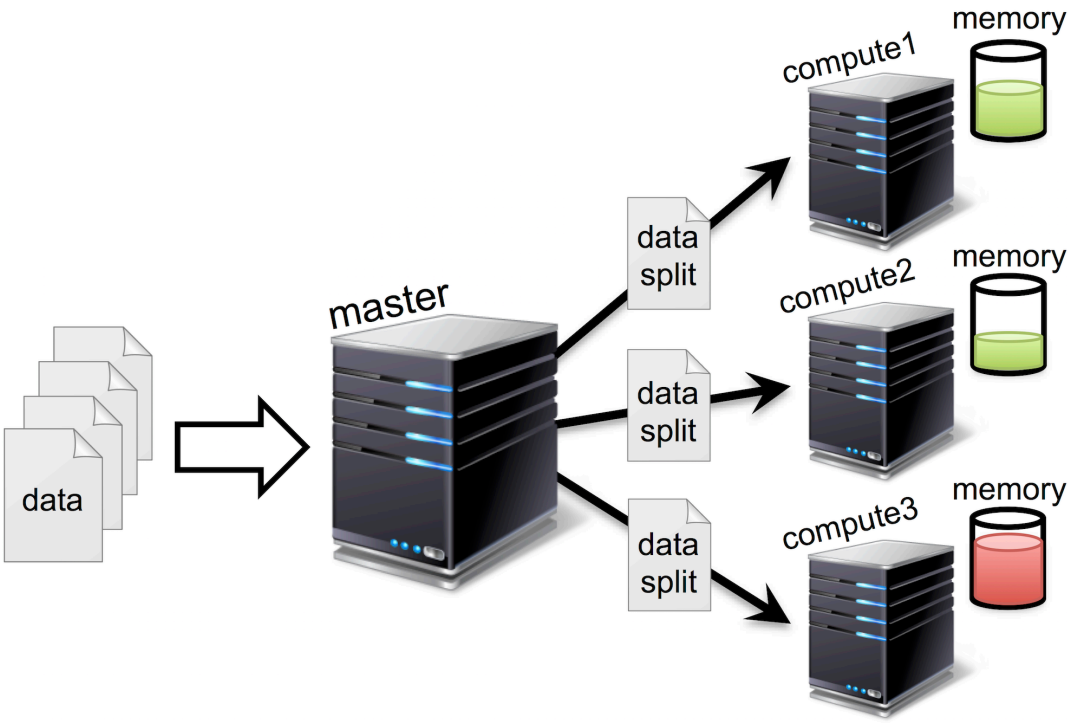

Figure 1.4: A simple example of a cluster computing environment.

ity. Repetitively transferring data across the network can be expensive and, at times, unnecessary. Data that will need to be compared with one another must be on the same machine, so if this data can be loaded to the same machine initially, unnecessary expensive operations in the future can be prevented. This is one of the major contributions of Spark. Unnecessary shuffling of the data common to the MapReduce programming model were reduced significantly if not completely eliminated.

Although clusters can be interacted with in a manner similar to a single file system, limitations can necessitate a clear distinction. In a cluster, the memory pool is disjointed, so data on one node cannot be accessed directly from another node. This is why in many languages designed for cluster computing are functional: They are based on lambda calculus. All operations can be thought of as a series of transformations on the data. This makes manipulating the data, al-

though it physically partitioned across many machines, naturally parallelizable. Since we utilize this type of infrastructure, the methods we propose are naturally horizontally scalable, fault tolerant, and memory centric. As a result, we can develop higher throughput, and reliable computational pipelines that are appropriate for large-scale data analytics.

With the ubiquity of distributed computing frameworks such as Apache MapReduce and Apache Spark, it has become significantly more affordable for people to ingest, transform, and visualize large amounts of data. Frameworks such as Hadoop and Spark are completely open-source and have strong communities providing support and advancements on a regular basis. Part of the appeal of these frameworks is that they are open-source, but can also run on commodity hardware. Creating a big data ecosystem is now as easy as setting up cluster of relatively-inexpensive nodes connected over an Ethernet network.

In recent years, advancements have been made in the in-memory cluster computing space, in particular, the Apache Spark project. These frameworks quickly became known throughout the big data community after setting the world data sorting record at the annual Terasort competition. In 2013, a Hadoop cluster of 2,100 nodes (2 2.3Ghz hexcore Xeon E5-2630, 64 GB memory, 12x3TB disks) set the record for sorting 102.5 terabytes of data in 4,328 seconds, approximately 1.42 terabytes per minute. In 2014, Apache Spark surpassed this record with a runtime of 1,406 seconds, approximately 4.27 terabytes per minute, and used a fraction of the hardware resources: 207 nodes (32 vCores - 2.5Ghz Intel Xeon E5-2670 v2, 244GB memory, 8x800 GB SSD).

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we will discuss one of the bottleneck for the original Apriori algorithm, the Cartesian Product. Our research focused on optimizing Cartesian operations in distributed computing environments. In Chapter 3, we propose a novel FPM paradigm. We redefine the problem as frequent hierarchical pattern mining and propose the FHPTree, a data structure designed to efficiently identify long frequent patterns. In Chapter 4, we present advances to the frequent hierarchical pattern mining paradigm including extensions to distributed computing environments. In Chapter 5, the dissertation is concluded and the contributions are summarized. Lastly, a brief overview of the Author is given.

Chapter 2

Cartesian Operations and the Distributed Apriori Algorithm

The classic Apriori algorithm relies on a Cartesian product to be performed iteratively on increasingly large itemsets. This Cartesian product, the candidate generation step, scales proportional to $O(n^2)$ and is a major bottleneck for the Apriori algorithm. Our goal was to implement the Apriori algorithm in a distributed computing environment, and we found that the performance of Cartesian operations on distributed datasets posed an issue. In this section, we discuss a technique designed to optimize Cartesian products on distributed computing environments. As a result, we are able to extend the scalability of Cartesian operations by introducing additional hardware resources.

2.1 Cartesian Scheduler

The first step for the Cartesian Scheduler is to provide a mechanism for partitioning the workload of a CP. As shown in Figure 2.1, we utilize a randomized virtual partitioning technique to uniformly distribute the data across a collection of compute nodes. Next, we collocate all virtual partitions in preparation for a series of partial CPs. This is accomplished by following the virtual partition

pairing protocol. For each virtual partition pair, we perform a CP. By following the proposed protocol, we guarantee the aggregation of all partial CPs is equivalent to a full CP on the original data. In the following theorem, we prove that performing all of the partial CPs defined by our VP pairing protocol is equivalent to performing the full CP between the two original RDDs.

Theorem 1 (Partial Cartesian Products)**.** *Every Cartesian product between two RDDs can be represented as a collection of partial Cartesian products.*

Proof. We show that every comparison performed in the full CP also exists in one of the partial CPs. For $x \in RDD_1$ and $y \in RDD_2$, let (x, y) be an arbitrary single comparison performed in the full CP. Suppose we partition the data; we are guaranteed that x and y are each contained within a partition. Without loss of generality, suppose $x \in P_1$ and $y \in P_2$. Then, a comparison (x, y) will occur when performing the partial CP between P_1 and P_2. Thus, every comparison that occurs in the full CP also occurs in at least one of the partial CPs. Furthermore, since $P_i \cap P_j = \varnothing$ for all $i \neq j$, we are guaranteed comparison (x, y) will occur in exactly one of the partial CPs. Therefore, the global CP can be represented as an equivalent set of partial CPs. □

Theorem 1 solidifies the logic utilized by the Apache Spark framework. We consider this the basic and classic distributed Cartesian product approach. Similar to the classic approach, the concept is to split the steps involved in a full CP into a functionally equivalent collection of partial CPs. Figure 2.2 demonstrates this concept as a graph where the nodes are partitions of data, and edges are a representation of partial CPs.

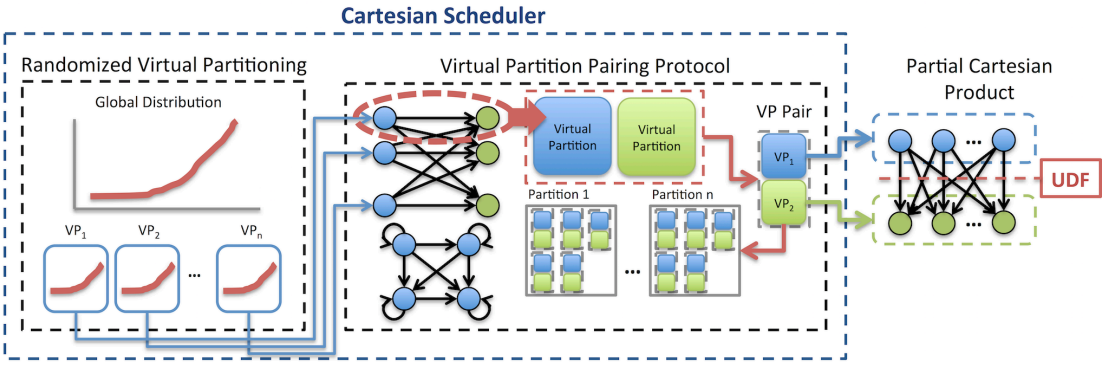

Figure 2.1: Overall architecture for the Cartesian scheduler. The first phase is randomized virtual partitioning. The results are fed through virtual partition pairing protocol. Finally, partial Cartesian products are performed on the virtual partition pairs, and the results are aggregated.

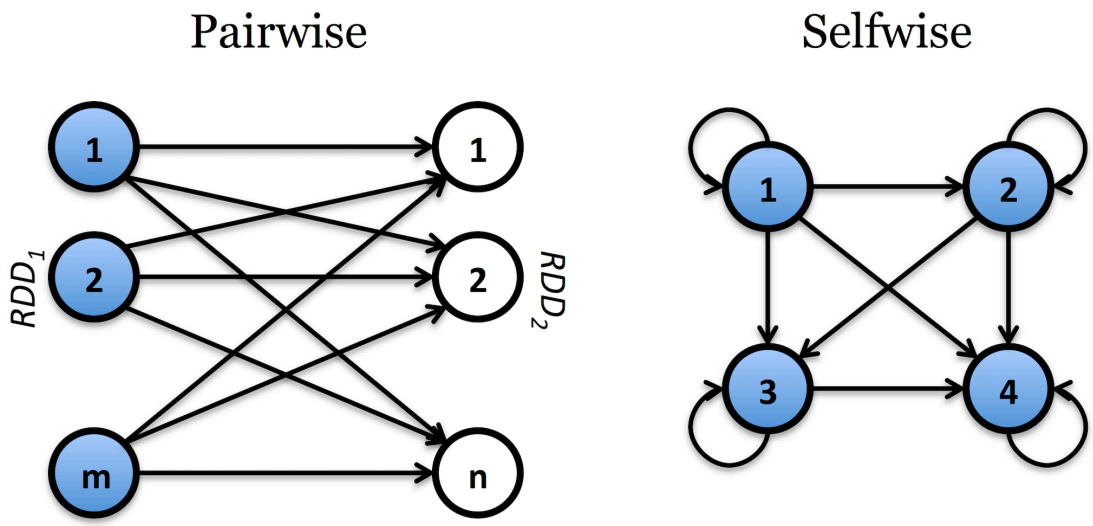

Figure 2.2: Each node in the pairwise graph (left) corresponds to a partition of one of the RDDs involved in the Cartesian product. The solid color nodes belong to one RDD, and white nodes belong to the other. The selfwise graph (right) demonstrates the differences between the full pairwise and selfwise comparisons. Each edge in a graph corresponds to one of the partial CPs also refered to as VP pairs.

The Cartesian Scheduler incorporates an added layer of partitioning logic, and addresses several limitations of the classic approach. The fundamental distinctions of this work are randomized virtual partitioning and the virtual partition pairing protocol. Both of these concepts and commensurations for optimizations in implementations are discussed in detail in subsequent sections.

2.1.1 Randomized Virtual Partitioning

Virtual partitioning is one of the key components of the Cartesian Scheduler. A virtual partition (VP) is an added abstraction layer of partitioning logic on a dataset. That is, in addition to the physical partitioning scheme, we introduce another variable-grouping paradigm that provides control over the size and granularity of physical partitions. We require the virtual partitions to be uniform with respect to dimensionality distribution and size to ensure balanced workloads. Our goal is to ensure each VP contains proportional collections of vectors. If one VP acquires all vectors with the largest dimensionality, it will likely incur greater runtime delays. As a result, we would not achieve a balanced workload. Figure 2.3 demonstrates an ideal data distribution.

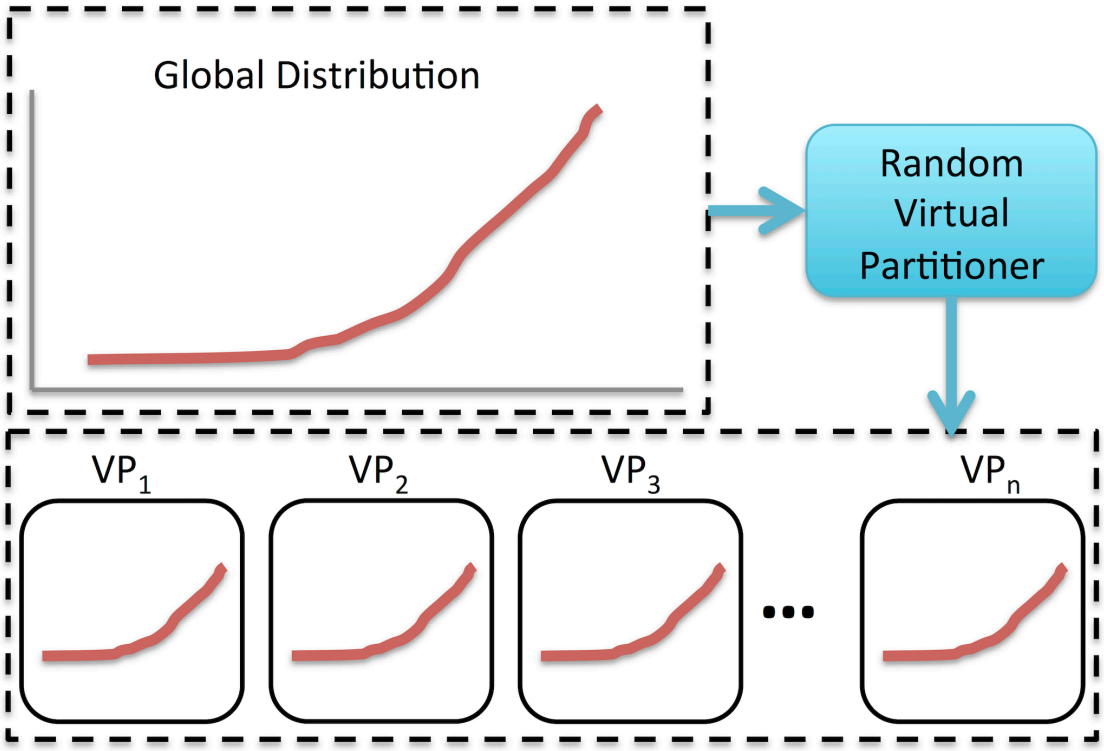

Figure 2.3: The main goal of virtual partitioning is achieving a uniform distribution of data. The distribution of data on each VP should be proportional to the global data distribution.

We design a randomized virtual partitioning scheme to ensure a uniformly

distributed workload. By definition, randomness provides a uniform distribution. To argue that randomization is a proper solution, we focus on the probability of encountering the worst case. In this domain, the worst case would be all data points getting allocated to the same virtual partition. The probability of this occurring is $\frac{1}{k^n}$ where k is the number of virtual partitions in the cluster and n is the size of the RDD. As the RDD becomes large, this probability becomes negligible. There are numerous near-worst-case scenarios that we also want to avoid; one VP has most of the data. More generally, a single VP acquires more data than other VPs, hence increasing the likelihood of stragglers. To account for near-worst-case scenarios, we generalize the probability function as follows:

$$\sum_{k=\frac{n}{k}+\beta}^{n} \frac{(k-1)^{n-m}}{k^n} = \frac{1-(k-1)^{\frac{n(k-1)}{k}+\beta+1}}{(2-k)k^n} \tag{2.1}$$

where k is the number of nodes in the cluster, n is the size of the RDD, m is the size of an arbitrary VP; VP_0, and β represent the degree to which VP_0 differs from the average, expected size of a VP.

Suppose our data points are not uniform; that is, each data point need not be the same size or dimensionality. Reusing the variable length vector example, a case that we must avoid is where all of the largest vectors are added to the same VP. In this case, the computations involving this VP will take much longer than the average case. By randomly allocating the vectors to VPs, we are able to reduce the potential of encountering this case; thus, the probability is $\frac{1}{k^x}$ where x is the number of abnormally large vectors in the dataset.

An additional important benefit to utilizing a randomization, no additional communication overhead is required to facilitate the balancing of VPs. The VP

assignments are determined independently of one another which lends itself to the distributed computing paradigm.

Algorithm 1 Randomized Virtual Partitioning

```
RDD.map(
a → (Random.nextInt( NumVP ), a)
).groupByKey
```

The pseudocode for this randomized partitioning logic is given in Algorithm 1. We perform a map transformation on *RDD* that randomly assigns each datapoint, a, to a VP. The total number of VPs is denoted by *NumVP*. The partitioning is performed by constructing a key-value pair; the key is the VP identifier, and the value is the original datapoint, a. As a post-processing step, a groupByKey action is performed to aggregate and physically relocate data based on VP assignment. Furthermore, for any two VPs, VP_1 and VP_2, we are guaranteed that $VP_1 \cap VP_2 = \varnothing$

The theoretical discussion in this section relies on true randomness. As a result, performance is dependent on the random function implementation of the underlying programming language. In our case, we rely on the performance of Java's standard library Random function.

2.1.2 Virtual Partition Pairing Protocol

Next, we perform an all-against-all comparison between the VPs from opposing RDDs. This is similar to the classic approach; however, we are operating on the VPs as opposed to the physical partitions. In addition, we collocate all VP pairs before performing the partial Cartesian products. That is, we shuffle the data to ensure the VPs from a VP pair are colocated on a single physical machine.

Thus, there will be no additional shuffling performed once the partial Cartesian products have begun. Each comparison between VPs consists of a partial CP. That is, we perform a full CP between the data of one VP and the data of another VP.

The proof of Theorem 1 applies to VPs the same way as physical partitions. We are able to focus our attention on effective scheduling where and when each partial CP is performed. The pseudocode for this phase is in Algorithm 2. The first step is performing a transformation on each virtual partition for both RDDs (line 8). This transformation is accomplished by first passing each VP through the *pairVPs* function. This function takes three parameters: a VP ID, the number of VPs in the opposing RDD, and a flag that denotes which RDD the VP came from. Suppose VP_1 from RDD_1 is passed through this function. The output will be a collection of tuples denoting the VP ID mapping for all VP comparisons involving VP_1. To clarify, the output collection is $\{(VP_1, x) \mid x \text{ is a VP from } RDD_2\}$. After all VP IDs from both RDDs are passed through *pairVPs*, union the results and perform a *groupByKey* operation to aggregate and collocate VPs. The resulting RDD is a collection of virtual partition pairs awaiting partial CPs to be performed.

Next, we provide an example walkthrough of Algorithm 2. Suppose we have datasets, $a = [1, 2, 3, 4, 5, 6]$ and $b = [7, 8, 9, 10, 11, 12]$, and a and b are both virtually partitioned into 2 parts. Equation 2.2 provides the list of VPs generated.

$$\begin{aligned} VP_{a,1} &= [1, 2, 3], \quad VP_{a,2} = [4, 5, 6] \\ VP_{b,1} &= [7, 8, 9], \quad VP_{b,2} = [10, 11, 12] \end{aligned} \tag{2.2}$$

Algorithm 2 Virtual Partition Pairing

```
1: function pairVPs(VP_ID,numVP,isRDD_1)
2:     if isRDD_1 then
3:         return (0 until numVP).map(i → (VP_ID, i))
4:     else
5:         return (0 until numVP).map(i → (i, VP_ID))
6:     end if
7: end function
8: RDD_1.flatMap( vp →
9:      pairVPs(vp.ID,numVP_2,true)
10:     .map(a→(a,vp.data))
11: ).union(RDD_2.flatMap(vp →
12:     pairVPs(vp.ID,numVP_1,false)
13:     .map(a→(a,vp.data))
14: )).groupByKey
```

To clarify, the format of Equation 2.2 is $VP_{ID} = DATA$. Line 8 shows that for each VP, we perform a map transformation. The first step in transforming $VP_{a,1}$, line 9, is generating *VP.ID* pairs, corresponding to the VP Pairs involving $VP_{a,1}$. Equation 2.3 provides the result.

$$VP_{a,1} \rightarrow [(VP_{a,1}, VP_{b,1}), (VP_{a,1}, VP_{b,2})] \tag{2.3}$$

On line 10, we package the data from $VP_{a,1}$ with the newly constructed *VP.ID* pairs. Continuing on from Equation 2.3, Equation 2.4 provides the result.

$$[((VP_{a,1}, VP_{b,1}), [1, 2, 3]), ((VP_{a,1}, VP_{b,2}), [1, 2, 3])] \tag{2.4}$$

By letting the *VP.ID* pairs form the key of a tuple, we are able to perform a groupByKey operation to aggregate data and form VP pairs. One of the resulting VP pairs is given in Equation 2.5.

$$((VP_{a,1}, VP_{b,1}), ([1, 2, 3], [7, 8, 9])) \tag{2.5}$$

Referring back to Figure 2.2, the VP pair defined in Equation 2.5 corresponds to one edge in the pairwise graph.

2.1.3 Number of Virtual Partitions

Determining an appropriate number of virtual partitions is a key challenge of the Cartesian Scheduler. The number of VPs directly determines the number of partial CPs that will be performed. Recall that our goal is to provide a method of moderating the granularity and size of the physical partitions. Since the number of physical partitions is predefined, define the number of VPs to remain proportional to the number of physical partitions. Since the all-against-all comparison between the virtual partitions from opposing RDDs, $a * b$ VP pairs (partial CPs) will be formed, where a is the number of VPs in RDD_1, and b is the number of VPs in RDD_2.

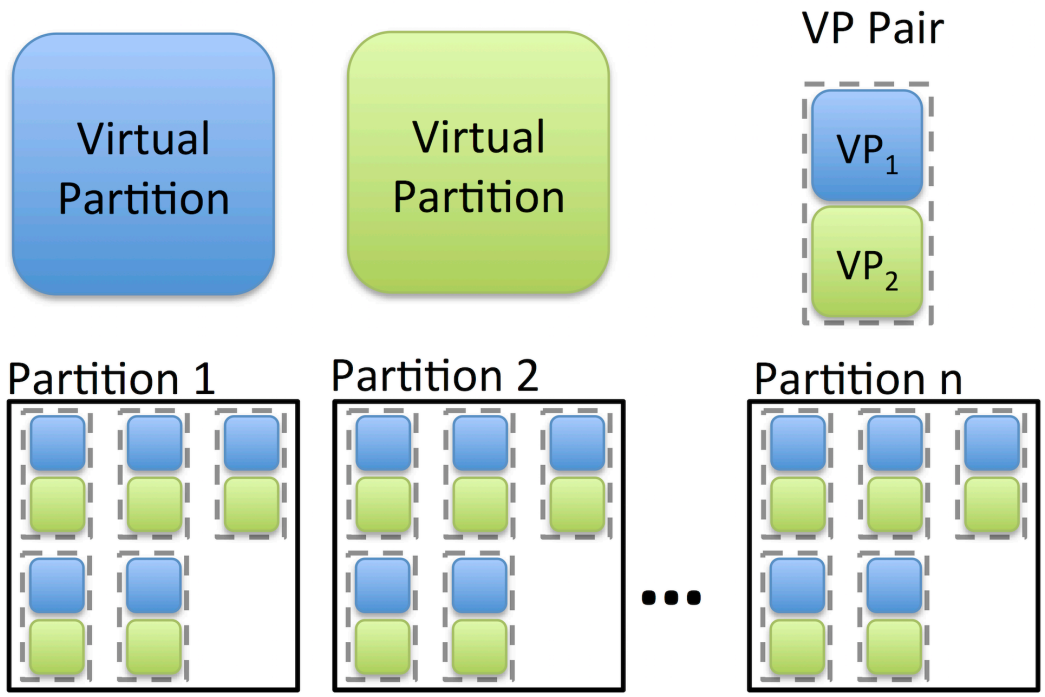

Figure 2.4: The number of VP pairs should be proportional to the number of physical partitions. As a result, each partition should have the same number of VP pairs.

To preserve the number of partitions requested by the user, m and n are the size of RDD_1 and RDD_2, respectively, and choose a and b such that the following equation is satisfied.

$$a * b = k(m + n) \tag{2.6}$$

Since the number of VP pairs is evenly divisible by the number of physical partitions, each physical partition can acquire the same number of VP pairs. As a result, each physical partition can house the same number of partial CPs, which promotes a uniform workload. Figure 2.4 provides a visual representation of this logic. This constraint, however, is not strong enough to provide us with values for a and b; there are infinitely many solutions to Equation 3.2. To further constrain the formulation, consider the following constraint.

$$\frac{a}{b} = \frac{m}{n} \tag{2.7}$$

The goal is to scale the partitions of opposing RDDs by the same factor. Thus, the ratio of a and b must be consistent with m and n.

Consider the task of choosing k, the sharding factor, which directly determines the number of partial CPs that will be performed on each physical partition. As a result, the sharding factor also determines the size of each partial CP. In the current status of the Cartesian Scheduler, various empirically evaluated sharding factors are considered, and a selection is made based on runtime. The variables required to determine an appropriate sharding factor are cluster size and input data size. By adding more nodes to a cluster, the sharding factor may be reduced. Larger datasets require a larger sharding factor. For each of our ex-

periments, we provide the sharding factor used to achieve the given runtimes.

2.1.4 Partial Cartesian Operations

Once we have formed the VP pairs, we may begin performing the partial CPs. As part of the Cartesian Scheduler API, we request a user defined function be provided as the comparator between the elements from one VP and the opposing VP. Each partial CP will take place on one of the nodes in our cluster, i.e., the same node that received the corresponding VP pair during the VP Pairing Protocol. We then perform a traditional pairwise comparison. The logic for this is given in Algorithm 3.

Algorithm 3 Partial Cartesian Product

```
1: function partialCartesian(VP1,VP2,function)
2:     return VP1.flatMap(a →
3:         VP2.map(b → function(a, b))
4:     )
5: end function
6: VPPairRDD.flatMap( a →
7:     partialCartesian(a._1,a._2)
8: )
```

The partial CP is performed using a VP pair as shown on lines 6 – 8. The CP is perform using a nested for-loop to perform each comparison between opposing VPs. Since we are utilizing a functional programming language, this translates into nested maps as shown on lines 2 and 3.

2.1.5 Selfwise Cartesian Product

There is an important special case of the Cartesian product to mention, the selfwise Cartesian product. As shown in Figure 2.2, the number of required comparisons is halved; the full Cartesian between an RDD of size n and itself is n^2,

but the selfwise Cartesian product only requires $\frac{n(n-1)}{2}$. Using the Spark API, we must compute the full pairwise comparison, and filter out the redundant comparisons as a post-processing step. We optimize our approach to negate redundant computation, effectively, halving the runtime for selfwise cartesian products. The pseudocode given in Algorithm 4 details the logic for handling this special case. Notice the subtle differences compared to Algorithm 2.

Algorithm 4 Selfwise VP Pairing

```
 1: function pairVPs(VP_ID,numVP)
 2:     return (0 until numVP).map(i →
 3:     if i < VP_ID then (i,VP_ID)
 4:     else (VP_ID,i)
 5:     end if
 6:     )
 7: end function
 8: RDD.flatMap( vp →
 9:      pairVPs(vp.ID,numVP)
10:      .map(a→(a,vp.data))
11: ).groupByKey
```

The logic defined in Algorithm 4 aligns with that of Algorithm 2. We begin by transforming the RDD of VPs using the *pairVPs* function. This function takes two parameters: a VP, the total number of VPs in the RDD. Suppose we pass VP_1 from *RDD* through this function. The output collection is $\{(x, y) \mid x < y$, x and y are VP IDs from *RDD*, and x or y is the ID for VP_1 $\}$. After all VP IDs pass through *pairVPs*, we perform a *groupByKey* operation to aggregate and collocate VPs. Just like the full pairwise case, the resulting RDD is a collection of virtual partition pairs awaiting partial CPs to be performed.

2.2 Experimental Results and Validation

The Cartesian Scheduler is evaluated on uniformness, runtime, and scalability. In addition, we conducted an analysis focused on characterizing the sharding factor. The goal is to demonstrate the impact it has on the overall performance of the Cartesian Scheduler. To determine the advantages and potential limitations, five experiments were conducted. First, we considered a heterogeneous dataset to validate the uniform distribution properties of virtual partitioning. Second, we took a closer look at the sharding factor and demonstrated its impact on the overall performance of the Cartesian Scheduler. As an extension of the second experiment, our third evaluated a pairwise difference between two large RDDs of integers to demonstrate the control, in terms of parallelism, gained by the sharding factor. Next, we performed a horizontal scalability analysis, characterizing the effect additional compute nodes have on runtime. Finally, we achieved a pairwise vector distance benchmark by performing a document similarity analysis on Reuters-21578, Distribution 1.0 [54]. For this experiment, we collected runtimes for both the Cartesian Scheduler and the classic Cartesian approach packaged in the Apache Spark 1.3.0 release.

We ran the following experiments on a standalone Spark 1.3.0 computing cluster. The cluster used in these experiments was composed of eight nodes; each node consisting of an 8-core processor and 80 GB of RAM. One of these nodes is the designated master of the cluster, and the remaining seven are compute nodes. We will refer to this hardware as Cluster 1. Throughout the course of these experiments, we utilize 56 physical partitions, one partition per CPU

core in the cluster, to promote full resource utilization.

2.2.1 Validation Criteria

Accuracy tests are conducted during each experiment that compares the Cartesian Scheduler and the classic algorithm implemented in Spark. We ensure that the output from the Cartesian Scheduler is identical to that of Spark's Cartesian function during every test that is conducted. This validation check is performed, and both methods consistently yield identical results. In addition to verifying the accuracy, we validate our approach based on its ability to achieve a uniformly distributed workload. By ensuring a relatively similar number of data points fall into each VP, we achieve a uniform distribution. When the dataset is a collection of vectors that vary in dimensionality, the task becomes more challenging; however, we show that a randomized paradigm addresses this case as well.

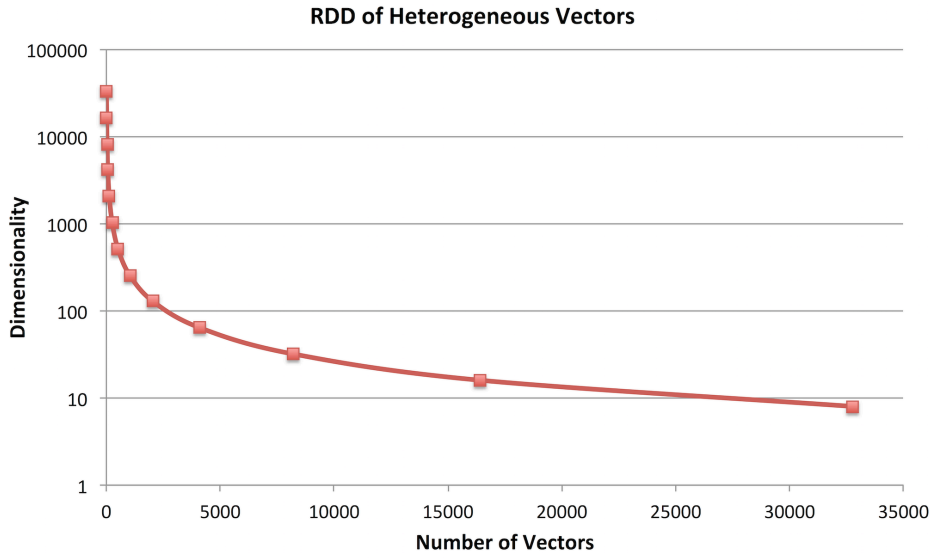

Figure 2.5: Dataset 1 follows a roughly power law distribution of vectors with respect to dimensionality. Few vectors have high dimensionality, and many vectors have low.

Dataset 1 is a generated heterogeneous dataset that exhibits a power law distribution with respect to dimensionality. Figure 2.5 characterizes Dataset 1 and

is used to empirically evaluate the overall effectiveness of our randomization technique in achieving a uniformly distributed workload. It is important to note that the vertical axis in Figure 2.5 uses a log scale. Dataset 1 follows a power law distribution which poses a greater potential for imbalanced workloads in comparison to a random distribution. This simple fact is why a dataset with a power law distribution is used in this experiment.

As shown in Figure 2.6, the distribution with respect to data size is relatively uniform across all of the executors in the cluster. When analyzing Dataset 1, the Cartesian Scheduler produces 56 tasks, which are then distributed to one of the 7 executors. Through experimentation and based on the size of the computing cluster, we empirically determine 40 to be an effective sharding factor. We conduct this experiment 100 times, utilizing a sharding factor of 40, and evaluate the uniformness with respect to data size and runtime on each executor in Cluster 1. Recall that the sharding factor is the number of partial cartesian products that will be performed within each physical partition. In this experiment, the number of physical partitions was equivalent to the number of cores in the cluster, 56 partitions, so the sharding factor defined the number of partial cartesian products that are performed on each core.

In addition, Figure 2.6 shows runtime and data distribution are relatively uniform across all executors. The average data size for each task is around 3.5 MB with a standard deviation of 0.37 MB. The standard deviation with respect to runtime is 24 seconds, which is small relative to the average executor runtime, 405 seconds. Although this could be considered a small example, recall that by the probability function defined in the Virtual Partitioning section, we see the

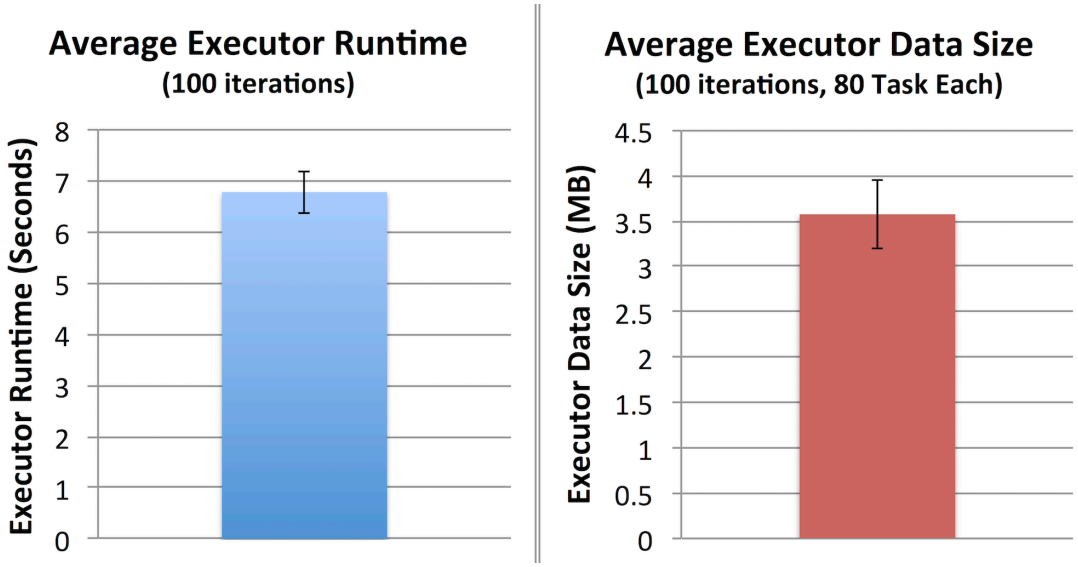

Figure 2.6: The graph on the left details the average runtime for each task; the vertical axis corresponds to runtime. The graph on the right details the average datasize for each task; the vertical axis corresponds to the data size. In addition, both graphs include the standard deviation acquired by each experiment. Runtimes were collected using Cluster 1.

probability of imbalance decreases as the input data size increases. Since the variance in runtime and data distribution between executors from the same jobs is low, we consider this evidence to support our claim of achieving a uniformly distributed workload.

2.2.2 Sharding Factor Analysis

In this experiment, we conduct a study on Dataset 2 which contains two RDDs of 200,000 integers. The operation we used to compare each pair of integers was subtraction. That is to say, compute the difference $r1 - r2$ of every $r1 \in R1$ with every $r2 \in R2$. Consider the following demonstration of this task. Let $R1 = [4, 2, 3]$ and $R2 = [1, 2, 6]$. The result of a pairwise difference $R1 \times_{diff} R2 = [3, 2, -2, 1, 0, -4, 2, 1, -3]$. Thus, our experiment consisted of approximately 40 billion individual comparisons. The psuedocode for this test is provided in Algorithm 5.

The goal was to demonstrate the importance of selecting an effective shard-

Algorithm 5 Pairwise Difference

```
1: function diff(a,b)
2:     return a – b
3: end function
4: // data0 = RDD[Int]
5: // data1 = RDD[Int]
6: var cs = new CartesianScheduler();
7: cs.run(data0,data1,diff);
```

ing factor. The sharding factor offers the ability to fine tune the distribution of the data to ensure a balanced workload. It determines the number of partial Cartesian products (virtual partition pairs) that will get mapped to each physical partition. In addition, the sharding factor controls the degree of redundancy introduced into the data, which helps promote full cluster utilization. We consider a variety of sharding factors when analyzing Dataset 2, beginning at 30 and increasing to 200,000.The results from 30 to 400 are represented visually in Figure 2.7.

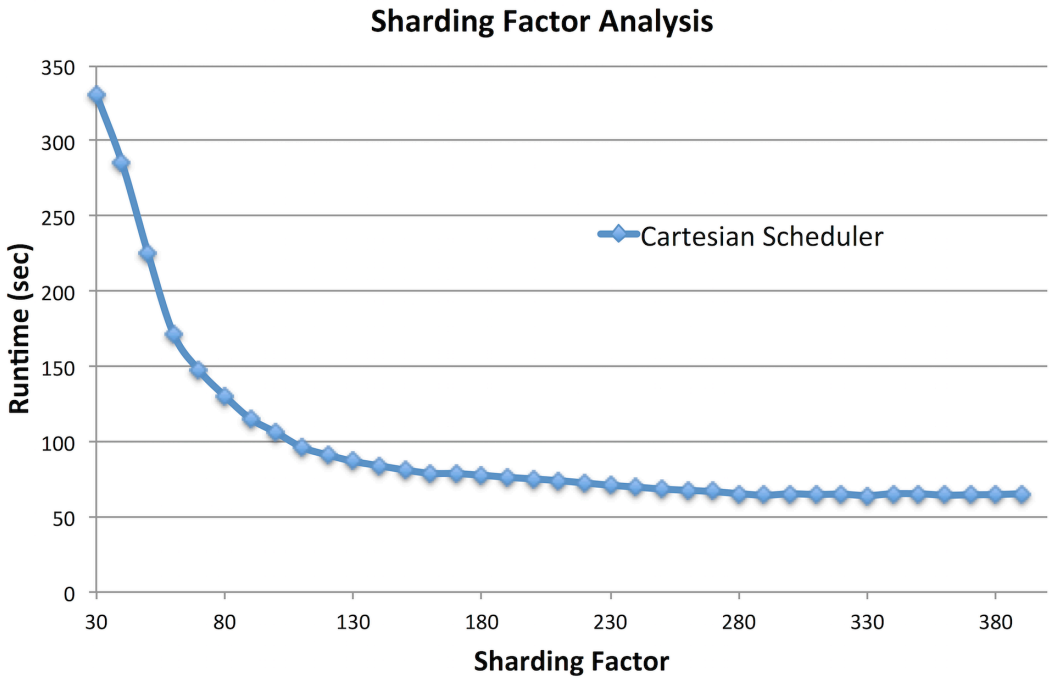

Figure 2.7: Runtimes collected from the pairwise difference experiment with various sharding factors. In this case, larger sharding factors yield faster runtimes.

When conducting the pairwise difference between large Integer RDDs, the Cartesian Scheduler will redundantly copy data points to several machines to reduce network dependencies in the computational pipeline. As shown in Figure 2.7, the degree of redundancy has a notable impact on performance. As the sharding factor increases, redundancy increases, and hence the level of parallelism can be increased. As a result, larger sharding factor can achieve shorter runtimes. However, there can be a point where the amount of redundancy becomes a burden; we may eventually exceed the resource capacity by creating too much redundancy. A sharding factor of 200,000 introduces the maximum amount of redundancy allowed by our Cartesian Scheduler implementation. The memory footprint was approximately $\sqrt{\frac{200{,}000}{400}} = 22$ times greater in comparison to a sharding factor of 400. Cluster 1 is still able to fit this inflated version of Dataset 2 in memory, roughly 50MB per worker node; however, the runtimes were equivalent.

Notice that the initial improvement from sharding factor 50 to 60 is dramatic relative to the improvement gained when incrementing the sharding factor from 140 to 150. Eventually, the percentage of computation spent on actually cpu cycles approaches 100% for each executor. At this point, we were not able to gain further improvements on runtime by increasing the sharding factor. As mentioned above, increasing the sharding factor further merely increases the storage overhead without yielding notable runtime improvements.

Next, we consider a study on how the input data size influences the sharding factor. We utilize integer RDDs, the same format as Dataset 2. The size is varied from 50,000 integers up to 300,000, and sharding factors ranging from 25 up

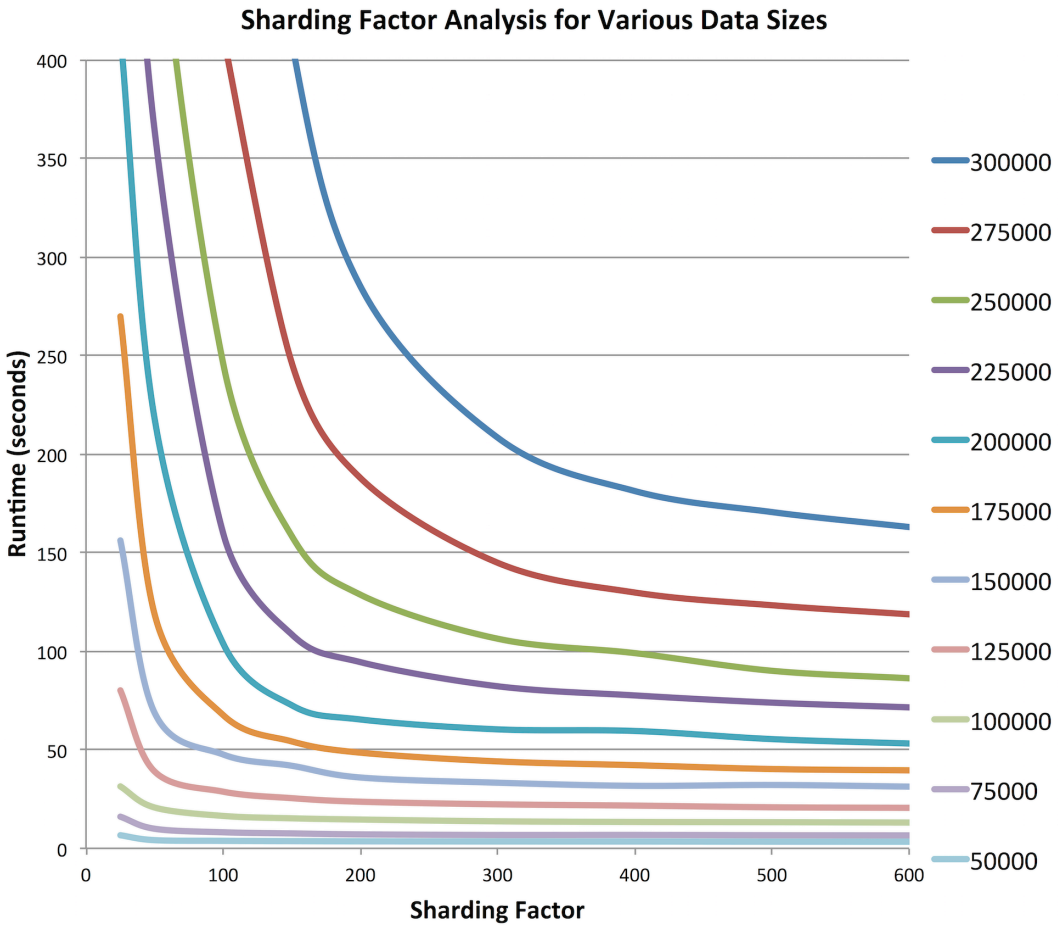

Figure 2.8: Runtimes collected from the pairwise difference experiment with various sharding factors. Integer RDD data size is varied from 50,000 integers up to 300,000. The order of the legend corresponds to the order of the curves.

to 600 were tested for each data size.

For each data size, the minimum viable sharding factor was determined using the runtimes displayed in Figure 2.8. For each curve, the minimum viable sharding factor is the smallest sharding factor at which increasing the sharding factor by 50 yields performance gains less than 10 percent. Further performance improvements can be made by increasing the sharding factor. This measure focuses on establishing a lower bound for the sharding factor. If the sharding factor is lowered, the method will suffer significant performance losses.

The key take-away is that the minimum viable sharding factor appears to scale quadratically as the input data size increases. Since the Cartesian product

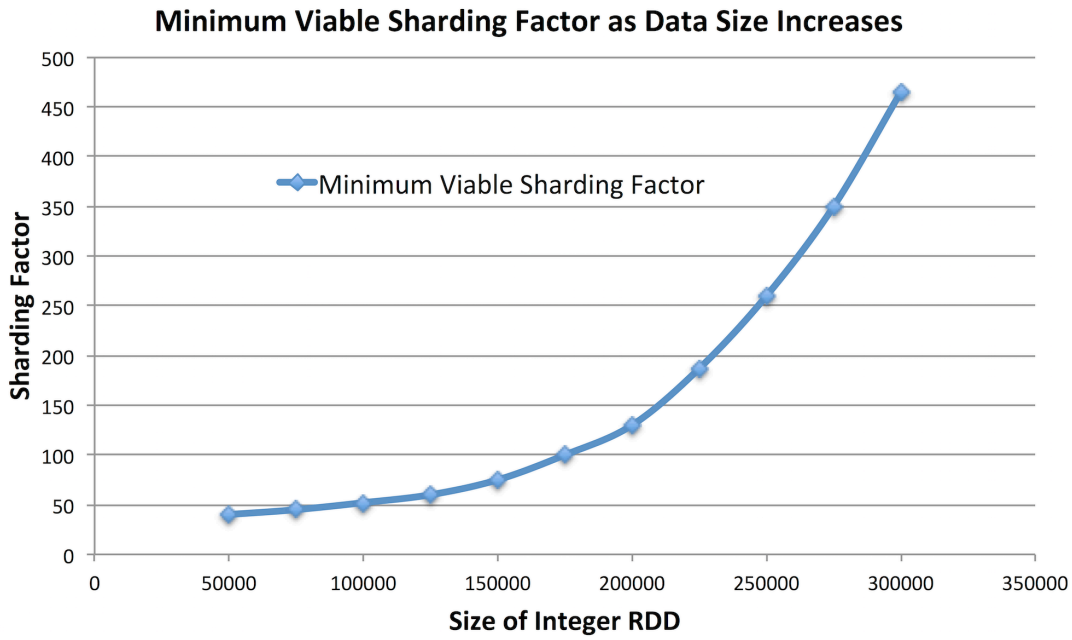

Figure 2.9: Minimum viable sharding factor as the data size increases.

is a quadratic operation, it is natural that the sharding factor scales proportionally. This is a demonstration of the Cartesian Scheduler's ability to control the granularity of the distributed CP. By identifying the ideal size for a partial CP, the Cartesian Scheduler can efficiently perform CPs in a distributed setting.

A recommendation for repetitive Cartesian analyses, as shown in this experiment, is to invest time in determining an effective sharding factor. Furthermore, this experiment introduces a new challenge, developing an automated method for determining the optimal sharding factor. We highlighted in this experiment that the size of the dataset has a dramatic impact on the minimum viable sharding factor. A variety of key factors must also be considered when identifying an appropriate sharding factor such as the size of the cluster, size of each data point, memory and storage limitations.

2.2.3 Horizontal Scalability

In this experiment, we vary the number of compute nodes in our Spark cluster from 3 nodes up to 18 nodes. The 18 nodes used to conduct this experiment have the same resources were nodes in Cluster 1. Similar to the last experiment, we considered Dataset 2 for the benchmark. It is important to mention that for each experiment with k nodes, we consider various sharding factors and report the fastest runtime achieved.

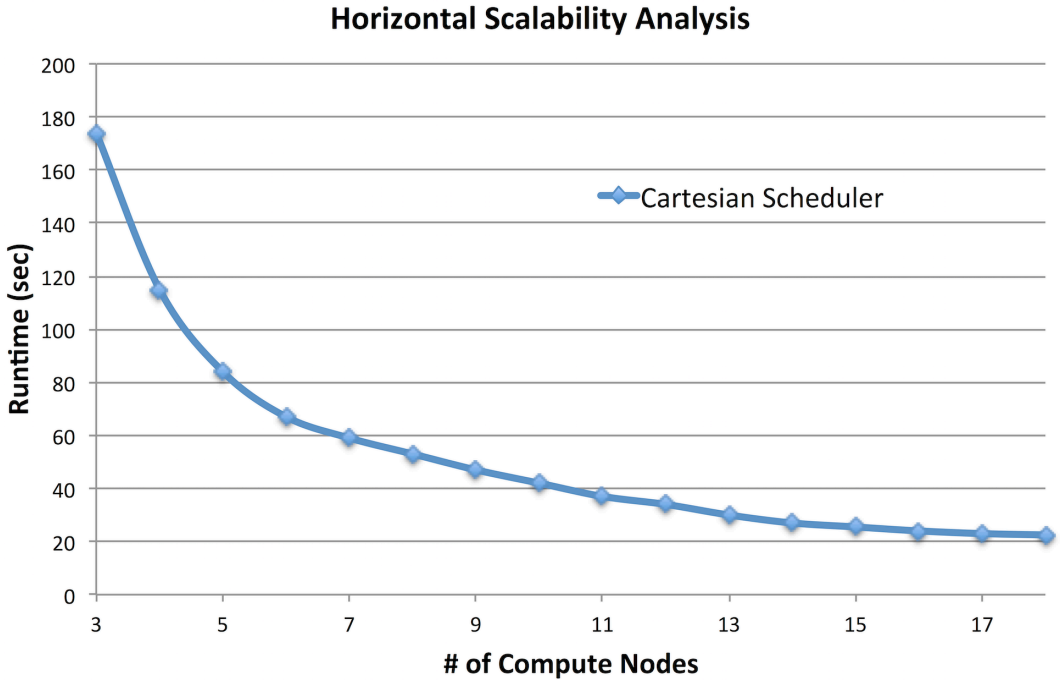

Figure 2.10: Runtimes collected from the pairwise difference experiment on 200,000 integers. The number of compute nodes in the cluster is varied from 3 to 18 nodes.

From Figure 2.10, we see that the Cartesian Scheduler is horizontally scalable. When the number of nodes is doubled, the Cartesian Scheduler achieves a 2x speed boost, halving the overall runtime. This is shown when increasing the number of nodes from 3 to 5; the runtime was improved from 174 seconds to 84 seconds. Since a Spark cluster has a master node which does not perform computation, a three node cluster has two compute nodes, and a five node cluster

has four. With twice the compute resources, the runtime is halved. The runtime trend is characterized by $runtime(ck) = \frac{runtime(k)}{c}$, where k is the initial number of nodes, and c is the multiple by which to increase the number of nodes in the cluster. This formula will eventually become inaccurate as the number of nodes in the cluster approaches the number of data points involved in the CP.

2.2.4 Pairwise Vector Distance Analysis

A common application for pairwise vector distances is information search and retrieval. When searching for the highest similarity between objects in a collection, the obvious solution is to compare each object with every other object and to pick out the two objects with the highest similarity. This approach is commonly referred to as brute force but is nothing more than a modified Cartesian product. In particular, it is a Cartesian product where the comparison operation performed between each data point is a distance function.

In this experiment we compare the classic Cartesian method as implemented in Apache Spark against the Cartesian Scheduler and consider Dataset 3, Reuters-21578, a corpus of approximately 19,000 articles. To conduct a full document similarity analysis, we generate feature vectors for each article in Dataset 3 and compute the distance between each pair of articles. The documents that have a relatively small distance are likely to be similar. The features are represented as binary vectors that associate terms with documents. That is, if some term t occurs in document d, the bit stored in column t of document d's feature vector would be set to 1. Before constructing these feature vectors, we determine which terms to consider relevant. Stop words are ignored, and we discard terms

that occur in at least five but no more than 5,000 articles; those discarded terms may not provide much information to support a correlation between documents. Each articles had 8,873 features.

Algorithm 6 Pairwise Vector distance

```
1: function vectorDistance(a,b)
2:     return (0 until a.size)
3:     .map( i→ (a[i] – b[i]) ∗ (a[i] – b[i]) )
4:     .reduce(_+_)
5: end function
6: // data = RDD[Vector[Int]]
7: var cs = new CartesianScheduler();
8: cs.run(data,vectorDistance);
```

To conduct scalability tests, we use random sampling to produce varying sizes of datasets. We considered eight data sizes beginning with 2,500 articles and incrementing by 2,500 each time until we reach 19,000. Algorithm 6 presents the pseudocode for this experiment. Notice on lines 3 and 4, we define a metric that is equivalent to the square of the Euclidean distance. Since the goal of this document similarity analysis was to rank documents, we remove the square root; it is a monotonic function.

We do not focus time on interpreting the results of this analysis relative to research domains such as text mining or journalism. Our interests are purely computational, focusing on scalability and reducing runtime. Before sharing the results of this analysis, we reflect on results reported in previous work [55]. Figure 2.11 provides a visual aid for the previous results.

In our previous work [55], we compare the classic Cartesian approach as implemented in Apache Spark 1.3.0 against the Cartesian Scheduler on a commodity Spark cluster consisting of nine nodes. Each node was equipped with 32 GB RAM and a 3.2 GHz Intel Xeon quad-core processor. We will refer to this

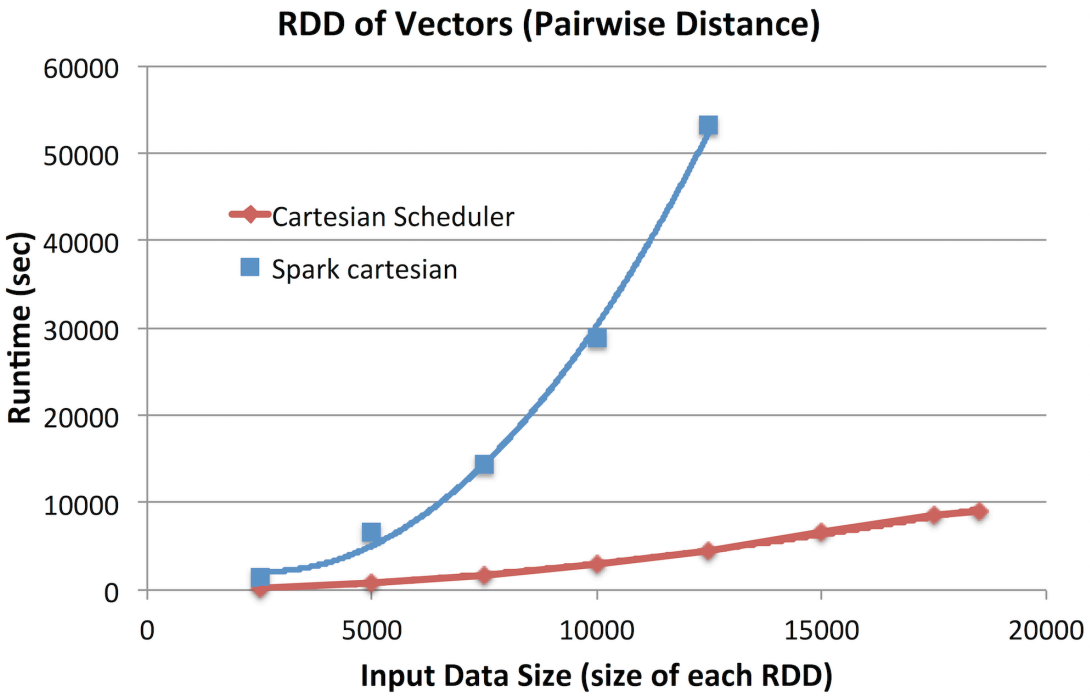

Figure 2.11: Runtimes collected from the document similarity experiments on Dataset 3 using Cluster 2.

hardware as Cluster 2. The Cartesian Scheduler consistently outperformed the native Cartesian method. In the example containing 5,000 articles, we outperform the native method by a factor of 6, and in the examples containing more than 12,500 documents, the native method failed to complete. Figure 2.12 characterizes the overall runtime differences between the Cartesian Scheduler and the classic approach when conducting these experiments. For the large datasets, the native method would crash after more than 20 hours of computation.

The results are consistent throughout this experiment. The Cartesian Scheduler performs the computation approximately twice as fast compared to the classic approach. This held true regardless of the size of the data. The greatest contrast in runtime occurred when processing the dataset of 10,000 articles; the Cartesian Scheduler required 1560 seconds (26 min) compared to the classic approach's 3575 seconds (60 min).

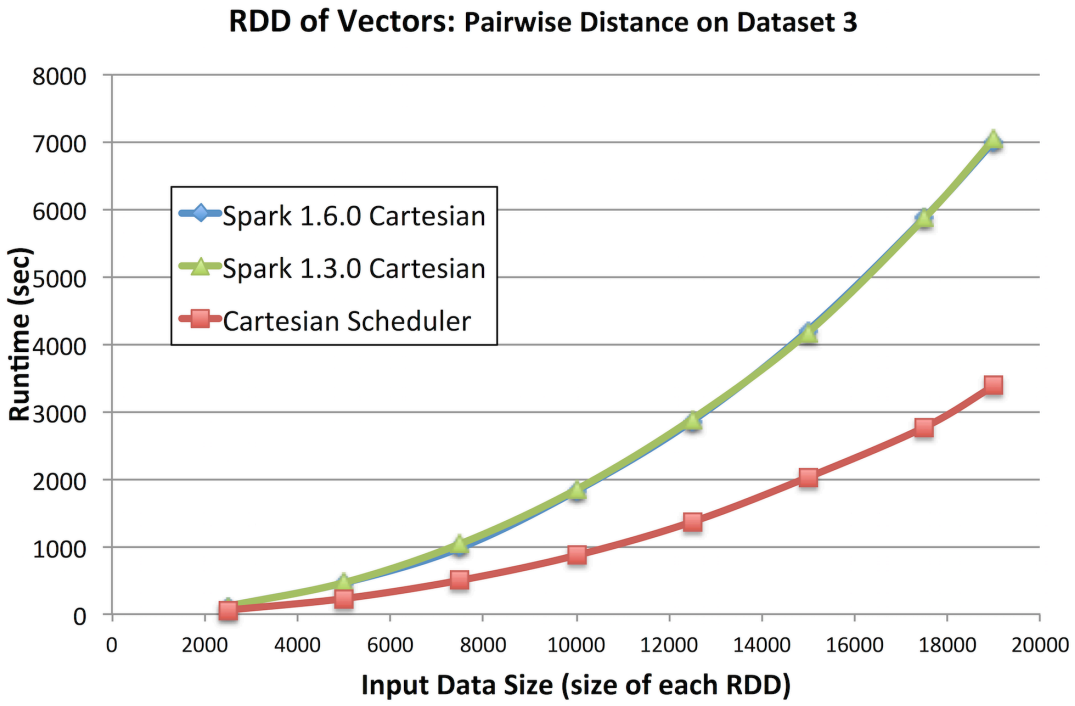

Figure 2.12: Runtimes collected from the document similarity experiments on Dataset 3 using Cluster 1.

We extend this experiment further and utilize a more powerful computing cluster running Apache Spark. Also, in addition to Dataset 3, we consider a larger Reuters dataset, Dataset 4. The maximum number of documents remains consistent at 19,000. However, we relax the constraint on stop words to increase the number of terms represented in each feature vector. The resulting feature vectors have a dimensionality of 16,244, roughly twice that of Dataset 3. This increases the complexity of each distance metric calculation that must be performed at each step of the Cartesian product. In addition to Spark version 1.3.0, we compare to Spark 1.6.0 on Cluster 1. The underlying cartesian implementation is consistent between version 1.3.0 and 1.6.0, so the performance is consistent between versions. We clarify the version to increase transparency in our experiments.

As shown in Figure 2.13, the difference in performance is consistent with that

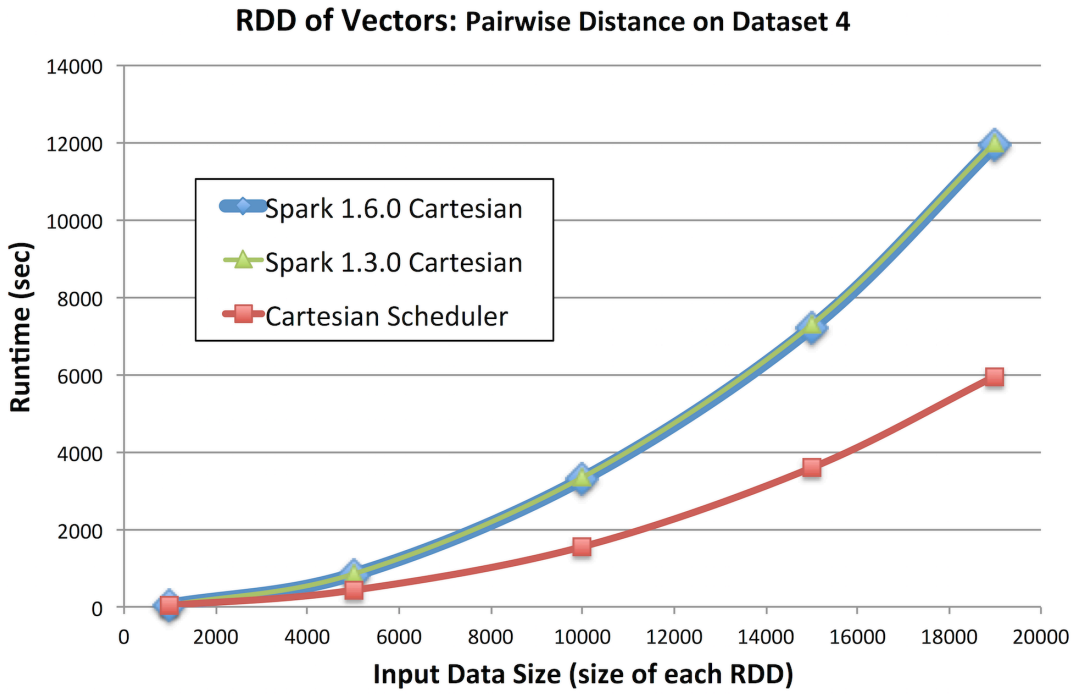

Figure 2.13: Runtimes collected from the document similarity experiments on Dataset 4 using Cluster 1. The Cartesian Scheduler consistently outperforms the built-in method. Also, the difference in performance of Spark 1.3.0 and 1.6.0 is negligible.

of the previous experiment shown in Figure 2.12. In addition, the performance difference between Spark 1.3.0 and 1.6.0 is negligible. This should not come as a surprise since the underlying Cartesian product execution plan remains unchanged. The amount of work required for each comparison is greater, which helps to highlight another advantage of the Cartesian Scheduler. The Cartesian Scheduler integrates the user-defined comparison operation into the Virtual Partition Pairing Protocol. That is, the comparison operation is performed immediately, whereas Spark's implementation of the classic approach simply accumulates all object pairs into a collection of tuples. The user-defined operation is then performed as a post-processing step on the collection of tuples. The Cartesian Scheduler consistently achieved at least a 2*x* speedup over the classic approach.

2.2.5 Potential Limitations

There are a few limitations of the Cartesian Scheduler. One is fundamental to all Cartesian operations, the underlying complexity. Since a Cartesian Product has complexity $O(n^2)$, any algorithm that uses it will be limited in scalability. Another potential limitation is imbalanced data. In this dissertation, we discussed the probability of this occurring approaches 0 as the data size increases; however, it is not impossible. There is also notable overhead associated with the Cartesian Scheduler. There is no redundant computation performed by the Cartesian Scheduler, we merely duplicate data across many machines. This is the tradeoff we accept to ensure that all computations, individual comparisons within the Cartesian product, are independent of one another, and that all of the network communication is performed as a preprocessing step to the sequence of partial Cartesian products. The magnitude of this overhead is influenced by the number of nodes in the cluster, virtual partitions, and sharding factor.

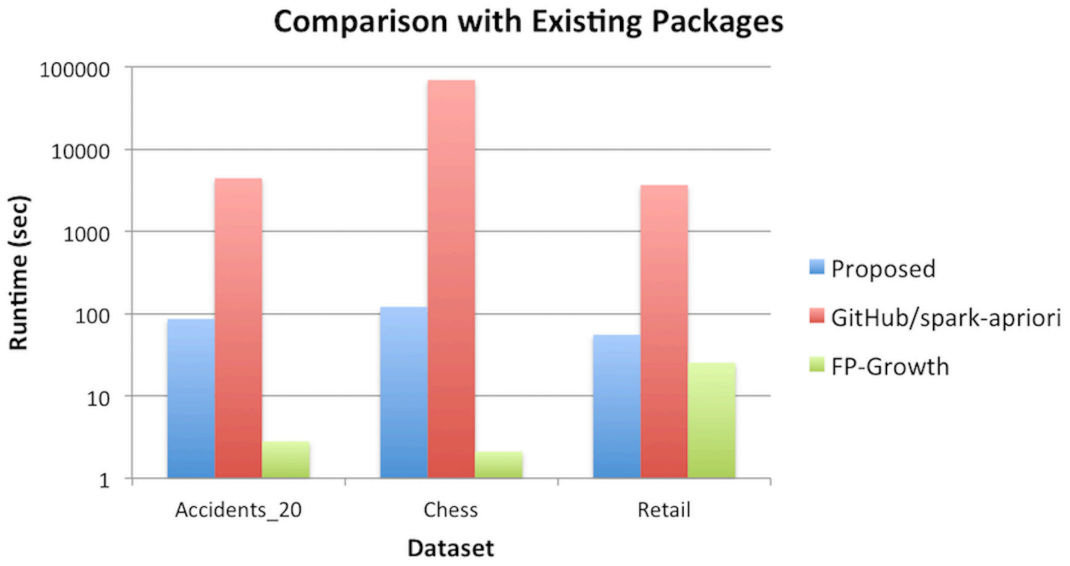

Figure 2.14: Runtimes collected from the frequent pattern mining experiments on using Cluster 2.

FPGrowth consistently outperforms both Apriori implementations on Accident_20 and chess. This is expected since both of those datasets are relatively dense datasets. However, the retail datasets is much more sparse, and we see the proposed Apriori Cartesian Scheduler approach is competetive with the FP-Growth approach.

There are a few limitations of the Cartesian Scheduler. One is fundamental to all Cartesian operations, the underlying complexity. Since a Cartesian Product has complexity $O(n^2)$, any algorithm that uses it will be limited in scalability. Another potential limitation is imbalanced data. In this dissertation, we discussed the probability of this occurring approaches 0 as the data size increases, however, it is not impossible. There is also notable overhead associated with the Cartesian Scheduler. It is important to clarify, there is no redundant computation performed by the Cartesian Scheduler, we merely duplicate data across many machines. This is the tradeoff we accept to ensure that all computations, individual comparisons within the Cartesian product, are independent of one another, and that all of the network communication is performed as a preprocessing step to the sequence of partial Cartesian products. The magnitude of this overhead is influenced by the number of nodes in the cluster, virtual partitions, and sharding factor.

2.3 Conclusion and Future Work

High-level goals for any distributed computing pipeline are to ensure a uniform distribution of the data and workload, maximize CPU utilization, and to mitigate continual network communication. These goals are also the limitations of the

classic Cartesian product (CP) approach. The philosophy we adopted is to precompute and execute all shuffle operations simultaneously as a preprocessing step, eliminating continual network communication and leaving the remaining time for uninterrupted computation. As a result, we must introduce redundant copies of data to ensure that every worker node has its own copy of the necessary data. However, data redundancy poses the challenge of preventing redundant comparisons in the CP. In this work, we propose virtual partitioning and the virtual partition pairing protocol to manage the degree of redundancy while guaranteeing no redundant computation is performed.

Virtual Partitioning is a variable grouping paradigm we proposed that gives control over the granularity of the partial CPs. A Virtual Partition (VP) functions as an irreducible building block for partial CPs, so redundant copies of VPs are created and copied to relevant compute nodes. Since partial CPs are performed between VPs, the size and number of partial CPs is managed by the VP size. This is valuable since the size of each partial CP affects how well the hardware can execute the instructions. The Virtual Partition Pairing Protocol preprocesses and schedules all of the partial CPs necessary to be equivalent to a global Cartesian product. This protocol facilitates the introduction of redundancy while guaranteeing no comparisons are redundant. By construction, the protocol prevents redundant comparisons, so additional filtering or duplicate checks are not necessary.

We have shown that the Cartesian Scheduler addresses several limitations and outperforms the classic approach by notable margins. Apache Spark does not provide a mechanism for introducing data redundancy without taking on

redundant computation. Preprocessing the shuffle operations and introducing data redundancy allowed us to increase throughput and reduce runtimes. In our experiments, we were able to achieve up to a 40x speedup when compared to Spark on a small commodity cluster. When the comparison is made on a high performance cluster, the advantage becomes less drastic, achieving a 2x speedup over the classic approach. In addition, the uniform distribution analysis demonstrated how well the Cartesian Scheduler handles heterogeneous data by achieving a balanced workload.

In addition to Cartesian operations, similar workloads that require massive amounts of data shuffling may benefit from this concept. Our experiments suggest preprocessing shuffle operations and introducing data redundancy can increase throughput and reduce runtimes. We intend to share this work with the community by making our open source software distribution available on GitLab [56]. In addition to our library's source code, we will distribute the code used to conduct all of the experiments detailed in this dissertation.

Future work includes automating the sharding factor selection process. This variable plays a great role in determining the overall runtime performance for the Cartesian Scheduler. Automating this selection would release a burden off of users. It would also be interesting to see the performance on a larger cluster, containing tens or hundreds of nodes. We are also seeking a generalized Cartesian Scheduler that supports n-fold Cartesian products, performing a Cartesian product between n collections of data. In addition, we will attempt to submit the Cartesian Scheduler for integration into a future release of Apache Spark.

Chapter 3

Frequent Hierarchical Pattern Mining

In this chapter, we propose a data structure, Frequent Hierarchical Pattern Tree (FHPTree), that does not suffer from the large candidate generation issue associated with the Apriori algorithm, and the number of nodes in the tree structure is linearly dependent on the number of unique items in the database. In addition, this data structure enables us to discover frequent patterns in a top-down fashion, locating maximal itemsets before any of their subsets. In contrast to existing top-down methods, the FHPTree allows a collection of items to be pruned simultaneously.

The FHPTree also serves as a persistent data structure that can be used as an index for frequent pattern databases, making targeted pattern mining and reoccurring data mining studies more efficient. Frequent pattern mining workflows tend to be an iterative discovery process. That is to say, the minimum support threshold and additional filtering criteria may be varied iteratively, and the pattern mining algorithm would execute repeatedly. It is advantageous to reuse the data structure. Search is also a critical component of the FHPTree. The proposed data structure offers an inclusive search features that, in general, eludes bottom-up approaches. This search technique discovers only those pat-

terns that contain a set of items of interest. The FHPTree supports *insert*,*update*, and *delete* operations, enabling the underlying transaction database to evolve. It may not be necessary to rebuild the entire data structure if a new item is introduced into the dataset or when new transactions are created. Apriori and the FPTree are not ideal for dynamic data and struggle with the inclusive search feature [32]. Thus, they may not be well suited to serve as or utilize persistent data structures.

A brief overview of the traditional terminology: Let $I = i_1, i_2, .., i_m$ be the set of all items. D is a database of transactions where each transaction $T \subset I$. T_i is the set of all transactions containing i. A collection of items $X \subset I$ is referred to as an itemset. X is contained in T if, and only if, $X \subset T$. An itemset is said to have support s if and only if $s\%$ of all transactions in D contain $R_a \cup Rc$. We say an itemset is frequent if and only if its corresponding support is greater than or equal to some user defined minimum support threshold, *min_support*. An itemset of size k is referred to as a k – *itemset* [57].

Enumerating all frequent pattern is NP-hard [58]. The worst-case scenario, a dataset containing k items has $O(2^k)$ subsets. Suppose all of them cooccur frequent. Regardless of which algorithm is selected for the problem, the resulting set will consist of $O(2^k)$ patterns. We propose a top-down frequent pattern mining paradigm that focuses on detecting maximal frequent patterns without enumerating non-maximal patterns. As shown in Figure 3.1, the proposed algorithm consists of two parts: FHPTree and FHPGrowth. In the FHPTree section, we define the tree structure utilized throughout this research. In Related Work, an overview of existing solutions and prior art is discussed. In FHPGrowth, we

discuss how to extract frequent patterns from the FHPTree. In Performance Evaluation, we detail the experiments used to benchmark the FHPTree and FH-PGrowth. We conclude the chapter and allude to future research opportunities in Conclusion and Future Work.

3.1 Related Work

3.1.1 Apriori-based approaches

Many optimizations and extensions have been proposed for the Apriori algorithm. The GSP algorithm is a generalization of Apriori to sequential pattern mining [59]. Algorithms that manage transaction sets, such as SETM, can benefit by utilizing diffsets to reduce the memory footprint [60]. Hashing techniques have also been applied to offer substantial improvements over Apriori on the candidate generation phase. The DHP Algorithm can achieve improvements orders of magnitude over Apriori in the size of intermediate candidate generation [15]. The BORDERS algorithm traverses and itemset lattice to identify a bounding region for frequent itemsets [18]. A-CLOSE is an Apriori-based algorithm designed to identify closed frequent itemsets [17]. CHARM focused specifically on identifying closed frequent itemsets was able to prune more intermediate itemsets, which resulted in an improved runtime [16]. Max-Miner was designed to focus on maximal itemsets [61]. A depth-first traversal on the enumeration tree was proposed as an advancement over Max-Miner [62]. MAFIA prunes large amounts of intermediate itemsets by narrowing the search-space [63]. CHARM-MFI is an post-processing technique for CHARM to identify closed and maximal patterns [64]

Methods have also be proposed to handle streaming data [65]. Probabilistic methods such as Fuzzy Apriori have been proposed and use fuzzy logic to determine frequent patterns [21]. High-utility pattern mining is gaining attention and has Apriori-based solutions [66]. Frequent pattern mining on uncertain datasets has a wide range of applications and has roots in the Apriori algorithm [67]. Apriori implementations have been ported to distributed computing environments [68].

3.1.2 Growth-based approaches

Similar to Apriori, many optimizations and extensions have been proposed for FPGrowth. PrefixSpan is a generalization of FPGrowth to sequential pattern mining [69]. The FPTree is not an ideal structure for handling dynamic transaction data. To address this issue a modified data structure called the CATs Tree was proposed [32]. The QFPGrowth algorithm was created to reduce the overhead associated with the FPTree and the number of conditional FPTrees generated [34]. Pfp, a parallel implementation of FPGrowth allows the work in creating and populating the FPTree to be distributed on multicore environments [35].

FPMax is an algorithm designed to focus on maximal frequent itemsets discovery [30]. CLOSET and an extension, CLOSET+, are Growth-based approaches used to identify closed itemsets [29]. TFP offer performance improvements and also extracts closed itemsets [70]. In recent years, high-utility pattern mining has become a popular data mining technique; UP-Growth [71] and TKU algorithm [72] are Growth-based solutions. Extracting frequent patterns from

steaming datasets has also been addressed by Growth-based approaches [73]. Another popular topic is data uncertainty [74]; tool kits have been developed because of its wide range of applications [75, 76]. Another popular topic is data uncertainty; tool kits have been developed to tackle this problem because of its wide range of applications [76].

3.1.3 Top-down approaches

Most of the algorithms discussed are bottom-up approaches; subset patterns are discovered before superset patterns. As the length of a pattern increases, bottom-up approaches begin to experience delays. The complexity to discover a single pattern is proportional to it's length. In many cases, such as Apriori, the relationship is 2^{length}.

Top-down approaches discover superset patterns before subset patterns. As a result, the complexity to discover a single pattern is inversely proportional to it's length. For example, Carpenter is a top-down mining approach from an item perspective; however, it builds the support by aggregating transactions in a bottom-up fashion [22]. The TD-Close algorithm utilizes the reverse enumeration tree to discover frequent patterns [77]. The approach begins with the set of all items and removes one item at a time until a frequent pattern is detected. Max-Clique is another top-down approach that focuses on maximal pattern detection and employs a probabilisitic strategy to improve performance [78]. The algorithm we propose in this chapter, FHPGrowth, is a frequent pattern mining paradigm similar to that of TD-Close. Both approaches begin the traversal with the set of all items, the reverse enumeration tree removes a single item at a time,

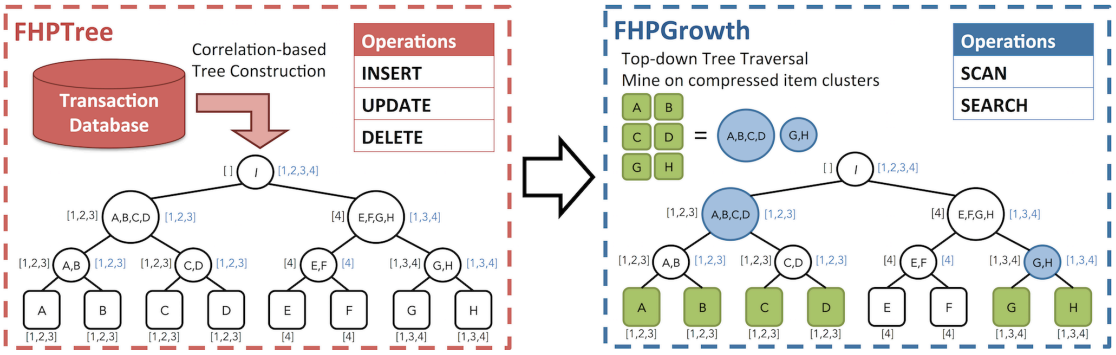

Figure 3.1: High-level overall architecture for the FHPTree and FHPGrowth. The first phase is constructing the FHPTree data structure. Next, frequent patterns are extracted using FHP-Growth.

while the FHPTree allows multiple items to be pruned simultaneously.

3.2 FHPTree: Frequent Hierarchical Pattern Tree

This section is organized as follows. First, we discuss the motivation for the the FHPTree. Next, we provide a formal definition for the proposed data structure. Then, in Tree Construction we discuss a strategy used to build an FHPTree. Insert, Update, and Delete provide details about their corresponding FHPTree operations.

3.2.1 Motivation

The conceptual overview for our proposed algorithm can be described as follows. Suppose we bifurcate I into two equal sets, I_1 and I_2. Consider I_1 and I_2 as items such that $T_{I_1} = \bigcup T_i$ for all $i \in I_1$ and $T_{I_2} = \bigcup T_i$ for all $i \in I_2$. T_{I_1} and T_{I_2} can be thought of as candidate transaction sets. We ask the question, is $\{I_1, I_2\}$ a frequent itemset based on this candidate transaction support? If the answer is no, then the itemset $\{i,j\}$ cannot possibly be frequent for any $i \subset I_1, j \subset I_2$, where i and j are nonempty. This conditional statement, $T_{I_1} \bigcap T_{I_2} >= min\_support$, can reduce our search space by $2^{|I_1|+|I_2|} - 2^{|I_1|} - 2^{|I_2|}$. For example, with 10 items

the search space is reduced from 1024 by $2^{10} - 2^5 - 2^5 = 960$; only 64 potential patterns remain. If the answer is yes, then $\{i,j\}$ may be a frequent itemset for any $i \subset I_1, j \subset I_2$, so we continue our search by bifurcating I_1 or I_2, potentially yielding I_1, I_3, and I_4. We then check if $\{I_1, I_3, I_4\}$ is a frequent itemset where $T_{I_\alpha} = \bigcup T_i$ for all $i \in I_\alpha$. If no, then $\{i,j,k\}$ cannot possibly be frequent for any $i \in I_1, j \in I_3$, and $k \in I_4$, reducing our search space by the amount defined in Equation 3.1.

$$\begin{aligned} & 2^{|I_1|+|I_2|+|I_3|} - (2^{|I_1|+|I_2|} - 2^{|I_1|} - 2^{|I_2|}) \\ & - (2^{|I_1|+|I_3|} - 2^{|I_1|} - 2^{|I_3|}) \\ & - (2^{|I_2|+|I_3|} - 2^{|I_2|} - 2^{|I_3|}) \\ & - 2^{|I_1|} - 2^{|I_2|} - 2^{|I_3|} \end{aligned} \tag{3.1}$$

This filtering technique can yield an exponential reduction in the search space. Equation 3.2 characterizes the potential reduction in search space. Let A be a set of sets of items, referred to as a *hierarchical pattern*, resulting from the recursive bifurcation of I. $P(A)$ is the power set of A.

$$z(A) = 2^{\sum_{a \in A} |a|} - \sum_{A' \in P(A) - A} z(A') \tag{3.2}$$

This recursive bifurcation process is also referred to as agglomerative clustering [79] and is a way to define the structure of an FHPTree. Notice that each step in the traversal attempts to increase the hierarchical pattern length by one. The FHPTree is used to aggressively grow frequent patterns while quickly pruning the search space.

A fundamental advantage of the FHPTree is its ability to represent multiple items as a single node in the tree. By evaluating combinations of FHPTree nodes, a *hierarchical pattern*, we effectively evaluate the support of many itemsets simultaneously, which enables multiple itemsets to be filtered at once. The FHP-Growth section of Figure 3.1 demonstrates how the FHPTree can be leveraged to find large itemsets by considering few nodes in the tree.

3.2.2 Definitions

One of the main contributions offered by the FHPTree, the structure of the tree is not dependent on the data distribution. The number of nodes in an FHP-Tree increases linearly relative to the number of unique items in the transaction database. However, the size of each node may increase as the number of transactions increases.

First, each node consists of u and T_u, the label of the node and a set, respectively. T_u can be a set, or a set of sets. Second, the FHPTree topology is bound by $T_u = f(T_C)$, where C is the set of children of u; child nodes determine the set stored in their parent node. From that perspective, it is natural to build an FHPTree from the bottom-up.

We let the frequent single items form the leaf nodes. cT_u is referred to as the item's candidate transaction list, and xT_u is the exact transaction set. This dissertation presents several traversal schemes, and some only make use of candidate transactions, while others leverage both candidate and exact transaction sets. We consider the exact transactions as an optional feature for the FHPTree.

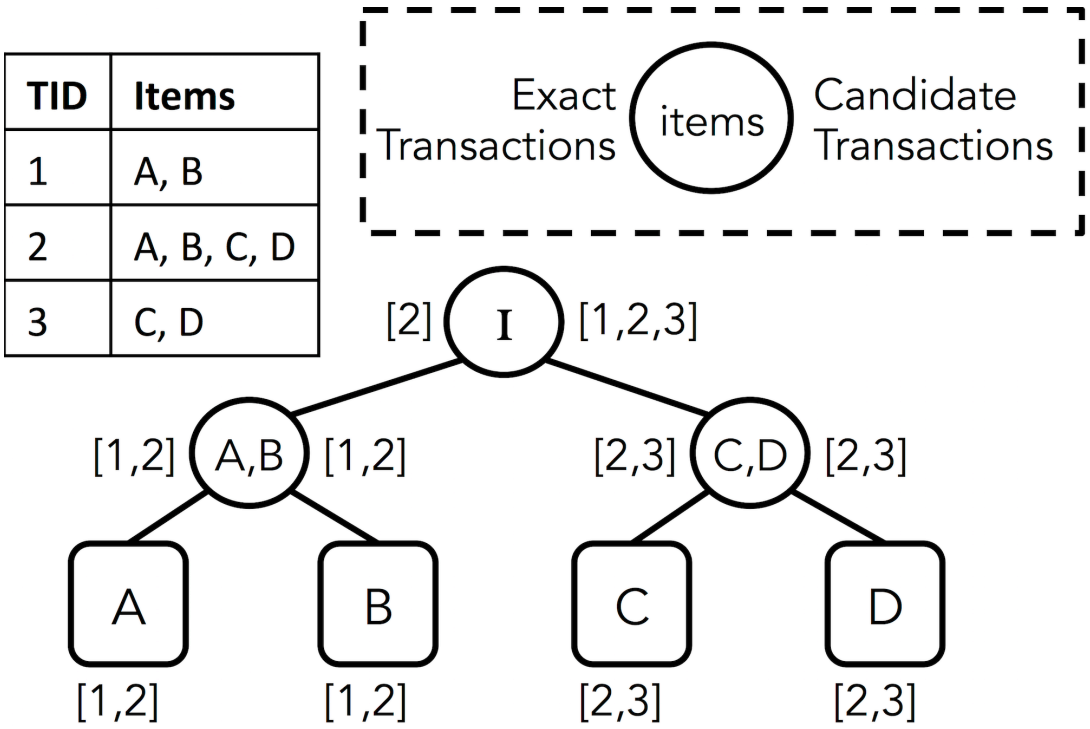

Figure 3.2: An example FHPTree produced for a simple transaction database. Each leaf node corresponds to an item. Each node also contains two transaction sets: The set to the left of a node is the exact transaction set and to the right is the candidate transaction set.

Definition 1 (FHPTree)**.** *A Frequent Hierarchical Pattern Tree (FHPTree) is a recursively defined tree structure having nodes of the form* (u, cT_u, xT_u)*, composed of a label, u, and two sets,* cT_u *and* xT_u*. The edge set is defined such that for every u,* $cT_u = \bigcup_{c \in C} cT_c$ *and* $xT_u = \bigcap_{c \in C} xT_c$ *where C is the set of children of u.*

There are a few points to make using Definition 1. The set union and set intersection operations constrain the candidate and exact transaction sets, respectively. As a result, each node or collection of nodes will have a candidate frequency and an exact frequency. If a collection of nodes is said to be candidate frequent, that implies the candidate frequency exceeds the minimum support threshold. If nodes are referred to as frequent, that references the exact support. Candidate frequency is always greater than or equal to the exact frequency, so

if a collection of nodes is not candidate frequent, it cannot be frequent. Also, $xT_u = cT_u$ for all leaf nodes, so it is only necessary to store one copy. Further detail regarding the candidate and exact transaction sets and their distinct usage is provided in the scan portion of the FHPGrowth section. Figure 3.2 provides an example FHPTree.

Another key point to notice is the similarity between this recursive bifurcation strategy discussed previously and the FHPTree. From a top-down perspective, the FHPTree is defined such that nodes are bifurcated to form children. This is what allows the FHPTree to aggressively grow frequent patterns while quickly filtering the search space. Mining efficiency using an FHPTree is possible because of Theorem 2. Let U be the set of all FHPTree nodes.

Theorem 2 (FHPTree Property)**.** *If a hierarchical pattern $V \subset U$ is candidate frequent, then for any $A = \{ancestor(V_1), ancestor(V_2), \ldots, ancestor(V_k)\}$ where $k = |V|$, must also be candidate frequent.*

Proof. Let $V \subset U$ be a candidate frequent hierarchical pattern and A_i be an ancestor of V_i. Then, $cT_{V_i} \subset cT_{A_i}$ for all $i \in [1, |V|]$, by the FHPTree definition. Since V is candidate frequent, $|\bigcap_{v \in V} cT_v| >= min\_support\_count$. Thus, $|\bigcap_{a \in A} cT_a| >= min\_support\_count$, and therefore, A is candidate frequent. □

We rely on the contrapositive of this theorem as a filtering criteria. Corollary 1 is a trivial result of Theorem 2 and is defined as follows.

Corollary 1 (Subtree Filtering)**.** *If $V \subset U$ is not candidate frequent, then none of it's descendants are candidate frequent.*

To clarify the meaning of descendant, a descendant of $V \subset U$ is a set $C \subset U$

such that C_i is a member of the subtree rooted at A_i for every $i \in [1, |V|]$. The FP-Growth section of Figure 3.1 demonstrates an example of a descendant-ancestor relationship. The highlighted leaf nodes are descendants of the highlighted non-leaf nodes. Corollary 1 allows us to prune the search space during the mining process. If a hierarchical pattern is not candidate frequent, there is no need to consider any of its descendants.

3.2.3 FHPTree Construction

Before we begin building data structures or mining data, we set the user defined minimum support threshold, *min_support*. Next, we scan the database, D, and calculate the support for each unique item, i. We discard any i where $support(i) < min\_support$. It is important to note that this *min_support* threshold will serve as a lower bound for any frequent pattern mining analysis. That is to say, this tree can be mined for frequent patterns at any higher support threshold. Let F be the set of all frequent items, and let the frequent items in F form the leaf nodes of an FHPTree.

The pseudocode for this generalized construction procedure is provided in Algorithm 7. In this example and throughout the remainder of this dissertation, we focus on binary FHPTrees. To form the next layer in the tree, we arrange all of the nodes formed by F into $\left\lceil \frac{|F|}{2} \right\rceil$ non-overlapping pairs. Each pair (i,j) is merged to form a node u, calculating the new candidate transactions union of their sets, $cT_u = cT_i \cup cT_j$. This merge, also forms two edges connecting i and u as well as j and u. In the case where there are an odd number of nodes, we cannot form a set of non-overlapping pairs that covers all nodes at a given level

of the tree, so we simply move the single odd node up to the next level of the FHPTree. This process is executed recursively until one node remains. Let U be the set of all nodes in the FHPTree.

Algorithm 7 FHPTree Construction

```
1: while current layer of tree L_i, |L_i| > 1 do
2:     for a : from 0 until |L_i| by 2 do
3:         Merge nodes L_i(a) and L_i(a + 1) and add to L_{i+1}
4:     end for
5:     i++
6: end while
```

There are a few points to emphasize. The construction process requires only 1 database scan. The number of nodes in the tree scales linearly as the number of distinct frequent items in the transaction database; given n frequent items, there are $(2n - 1)$ nodes in the FHPTree. The *min_support* threshold defined when building the tree sets an absolute minimum support threshold for mining frequent patterns. That is to say, we are able to mining for frequent patterns having any *min_support* value greater than this absolute minimum support defined during construction.

Correlation-based FHPTree

What does an ineffective FHPTree look like? Let n be a node that has two children x and y with transaction sets that differ significantly. In the worst case, suppose $cT_x \bigcup cT_y = T$ and $cT_x \bigcap cT_y$ is nonempty. Then, $cT_n = T$ and the candidate support of n is 100%, while the support is 0%. As a result, every itemset containing n or an ancestor of n is guaranteed to be frequent, however the likelihood of x and y being part of a frequent itemset is nil. This is the worst case scenario, as all comparisons involving n are wasted computation. An FHPTree

should be constructed to ensure that sibling nodes, such as x and y, are similar in terms of their transaction sets.

To accomplish this, we utilize an agglomerative or hierarchical clustering technique [79]. Similar to the classic hierarchical clustering techniques, we build the FHPTree from the bottom up. When constructing the next level in the tree, our goal is to identify pair clusters such that each cluster consists of two nodes and a node cannot belong to more than one cluster. Each cluster forms a node in the next level of the FHPTree. To clarify, for node n with children x and y, n was formed by the pair cluster containing x and y.

Algorithm 8 FHPTree Correlation-based Construction

```
1: while current layer of tree L_i, |L_i| > 1 do
2:     for a : node in L_i do
3:         Find most similar b ∈ L_i, a ≠ b
4:         Merge nodes a and b
5:         Remove a and b from L_i
6:     end for
7:     i++
8: end while
```

In practice we utilize the Jaccaard Index as a similarity measure between transaction sets, as shown in Equation 3.3. Since we have both candidate and exact transaction sets, we utilize a linear combination of both Jaccaard indices.

$$\begin{aligned} JaccaardDistance(T_x, T_y) &= 1 - \frac{|(T_x \cap T_y)|}{|(T_x \cup T_y)|} \\ Similarity(x, y) &= c_1 * J(cT_x, cT_y) \\ &\quad + c_2 * J(xT_x, xT_y) \end{aligned} \tag{3.3}$$

Other similarity measures were considered such as intersection cardinality, Euclidean distance, Hamming distance, and Sorensen-Dice coefficient as well. Al-

though other measures may potentially achieve better FHPTrees, the Jaccaard index proved most useful thus far, improving scan and search performance. Correlation-based construction is essential for the FHPTree to be useful in any case. Even though we discussed binary FHPTrees in this example, other schemes can be applied to define the connectivity of the FHPTree.

FHPForest

We have shown that connecting similar nodes is advantageous. In this section, we extend that concept a step further. Nodes that never cooccur should not be apart of the same tree. That is to say, non-overlapping clusters of items that never cooccur, should not be a part of the same tree. As a result, the tree construction process may yield an FHPForest, a collection of FHPTrees.

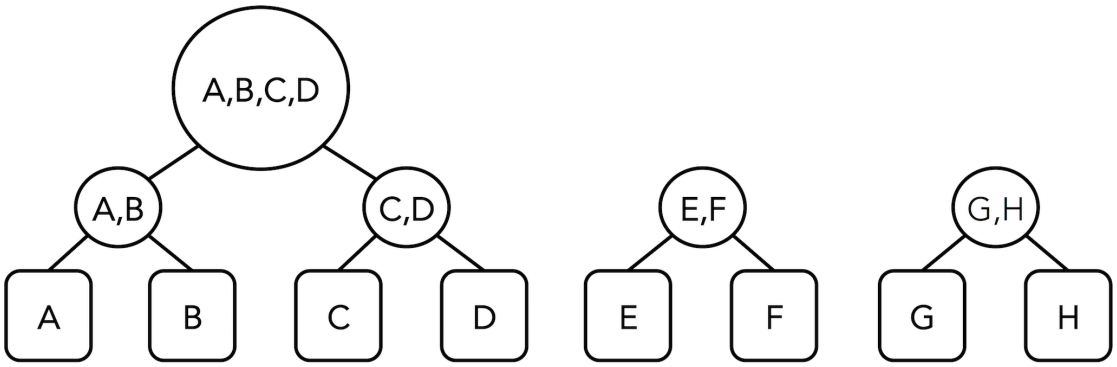

Figure 3.3: An example FHPForest where x and y do not cooccur in any transaction for $x \in \{A, B, C, D\}$ and $y \in \{E, F\}$.

Figure 3.3 provides an example FHPForest. Sparse datasets and datasets that are composed of multiple subpopulations will benefit from this technique as unnecessary computation is reduced during the frequent pattern extraction process.

3.2.4 Insert

We consider the insert operation as the addition of a new node to an FHPTree. A new node is created when a novel item is detected in the transaction database, or the minimum support threshold used to build the tree is reduced. Let $u \in I$ be a new item. To integrate u into the FHPTree, we define a new node, (u, T_u). The primary goal is to preserve the existing connections while pairing u with the most mutually-similar node. Figure 3.4 provides a visual representation of an insert operation performed on an FHPTree.

The pseudocode for the insert operation is provided in Algorithm 9. To ensure the new node u is paired with its most mutually-similar node, we perform a top-down insertion. We traverse the tree beginning at the root, and evaluate the correlation between it's children, C. Let $C' = \{c \in C | Distance(u, c) < Distance(c, C - c)\}$. If C' is not empty, we perform a recursive call on the $c \in C'$ with the minimum *Distance*. Otherwise, if C' is empty, we pair u with the last node visited, *node*. The details of the *Pair* function are illustrated through color in Figure 3.4, however, are omitted from Algorithm 9 for readability. Similarly, *UpdateAncestorTransactions* is not detailed, but the logic consists of a traversing the path from *node* to the root node, updating the transaction sets along the way.

TID	Items
1	A, B
2	A, B, C, D
3	C, D
4	A, B, E

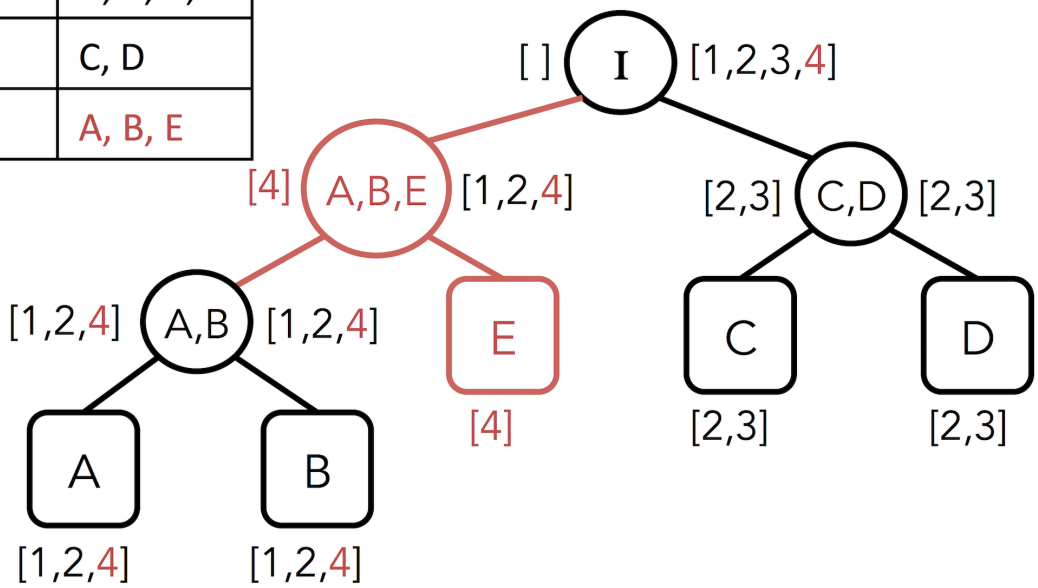

Figure 3.4: A new node, *E*, is inserted into an FHPTree. The highlighted nodes, edges, and transaction IDs are created or modified as part of the operation.

Algorithm 9 FHPTree Insert Operation

```
1:  newNode is required
2:  function checkPair(node)
3:      C = node.children, C′ = []
4:      for c ∈ C do
5:          if Dist(newNode, c) < Dist(c, C − c) then
6:              C′+ = c
7:          end if
8:      end for
9:      if C′ is not empty then
10:         minNode = c ∈ C′ with minimum distance
11:         checkPair(minNode)
12:     else
13:         Pair(node)
14:         UpdateAncestorTransactions(node)
15:     end if
16: end function
```

An alternative solution for building an FHPTree may be sequentially inserting items. It is important to note that this insert operation is greedy, and the in-

sertion order influences the quality of the tree. That is to say, under this scheme, it is not guaranteed that the insert operation will yield the best tree. As a result, after a series of insert operations, it may be advantageous to rebuild the tree.

3.2.5 Update

We consider the update operation as the modification of an existing node. This operation does not modify the underlying structure of the FHPTree. Situations that would utilize an update operation include the addition of a new transaction containing existing items, the removal of a transaction, or changing the label of a node. Changing the label is trivial. Updating the transactions sets required more effort and is a bottom-up approach. When adding or deleting a transaction, each item in the transaction is a leaf node in the FHPTree; each of those nodes will be updated by modifying the transaction set accordingly. Next all of the ancestors of the modified nodes must be updated as well.

This can be an expensive operation depending on the size of the transaction added or removed. The requirement of updating the ancestors introduces overhead. In the worst case, all nodes will be modified. With this in mind, it is important to note that multiple update operations may be performed simultaneously. Updates should be buffered and executed at once to reduce the impact of ancestor overhead.

There is also a case where new transactions can boost the support of an item such that it becomes frequent. This newly frequent item is not currently in the FHPTree, so it needs to be inserted. We store a hash map to track infrequent items and their corresponding transaction sets. If a new transaction causes an

item to surpass the *min_support* threshold, we call the insert operation.

3.2.6 Delete

The delete operation is the inverse of the insert operation. That is to say, the removal of a node from an FHPTree. Figure 3.5 provides a visual representation of the delete operation.

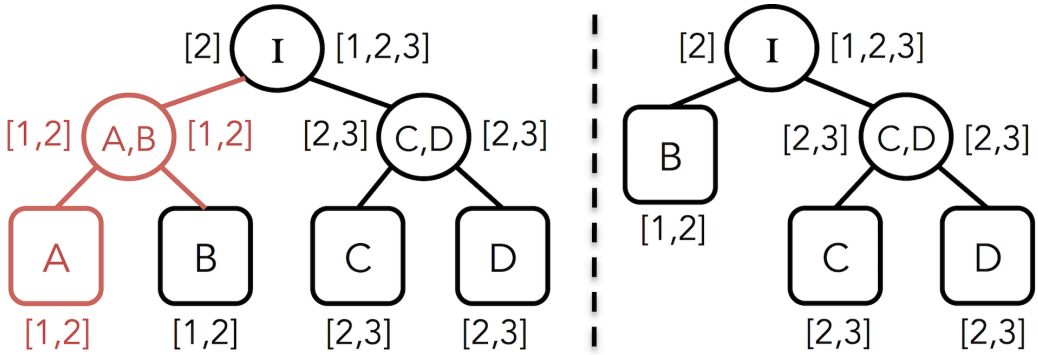

Figure 3.5: A item, *A*, is deleted from an FHPTree. The highlighted nodes, edges, and transaction sets are removed as part of the operation.

This is a bottom-up approach. If the node has multiple siblings, we simply delete the node. If the node has a single sibling, we delete the node and parent. The sibling node assumes the role of its former parent. The pseudocode for the delete operation is given in Algorithm 10.

Algorithm 10 FHPTree Delete Operation

```
1: function removeNode(node)
2:     if node.siblings.size == 1 then
3:         grandParent = node.parent.parent
4:         node.sibling.parent = grandparent
5:         delete(node.parent)
6:     end if
7:     UpdateAncestorTransactions(node.sibling)
8:     delete(node)
9: end function
```

Similar to the insert operation, deleting nodes also affects the transaction sets of their surviving ancestors. As a result, we must update the transaction

sets for those surviving ancestors using the logic defined in Tree Construction.

3.3 FHPGrowth: Frequent Hierarchical Pattern Growth

The structure of the FHPTree scales linearly relative to the number of distinct items, it serves as a persistent data structure, and supports insert, update, and delete operations. FHPGrowth is a general term used to denote the process of frequent pattern mining using an FHPTree. The algorithms described in this section, Scan and Search, are instances of FHPGrowth. We will start by providing a base traversal scheme and incrementally apply two optimization techniques. Finally, we walk through an example of the optimized approach.

3.3.1 Scan

Scan is the familiar task of extracting all frequent patterns. The traversal paradigm we employ is a recursive depth-first strategy where each state of the traversal consists of a collection of nodes $A \subset U$ that are currently *visited*. With this in mind, we define the traversal as a transition between states and define each state as $A_i \subset U$, the set of nodes visited during the i^{th} step of the traversal.

Next, we define how transitions are performed. That is to say, we define a mapping g such that $g(A_i) = A_{i+1}$. However, each transition on a FHPTree traversal yields three states $\{A_{i+1}, A_j, A_k\}$, so we write the mapping as $g(A_i) = \{A_{i+1}, A_j, A_k\}$. A_{i+1} is referred to as a descendant state of A_i. Furthermore, any descendant state of A_{i+1} is also a descendant of A_i. The mechanism to determine these states selects a non-leaf node $\alpha \in A_i$ and analyzing its children. Equa-

Algorithm 11 FHPGrowth: Scan 1

```
1: function fhpg(A: Array of nodes in U)
2:     cT = ⋂_{a∈A} cT_a
3:     if cT.size < min_support then
4:         return
5:     end if
6:     leaves = {a | a ∈ A & a.children.size = 0}
7:     nonLeaves = A − leaves
8:     if nonLeaves.size > 0 then
9:         splitNode = nonLeaves.head
10:        rNodes = (nonLeaves − splitNode) + leaves
11:        fhpg(rNodes + splitNode.children)
12:        fhpg(rNodes + splitNode.rightChild)
13:        fhpg(rNodes + splitNode.leftChild)
14:    else
15:        save(A.itemset)
16:    end if
17: end function
```

tion 3.4 defines the transitioning mechanism:

$$\begin{aligned} A_{i+1} &= (A_i - \alpha) \cup children(\alpha) \\ A_j &= (A_i - \alpha) \cup leftChild(\alpha) \\ A_k &= (A_i - \alpha) \cup rightChild(\alpha) \end{aligned} \tag{3.4}$$

A_j and A_k are not visited immediately. They are visited during a backtracking phase. The order in which these states are visited determines whether frequent patterns are discovered in a top-down fashion or bottom-up. By visiting the state containing the longer hierarchical pattern, A_{i+1}, first, we guarantee maximal itemsets are discovered before their subsets.

Before considering a transition to state A_{i+1}, we check the support of A_i, which is defined as $|\bigcap_{v \in V} cT_v|$. If it meets the *min_support* threshold and A_i contains at least one non-leaf node, we continue on to state A_{i+1}. If all $\alpha \in A_i$ are leaf-nodes or A_i does not meet the *min_support* threshold, then the descendants of A_i are

pruned, and the traversal begins backtracking. In all cases, the traversal begins at the initial state, $A_0 = \{root\}$, and continues until all states have been visited or pruned.

Maximal Frequent Itemset Detection

Since all non-maximal frequent itemsets are subsets of and therefore a direct implication of some maximal frequent itemset, non-maximal itemsets can been viewed as redundant and unnecessary information. The FHPGrowth strategy we demonstrated enables us to identify maximal itemsets before their subsets. Now, the task is preventing the discovery of non-maximal frequent itemsets. Before a branch of the FHPTree is traversed, we check if the branch has already been "covered." The follow definitions describe the relationship between a pattern and a cover.

Definition 2 (Pattern Cover)**.** *Let $V \subset U$ be a frequent pattern. $C \subset U$ is a cover of V if and only if for every $v \in V$, there exists a $c \in C$ such that v is a descendant of c.*

The trivial pattern cover in an FHPTree is the root node. Since every node is a descendant of the root, any combination of those nodes will also be a descendant of the root.

Definition 3 (Perfect Pattern Cover)**.** *Let $V \subset U$ be a frequent pattern, and $C \subset U$ be a cover of V. C is a perfect cover if and only if for every $c \in C$, there exists a $v \in V$ such that c is an ancestor of v and for every $c_1, c_2 \in C, c_1$ is not a descendent of c_2.*

A perfect pattern cover has an added constraint that the cover cannot contain extraneous items. This reduces the number of covers; the trivial cover is only a perfect cover for the pattern containing all items.

Definition 4 (Maximal Perfect Pattern Cover)**.** *Let $V \subset U$ be a frequent pattern, and $C \subset U$ be a perfect cover of V. C is maximal if and only if there exists no perfect cover of V, $S \subset U$, such that $S \neq C$ and S is a perfect cover of C.*

Each pattern has a unique maximal perfect pattern cover (MPPC). In addition, each MPPC has a unique frequent pattern. In order to prevent unnecessary traversal steps in FHPGrowth, we must store the MPPC.

Algorithm 12 FHPGrowth: Scan 2

```
11:     fhpg(rNodes + splitNode.children)
12:     if splitNode.rightChild not covered then
13:         fhpg(rNodes + splitNode.rightChild)
14:     end if
15:     if splitNode.leftChild not covered then
16:         fhpg(rNodes + splitNode.leftChild)
17:     end if
18: else
19:     saveMPPC(A.itemset)
20:     save(A.itemset)
21: end if
```

Algorithm 12 is an extension of Algorithm 11. On line 19 of Algorithm 12, in addition to saving frequent itemsets, MPPCs for said patterns are also saved. As shown on lines 12 and 15, the next state in the FHPGrowth traversal cannot be covered to proceed. It is only necessary to check when traversing to individual children; both children cannot possibly be covered or the previous state would have been covered.

Frequent MPPC Mining

There is a one-to-one mapping from maximal frequent patterns to MPPC. As a result, patterns may be considered functionally equivalent to their respective MPPCs. We can simply mine for MPPCs and reduce the traversal depth required during FHPGrowth. In addition to utilizing *cT*, the candidate transactions, at each node, this requires utilizing the exact transactions, *xT*. Using *xT*, for any collection of nodes in an FHPTree, we know the exact support for frequent pattern it represents.

Algorithm 13 FHPGrowth: Scan 3

```
2:  cT = ⋂_{a∈A} cT_a, xT = ⋂_{a∈A} xT_a
3:  if cT.size < min_support then
4:      return
5:  end if
6:  if xT.size >= min_support then
7:      saveMPPC(A.itemset)
8:      save(A.itemset)
9:      return
10: end if
11: leaves = {a | a ∈ A & a.children.size = 0}
```

Algorithm 13 is an extension of Algorithms 11 and 12. By adding lines 5-9 in Algorithm 13, we are able to identify maximal frequent itemsets using MPPCs. Referring back to the FHPGrowth section of Figure 3.1, the MPPC is represented by the two highlighted non-leaf nodes. The corresponding maximal frequent itemset is represented by the six highlighted leaf nodes.

Example

We provide examples to demonstrate how frequent patterns can be discovered using the FHPTree. Consider the transaction dataset defined below.

TID	Items
1	A B C D G H
2	A B C D
3	A B C D G H
4	E F G H

Suppose we build the tree with a *min_support* threshold of 25%. Since there are four transactions in total, this implies that all items are frequent, and $F = \{A, B, C, D, E, F, G, H\}$. We build the FHPTree from the bottom up where F is the set of leaf nodes. Each node contains a candidate transaction set and an exact transaction set. Figure 3.6 provides a visual of the resulting FHPTree.

We will mine the FHPTree with a *min_support* threshold of 50%, so a pattern must occur in two transactions to be frequent. Now, to identify frequent itemsets, we begin the traversal following the pseudocode in Algorithm 13. This procedure is detailed visually in Figure 3.7.

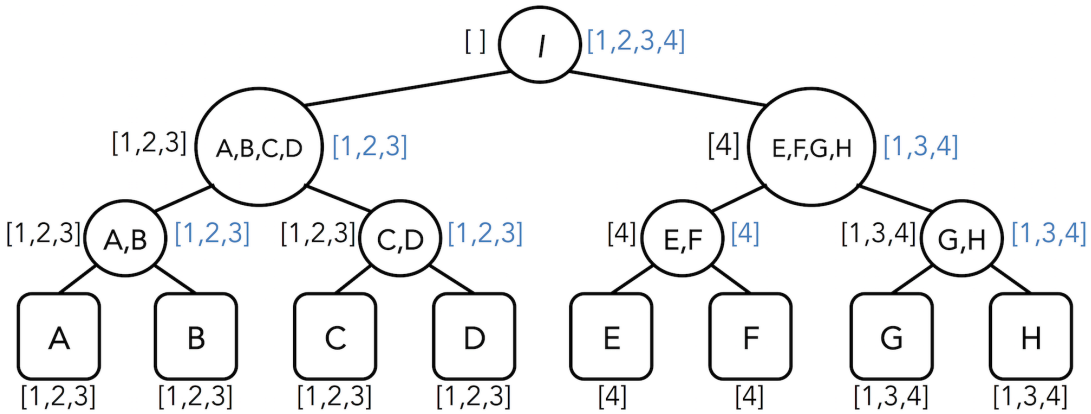

Figure 3.6: Leaf nodes correspond to an item from the transaction database. Each non-leaf node contains two sets; to the left of the node is the exact transaction set and to the right is the candidate transaction set.

The initial state of the traversal (a) is $A_0 = I$. First, we consider the candidate support, which exceeds the *min_support* threshold, so the exact support is

considered. The exact transaction set for the root node is empty, which implies that no transactions contain all items in F. Since the current state is candidate frequent, the traversal will continue by splitting a node. Using the traversal mapping, g, to determine the next state in the traversal. $g(A_0) = \{A_1, A_j, A_k\}$. Equation 3.5 provides more details about the next descendant states.

$$\begin{aligned} A_1 &= \{ABCD, EFGH\} \\ A_{j,1} &= \{ABCD\} \\ A_{k,1} &= \{EFGH\} \end{aligned} \tag{3.5}$$

Before proceeding, we check that the next state has not been covered. No patterns have been detected yet, so it cannot be covered. Recall that A_1 is the next state and $A_{j,1}$ and $A_{k,1}$ will be visited during backtracking. The next step is to analyze the candidate support of A_1. The candidate support of A_1 is $|\{1, 2, 3\} \cap \{1, 3, 4\}| = 2$, which exceeds the *min_support* threshold. Check the exact support of A_1, $|\{1, 2, 3\} \cap \{4\}| = 0$; we have not discovered a frequent pattern. A_1 contains non-leaf nodes, so the traversal continues, $g(A_1) = \{A_2, A_{j,2}, A_{k,2}\}$. More detail about the next descendant states is given in Equation 3.6.

$$\begin{aligned} A_2 &= \{ABCD, EF, GH\} \\ A_{j,2} &= \{ABCD, EF\} \\ A_{k,2} &= \{ABCD, GH\} \end{aligned} \tag{3.6}$$

The candidate support of A_2 is $|\{1, 2, 3\} \cap \{4\} \cap \{1, 3, 4\}| = 0$ fails to meet the *min_support* threshold, so we halt the traversal and consider state $A_{k,2}$ next.

$A_{k,2}$ hasn't been covered yet, and the candidate support is $|\{1, 2, 3\} \cap \{1, 3, 4\}| =$

2, passing the *min_support* threshold. The exact support of $A_{k,2}$ is $|\{1, 2, 3\} \cap \{1, 3, 4\}| = 2$, meeting the *min_support* threshold and implying that we have discovered a frequent pattern. We save the itemset $\{A, B, C, D, G, H\}$ and the MPPC $\{ABCD, GH\}$. Next, the algorithm would continue to state $A_{j,2}$ where it would fail the *min_support* checks and backtracking would begin. This process is continued until all states are visited and all maximal patterns have been detected.

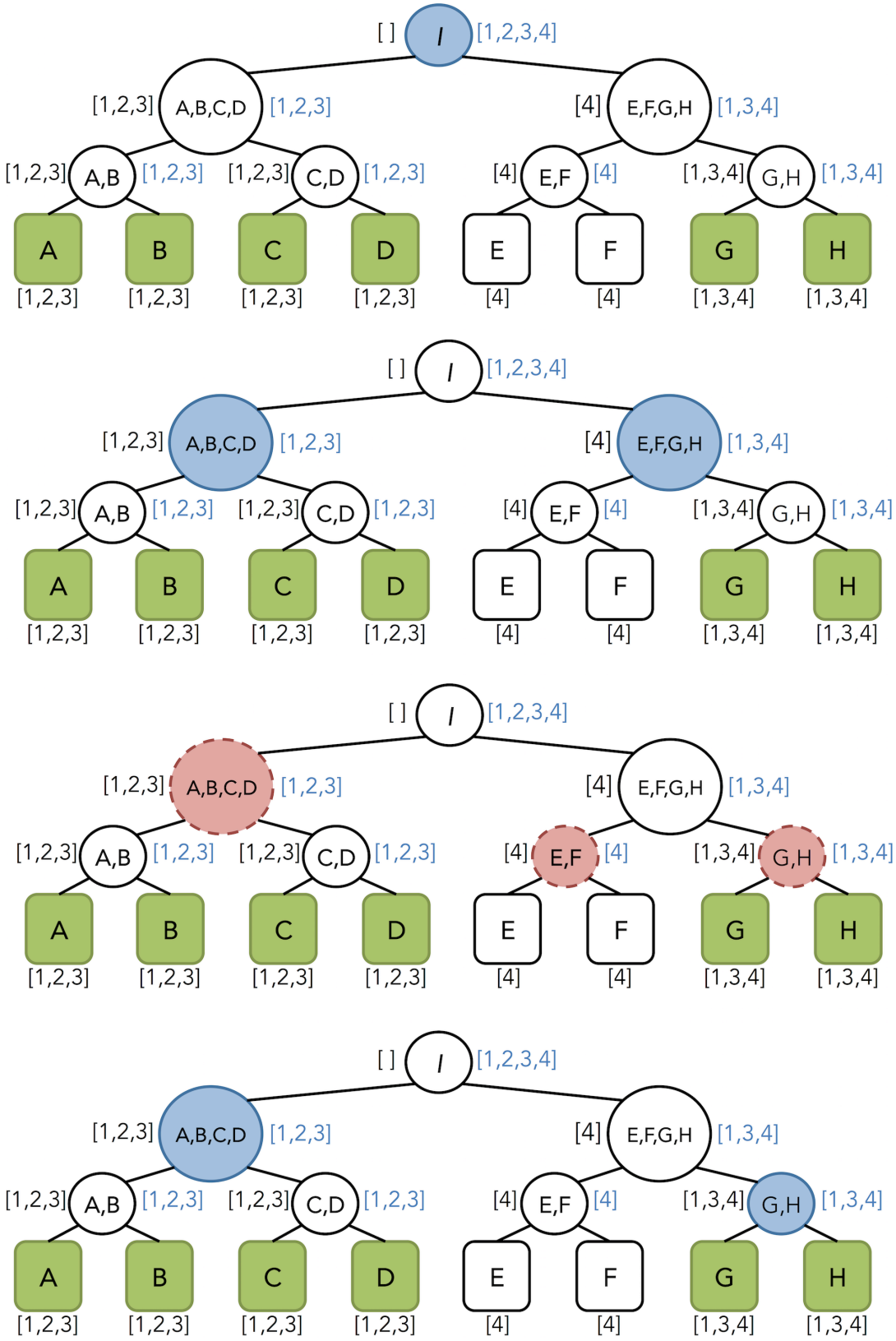

Figure 3.7: An FHPGrowth traversal to detect the frequent pattern, $\{A, B, C, D, G, H\}$

3.3.2 Search

The search operation takes a query item or items as input. The result is the collection of all frequent patterns containing the query item or items. Bottom-up approaches struggle with this feature and may perform it as a post processing technique. From a top-down perspective, if the current state does not contain our query item, then its subsets, descendant states, do not either. In contrast, descendant states in bottom-up approaches contain new items and may not be capable of this sort of on-the-fly inclusive filter.

Our mission is to prevent the enumeration of all frequent patterns as an intermediate step and provide a targeted mining feature. Fortunately, the logic for this approach is the quite similar as that of the scan operation. The fundamental difference involves restricting the traversal of FHPGrowth.

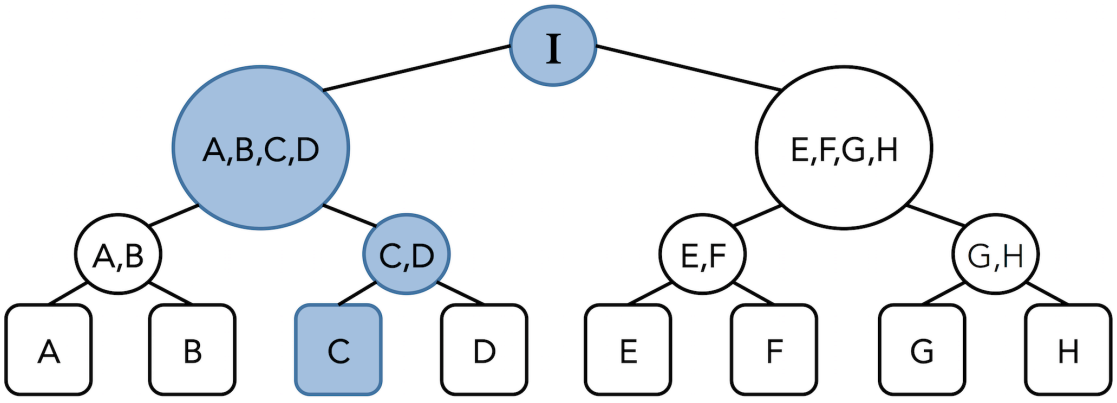

Figure 3.8: When searching for patterns involving item *c*, at least one of the highlighted nodes must be present in each state of FHPGrowth.

Figure 3.8 highlights the ancestors of a query item, *c*. All states of the FHP-Growth traversal must include at least one of the ancestors of *c*. This enables us to detect all patterns containing the query items, while ignoring unrelated items.

The pseudocode for Search is provided in Algorithm 14, and is a extension

Algorithm 14 FHPGrowth: Search

```
1: query = item(s)
2: ancestors = ancestors of query
3: function fhpg(A: Array of nodes in U)
4:     if A ∩ ancestors.size == 0 then
5:         return
6:     end if
7:     cT = ⋂_{a∈A} cT_a, xT = ⋂_{a∈A} xT_a
8:     if cT.size < min_support then
9:         return
10:    end if
11:    if xT.size >= min_support then
12:        saveMPPC(A.itemset)
13:        save(A.itemset)
14:        return
15:    end if
16:    leaves = {a | a ∈ A & a.children.size = 0}
17:    nonLeaves = A − leaves
18:    if nonLeaves.size > 0 then
19:        splitNode = nonLeaves.head
20:        rNodes = (nonLeaves − splitNode) + leaves
21:        fhpg(rNodes + splitNode.children)
22:        if splitNode.rightChild not covered then
23:            fhpg(rNodes + splitNode.rightChild)
24:        end if
25:        if splitNode.leftChild not covered then
26:            fhpg(rNodes + splitNode.leftChild)
27:        end if
28:    else
29:        saveMPPC(A.itemset)
30:        save(A.itemset)
31:    end if
32: end function
```

of the logic defined in the Scan section. This pseudocode provides the complete pseudocode for the Scan operation with the addition of two if conditions. The if-conditions require that at each state of the traversal the nodes cover the items provided in the query.

3.4 Performance Evaluation

We conducted a series of experiments to evaluate the performance of FHPTrees and FHPGrowth. Evaluations are based on runtime and scalability. The Tree Construction, Insert, Delete, Scan, and Search methods described in this chapter are considered in the following experiments. For Scan, existing frequent pattern mining algorithms, CHARM-MFI and FPMax, are considered as baselines for performance. The algorithms we compare to are distributed in the Java Open-Source Pattern Mining Library (SPMF), a large suite of frequent pattern mining and sequential pattern mining algorithms [80]. Currently, the most efficient maximal pattern mining algorithms distributed in SPMF are CHARM-MFI and FPMax. CHARM-MFI is Apriori-based and an extension of the CHARM algorithm; first closed patterns are detected and a post-processing filter identifies those that are also maximal. FPMax, an extension of FPGrowth, utilizes the FP-Tree and directly identifies maximal frequent patterns.

Throughout the experiments, we use five datasets: chess, chainstore, connect, pumsb, and a series of synthetic datasets. Chess and chainstore are classic datasets commonly used for benchmarking frequent pattern mining algorithms [80]. Chess consists of 75 distinct items across 3196 transaction that have an average size of 37 items. Chainstore contains digital customer trans-

actions from a retail store. There are roughly 46,000 distinct items, 1.1 million transactions, and each transaction has an average size of seven items. Connect is composed of spacial information collected from the connect-4 game. This data contains 67,557 transactions and 129 distinct items. Pumsb consists of census data for population and housing. It is composed of 49,000 transactions with more than 2,100 distinct items. The synthetic datasets are simulated transaction databases of increasing size and are used for the Insert and Delete experiments. All experiments are conducted on a single-core server with 120 GB of RAM.

3.4.1 Tree Construction

In this experiment we benchmark the runtime for building the FHPTree using the chess, connect, pumsb, and mushroom datasets. In addition, we discuss the factors that determine the size of FHPTrees and evaluate the memory footprint. The goal is to illustrate the worst case performance and discuss the critical factors that determine the runtime. The most expensive operation in the tree construction approach is the iterative pairwise comparison. When searching for the most similar pair of items, all cooccurring items are evaluated in order to find the best matches. In the worst case, all items cooccur, and $\frac{n(n-1)}{2}$ comparisons are performed when forming each layer of the tree, where n is the number of nodes in the current layer of the tree. As a result, our concern is the scalability as the number of distinct items increases.

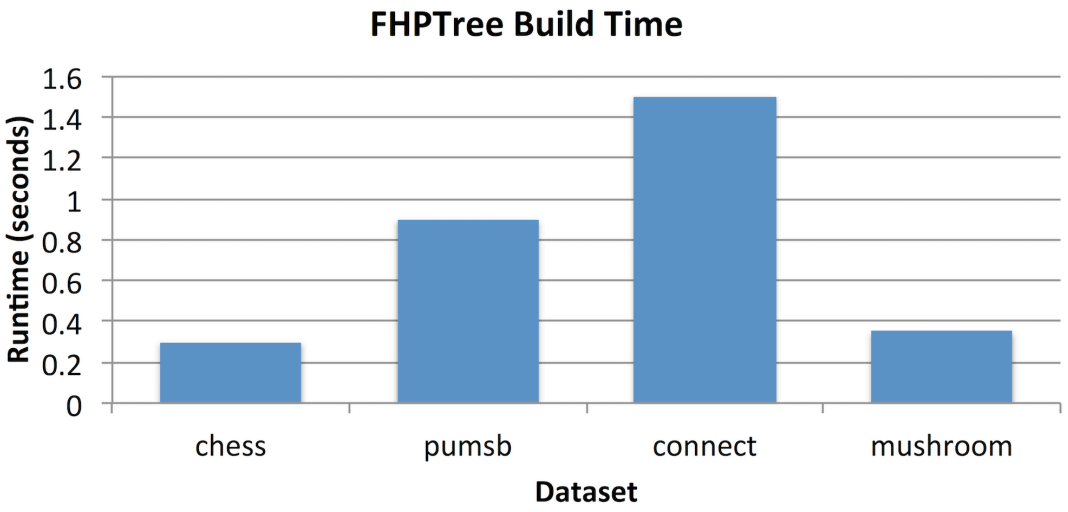

Figure 3.9: The runtime for Tree Construction on the following dataset and *min_support* combinations: chess (1%), connect (1%), pumsb (1%), and mushroom (0.01%)

Each recursively defined layer requires $O(n^2)$ comparisons, each layer has roughly half the items from the previous layer, and there are $log(n)$ layers. Equation 3.7 defines an upper bound for the number of comparisons required to build an FHPTree from n leaf nodes.

$$n^2 \times \sum_{k=0}^{log(n)} \left(\frac{1}{2}\right)^{2k} < n^2 + \frac{n^2}{3} \tag{3.7}$$

It is important to recall that the number of nodes in an FHPTree is not dependent on the distribution of the data. Given k distinct items, there will always be $2k - 1$ nodes and $2k - 2$ edges in the FHPTree. Also, in practice, we utilize a *BitSet* data structure to represent transaction sets, so the footprint may be small. This notion makes it easy to estimate the size of an FHPTree and thus, easier to estimate hardware requirements. Figure 3.10 provides a comparison between the FHPTree and FPTree in terms of their memory footprint.

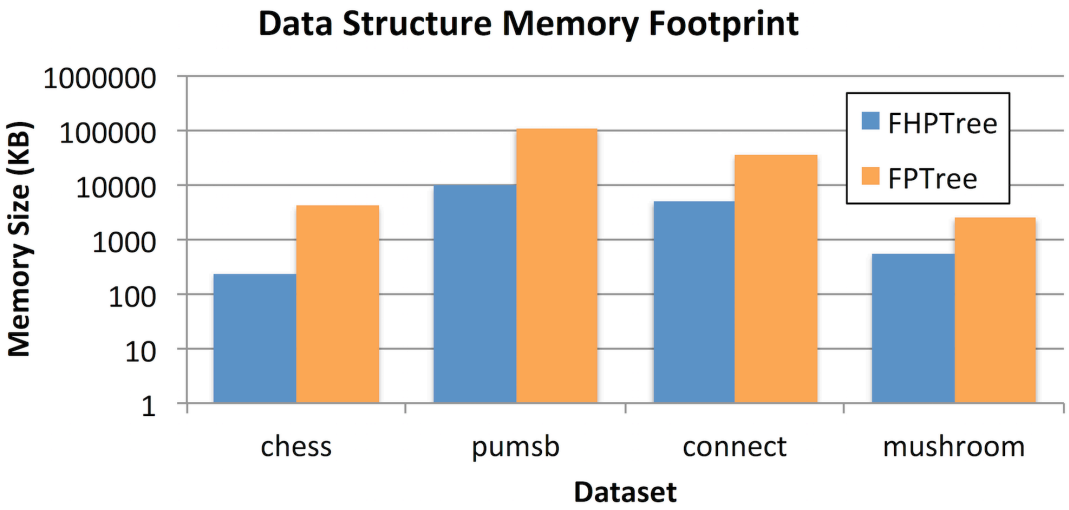

Figure 3.10: The memory footprint for the FHPTree and FPTree on the following dataset and *min_support* combinations: chess (1%), connect (1%), pumsb (1%), and mushroom (0.01%). The vertical axis is a log-scale and is measured in KB.

The FHPTree has a small footprint, up to 10x smaller than the FPTree for these select datasets. At each node, the candidate transactions may contain redundant information since the exact transactions are a subset, which suggests that the footprint could be reduced further.

The performance difference between chess and connect in Figure 3.9 suggests the runtime is impacted by the distribution of data and the number of transactions. Similarly there is a noticeable impact on the memory footprint. The chess and connect datasets FHPTrees have nearly the same number of nodes; however, the density and transaction count make connect more computationally intensive. Since the FHPTree serves as a persistent data structure, this does not have to be a reoccurring challenge. Future tree building strategies may employ a *k*-nearest neighbor search to reduce the complexity.

3.4.2 Insert and Delete

To evaluate the performance of insert and delete operations, we built FHPTrees of various sizes and perform insert and delete operations. The runtime is recorded when performing 10 of the respective operations. The goal is to characterize the impact the FHPTree size has on performance.

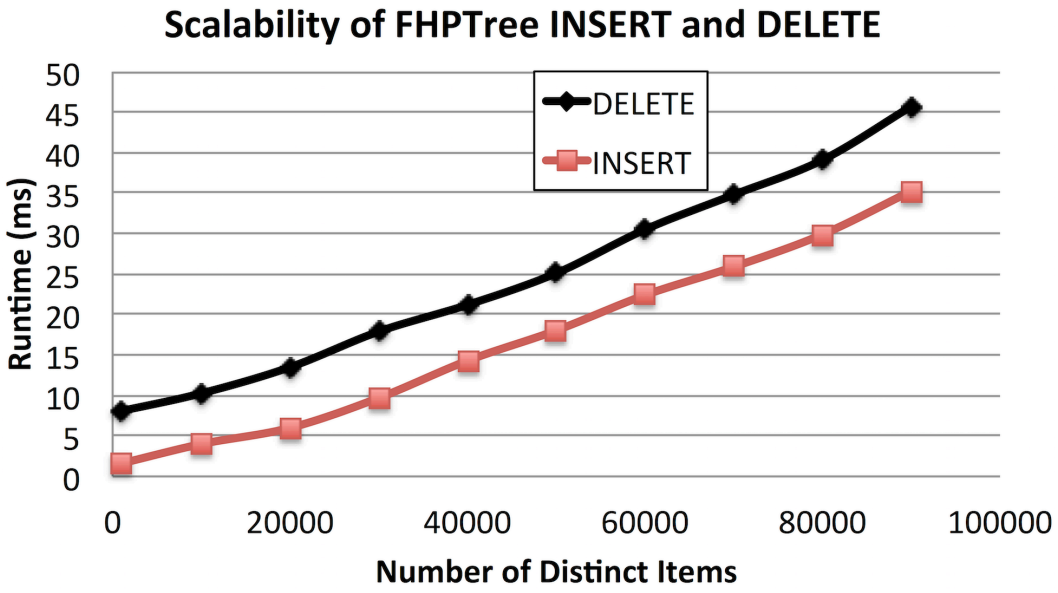

Figure 3.11: The time required to perform insert and delete operations on FHPTrees of various sizes. These operations are fast and scalable.

Even on the largest tree, consisting of 200,000 nodes, the overall runtime required for 10 insert or delete operations is less than 50 ms. The number of traversal steps for either operation is proportional to $log(n)$, where n is the number of leaves in the FHPTree. The four datasets discussed previously were also considered. Each tree was built while excluding an item. Then, the excluded item was inserted into the tree. Every item was excluded and inserted, and the average runtime was collected. For each dataset, the average insert time was approximately 1 ms. Results were consistent for the delete operation as well.

Runtimes this fast suggest that an alternative solution for building an FHP-

Tree may be sequentially inserting items. One of the main drawbacks is that the approach may not yield an effective FHPTree; the order in which the items are inserted can affect the topology of the tree.

3.4.3 Scan

Since this is the classic problem of frequent pattern mining, we compare the performance of FHPGrowth with existing approaches: FPMax and CHARM-MFI [80]. These approaches are implemented in Java; this implementation of FHPGrowth was written in Scala, which compiles to Java. It is important to mention that the runtimes reported for the scan operation do not include the time required for tree construction as the tree is a persistent structure, i.e., it is only built once and used by all scans. The first test is extracting maximal frequent patterns from the chess dataset. Figure 3.12 characterizes the runtime relative to the *min_support* threshold. At high support values, FPMax is fastest by a narrow margin, but as the *min_support* threshold becomes small, the number of maximal frequent patterns increases, and FPMax begins to slow. At 40% support, both FPMax and FHPGrowth are more than 300x faster than CHARM-MFI. At the lowest support value of 15%, FHPGrowth is 14x faster than FPMax. We also consider the runtime relative to the number of frequent patterns detected, shown in Figure 3.13.

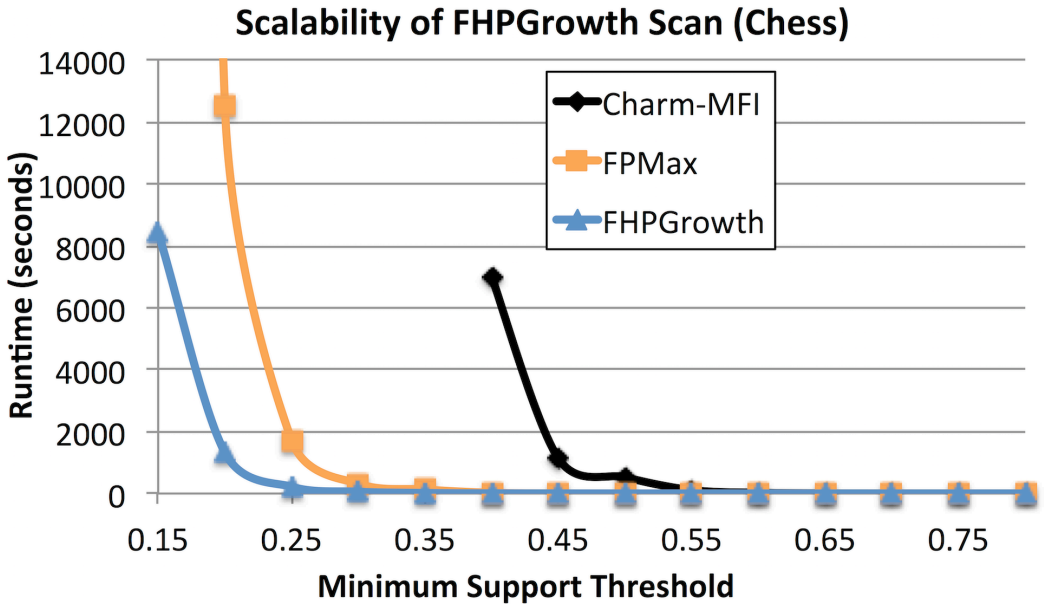

Figure 3.12: The runtime comparison between FHPGrowth, FPMax, and CHARM-MFI based on *min_support* using the chess dataset.

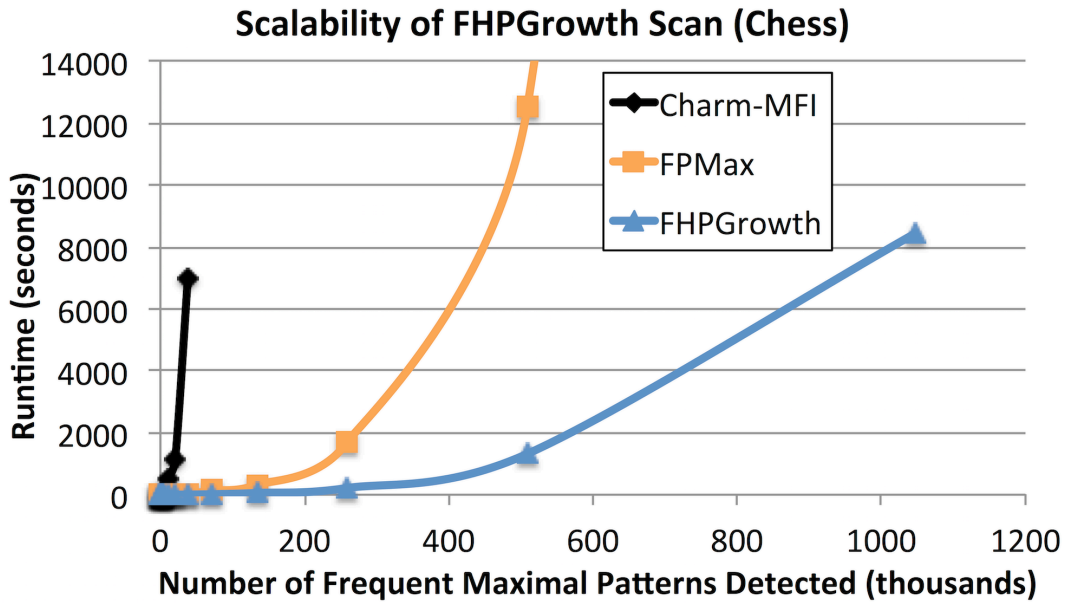

Figure 3.13: The runtime comparison of FHPGrowth, FPMax, and CHARM-MFI based on the number of maximal patterns in the chess dataset.

The chess dataset used in this experiment is quite small. Its density, high degree of connectivity between items, is what makes the dataset challenging. The CHARM-MFI algorithm wasn't able to survive below 35% support. FPMax and FHPGrowth were comparable in performance until around 45% support where

FHPGrowth begins to gain a significant advantage. FHPGrowth was able to detect the top one million patterns faster the FPMax could detect the top 500 thousand.

Next, we consider the connect dataset shown in Figure 3.14. This dataset is larger than chess in terms of transactions. Connect contains fewer but longer maximal frequent patterns.

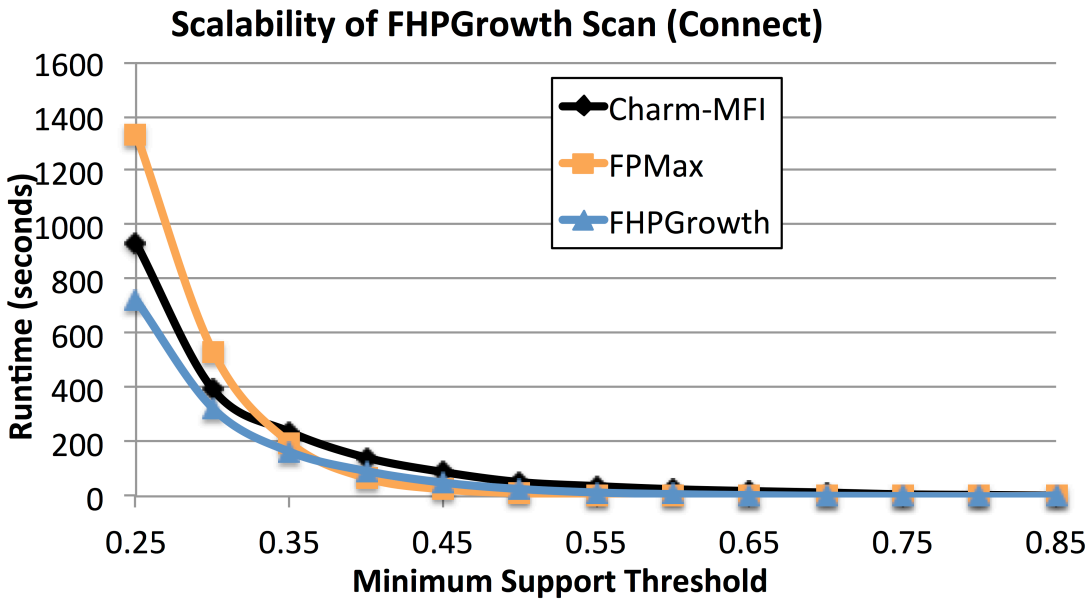

Figure 3.14: The runtime comparison between FHPGrowth, FPMax, and CHARM-MFI based on *min_support* using the connect dataset.

At high support values, FPMax is the fastest, but once *min_support* drops below 35%, CHARM-MFI and FHPGrowth take the advantage. At the lowest support value, FHPGrowth is roughly 20% faster than CHARM-MFI and 2x faster than FPMax.

In the next performance comparison, we utilize the pumsb dataset. This dataset contains the most maximal patterns of any dataset we consider in this study. Shown in Figure 3.15, as the *min_support* value decreases, FHPGrowth became increasingly faster than FPMax and CHARM-MFI. At 55% support, FH-

PGrowth is 60x faster than FPMax. CHARM-MFI had performance woes for *min_support* below 70%, and FPMax becomes significantly slower than FHP-Growth for *min_support* below 60%. We also consider the runtime relative to the number of frequent patterns detected, as shown in Figure 3.16.

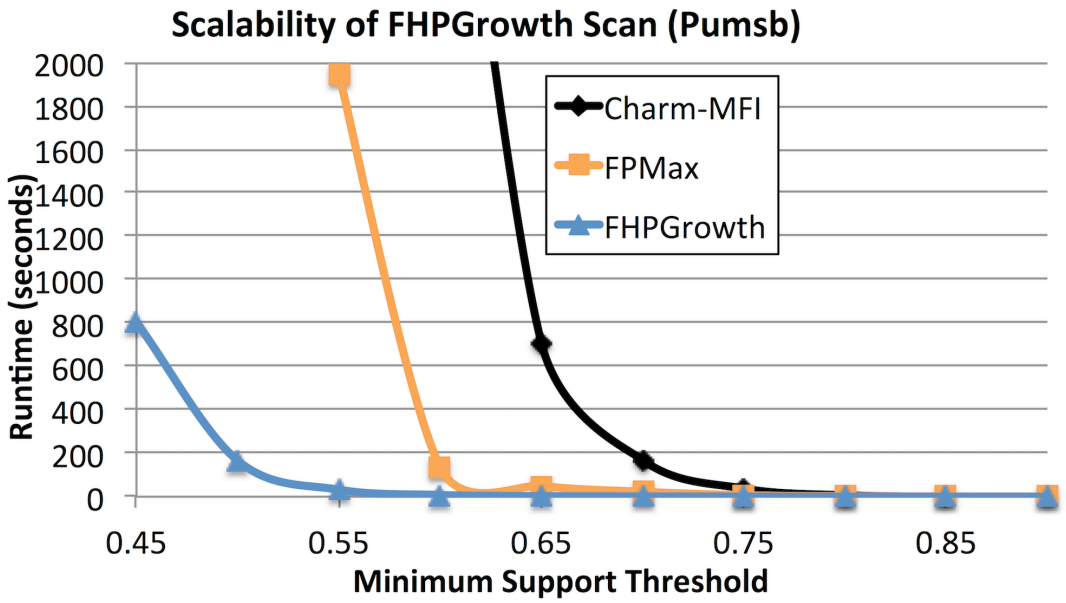

Figure 3.15: The runtime comparison between FHPGrowth, FPMax, and CHARM-MFI based on *min_support* using the pumsb dataset.

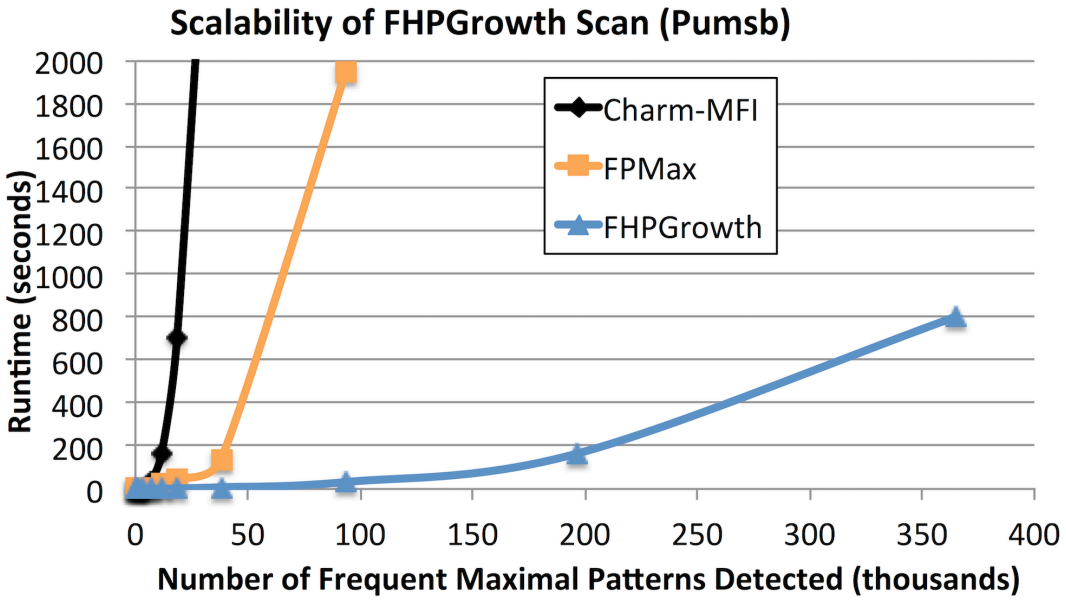

Figure 3.16: The runtime comparison between FHPGrowth, FPMax, and CHARM-MFI based on the number of maximal patterns detected using the pumsb dataset.

FHPGrowth detected the top one million patterns faster than FPMax could detect the top 500 thousand. This sort of dense dataset is appropriate for the FHPGrowth paradigm. The last dataset we consider is mushroom, which is much more sparse in comparison to chess, connect, and pumsb.

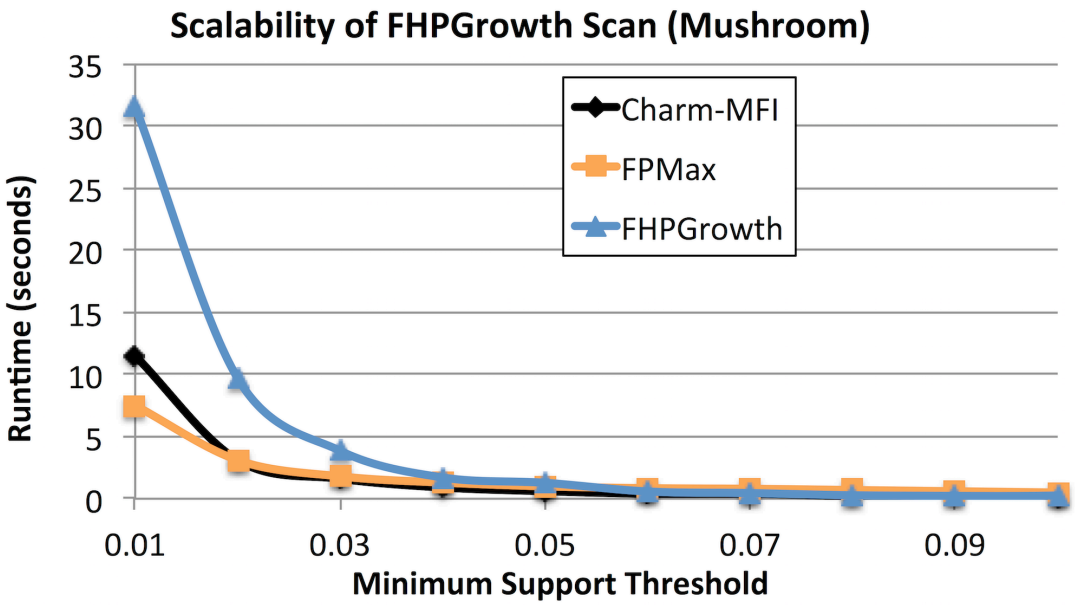

Figure 3.17: The runtime comparison between FHPGrowth, FPMax, and CHARM-MFI based on *min_support* using the mushroom dataset.

As shown in Figure 3.17, FPMax and CHARM-MFI outperform FHPGrowth on this sparse dataset. The FHPTree is not able to effectively make use of the hierarchical node structure, and the performance suffers. That is to say, there are many hierarchical patterns that are candidate frequent but were not frequent.

As a theoretical evaluation of our performance and method of reflection, we consider the number of states in a traversal and compare it with the ideal case having the minimum number of states required to extract all maximal frequent patterns. A direct traversal from root to pattern of size k will require $O(k*log(n))$ steps for an FHPTree with n leaves. If there are p patterns of size less than or equal k, then we may suggest $p * k * log(n)$ steps are required. However, our

method does not start at the root before discovering each item. Item detection is a chain of events occurring along the traversal. Moreover, 1-off patterns may only be separated by a single state. At this point $p * k * log(n)$ may begin to seem like a conservative estimate for the minimum number of required steps. Furthermore, since a k-itemset may only require $j \leq k$ nodes, due to the nature of pattern covers in the FHPTree, $p * k * log(n)$ can be reduce to $p * j * log(n)$. As an example, using the connect dataset, the current FHPGrowth implementation utilizes 378,000 states to discover 2,103 itemsets; the longest itemset consists of 20 items. Assuming all items are of length 20 and utilizing the conservative estimate of $p * k * log(n)$, we find 294,420 states could identify all 2,103 itemsets. The Scan experimental results suggest the FHPTree is efficient and can effectively prune the search space; this discussion suggests there is still room for improvement.

3.4.4 Search

We evaluate the performance of search using the chess dataset. In this experiment, our goal is to find the maximal frequent itemsets containing an item of interest. Similar to the scan experiments, we vary the *min_support* threshold and evaluate the effect on runtime.

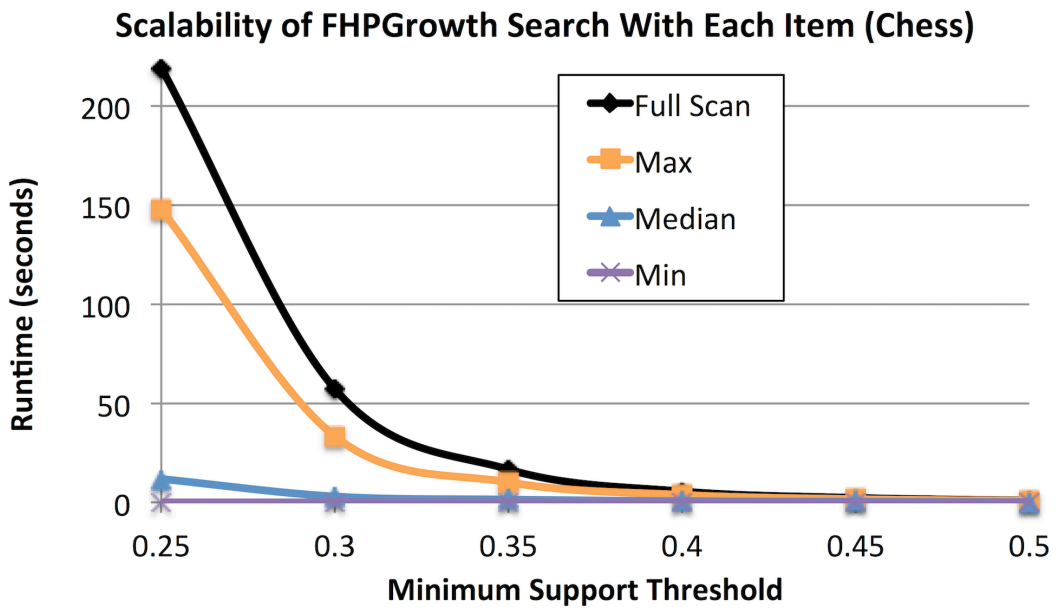

Figure 3.18: The performance comparison between the search and scan operations. There was a lot of variance in the runtimes of search operation depending on the query item, so the minimum, maximum, and median search runtimes are reported as well.

Figure 3.18 provides an overview of the search performance. The number of maximal frequent patterns containing some item x varies significantly for different x. The maximum search time was consistently less than the full scan operation. The median search time at 25% support was 11 seconds, a 20x reduction in runtime compared to the full scan. These results suggest the search approach is favorable to a post-processing technique to identify patterns containing the query item.

3.5 Conclusion and Future Work

In this chapter, we proposed the FHPTree, a hierarchical cluster tree of items, and FHPGrowth, a top-down mining scheme for extracting frequent patterns. The number of nodes required for the FHPTree scales linearly as the number of distinct items, while the FPTree is highly dependent on the distribution of the

data and can scale exponentially in certain scenarios. Furthermore, we achieved a 10-fold reduction in the memory footprint over the FPTree. In addition, for re-occurring pattern mining analyses, utilizing a persistent data structure reduces redundant computation. Since the FHPTree supports insert, update, and delete operations, it is not necessary to continually rebuild before each analysis or when the transaction database is updated. FHPGrowth was competitive when compared to existing state-of-the-art approaches, CHARM-MFI and FPMax. FHPGrowth outperformed both approaches on dense datasets, achieving up to a 60-fold reduction in runtime. In addition, the search operation enables targeted pattern mining analyses to be conducted efficiently. The median runtime for search was a dramatic reduction in runtime compared to a full scan.

Experimental results are promising and a testament to the frequent hierarchical pattern mining paradigm. Furthermore, there are many optimizations to further improve and refine the concept. We have conjectured several potential improvements to the FHPTree structure, FHPTree construction process, FHPGrowth Scan, and FHPGrowth Search. During the Scan discussion of the Performance Evaluation section, we provide evidence that the efficiency has significant room for improvement. FHPGrowth could also benefit from multithreading and GPU acceleration; the transaction set operations may be a starting place for parallelization.

The next phase of this research may include defining optimal FHPTrees, contrast set mining, and sequential pattern mining. In this chapter, we proposed an effective strategy for construction FHPTrees; however, it is certainly not the optimal strategy. The question remains, what does a perfectly choreographed

traversal, FHPGrowth, look like? We define an optimal FHPTree such that when scanned, the traversal will extract all maximal frequent patterns in the minimum number of steps. Are there efficient strategies to ensure FHPTree optimality? Datasets may be packaged into an optimal FHPTree and shared among researchers. In this way, much of the mining process can be preserved, encoded into the small footprint of an FHPTree, and therefore negated in subsequent computations.

Chapter 4

Distributed Computing and Frequent Hierarchical Pattern Mining

In the previous chapter, we demonstrated promising results for the FHPTree and FHPGrowth when compared to the current state of the art maximal frequent pattern mining algorithms. In this chapter, we will explore a few of those opportunities including an iterative search and delete process, and a distributed computing strategy.

We have discussed the efficiency of the FHPGrowth traversal and pointed out that there is room for improvement. Specifically, the number of required states in the traversal may be notably less than the number used by the current traversal. On the other hand, we saw the power of the search operation, and how it efficiently prunes the search space and demonstrated promising runtime performance. Our hypothesis is that the advantages of the search operation can be used to optimize the full scan.

Since frequent pattern mining is a computationally intensive task, parallelizing FHPGrowth has the potential to improve performance and significantly reduce runtimes. In this way, hardware and computing clusters can be scaled in order to achieve desired performance. We will propose a mechanism for per-

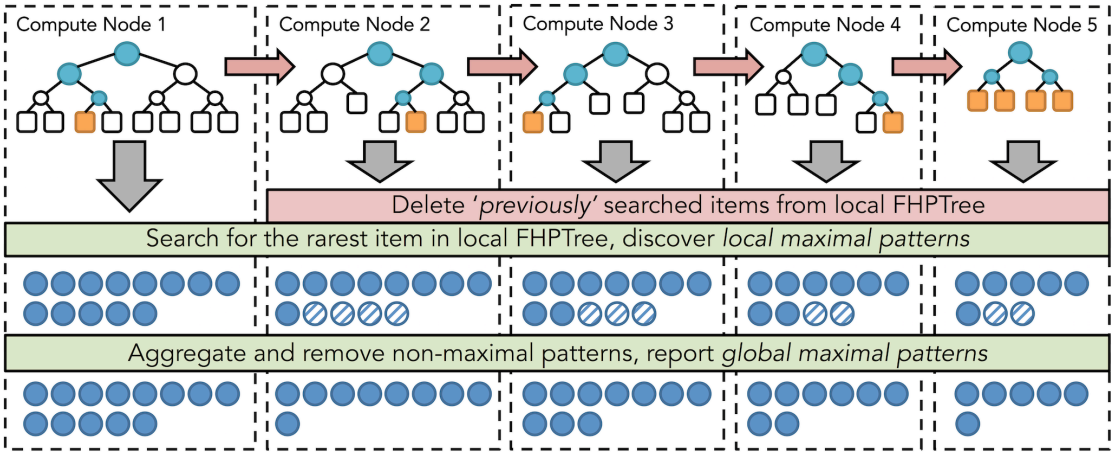

Figure 4.1: High-level overall architecture for the distributed FHPGrowth algorithm. The process involves copying the data structure to each compute node and performed a collection of targeted search operations.

forming a full scan as a collection of targeted search operations. In addition, we will demonstrate how this can be used to devise a distribution strategy appropriate for distributed computing environments.

The remainder of this chapter focusses on how to perform a full scan operation as a collection of targeted search operations. In Parallelizing FHPGrowth, we present the algorithm details of this process and discuss how it can be parallelized and distributed for cluster computing environments. Performance Evaluation provides a variety of experimental results to characterize the runtime performance of our proposed methods. In Conclusion, we summarize our contributions and allude to future research opportunities.

4.1 Parallelizing FHPGrowth

The first component to consider for distributing the approach is the underlying data structure. When distributing the FHPTree, we have 2 choices: duplicate the data structure on each machine, or partition the data structure and distribute the resulting collection of subtrees onto various machines. The traversal scheme of

FHPGrowth would not lend itself nicely to the latter as each state may involve many nodes in the tree, which could create significant network overhead. Thus, we adopt the strategy of copying or broadcasting the data structure onto each machine. It is important to recall that the memory footprint is relatively small for the FHPTree.

Another goal is to define a mechanism for partitioning the FHPGrowth traversal; it comprises the majority of the workload. We will do this by using the FHPTree's efficient inclusive search filter. By performing a search for each individual item, we are guaranteed to find all maximal frequent patterns. Based on the results from the previous chapter, we are interested in knowing whether searching for each item individually could be faster than the typical full scan; it certainly presents a distribution strategy. We would be able to search for different items on specific CPUs or physical machines, in parallel. In the previous study, we demonstrated how individual search operations are faster than a full scan. Thus, if the computing environment is large enough and all search operations can be performed simultaneously, this distribution technique will provide improves in runtime.

There are a few limitations to this approach. First, searching for an item with high support may take significantly longer than for a low support item. This may cause the distributed workload to become imbalanced, leading to straggler tasks. Second, duplicate patterns will be detected. For example, $\{A, B, C\}$ will be detected when searching for A, B, and C. Our proposed method will ensure a balanced workload and eliminate discovery of duplicate results.

The remainder of this section details the search and scan operations. Since

we broadcast the FHPTree to each node, the insert, update, and delete operations remain unchanged from the classic approach. A given operation is performed on the head node of the compute cluster, and the updated FHPTree is broadcasted to all nodes.

4.1.1 Scan

The sequential search strategy discussed in the second paragraph of this section is detailed in Algorithm 15. Figure 4.2 provides a visualization of the number of patterns returned by each search; the number of results varies significantly for different query items. Specifically, the items with higher support tend to be involved in more maximal frequent patterns.

Algorithm 15 FHPGrowth: Distributed Scan

```
1: queries = []
2: for item ∈ I do
3:     n = newQuery()
4:     n.search = item
5:     queries.append(n)
6: end for
7: FHPTree.broadcastToNodes
8: for q ∈ Distribute(queries) do
9:     localFHPTree.search(q.search)
10: end for
```

To address these concerns, we utilize an iterative search and delete process. The philosophy is that after searching for an item, we have all the patterns containing said item, and thus, that item may be deleted after the search. In addition, a classic strategy in query optimization is to evaluate the most selective query condition first. Similarly, we search for the least frequent item first since they execute the fastest. Then, the item is deleted, and the next least frequent item is searched. This process is continued until the tree is empty. This itera-

tive search and delete technique is discussed as a serial process. However, the method is parallelized by precomputing the necessary delete operations associated with each query item and distributing the corresponding instructions to appropriate machines in the cluster.

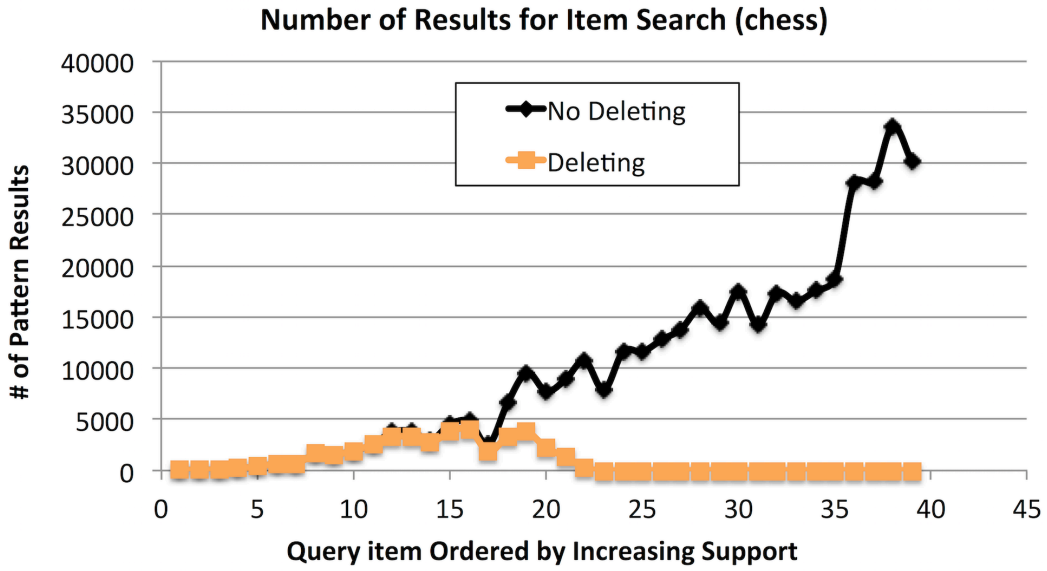

Figure 4.2: Illustrating the number of results returned when searching for each item and the impact of the iterative search and delete process. By deleting 'previously' search items, redundant computation is reduced.

Throughout this chapter, we will refer to queries as being subsequent or occurring after other queries. If query A occurs "after" query B, this simply states that A will be deleted from the tree before searching for B. As a result, searching for A and searching for B become independent operations. In general, for any query Q_1 that occurs before another query Q_2, the items contained in Q_1 will be deleted from the FHPTree before searching for Q_2. From this point forward, we refer to this sequential search and delete strategy as the SD approach and is detailed in Algorithm 16.

By deleting from the FHPTree, the most frequent items are contained in progressively smaller trees. Items that previously took the longest become the

Algorithm 16 FHPGrowth: Distributed Scan (Search And Delete)

```
1: queries = []
2: sortItems = I.sortByIncreasingSupport
3: for item ∈ sortItems do
4:     n = newQuery()
5:     n.search = A.append(item)
6:     n.previousItems = sortItems.subseq(0, item)
7:     queries.append(n)
8: end for
9: FHPTree.broadcastToNodes
10: for q ∈ Distribute(queries) do
11:     localFHPTree.delete(q.previousNodes)
12:     localFHPTree.search(q.search)
13: end for
```

fastest. Figure 4.2 demonstrates the reduction in results for the highest support items achieved by this technique. Although, this technique has a significant impact on workload distribution and reducing redundant computation, both issues remain. Now the first 50% of items, when sorted in ascending order by support, take on the burden of retrieving the most results, and some non-maximal patterns are detected due to item deletion. Another point to notices, there is a long tail effect; at a certain point, all patterns have been detected and subsequent queries do not return any patterns.

We will come back to the issue of non-maximal patterns being detected and the long tail of unnecessary queries. As for workload balancing, conceptually, these expensive search queries need to be split into a collection of more selective queries that yield equivalent results. Those new queries could then be executed on various CPUs or compute nodes, effectively distributing the burden of the initially expensive query. For example, if the search for A yields too many results, searching for $\{A, B\}$, $\{A, C\}$, and $\{A, D\}$ separately may partition this burden. These secondary items are also searched for in order of increasing support,

so the most selective query is evaluated first. We also delete the secondary items after searching. That is, after searching for $\{A, B\}$, B is deleted, so the search for $\{A, C\}$ will not return any patterns containing B. We let each node in item A's corresponding FHPTree be a secondary query item to ensure all patterns involving A have been detected.

At this point, we have defined a mechanism for partitioning and distributing the workload of the FHPGrowth traversal. Next, we discuss how to prevent non-maximal patterns from being detected and how to remove the long tail of unnecessary queries.

Removing Non-maximal Patterns

Removing subsets can be a computationally intensive process. The naive approach is to perform an all-against-all comparison between the resulting patterns, and remove those that occur as a subset. That approach is expensive, scaling as $O(n^2)$. The number of non-maximal patterns is linearly dependent on the number of maximal patterns, and these subsets follow a predictable pattern. For example, suppose $\{A, B, C\}$ is detected when searching for A. Then, $\{B, C\}$ may be detected when subsequently searching for B. For each pattern P detected when searching for A, we generate an anti-pattern $P - A$ designed to negate the subsets detected during subsequent searches. In the final aggregation of the pattern results, these anti-patterns will negate and eliminate their non-maximal counterpart, and as a result, only maximal patterns will remain.

In the cases where secondary items are utilized in the query, the deletion logic becomes slightly more complex. Suppose we search for $\{A, B\}$, and dis-

cover the maximal pattern $\{A, B, C, D\}$. After deleting the secondary item B and searching for $\{A, C\}$, the pattern $\{A, C, D\}$ will be detected. After all of A's secondary items are exhausted, A is deleted from the FHPTree and B is searched. At this time, $\{B, C, D\}$ will be detected. Given that $\{A, B, C, D\}$ was detected while searching for $\{A, B\}$, we create anti-patterns $\{A, B, C, D\} - B$ and $\{A, B, C, D\} - A$. To generalize, if K is a collection of items in the query, and P is a pattern detected, $P - k_i$ is a subset that may be detected by subsequent searches for all $k_i \in K$.

Removing the Long Tail

The long tail can be viewed in Figure 4.2 and refers to the collection of queries that are unnecessary because they do not return any results. The results for these queries have already been covered by previous search operations. To account for this, we only evaluate the first $k\%$ of queries. There is a chance that the last $(1 - k)\%$ of queries contain novel pattern results. Therefore, we must construct one final query that covers the $(1 - k)\%$ that have been discarded. In this way, we guarantee comprehensive results while consolidating many unnecessary operations.

For example, let $\{A, B, C, D, E, F\}$ be the query items sorted in ascending order based on support. Suppose we evaluate the first 50% of queries. Then $\{A, B, C\}$ is searched using the SD approach. The remaining nodes $\{D, E, F\}$ are searched by performing a full scan on the FHPTree containing only D, E, and F.

This long tail effect also presents itself after we slit a query into a collection of more selective queries. For example, let A be a query that is split into a collection of more selective queries. Let $\{(A, B), (A, C), (A, D), (A, E), (A, F)\}$ be the collection

of queries sorted in ascending order based on support. If we discard the last 50%, we are left with $\{(A, B), (A, C), (A, D)\}$. The covering query would be a search for item A, on the FHPTree containing $\{A, E, F\}$. If any remaining patterns contain (A, E) or (A, F), they will be detected by this covering query. In general, for a query Q that is split into a collection CQ, the covering query involves searching for Q on the FHPTree containing all items in Q and the long tail removed from QC.

4.1.2 Search

A useful technique in frequent pattern mining is having the ability to execute targeted data mining tasks. That is to say, we would like to find all maximal frequent patterns that contain a collection of items of interest. Similar to the classic FHPGrowth approach, the distributed search utilizes the same logic as the distributed scan. However, rather than querying all items, we generate a collection of queries that include the items of interest. Algorithm 17 demonstrates this concept.

Algorithm 17 FHPGrowth: Distributed Search

```
1:  function Search(A: Array of items)
2:      queries = []
3:      sortItems = I.sortByIncreasingSupportWith(A)
4:      for item ∈ sortItems do
5:          n = newQuery()
6:          n.search = A.append(item)
7:          n.previousItems = sortItems.subseq(0, item)
8:          queries.append(n)
9:      end for
10:     FHPTree.broadcastToNodes
11:     for q ∈ Distribute(queries) do
12:         localFHPTree.delete(q.previousNodes)
13:         localFHPTree.search(q.search)
14:     end for
15: end function
```

In this passage, we provide an example of the collection of queries generated for a specific search. Let $\{A, B, C, D, E, F\}$ be the set of all items, and let A be the search item. The set of queries we generate are $\{(A, B), (A, C), (A, D), (A, E), (A, F)\}$. After removing the long tail, the resulting set of queries may be $\{(A, B), (A, C), (A, D), A\}$, where the singleton A is the covering query.

4.2 Performance Evaluation

In this section we evaluate the efficiency of the distributed FHPGrowth in terms of the search and scan operations. We compare with the single node implementation of FHPGrowth detailed in the previous chapter. In addition, we compare with a single node implementation that utilizes the SD approach.

Throughout the experiments, we use four datasets: chess, chainstore, connect, and pumsb. Chess and chainstore are classic datasets commonly used for benchmarking frequent pattern mining algorithms [80]. Chess consists of 75 distinct items across 3196 transactions that have an average size of 37 items. Chainstore contains digital customer transactions from a retail store. There are roughly 46,000 distinct items, 1.1 million transactions, and each transaction has an average size of seven items. Connect is composed of spacial information collected from the connect-4 game. This data contains 67,557 transactions and 129 distinct items. Pumsb consists of census data for population and housing. It is composed of 49,000 transactions with more than 2,100 distinct items. All experiments are conducted on an Apache Spark cluster consisting of 8 nodes, each with 8 CPU cores and 120 GB of RAM.

4.2.1 Horizontal Scalability

The goal of this experiment is to demonstrate the performance gains associated with increasing computing resources. In this experiment, we consider the chess dataset at 15% support and vary the number of compute nodes in our cluster from 1 to 8 and report the runtime for the scan operation.

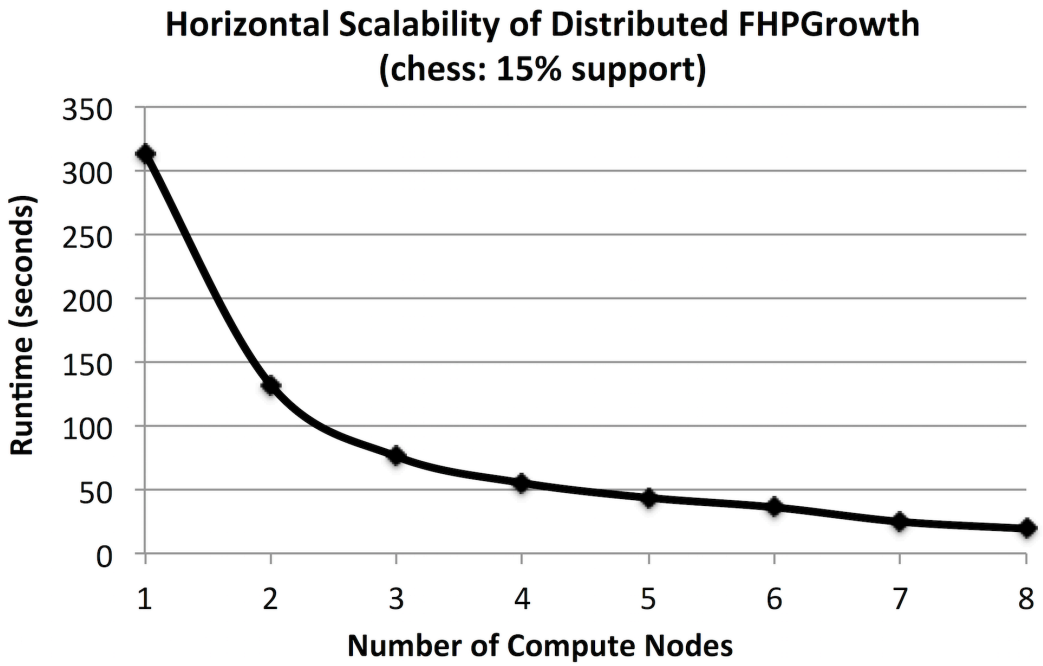

Figure 4.3: A horizontal scalability analysis demonstrating how the number of compute nodes affects the runtime.

Shown in Figure 4.3, as the number of nodes increases, the runtime decreases. For this dataset, the workload distribution is balanced, so we see an 8*x* speed up when comparing the 8 node cluster to the single node execution.

4.2.2 Scan

The next experiment is mining all maximal frequent patterns from the chess dataset. Figure 4.4 characterizes the runtime relative to the *min_support* threshold. The improvements offered by the SD process are significant. For minimum

support thresholds greater than 35%, the SD approach is the slowest, nearly 2*x* slower than the original FHPGrowth. Below 35%, the improvement becomes significant, offering a 20*x* speedup at *min_support* = 15%. The distributed approach consistently offers approximately 8*x* speedup over the SD approach, and thus, is roughly 160*x* faster than the original FHPGrowth.

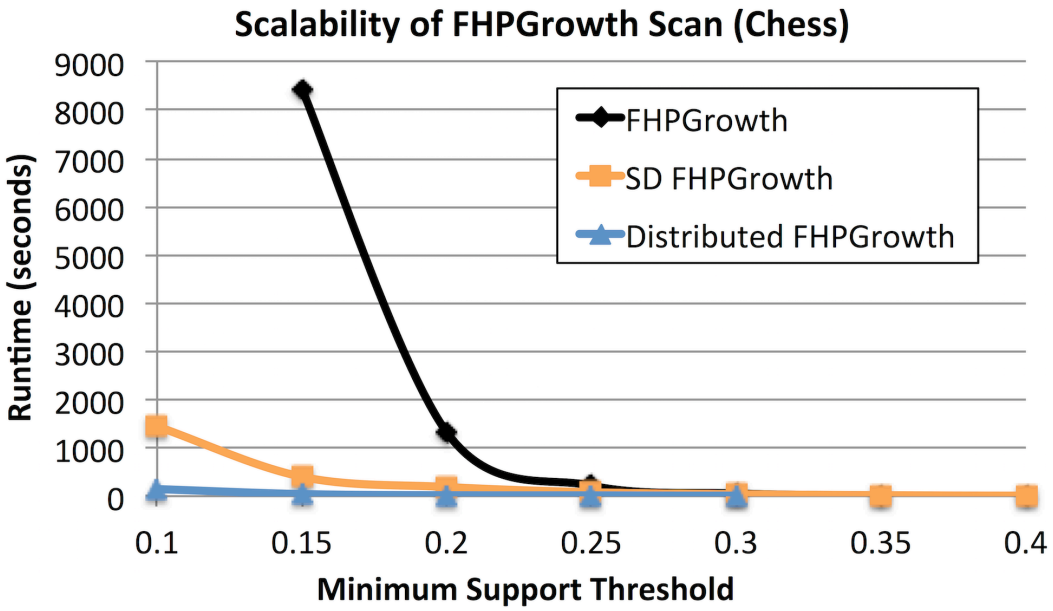

Figure 4.4: A comparison between the single server FHPGrowth and distributed FHPGrowth relative to runtime and minimum support threshold on the chess dataset.

As the number of patterns increases, the SD approach is more efficient. Figure 4.5 demonstrates this concept.

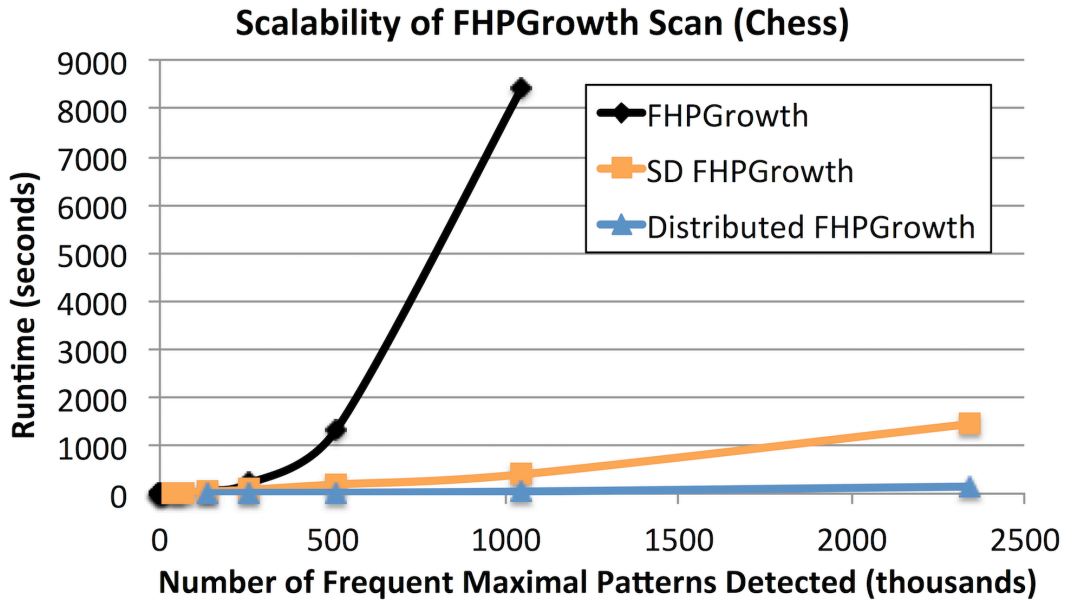

Figure 4.5: A comparison between the single server FHPGrowth and distributed FHPGrowth relative to runtime and number of pattern results on the chess dataset.

In this next performance comparison, we utilize the connect dataset. In contrast to the previous results, the SD approach is 2x slower than the original FHPGrowth. This may seem contradictory; however, it is aligned with the previous results at high minimum support thresholds.

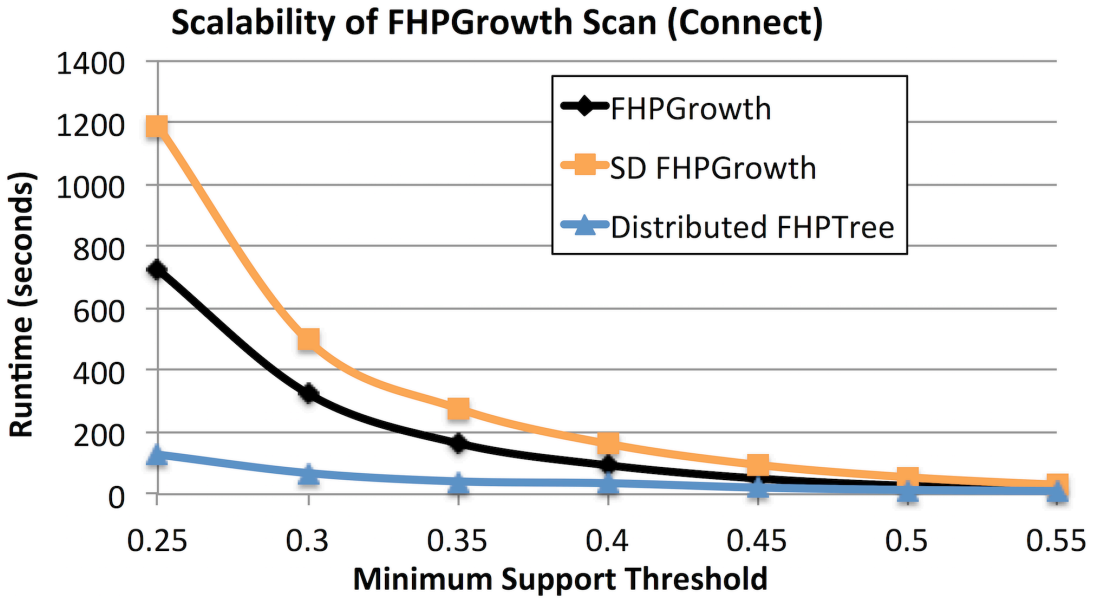

Figure 4.6: A comparison between the single server FHPGrowth and distributed FHPGrowth relative to runtime and minimum support threshold on the connect dataset.

The number of results for this dataset is far less than that of chess; at 25% support, there are roughly 18,000 results. As a result, the overhead of the SD process makes it less efficient in this scenario. Figure 4.7 provides more detail about the runtime performance relative to the number of maximal patterns.

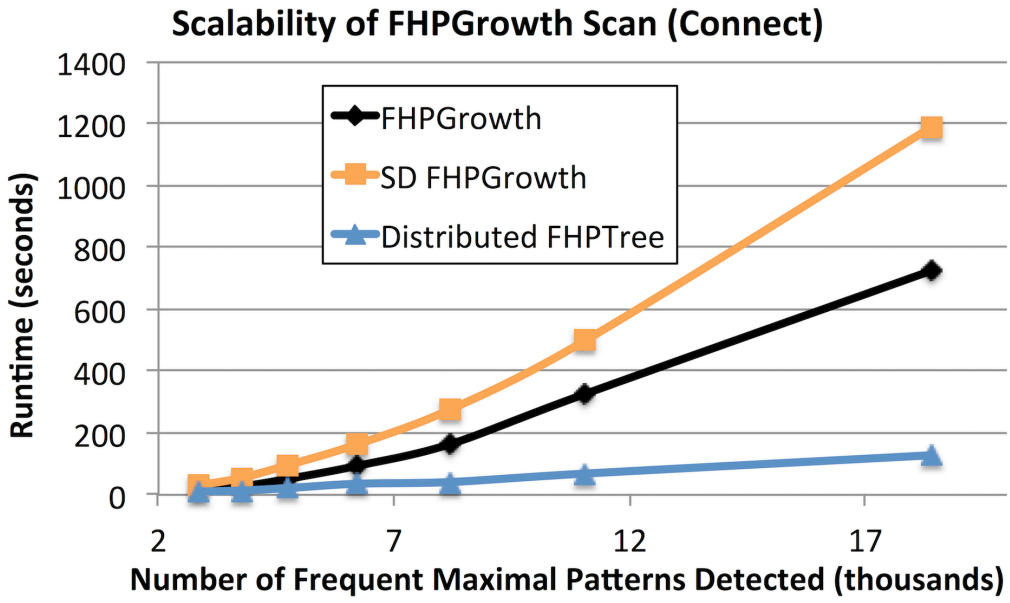

Figure 4.7: A comparison between the single server FHPGrowth and distributed FHPGrowth relative to runtime and number of pattern results on the connect dataset.

Next, we utilize the pumsb dataset to evaluate performance. The FHPTree is efficient on this dataset; in our previous work, we demonstrated a 60*x* speedup over the classic approaches, FPMax and Charm-MFI. As shown in Figure 4.8, the SD approach offers a 4*x* speedup over the original FHPGrowth, and the distributed approach achieved a 3*x* speedup over the SD approach.

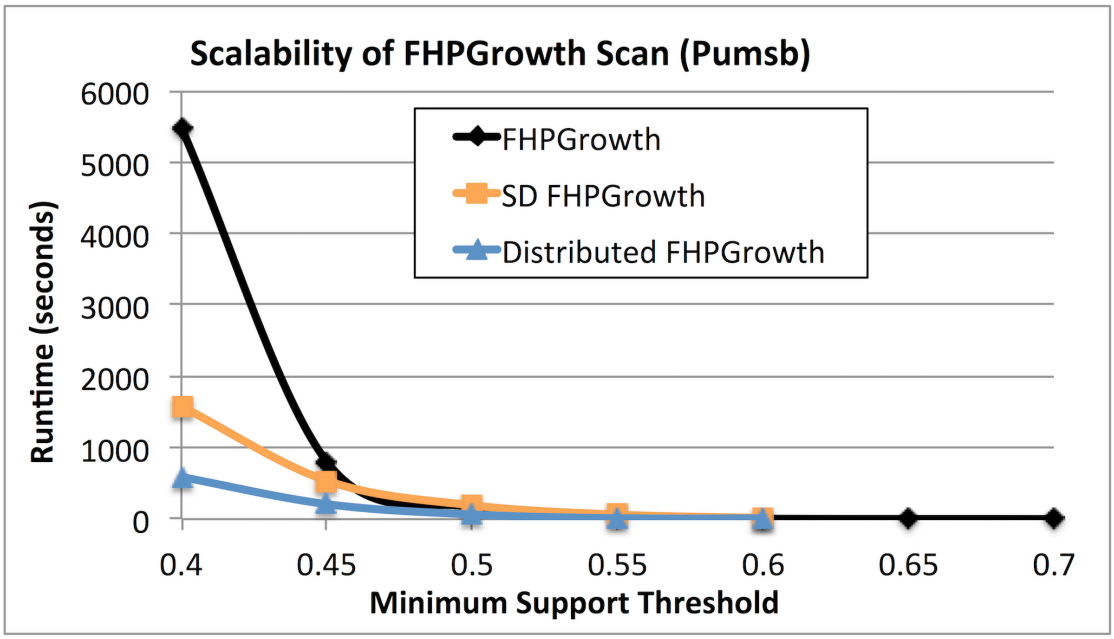

Figure 4.8: A comparison between the single server FHPGrowth and distributed FHP-Growth relative to runtime and minimum support threshold on the pumsb dataset.

If we consider the number of maximal patterns detected, shown in Figure 4.9, we see a similar trend compared to the chess dataset. When there are hundreds of thousands of maximal patterns detected, the SD approach is faster than the original FHPGrowth. Since the distributed approach achieves a 3*x* speedup, rather than 8, we know that the workload distribution was not perfectly balanced.

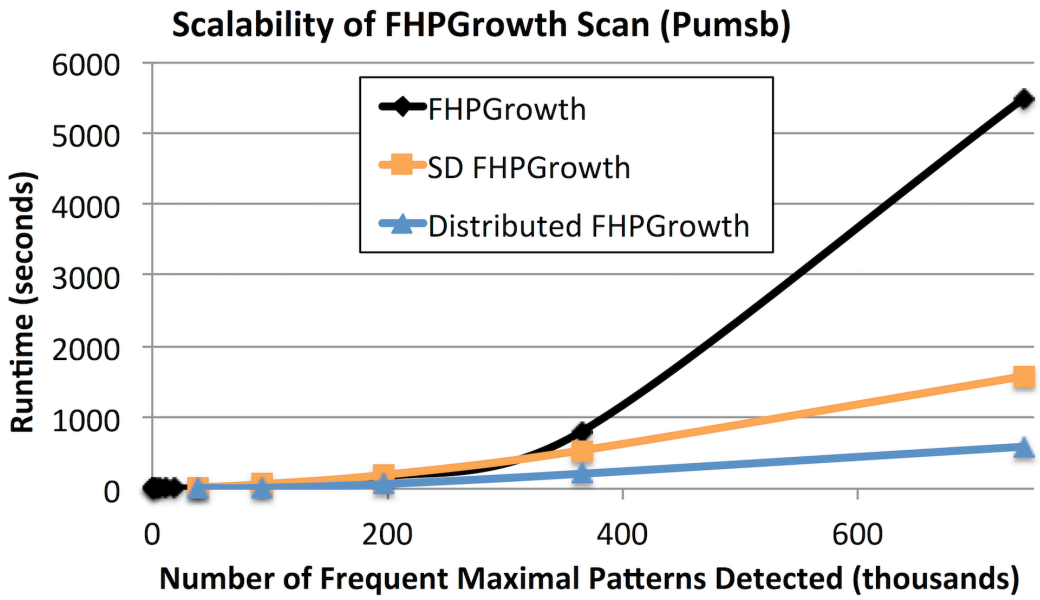

Figure 4.9: A comparison between the single server FHPGrowth and distributed FHPGrowth relative to runtime and number of pattern results on the pumsb dataset.

The final dataset we consider is mushroom, which is much more sparse in comparison to chess, connect, and pumsb. As a result, the original FHPTree was not efficient. This is because many traversal paths lead to patterns that do not meet the minimum support threshold. The SD approach is able to significantly reduce this effect, offering a 3*x* speedup for *min_support* < 0.5%. The distributed approach consistently offered an 8*x* speedup over the SD approach.

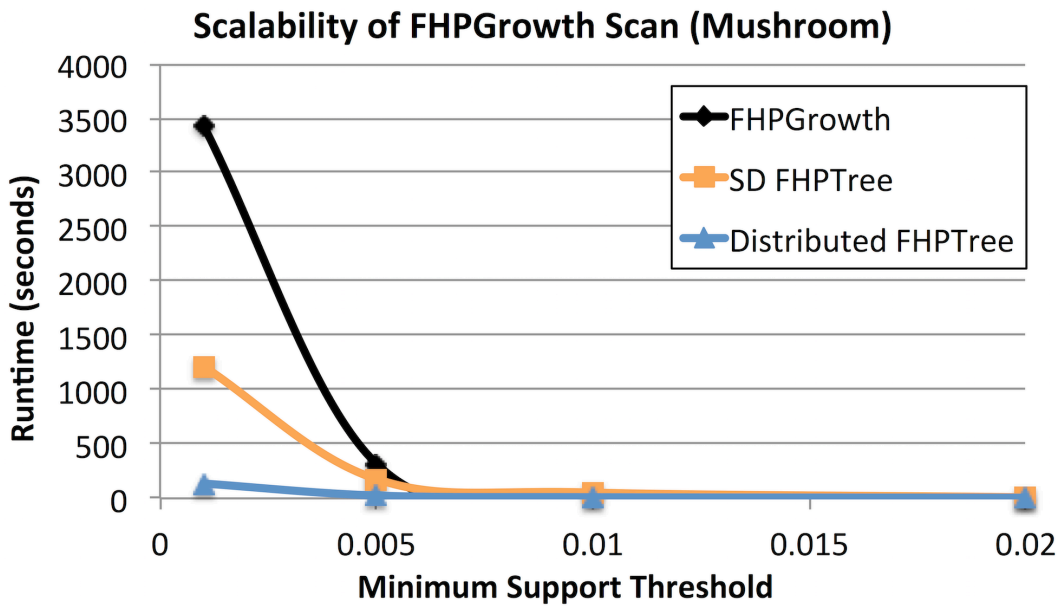

Figure 4.10: A comparison between the single server FHPGrowth and distributed FHP-Growth relative to runtime and minimum support threshold on the mushroom dataset.

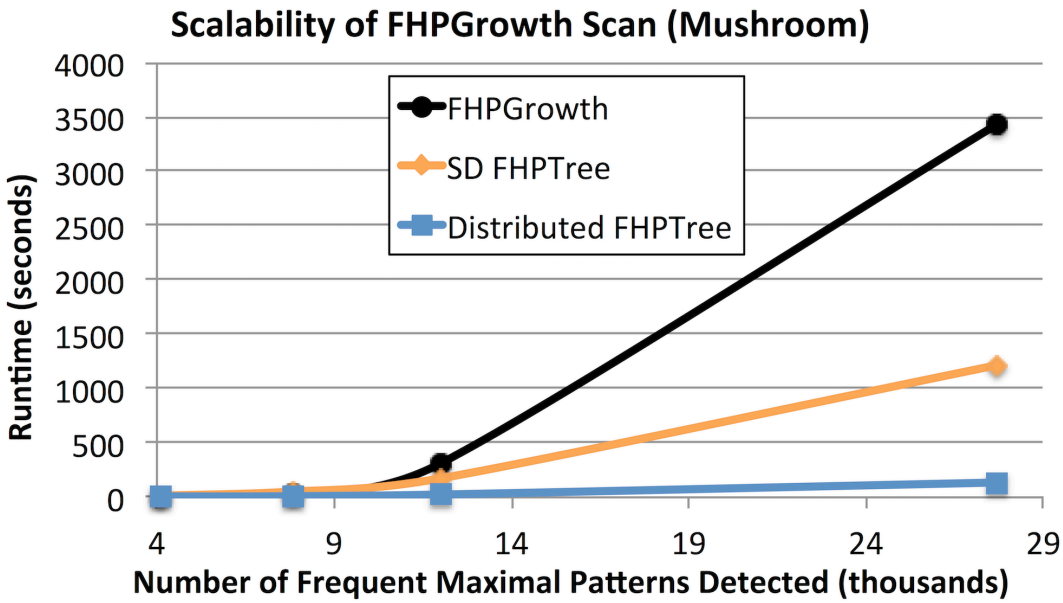

Figure 4.11: A comparison between the single server FHPGrowth and distributed FHP-Growth relative to runtime and number of pattern results on the mushroom dataset.

4.2.3 Search

We evaluate the performance of the distributed search operation using the chess dataset. In this experiment, our goal is to find the maximal frequent itemsets containing an item of interest. Since the search time varies significantly for dif-

ferent items, we consider the maximum, minimum, and median runtimes when characterizing the performance.

As shown in Figure 4.12, the maximum time taken during search is always less than that of the full scan. This is intuitive as there are fewer results returned in a targeted search. The median search time is consistently more than 4*x* faster than the full scan. The minimum runtime will be achieved when searching for the rarest items. These rarest items took less than 1 second for all minimum support thresholds we tested, and at *min_support* = 10%, this search returns thousands of maximal patterns. These results are consistent with that of the single server approach shown in Figure 3.18

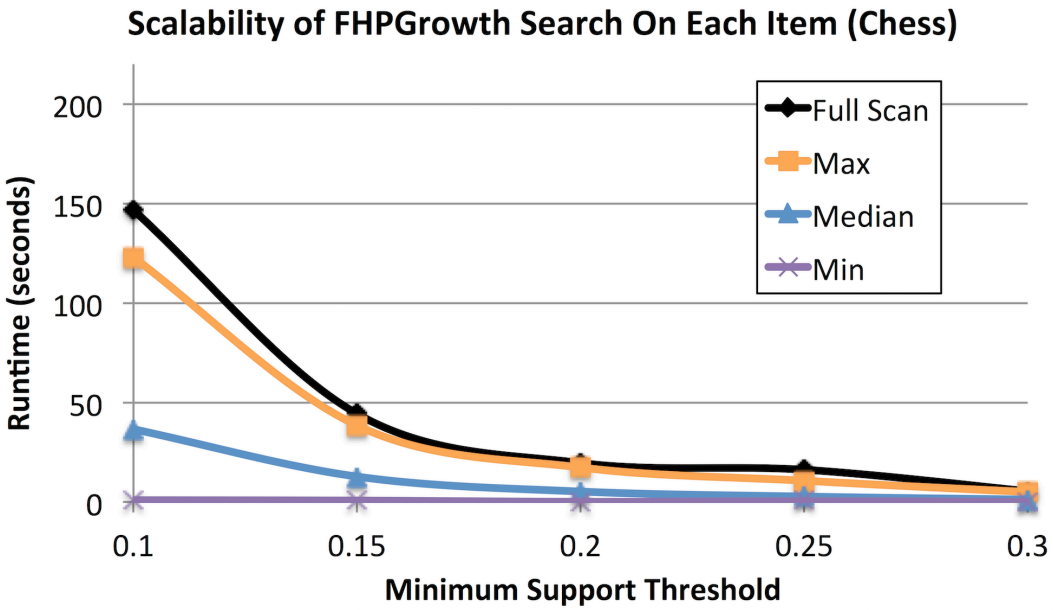

Figure 4.12: An analysis comparing targeted search maximum, minimum, and median runtimes with the full scan.

4.3 Conclusion and Future Work

In this chapter, we present an iterative search and delete process for the FHP-Growth algorithm that offered up to a 20x speed up over the original approach in

a single server environment. In addition, we discuss how this can used to parallelize the FHPGrowth traversal, porting the technology to distributed computing environments The distributed computing extension consistently outperformed the single node implementations by notable margins.. Our experiments demonstrated that an 8 node computing cluster with a balanced workload can achieve up to an 8*x* speed up over the single server environment. In total, we offered up to a 160*x* speedup over the original FHPGrowth algorithm. Furthermore, since the original traversal scheme was significantly faster than competing algorithms in the previous chapter, the evidence to support the FHPTree paradigm is clear and strong.

Through our experiments, we discovered that the density of a dataset and number of items it contains determines the effectiveness of the SD approach. At times when the number of maximal patterns remains small and the dataset is dense, the original traversal may be more efficient. However, in cases where there are hundreds of thousands of maximal patterns to be detected, the SD approach can offer improvements. In every case, the distribution strategy proved effective and the workload was distributed across several machines while yielding comprehensive results. Through our horizontal scalability experiments, we have shown that by increasing the resources in the computing cluster, the runtime is reduced.

At the core of this research, the key is the targeted search feature that is offered by the FHPTree. The ability to target broad or specific regions of the search space allows efficient retrieval of relevant information. Moreover, we can use this strategy to partition the original traversal into a collection of more efficient

traversals. Those traversals can be parallelized and distributed across multicore and cluster computing environments.

It is important to note that any improvements to the core FHPTree directly improve both of the methods detailed in this chapter. For example, in the previous chapter, we discussed several areas where the FHPTree may be improved such as defining a more effective similarity measure when building the tree. The results presented in this dissertation foreshadow a variety of useful applications for the FHPTree paradigm. Additional extension on the FHPTree include a generalization to sequential pattern mining, contrast mining, handling uncertainty data, among many others. This work demonstrates that further extensions may also be effective in distributed computing environments.

Chapter 5

Conclusion

Applications of frequent pattern mining and association rule mining are abundant and in diverse research domains. In this dissertation, we emphasize the complexity of the pattern mining problem and demonstrate the computational need for innovative algorithms. Our initial studies utilize classic algorithms, such as Apriori, on distributed computing environments in order to acquire the amount of computational resources necessary to quickly extract patterns. We presented a scheduling method, Cartesian Scheduler, to optimize the Cartesian operations on distributed datasets and improve the performance of the self-join operation embedded in the distributed Apriori algorithm. Next, we reconsidered the problem of frequent pattern mining and proposed a novel paradigm, frequent hierarchical pattern mining. The FHPTree is the persistent, dynamic data structure at the core of this paradigm; it provides targeted search capabilities that were previously not possible using classic approaches. FHPGrowth is the top-down traversal algorithm that offer significant improvements over the classic approaches and lends itself well to distributed computing environments. We also offered a distributed FHPGrowth technique that offered significant runtime improvements over the single server solution. The remainder of this chap-

ter provides more details about the contributions made with these technologies.

5.1 Distributed Cartesian Operations and The Apriori Algorithm

In chapter 2, we worked to optimize the bottleneck associated with the Apriori algorithm, the self-join. This classic algorithm utilizes a Cartesian product (CP) to build larger itemsets. The philosophy we adopted was to precompute and execute all shuffle operations simultaneously as a preprocessing step, eliminating continual network communication and leaving the remaining time for uninterrupted computation. As a result, we must introduce redundant copies of data to ensure that every worker node has its own copy of the necessary data. However, data redundancy poses the challenge of preventing redundant comparisons in the CP. In this work, we proposed virtual partitioning and the virtual partition pairing protocol to manage the degree of redundancy while guaranteeing that no redundant computation is performed.

Virtual partitioning is a variable grouping paradigm we proposed that gives control over the granularity of the partial CPs. A virtual partition (VP) functions as an irreducible building block for partial CPs, so redundant copies of VPs are created and copied to relevant compute nodes. Since partial CPs are performed between VPs, the size and number of partial CPs is managed by the VP size. This is valuable since the size of each partial CP affects how well the hardware can execute the instructions. The virtual partition pairing protocol preprocesses and schedules all of the partial CPs necessary to be equivalent to a global Cartesian product. This protocol facilitates the introduction of redundancy while guar-

anteeing that no comparisons are redundant. By construction, the protocol prevents redundant comparisons, so additional filtering or duplicate checks are not necessary.

Limitations for this approach include the automatic selection of the sharding factor. We demonstrated the importance of the sharding factor in determining the overall performance of a distributed CP. In our experiments, we were able to achieve up to a 40x speedup when compared to Spark on a small commodity cluster. When the comparison was made on a high performance cluster, the advantage becomes less drastic, achieving a 2x speedup over the classic approach. In addition, we demonstrated how well the Cartesian Scheduler handles heterogeneous data by achieving a balanced workload, which is common in the Apriori algorithm.

5.2 Frequent Hierarchical Pattern Mining

In this chapter, we proposed the FHPTree, a hierarchical cluster tree of items, and FHPGrowth, a top-down mining scheme for extracting frequent patterns. The number of nodes required for the FHPTree scales linearly as the number of distinct items. Furthermore, we achieved a 10-fold reduction in the memory footprint over the FPTree. In addition, for reoccurring pattern mining analyses, utilizing a persistent data structure reduces redundant computation. Since the FHPTree supports insert, update, and delete operations, it is not necessary to continually rebuild before each analysis or when the transaction database is updated. FHPGrowth was competitive when compared to existing state-of-the-art approaches, CHARM-MFI and FPMax. FHPGrowth outperformed both ap-

proaches on dense datasets. In addition, the search operation enables targeted pattern mining analyses to be conducted efficiently. The median runtime for search was a dramatic reduction in runtime compared to a full scan.

Limitations for the FHPTree revolve around sparse data. We discussed scenarios to avoid when building the tree; however, at times those situations may not be avoidable. Using new correlation metrics to build the FHPTree could help to further alleviate these concerns. Experimental results were promising and a testament to the frequent hierarchical pattern mining paradigm. Additionally, we conjectured several potential improvements to the FHPTree structure, FHP-Tree construction process, FHPGrowth Scan, and FHPGrowth Search.

5.3 Distributed Frequent Hierarchical Pattern Mining

We presented an iterative search and delete process for the FHPGrowth algorithm that offered up to a 20*x* speed up over the original FHPGrowth approach in a single server environment. We also discussed how this can be used to parallelize the FHPGrowth traversal, porting the technology to distributed computing environments The distributed computing extension consistently outperformed the single node implementations by notable margins.. Our experiments demonstrated that an 8 node computing cluster with a balanced workload can achieve up to an 8*x* speed up over the single server environment. In total, we offered up to a 160*x* speedup over the original FHPGrowth algorithm and a 2400*x* speedup over the classic FPMax.

Limitations for this approach are similar to that of the single server approach; sparse datasets continue to be a challenge. Through our experiments, we discov-

ered that the density of a dataset and number of items it contains determines the effectiveness of the search and delete approach. At times when the number of maximal patterns remains small and the dataset is dense, the original traversal may be more efficient. However, in cases where there are hundreds of thousands of maximal patterns to be detected, the search and delete approach can offer improvements. In every case, the parallelization strategy proved effective and the workload was distributed across several machines while yielding comprehensive results. Through our horizontal scalability experiments, we showed that by increasing the resources in the computing cluster, the runtime was reduced.

At the core of this research, the key is the targeted search feature that is offered by the FHPTree. The ability to target broad or specific regions of the search space allows efficient retrieval of relevant information. Moreover, we can use this strategy to partition the original traversal into a collection of more efficient traversals. Those traversals can be parallelized and distributed across multicore and cluster computing environments.

5.4 Contributions in Computer Science and Applications in Biomedicine

The main contribution of this dissertation is the frequent hierarchical pattern mining paradigm. Classic data structures used for frequent pattern mining were not well suited to serve as persistent, dynamic indexes for frequent pattern data, as they do not provide targeted search capabilities like FHPGrowth can. The results presented in this dissertation foreshadow a variety of useful applications

for the FHPTree paradigm. The frequent hierarchical pattern mining paradigm will serve as a catalyst for deep, targeted associative mining analyses.

The FHPTree achieved massive performance improvements over state-of-the-art approaches, and the memory footprint was smaller than existing data structures. The approaches offered in this dissertation generate a variety of research opportunities in the form of extensions into similar pattern mining domains. The generality of the frequent pattern mining problem suggests that the FHPTree paradigm may have a broad impact on related data mining areas.

Several other research projects were instrumental to this research process; however, they are not detailed in this dissertation. First, a biological application of frequent pattern mining was explored. The high-level goal was to extract repetitive DNA sequences from massive genomic sequence datasets using the Apache Hadoop MapReduce distributed computing framework [81, 82]. Applications of contrast mining in the medical domains were also explored [83, 84]. Big data technologies were employed for these studies as well to promote scalability, as EMR data continues to increase in volume and variety.

5.5 Limitations and Future Work

The methods and discussions presented in this dissertation open up a variety of research opportunities. Regarding Cartesian operations, the automatic selection of the sharding factor will alleviate a burden from developers and improve performance across the board. Such improvements will directly impact the performance of the distributed Apriori algorithm presented in chapter 2.

As with Apriori and FPGrowth, a common next step is to seek generalizations,

extensions, and optimizations in sequential pattern mining, high-utility pattern mining, uncertain datasets, streaming data, GPU architectures, and many other areas of research. The FHPTree may offer advantages in these areas as well. These extensions and generalizations will require additional research as the FHPTree may not be ready out of the box. For example, sequential pattern mining requires the the order of items to be tracked where the order of items is not considered in frequent pattern mining.

Several limitations were discussed, which could also create potential research opportunities. Addressing the limitations of the FHPTree in terms of sparse datasets would make FHPGrowth more generally applicable to arbitrary frequent pattern mining analyses.

References

[1] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[2] C. Silverstein, S. Brin, and R. Motwani, "Beyond market baskets: Generalizing association rules to dependence rules," *Data mining and knowledge discovery*, vol. 2, no. 1, pp. 39–68, 1998.

[3] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," in *ACM SIGMOD Record*, vol. 26, pp. 255–264, ACM, 1997.

[4] S. E. Brossette, A. P. Sprague, J. M. Hardin, K. B. Waites, W. T. Jones, and S. A. Moser, "Association rules and data mining in hospital infection control and public health surveillance," *Journal of the American medical informatics association*, vol. 5, no. 4, pp. 373–381, 1998.

[5] C. Ordonez, N. Ezquerra, and C. A. Santana, "Constraining and summarizing association rules in medical data," *Knowledge and Information Systems*, vol. 9, no. 3, pp. 1–2, 2006.

[6] A. Wright, E. S. Chen, and F. L. Maloney, "An automated technique for identifying associations between medications, laboratory results and

problems," *Journal of biomedical informatics*, vol. 43, no. 6, pp. 891–901, 2010.

[7] J. Li, A. W.-c. Fu, H. He, J. Chen, H. Jin, D. McAullay, G. Williams, R. Sparks, and C. Kelman, "Mining risk patterns in medical data," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pp. 770–775, ACM, 2005.

[8] J. Nahar, T. Imam, K. S. Tickle, and Y.-P. P. Chen, "Association rule mining to detect factors which contribute to heart disease in males and females," *Expert Systems with Applications*, vol. 40, no. 4, pp. 1086–1093, 2013.

[9] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Acm sigmod record*, vol. 22, pp. 207–216, ACM, 1993.

[10] K. Ishimoto, "Incremental mining of constrained association rules," in *Proceedings of the 2001 SIAM International Conference on Data Mining*, Society for Industrial and Applied Mathematics, 2001.

[11] J. Hipp, U. Güntzer, and G. Nakhaeizadeh, "Algorithms for association rule mining?a general survey and comparison," *ACM sigkdd explorations newsletter*, vol. 2, no. 1, pp. 58–64, 2000.

[12] R. Agrawal, R. Srikant, *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, pp. 487–499, 1994.

[13] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data mining and knowledge discovery*, vol. 8, no. 1, pp. 53–87, 2004.

[14] M. Houtsma and A. Swami, "Set-oriented mining for association rules in relational databases," in *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pp. 25–33, IEEE, 1995.

[15] J. S. Park, M.-S. Chen, and P. S. Yu, *An effective hash-based algorithm for mining association rules*, vol. 24. ACM, 1995.

[16] M. J. Zaki and C.-J. Hsiao, "Charm: An efficient algorithm for closed itemset mining.," in *SDM*, vol. 2, pp. 457–473, SIAM, 2002.

[17] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Efficient mining of association rules using closed itemset lattices," *Information systems*, vol. 24, no. 1, pp. 25–46, 1999.

[18] Y. Aumann, R. Feldman, O. Lipshtat, and H. Manilla, "Borders: An efficient algorithm for association generation in dynamic databases," *Journal of Intelligent Information Systems*, vol. 12, no. 1, pp. 61–73, 1999.

[19] D. Burdick, M. Calimlim, and J. Gehrke, "Mafia: A maximal frequent itemset algorithm for transactional databases," in *Data Engineering, 2001. Proceedings. 17th International Conference on*, pp. 443–452, IEEE, 2001.

[20] A. Sharma and N. Tivari, "A survey of association rule mining using genetic algorithm," *Int J Comput Appl Inf Technol*, vol. 1, pp. 5–11, 2012.

[21] J. Alcalá-Fdez, R. Alcalá, M. J. Gacto, and F. Herrera, "Learning the membership function contexts for mining fuzzy association rules by using genetic algorithms," *Fuzzy Sets and Systems*, vol. 160, no. 7, pp. 905–921, 2009.

[22] F. Pan, G. Cong, A. K. Tung, J. Yang, and M. J. Zaki, "Carpenter: Finding closed patterns in long biological datasets," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 637–642, ACM, 2003.

[23] B. Nair and A. K. Tripathy, "Accelerating closed frequent itemset mining by elimination of null transactions," *Journal of Emerging Trends in Computing and Information Sciences*, vol. 2, no. 7, pp. 317–324, 2011.

[24] A. J. Lee, W.-K. Tsao, P.-Y. Chen, M.-C. Lin, and S.-H. Yang, "Mining frequent closed patterns in pointset databases," *Information Systems*, vol. 35, no. 3, pp. 335–351, 2010.

[25] N. Li, L. Zeng, Q. He, and Z. Shi, "Parallel implementation of apriori algorithm based on mapreduce," in *Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), 2012 13th ACIS International Conference on*, pp. 236–241, IEEE, 2012.

[26] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, "Apriori-based frequent itemset mining algorithms on mapreduce," in *Proceedings of the 6th international conference on ubiquitous information management and communication*, p. 76, ACM, 2012.

[27] S. Rathee, M. Kaul, and A. Kashyap, "R-apriori: an efficient apriori based algorithm on spark," in *Proceedings of the 8th Workshop on Ph. D. Workshop in Information and Knowledge Management*, pp. 27–34, ACM, 2015.

[28] H. Qiu, R. Gu, C. Yuan, and Y. Huang, "Yafim: a parallel frequent itemset mining algorithm with spark," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pp. 1664–1671, IEEE, 2014.

[29] X. Yan and J. Han, "Closegraph: mining closed frequent graph patterns," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 286–295, ACM, 2003.

[30] G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using fp-trees," *IEEE transactions on knowledge and data engineering*, vol. 17, no. 10, pp. 1347–1362, 2005.

[31] G. Grahne and J. Zhu, "Efficiently using prefix-trees in mining frequent itemsets.," in *FIMI*, vol. 90, 2003.

[32] W. Cheung and O. R. Zaiane, "Incremental mining of frequent patterns without candidate generation or support constraint," in *Database Engineering and Applications Symposium, 2003. Proceedings. Seventh International*, pp. 111–116, IEEE, 2003.

[33] I. Pramudiono and M. Kitsuregawa, "Fp-tax: Tree structure based generalized association rule mining," in *Proceedings of the 9th ACM SIGMOD*

workshop on Research issues in data mining and knowledge discovery, pp. 60–63, ACM, 2004.

[34] Y. Qiu, Y.-J. Lan, and Q.-S. Xie, "An improved algorithm of mining from fp-tree," in *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, vol. 3, pp. 1665–1670, IEEE, 2004.

[35] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "Pfp: parallel fp-growth for query recommendation," in *Proceedings of the 2008 ACM conference on Recommender systems*, pp. 107–114, ACM, 2008.

[36] P. Comninos, *Mathematical and computer programming techniques for computer graphics*. Springer Science & Business Media, 2010.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, p. 10, 2010.

[38] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, USENIX Association, 2012.

[39] U. Kumar and J. Kumar, "A comprehensive review of straggler handling algorithms for mapreduce framework," *International Journal of Grid and Distributed Computing*, vol. 7, no. 4, pp. 139–148, 2014.

[40] M. Raab and A. Steger, "Òballs into binsÓÑa simple and tight analysis," in *Randomization and Approximation Techniques in Computer Science*, pp. 159–170, Springer, 1998.

[41] C. J. Date and H. Darwen, *A Guide To Sql Standard*, vol. 3. Addison-Wesley Reading, 1997.

[42] M. W. Berry, Z. Drmac, and E. R. Jessup, "Matrices, vector spaces, and information retrieval," *SIAM review*, vol. 41, no. 2, pp. 335–362, 1999.

[43] F. Holzschuher and R. Peinl, "Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pp. 195–204, ACM, 2013.

[44] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," in *International semantic web conference*, vol. 4273, pp. 30–43, Springer, 2006.

[45] J. Clark, S. DeRose, *et al.*, "Xml path language (xpath) version 1.0," 1999.

[46] C. Tang, Z. Xu, and S. Dwarkadas, "Peer-to-peer information retrieval using self-organizing semantic overlay networks," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 175–186, ACM, 2003.

[47] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, *et al.*, "F1: A distributed sql database that scales," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1068–1079, 2013.

[48] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394, ACM, 2015.

[49] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[50] T. Elsayed, J. Lin, and D. W. Oard, "Pairwise document similarity in large collections with mapreduce," in *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, pp. 265–268, Association for Computational Linguistics, 2008.

[51] J. Lin, "Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce," in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pp. 155–162, ACM, 2009.

[52] F. Crestani, M. Lalmas, C. J. Van Rijsbergen, and I. Campbell, "Òis this document relevant?É probablyÓ: a survey of probabilistic models in information retrieval," *ACM Computing Surveys (CSUR)*, vol. 30, no. 4, pp. 528–552, 1998.

[53] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey, "Scatter/gather: A cluster-based approach to browsing large document collec-

tions," in *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 318–329, ACM, 1992.

[54] "Reuters-21578." http://www.daviddlewis.com/resources/testcollections/reuters2157 Accessed: 2015-10-02.

[55] M. Phinney, S. Lander, M. Spencer, and C.-R. Shyu, "Cartesian operations on distributed datasets using virtual partitioning," in *IEEE International Conference on Big Data Computing Service and Applications*, pp. 1–8, IEEE, 2016.

[56] "Cartesian scheduler gitlab source code repository." https://gitlab.com/idas-lab/CartesianScheduler.git. Accessed: 2017-3-20.

[57] C. C. Aggarwal and J. Han, *Frequent pattern mining*. Springer, 2014.

[58] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *Data Mining and Knowledge Discovery*, vol. 15, no. 1, pp. 55–86, 2007.

[59] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *Int'l Conf. on Extending Database Technology*, pp. 1–17, Springer, 1996.

[60] M. J. Zaki and K. Gouda, "Fast vertical mining using diffsets," in *Proc. of the 9th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pp. 326–335, ACM, 2003.

[61] R. J. Bayardo Jr, “Efficiently mining long patterns from databases,” *ACM Sigmod Record*, vol. 27, no. 2, pp. 85–93, 1998.

[62] R. C. Agarwal, C. C. Aggarwal, and V. Prasad, “Depth first generation of long patterns,” in *Proc. of the 6th ACM SIGKDD Int’l Conf. on Knowledge Discovery and Data Mining*, pp. 108–118, ACM, 2000.

[63] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu, “Mafia: A maximal frequent itemset algorithm,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 11, pp. 1490–1504, 2005.

[64] L. Szathmary, *Symbolic Data Mining methods with the Coron platform*. PhD thesis, Université Henri Poincaré-Nancy I, 2006.

[65] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz, “Catch the moment: maintaining closed frequent itemsets over a data stream sliding window,” *Knowledge and Information Systems*, vol. 10, no. 3, pp. 265–294, 2006.

[66] R. Chan, Q. Yang, and Y.-D. Shen, “Mining high utility itemsets,” in *3rd IEEE Int’l Conf. on Data Mining (ICDM)*, pp. 19–26, IEEE, 2003.

[67] T. Bernecker, H.-P. Kriegel, M. Renz, F. Verhein, and A. Zuefle, “Probabilistic frequent itemset mining in uncertain databases,” in *Proc. of the 15th ACM SIGKDD Int’l Conf. on Knowledge Discovery and Data Mining*, pp. 119–128, ACM, 2009.

[68] D. W. Cheung, J. Han, V. T. Ng, A. W. Fu, and Y. Fu, “A fast distributed algorithm for mining association rules,” in *4th Int’l Conf. on Parallel and Distributed Information Systems*, pp. 31–42, IEEE, 1996.

[69] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "Mining sequential patterns by pattern-growth: The prefixspan approach," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 11, pp. 1424–1440, 2004.

[70] J. Wang, J. Han, Y. Lu, and P. Tzvetkov, "Tfp: An efficient algorithm for mining top-k frequent closed itemsets," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 5, pp. 652–663, 2005.

[71] V. S. Tseng, C.-W. Wu, B.-E. Shie, and P. S. Yu, "Up-growth: an efficient algorithm for high utility itemset mining," in *Proc. of the 16th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pp. 253–262, ACM, 2010.

[72] V. S. Tseng, C.-W. Wu, P. Fournier-Viger, and P. S. Yu, "Efficient algorithms for mining top-k high utility itemsets," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 1, pp. 54–67, 2016.

[73] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu, "Mining frequent patterns in data streams at multiple time granularities," *Next generation Data Mining*, pp. 191–212, 2003.

[74] C. C. Aggarwal, Y. Li, J. Wang, and J. Wang, "Frequent pattern mining with uncertain data," in *Proc. of the 15th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pp. 29–38, ACM, 2009.

[75] Y. Tong, L. Chen, and P. S. Yu, "Ufimt: an uncertain frequent itemset mining toolbox," in *Proc. of the 18th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pp. 1508–1511, ACM, 2012.

[76] Y. Tong, L. Chen, Y. Cheng, and P. S. Yu, "Mining frequent itemsets over uncertain databases," *Proc. of the VLDB Endowment*, vol. 5, no. 11, pp. 1650–1661, 2012.

[77] H. Liu, J. Han, D. Xin, and Z. Shao, "Mining frequent patterns from very high dimensional data: A top-down row enumeration approach," in *Proc. of the 2006 SIAM Int'l Conf. on Data Mining*, pp. 282–293, SIAM, 2006.

[78] Y. Xie and P. S. Yu, "Max-clique: A top-down graph-based approach to frequent pattern mining," in *IEEE 10th Int'l Conf. on Data Mining (ICDM)*, pp. 1139–1144, IEEE, 2010.

[79] O. Maimon and L. Rokach, *Data Mining and Knowledge Discovery handbook*, vol. 2. Springer, 2005.

[80] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C.-W. Wu, V. S. Tseng, *et al.*, "Spmf: a java open-source pattern mining library.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3389–3393, 2014.

[81] H. Cao, M. Phinney, D. Petersohn, B. Merideth, and C.-R. Shyu, "Mining large-scale repetitive sequences in a mapreduce setting," *International Journal of Data Mining and Bioinformatics*, vol. 14, no. 3, pp. 210–228, 2016.

[82] H. Cao, M. Phinney, D. Petersohn, B. Merideth, and C.-R. Shyu, "Mrsmrs: Mining repetitive sequences in a mapreduce setting," in *Bioinformatics and Biomedicine (BIBM), 2014 IEEE International Conference on*, pp. 463–470, IEEE, 2014.

[83] L. Sheets, M. A. Phinney, S. Lander, J. C. Parker, and C. Shyu, "Data mining to predict healthcare utilization in managed care patients," in *AMIA 2015, American Medical Informatics Association Annual Symposium, San Francisco, CA, USA, November 14-18, 2015*, AMIA, 2015.

[84] M. A. Phinney, Y. Zhuang, S. Lander, L. Sheets, J. C. Parker, and C. Shyu, "Contrast mining for pattern discovery and descriptive analytics to tailor sub-groups of patients using big data solutions," in *MedInfo*, 2017.

Vita

Michael Phinney received both his Ph.D. degree in Computer Science (in May 2017) and his M.S. degree in Computer Science (in December 2015) from the University of Missouri-Columbia. He received dual B.S. degrees in Computer Science and Mathematics from the University of Central Missouri in May 2012.

Since 2012, he has worked as a Graduate Research Assistant in the Center for Interdisciplinary Data Analytics and Search (iDAS) under the direction of Dr. Chi-Ren Shyu at the University of Missouri. Mike was the recipient of the U.S. Department of Education Graduate Assistantship in Areas of National Need (GAANN) Fellowship which supported his education from 2012 to 2017. Mike received the 2017 University of Missouri Outstanding Computer Science PhD Student Award. In 2013, his Big Data project won the IBM Smarter Planet Big Data Student Project Award. He instructed a required undergraduate computer science course, CS3380: Database Applications and Information Systems, where he also managed three undergraduate teaching assistants. Over the last two years of Mike's degree, while completing his own Ph.D. program requirements, he mentored seven undergraduate researchers. Mike has also served as an advisory board member for the University of Central Missouri's Mathematics and Computer Science Department since 2013.

During his graduate studies at the University of Missouri, his research foci

were frequent pattern mining, data mining and analytics, big data technologies, distributed computing, and algorithm design. In addition, Mike applies his theoretical computer science research to biomedical domains, such as deep genomic sequence analysis across a large number of genomes and healthcare data mining for a project supported by the Centers for Medicare and Medicaid Services. In his dissertation, he proposed a novel frequent pattern mining paradigm that offered significant performance improvements over prior solutions. These works resulted in several publications, conference posters, and presentations.