2004 年度修士論文

PC クラスタ上における
頻出飽和パターン抽出並列化手法の提案

提出日:2005 年 2 月 2 日

指導：山名早人助教授

早稲田大学大学院理工学研究科

情報・ネットワーク専攻

学籍番号：3603u0167

岩橋永悟

概 要

近年，ネットワーク環境の整備，記憶装置の低価格化・大容量化にともなって，データの洪水化が進んでいる．そこで，大規模なデータから有用な知識を抽出するために，データマイニング技術が注目を集めている．

データマイニング分野における頻出パターン抽出問題では，大規模なデータに対して処理を行うため，メモリ容量不足やディスクアクセス増加といった問題に直面する．このようなリソース面の制約を緩め，現実時間で頻出パターンを抽出するために，様々な並列化手法が提案されている．しかし，従来の並列化手法の多くは全ての頻出パターンを抽出するため，結果として莫大な数のパターンが抽出されてしまう．大規模なデータに対しても，ユーザにとって解析する負担が少ない，冗長性を軽減したパターンを高速に提示する並列化手法が必要である．そこで，本稿では，FPclose に基づき頻出飽和アイテム集合を並列抽出する手法を提案する．さらに，並列化において問題となる，タスク負荷の偏りを平坦化する手法を提案する．提案した手法を PC クラスタ上で実装し，評価を行った結果，最小サポートを 2%と設定した場合，32PU 投入時に 30.9 倍の速度向上を得た．

目 次

第1章　はじめに

近年，ネットワーク環境の整備，記憶装置の低価格化が進むにつれて，大量のデータを蓄積することが可能となった．例えば，コンビニエンスストアの売り上げデータや，株価に代表される経済データが，活用するしないに関わらず，ネットワークを経由して大量に蓄積されている [1]．しかし，データは記号の列にすぎず，データから情報を見出すのは本来ユーザの仕事である．無秩序に集められた大規模なデータに対してユーザが目を通すのは不可能であるため，大規模なデータから有用な知識を抽出するデータマイニング技術が注目されている．

データマイニング技術の重要な問題として相関ルールがある．相関ルールとは，複数のアイテムまたはアイテムセット間の相関関係を表すものである．相関ルールを利用したアプリケーションとしては，スーパーなどの買い物データから併売パターンを抽出するバスケット解析，Web ログからユーザの行動パターンを抽出する Web ログ解析などが挙げられる．

相関ルール抽出問題はデータベースからアイテムセットの出現頻度を数える問題（**頻出パターン抽出問題**）に置き換えられる．そこで，データマイニング分野においては，巨大なデータベースに対してアイテムセットの出現頻度を効率よく数え上げる手法の研究が行われている．

頻出パターン抽出手法としては，Apriori[16] や FP-growth[9] が有名である．これらの手法では，抽出されるパターン数が膨大となり，結果が冗長になってしまうという問題がある．この問題点を解決するために，パターンの冗長性を削減した極大頻出パターンや飽和頻出パターンを抽出する手法 [10][6] が提案されている．

一方で，頻出パターン抽出では大規模なデータに対して処理を行うため，メモリ容量不足やディスクアクセス増加といった問題に直面する．このようなリソース面の制約を緩め，現実時間で頻出パターンを抽出するために，PC クラスタなどをターゲットとした並列化手法が提案されている．

従来提案されてきた並列化手法の多くは，Apriori をベースとする手法である [17][20]．最近では，FP-growth をベースとする並列化手法も提案されている [8]．しかし，これら

の並列化手法では，全ての頻出パターンを抽出するため，結果として莫大な数のパターンが抽出され，ユーザに負担が掛かるという問題がある．また，頻出パターン並列抽出処理においては，トランザクションデータベース中に出現するアイテムの特性により，ノード毎でタスク負荷が偏ってしまい，結果として並列処理効率が低下するという問題がある．

本論文では，ユーザにとって解析する負担が少ない，冗長性を軽減したパターンを高速に提示するために，2003 年 11 月時点で最速である FPclose をベースとして，飽和頻出パターンを並列抽出する手法を提案する．さらに，頻出パターン並列抽出において問題となる，タスク負荷の偏りを平坦化する手法を提案する．提案した手法を PC クラスタ上で実装し，性能評価を行なう．

本論文では，第 2 章で従来の頻出パターン抽出手法について，逐次アルゴリズムと並列アルゴリズムに分類して述べる．第 3 章では、FPclose アルゴリズムの並列化手法について述べる．第 4 章では，第 3 章で述べた手法の性能評価について述べる．第 5 章で，まとめをおこなう．

第2章　関連研究

第 2 章では相関ルール抽出問題について述べた後，相関ルール抽出の高速化に関連する研究について述べる．関連研究についてはシーケンシャルなアルゴリズムを紹介した後，並列アルゴリズムを紹介する．

2.1　相関ルール抽出問題

相関ルールは以下のように定義される．$I = \{i_1, i_2, \cdots, i_m\}$ をアイテムの集合とする．データベースを $D = \{t_1, t_2, \cdots, t_n\}(t_i \subseteq I)$ とする．各要素 t_i をトランザクションとよぶ．

相関ルールとは，$X \subseteq I$，$Y \subseteq I$，$X \cap Y = \emptyset$ であるような任意のアイテム集合 X，Y を使ってつくられる $X \Rightarrow Y$ というルールを示す．相関ルールは，サポート $sup(X \Rightarrow Y)$ と確信度 $conf(X \Rightarrow Y)$ の二つのパラメータを持つ．サポートは，トランザクションデータベース D 全体に対して X と Y をともに持つ割合 $sup(X \cup Y)$，確信度は，$sup(X \cup Y)/sup(X)$ と定義される．

相関ルール抽出問題はトランザクションデータベース D に対して，サポートの最小値および確信度の最小値が与えられたときに，これらを満足するルールを見つけ出すことである．ここで，ユーザが与えた最小サポート値以上のサポートを持つアイテムセットを頻出アイテムセット (frequent itemset) とよぶ．相関ルール抽出処理は，以下の 2 つのステップによって行われる．

ステップ 1 ユーザが与えた最小サポート値を満たすアイテムセット（頻出アイテムセット）を全て抽出する．

ステップ 2 得られた頻出アイテムセットから最小確信度を満たす相関ルールを得る．

ー

ステップ 2 はステップ 1 で求めた頻出アイテムセットを用いて相関ルールを導出する処理であり，比較的負荷が小さい処理である．一方，ステップ 1 は巨大なトランザクションデータベースに対して繰り返しスキャンを行い，アイテムセットのサポートを調査するた

め，処理時間の大半を占める．そこで相関ルール抽出アルゴリズムはステップ 1 を効率化することが主要課題である．この問題を**頻出パターン抽出問題**と呼ぶ．

相関ルール抽出問題の形式的な定義は，先述のとおりである．ここで，具体的に相関ルールが活用される例を挙げておく．表 2.1 はあるスーパーマーケットの購買データを蓄積したデータベース D である．

表 2.1: 入力データ例

データベース D

TID	Item
1000	A, B, C
1001	A, B, C, E
1002	B, D, E
1003	C, E
1004	A, B, C, D

表 2.1 において，各行がトランザクションを表す．TID が 1000 番の顧客は商品 A,B,C を購入し，TID が 1001 番の顧客が商品 A,B,C,E を購入したということを意味している．$X \Rightarrow Y$ という相関ルールの確信度が $c\%$ で，そのサポートが $s\%$ だとすると，商品の集合 X を購入した顧客のうち $c\%$ が商品の集合 Y も同時に購入していて，$X \cup Y$ を購入していた顧客の全体に対する割合は $s\%$ だったということである．

2.2　飽和頻出パターン抽出問題

従来の頻出パターン抽出問題では，結果として莫大な数のパターンが得られてしまう．莫大な数のパターンから，有用な知識を得ることは困難である．この問題点を解決するために，飽和頻出パターン（$CFI : ClosedFrequentItemset$）を抽出するアルゴリズムが提案されている [10][14]．パターン P が飽和パターンであるということは，以下の二つの条件を同時に満たす P' が存在しないことである．

1. P' が P のスーパーセットである．
2. P を含む全トランザクションで，P' も含まれる

飽和パターン P が，最小サポートを超えるサポート値を持っているとき，P は CFI である．

2.3　逐次頻出パターン抽出アルゴリズム

本節では，頻出パターン抽出を高速化する研究のなかでも，逐次頻出パターン抽出アルゴリズムの特徴と手順を紹介する．多くのアルゴリズムが水平データレイアウトを想定している．水平データレイアウトでは，一つのトランザクションに TID とアイテムが含まれている．一方，垂直データレイアウトを想定しているアルゴリズムもある．垂直データレイアウトでは，あるアイテム X を含む全ての TID を記録する”X についての tidlist”を利用する．

2.3.1　Apriori

IBM アルマデン研究所の Agrawal によって 1994 年に提案された Apriori アルゴリズムは，効率的にすべての頻出パターンを発見することができる手法である [16]．Apriori は，その後多くの派生的な改良を生んだベーシックなアルゴリズムである．Apriori はボトムアップに頻出アイテムセットを数え上げる．Apriori の手順を以下に示す．

まず，1 回目のデータベースのスキャンにおいて，全てのアイテムについてそれぞれサポート値を数え，アイテム数 1 の頻出アイテムセットを抽出する．次に，アイテム数 1 の頻出アイテムセットを組み合わせることにより，アイテム数 2 の候補アイテムセットを生成する．そして，2 回目のデータベースに対するスキャンにおいて，アイテム数 2 の候補アイテムセットのサポート値を数えるために，再びトランザクションデータをスキャンする．以降同様に，k 回目のデータベースのスキャンでは，アイテム数 k の候補アイテムセットについてトランザクションデータをスキャンすることにより，各アイテムセットのサポート値を算出し，アイテム数 k の頻出アイテムセットを抽出する．

Apriori での k 回目のデータベースのスキャン時（パス k）の処理（要素数 k の候補アイテムセットに対する処理）は次のようになる．ここで，パス k における候補アイテムセットを C_k，頻出アイテムセットを L_k とする．

1. データベースをスキャンして，C_k 中の各候補のサポートを調べる．
2. C_k 中の最小サポートを満足する部分を L_k とする．
3. L_k から候補 C_{k+1} を生成する．

候補アイテムセットがなくなるまで，上記の処理を続けることにより，全ての頻出アイテムセットを抽出する．表 2.2 のようなトランザクションデータが与えられたときに，2 回以上出現するアイテムセットを抽出するときの Apriori の動きを図 2.1 に示す.（最小サポート値=2）

表 2.2: トランザクション：Apriori の例

TID	Item
110	A, B, C, E
120	B, C
130	A, B, C
140	C, D, E

1 回目のスキャンでは各アイテムのサポートを数える．表 2.2 の例では，{D} は最小サポート値である 2 を満たさない．2 回目のスキャンでは {D} を除いた {A}, {B}, {C}, {E} から生成される要素数 2 の候補アイテムセットのサポートを求める．ここでは {A, E} と {B, E} が最小サポートを満たさないので頻出アイテムセットから除外される．3 回目のスキャンでは要素数 2 の頻出アイテムセットから生成できる候補アイテムセット {A, B, C} のサポートを数える．{A, B, C} のサポートは最小サポート値を満たすため，頻出アイテムセットである．要素数 3 のアイテムセットからは要素数 4 の候補アイテムセットを生成できないため，Apriori アルゴリズムはここで終了する．

Apriori アルゴリズムの欠点は，候補アイテムセットの数が莫大になるということである．仮に長さ 1 の頻出アイテムが 10^4 個あれば，Apriori アルゴリズムは長さ 2 の候補アイテムセットを 10^7 個生成する必要がある．また，候補アイテムセットをチェックするためにデータベースを繰り返しスキャンする必要がある．具体的には，データベース中の頻出アイテムセットを構成する要素数の最大値が k であるとすると，k 回のスキャンが必要である．

2.3.2　Dynamic Hashing Pruning

1997 年に IBM トーマス・J・ワトソン研究所の Park らによって提案された Dynamic Hashing Pruning(DHP) はハッシュ表を用いることによって，候補アイテムセット数を減

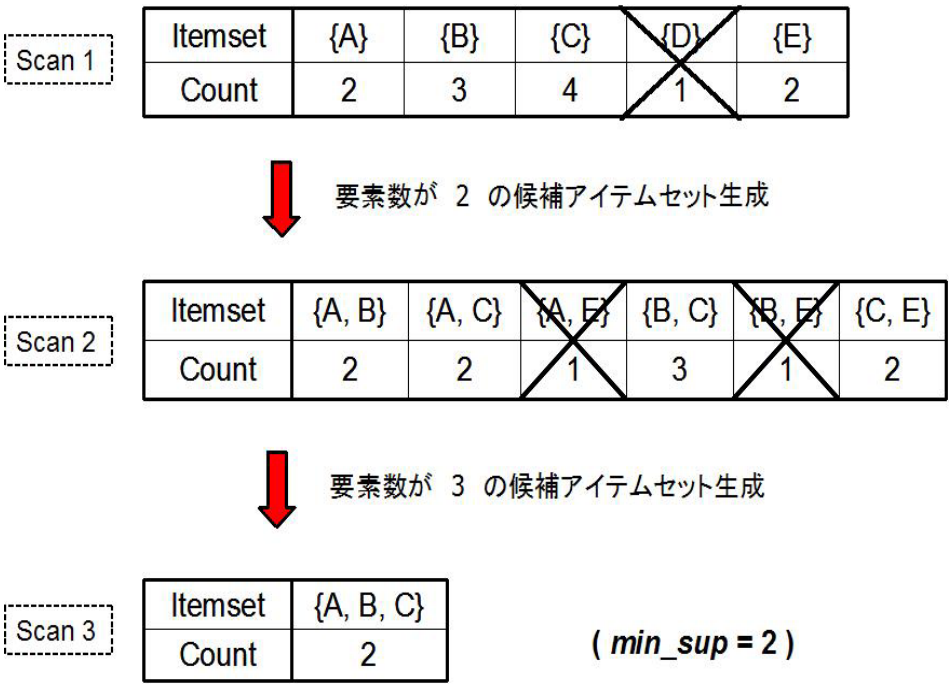

図 2.1: Apriori の動き

少させる手法である [12].

DHP では要素数 k のアイテムセットのサポートを調べる際に，各トランザクションから要素数 $k+1$ のアイテムセットを生成し，それらに対してハッシュ値を求める．各バケットにはハッシュされたアイテムセットの総数が格納されている．次のパスでは最小サポートを越えるバケットにハッシュされたアイテムセットのみを候補とする．DHP では，特に要素数 2 の候補アイテムセット数を減少させることができる．図 2.2 に，1 回目のスキャンが終了した時点でのハッシュテーブルを示す．最小サポート値 3 を満たさない値が格納されているバケットにハッシュされた {A, C} が，候補アイテムセットから除外される．

2.3.3　Sequential Efficient Association Rules

1995 年にメリーランド大学の Andreas Mueller によって提案された Sequential Efficient Association Rules Algorithm(SEAR)[2] は，候補アイテムセットをハッシュ木ではなく prefix tree に格納するという点を除いては Apriori と同じである．

prefix tree 構造では，各ノードが対応するアイテムセットの出現頻度を格納する．root

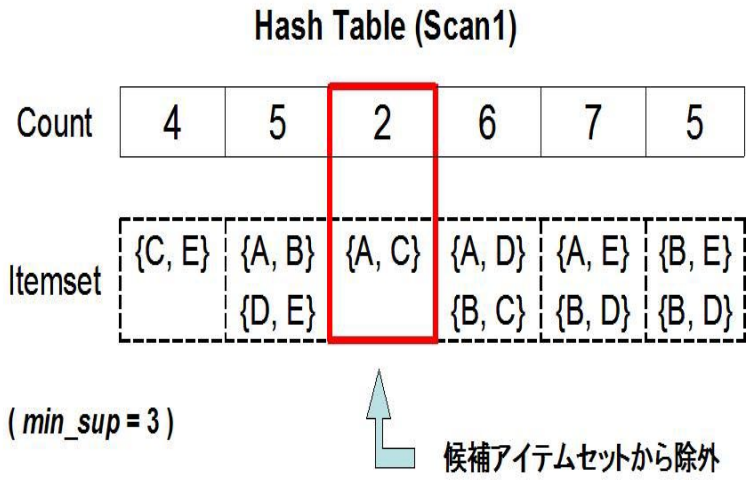

図 2.2: DHP における Scan1 終了時のハッシュテーブル

ノードは空アイテムセットを表す．全てのトランザクションは空アイテムセットを持つため，出現頻度はトランザクション数に等しい．候補セットは数えられるまでは 0 がカウントされている．図 2.3 に prefix tree の例を示す．図 2.3 の例においては全トランザクション数が 10 であり，出現するアイテムは A, B, C, D の 4 種類である．太字で描かれたパスは {A, C} というアイテムセットを表している．

ハッシュ木と異なる点は，prefix tree は頻出アイテムセットと候補アイテムセットが同じ木に格納されていることである．候補アイテムセットが数えられて頻出アイテムセットであることが決定すれば，候補アイテムセットは木においてその位置を変えることなく，頻出アイテムセットとなる．

Sequential Partitioning Efficient Association Rules(SPEAR) アルゴリズムは SEAR と類似しているが tidlist を使わずに Partition を行う．SPEAR では水平データフォーマットを用いて 2 度スキャンを行う．Partition と同様に 1 度目のスキャンでローカルな頻出アイテムセットを得て，2 度目のスキャンでグローバルなサポート値を得る．

Muller の目的は Partition を評価することであった．Muller は Partition は効果的でないと結論付けた．なぜなら，Partition によって，ローカルデータベースでは頻出であってもデータベース全体では頻出しないアイテムセットが多く発見されてしまうためである．

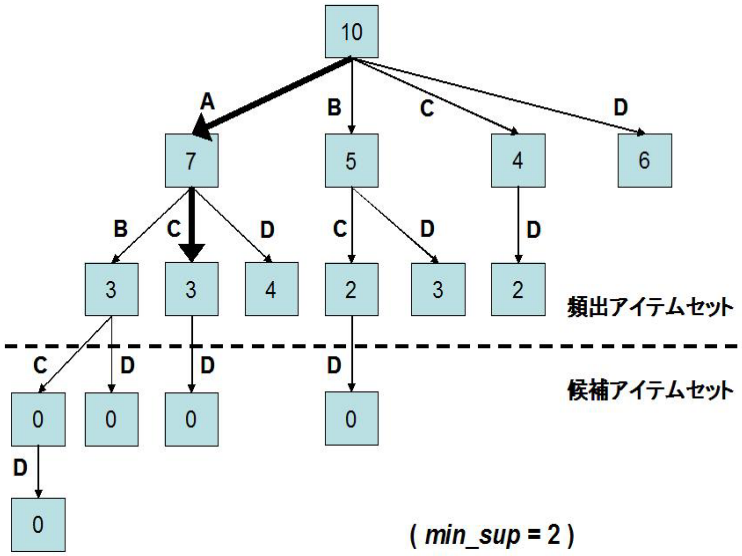

図 2.3: prefix tree 構造の例

2.3.4　Partition

1995 年にジョージア工科大学の Ashok らによって提案された Partition アルゴリズムはデータベースを分割し，水平データレイアウトを用いることによって，データベースに対して 2 回のスキャンで頻出アイテムセットを得る [3]．各 Partition が読み込まれると，各アイテムごとに tidlist(アイテムが現れた tid のリスト) が生成される．そして，tidlist からすべてのローカル頻出アイテムセットを生成する．各ローカル頻出アイテムセットをマージすることによって，グローバル候補アイテムセットを生成する．2 回目のスキャンで，グローバル候補アイテムセットのサポートを数え上げることによって，頻出アイテムセットを得ることができる．

2.3.5　Dynamic Itemset Counting

1997 年に Stanford 大学の Brin らによって提案された Dynamic Itemset Counting(DIC) アルゴリズムは Apriori アルゴリズムのスキャンコストを抑える手法である [19].

DIC ではデータベースを p 個に等分割する（Partition 1 〜 Partition p とする）．まず Partition 1 に対して，DIC は要素数 1 のアイテムのサポートを数える．Partition 1 だけで最小サポートを満たしているアイテムがあれば，それらのアイテムから要素数 2 の候補アイテムセットを生成する．次に DIC は Partition 2 を読み，要素数 1 のアイテムと要素

数 2 の候補アイテムセットのサポートを数える．これを Partition p まで繰り返す．つまり，DIC では 1 回目のデータスキャンにおいて，Partition k を読んでいるあいだに要素数 k のアイテムセットを数え始める．

DIC は Partition ごとでデータ内容に均一性がある場合に効果的である．Partition ごとでデータに偏りが生じていれば，ローカルなパーティションでは頻出アイテムセットであっても，グローバルな頻出アイテムセットではない候補アイテムセットを生成してしまう．

2.3.6　FP-growth

Apriori のように候補アイテムセットを生成すると，パターンが多く存在する場合，候補アイテムセットを格納するために必要となるメモリ容量が大きくなる．この問題を解決するために，候補アイテムセットを生成しない FP-growth アルゴリズムが 2000 年に Han らによって提案された [9]．このアルゴリズムは FP-tree 構造を利用しており，すべての頻出パターンを数え上げることができる．FP-tree は巨大なデータベースが小さく圧縮されたデータ構造であり，スキャンの繰り返しを減らすことができる．以下で FP-tree 構造の特徴および FP-tree の構築方法について述べた後，FP-tree 構造から頻出パターンを抽出する手法について述べる．

FP-tree 構造の特徴

FP-tree 構造は以下のような特徴を持つ．

- FP-tree 構造において，頻出アイテムのみが頻出アイテムセット抽出に使われる．頻出アイテムを見つけるために，データベースを 1 度スキャンする必要がある．
- 各トランザクションの頻出アイテムセットをコンパクトな FP-tree 構造に格納すると，繰り返しデータベースをスキャンする必要がなくなる．
- 複数のトランザクションが同一の頻出パターンを共有するのであれば，カウンタを一つにマージして FP-tree 構造に格納することができる．頻出アイテムがソートされていれば，二つのアイテムセットが同一かどうかをチェックするのは簡単である．
- 二つのトランザクションが共通の prefix を持っていれば，ソートされた頻出アイテムの順番に従って，共有する prefix 部分をマージして FP-tree 構造に格納することができる．頻出アイテムの頻度が降順に並ぶようにソートされていれば，より多くの prefix を共有する可能性が高くなる．

FP-tree の構築方法

1 回目のデータベーススキャンでは，各アイテムのサポートを求め，頻出アイテムを抽出する．抽出された頻出アイテムをサポートの値により，頻度が降順になるように並び替える（そのリストを F-list とする）．そして空（null）のラベルを持つ木のルートを作る（この木を T とする）．

2 回目のデータベーススキャンでは，各トランザクションごとに以下の処理を行う．

1. F-list に従って，トランザクションから頻出アイテムを抽出し，ソーティングを行う．
2. T が F-list の要素である子を持っていれば，その子のカウントを 1 増やす．T が F-list の子を持っていないときは，新しくカウント 1 を持つ子を作る．
3. F-list の最後の要素まで 1，2 の操作を繰り返す．

全てのトランザクションで処理を終えたら，同じ名前（アイテム ID）を持つノードにリンクを付ける．

具体的に表 2.3 のようなトランザクションデータから FP-tree 構造を構築する方法について説明する．ここで最小サポート値は 3 とする．/ 1 回目のスキャンでは，頻出アイテムを抽出し F-list を生成する．

$$\textbf{F-list} :< (f:4), (c:4), (a:3), (b:3), (m:3), (p:3) >$$

F-list では頻度が降順に並ぶようにソートされている．FP-tree の各パスがこの順序に従うため，この順序は重要である．F-list を生成したら，null とラベルされた木の root を生成する．

続いて 2 回目のスキャンを行う．1 番目のトランザクションのスキャンでは，最初の枝である <(f:1), (c:1), (a:1), (m:1), (p:1)> を得ることができる．トランザクション中の頻出アイテムは，リストによって降順にソートされているということに注意が必要である．2 番目のトランザクションにおいては，ソートされたアイテムリスト <f, c, a, b> が先ほどのパス <f, c, a, m, p> と <f, c, a> を共有するので，prefix 部分のノードのカウントがインクリメントされる．そして新しいノード (b:1) が生成され (a:2) の子としてリンクされる．もうひとつの新しいノード (m:1) が生成され，(b:1) の子としてリンクされる．3 番目のトランザクションでは，頻出アイテムリスト <f,b> がノード <f> のみを共有するので，f のサポートがインクリメントされる．そして新しいノード (b:1) が生成され，(f:3) の子としてリンクを張られる．4 番目のトランザクションのスキャンでは木の 2 本目の枝 <(c:1), (b:1), (p:1)> が生成される．5 番目のトランザクションでは，頻出アイテムリストが <f,

c, a, m, p> なので最初のトランザクションと同じであり，カウンタがパス中の各ノードで共有されインクリメントが行われる．

表 2.3: トランザクションデータ

TID	Items	Frequent Items
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

FP-growth

FP-growth アルゴリズムは FP-tree の以下のような性質を利用する．

性質 1 どんな頻出アイテム a_i に対しても，先頭の a_i を示すヘッダテーブルから，a_i のノードリンクをたどることにより，a_i を含む生成可能な頻出パターンをすべて得ることができる．

性質 2 パス P にあるノード a_i を含む頻出パターンを数えるためには，パス P におけるノード a_i の prefix-path を求めるだけでよい．そして prefix-path にあるノードのカウントは，ノード a_i のカウントと同じである．

図 2.4 を例にマイニングプロセスについて述べる．性質 1 よりあるアイテム a_i のヘッダから始めてノードリンクをたどることによって，a_i を含むすべてのパターンを集めることができる．ヘッダテーブルのアイテムからボトムアップに行うマイニングプロセスについて述べる．

ノード p に注目すると，頻出パターン (p:3) と FP-tree の 2 つのパス（<f:4, c:3, a:3, m:2, p:2>, <c:1, b:1, p:1>）が得られる．最初のパスは (f, c, a, m, p) というパターンがデータベースに 2 度現れることを意味している．<f, c, a> は 3 度，<f> は 4 度出現しているにもかかわらず，p とともに出現するのは 2 度だけであるということに注目したい．p とともにどのアイテムが出現しているかを見ることによって，p の prefix-path<f:2, c:2, a:2, m:2>

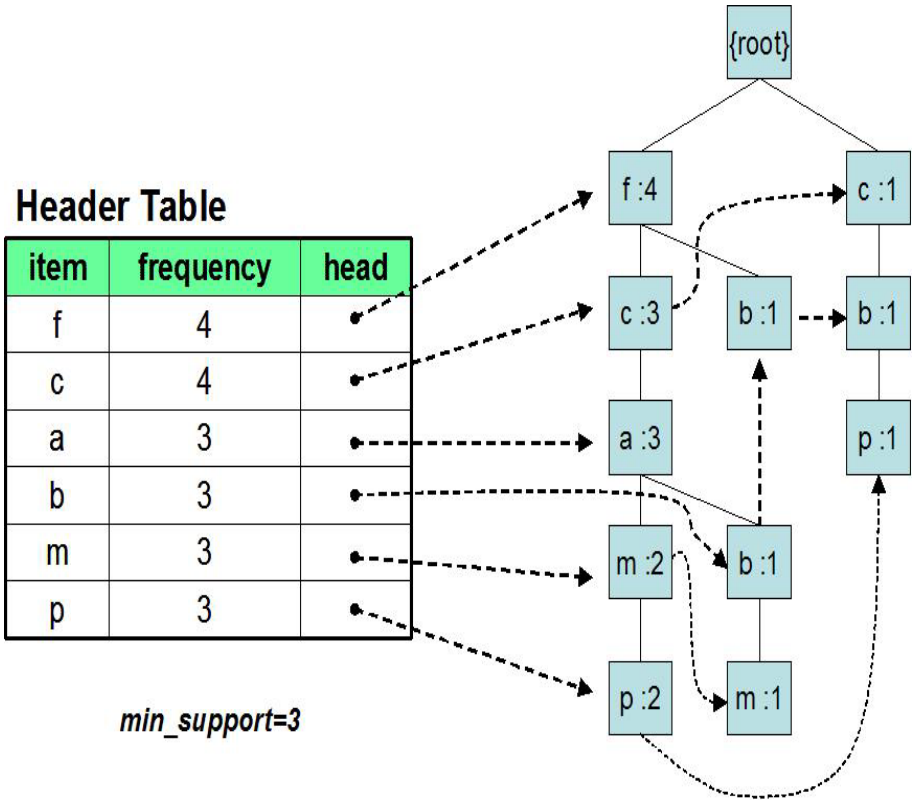

図 2.4: 表 2.3 から構築された FP-tree 構造 (文献 [9] より引用)

を数えることができる．同様に 2 つ目のパスは (c, b, p) が 1 度出現し，p の prefix-path が <c:1, b:1> であることを表す．p の sub-pattern を形成するこれら 2 つの p の prefix-path（<f:2, c:2, a:2, m:2>, <c:1, b:1>）は，p の conditional pattern base[1] とよばれる．p を含む条件のもとでの FP-tree の構築 (conditional FP-tree) からはたった一本の枝 (c:3) しか得られない．ゆえにたったひとつの頻出パターン (cp:3) しか得られない．これで p を含む頻出パターンの探索は終了である．

2.3.7　H-Mine

2001 年に Pei らによって H-Mine アルゴリズムが提案された [11]．H-Mine は，トランザクション DB を H-struct とよばれるハイパーリンクデータ構造に変換し，H-struct に対してマイニングを行う．H-Mine の手順を，表 2.4 から頻出パターンを抽出する例で示す．

頻出 1-アイテムのみが頻出パターンを構成するという Apriori の性質を利用する．1 回

[1] p が存在するという条件のもとでの sub-pattern base

表 2.4: トランザクションデータ

TID	Items	頻出アイテム射影
100	c, d, e, f, g, i	c, d, e, g
200	a, c, d, e, m	a, c, d, e
300	a, b, d, e, g, k	a, d, e, g
400	a, c, d, h	a, c, d

目のスキャンで頻出アイテム $\{a:3,\ c:3,\ d:4,\ e:3,\ g:2\}$ が抽出される．a, c, d, e, g という 5 つの頻出アイテムから構成される頻出アイテムセットは，以下の 5 つのパターンに分けられる．

パターン 1. a を含むパターン

パターン 2. c を含むが a を含まないパターン

パターン 3. d を含むが a も c も含まないパターン

パターン 4. e を含むが a も c も d も含まないパターン

パターン 5. g のみを含むパターン

ここで，頻出アイテムをアルファベット順に並べたものを，F-list(a-c-d-e-g) とする[2]．また，トランザクションから頻出 1-item を抽出したものを，頻出アイテム射影 (frequent item projection) とする．頻出アイテム射影の全てのアイテムは，F-list の順に従ってソートされる．例えば，TID100 のトランザクションの頻出アイテム射影は $cdeg$ である．頻出アイテムは，アイテム ID とハイパーリンクの 2 つの値を持つエントリに格納される．図 2.5 に，H-struct 構造の例を示す.

各頻出アイテムについて，アイテム名，サポート値，ハイパーリンクの 3 つの値を持つヘッダテーブル H が作られる．頻出アイテム射影が読み込まれると，最初に出現するアイテムが同じであるものが，キューとしてハイパーリンクでリンクされる．ヘッダテーブル H のエントリは，キューのヘッダの役割を果たす．例えば，ヘッダテーブル H におけるアイテム a のエントリは，トランザクション 200，300，400 の頻出アイテム射影をリン

[2]FP-growth の場合，頻出アイテムは頻度順にソートされていたが，ここでは説明を簡略化するためアルファベット順とする．

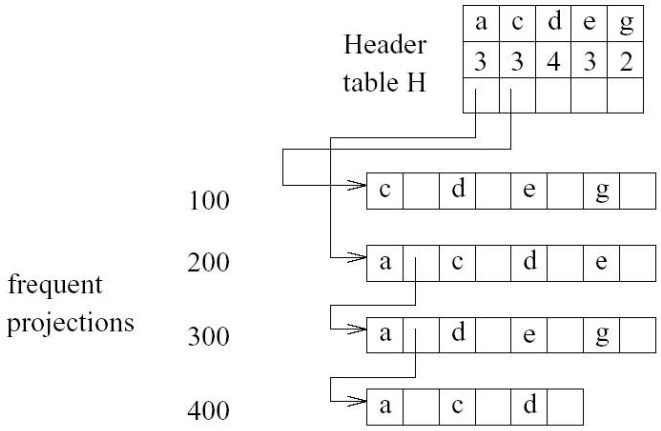

図 2.5: H-struct([11] より引用)

クする a キューのヘッダである．これら 3 つのトランザクションは全て，トランザクションで最初に出現するアイテムとして a を持っている．同様にトランザクション 100 の頻出アイテム射影は c キューとして，リンクされている．d キュー，e キュー，g キューは，これらのアイテムから始まる頻出アイテム射影が存在しないため，空である．

H-struct を構築するためには，トランザクションデータベースを一度スキャンする必要がある（2 度目のスキャン）．H-struct を構築すれば，データベースにある情報を参照することなく，マイニングを行うことができる．その後，5 つに分けられた頻出パターンが，以下のように一つずつマイニングされる．

まず，a を含む全頻出パターンを見つけ出す方法について述べる．このためには，アイテム a を含む頻出アイテム射影（a が射影されたデータベース：a-projected database）を探すことが必要となる．a が射影されたデータベースにおける頻出アイテム射影は，すでに a キューとしてリンクされているため，効率的に a を含む頻出アイテム射影をたどることができる．

a が射影されたデータベースをマイニングするために，a をヘッダとして持つヘッダテーブル H_a が作られる (図 2.6)．H_a では，a 自身を除く全頻出アイテムが，H と同じ 3 つの値（アイテム ID，サポート値，ハイパーリンク）を持っている．H_a でのサポートは，a が射影されたデータベースにおけるサポートが記録されている．例えば，アイテム c は a が射影されたデータベースでは 2 度現れているため，H_a における c のサポート値は 2 となる．

a キューを一度探索すると，a が射影されたデータベースでは最低 2 度出現しているような，ローカル頻出アイテムセットが見つかる．例えば，$\{c:2, d:3, e:2\}$ である．このスキャンの結果，$\{ac:2, ad:3, ae:2\}$ という頻出パターンが得られ，H_a のヘッダが図

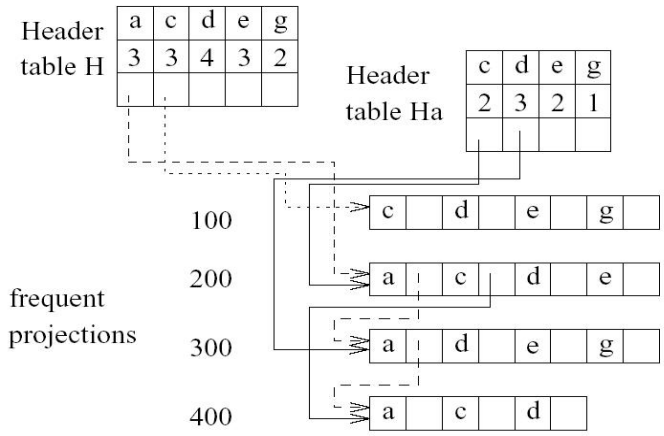

図 2.6: ヘッダテーブル H_a と ac キュー ([11] より引用)

2.6 のようにリンクされる．

同様に，H_a にある c キューを調べることによって，ac が射影されたデータベースで処理が続けられる．そして，図 2.7 に示すように，ac ヘッダテーブル H_{ac} が生成される．

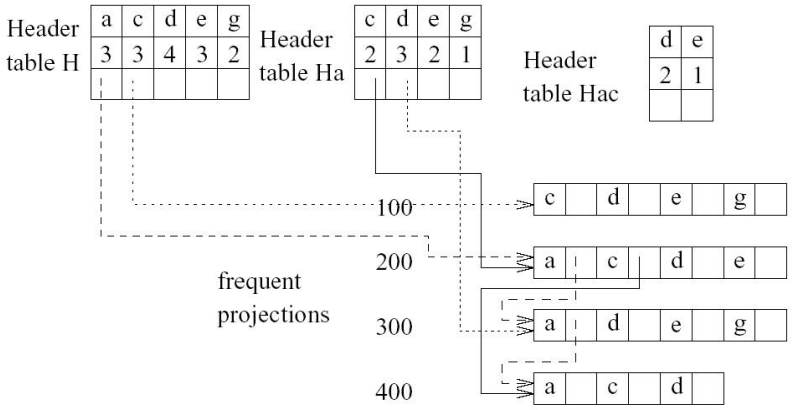

図 2.7: ヘッダテーブル H_{ac}([11] より引用)

ac が射影されたデータベースにおいて，d だけがローカルに頻出するため，acd のみが頻出パターンとして得られる．

その後，a と d を含み c を含まないパターンを見つけるために，同じことを行う．H_a で d から始まるキュー（ad キュー）は，a と d を含む（しかし c は含まない）全ての頻出アイテム射影をリンクする．ac キューで d を含む頻出アイテム射影を，ad キューに挿入することで，完全な ad が射影されたデータベースを得ることができる．ac キューの各頻出アイテム射影は，F-list に従って射影にある次の頻出アイテムのキューに追加される．図 2.8 に示すように，ac キューの頻出アイテム射影は全てアイテム d を持っているため，全

ての ac キューの頻出アイテム射影 (TID=200, 400) が ad キューに挿入される．

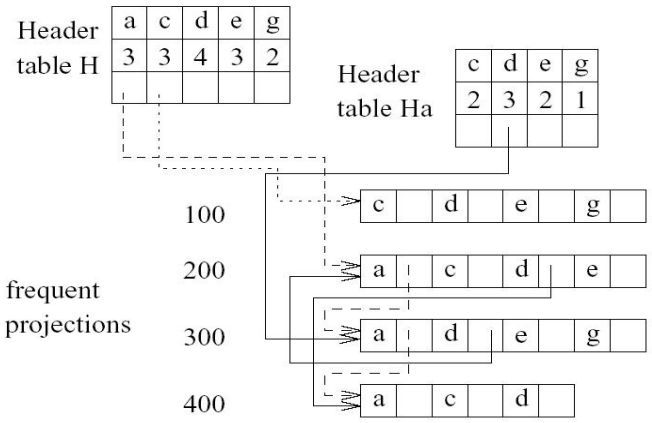

図 2.8: ヘッダテーブル H_a と ad キュー ([11] より引用)

ac キューの頻出アイテム射影を挿入すると，ad キューが a と d を含む頻出アイテム射影を全て集めていることが分かる．a と d を含む頻出パターンは再帰的に抽出される．ad が射影されたデータベースの頻出アイテム射影に c が出現するにもかかわらず，ac キューのマイニングの際に考慮されているため，c をローカルな頻出アイテムとして考えなくても良い．このフェーズでは，ade というパターンのみが抽出される．すでに H_{ac} に対する探索が行われているため，3 段階目のヘッダテーブル H_{ad} は H_{ac} を使うことができる．つまり，3 段階目の探索では，一つのヘッダテーブルのみを必要とする．その後の全マイニングプロセスにおいて，各段階で一つのヘッダテーブルのみが必要であるということがわかる．

続いて，ae が射影されたデータベースを調べると，e はリンクを持っていないため，頻出パターンを生成することなく，このフェーズは終了する．これで a を含む頻出パターンは全て抽出されたことになる．

a を含む頻出パターンが発見されたら，a が射影されたデータベース（a キュー）は不要となる．c キューは，a と c を両方含む射影を除いて（これは a キューにある），c を含む全頻出アイテム射影を含む．a を含まないで c を含む全頻出パターンを抽出するためには，a キューにある全頻出アイテム射影をキューに挿入しなければならない．

各頻出アイテム射影は，F-list の順に従って a の次にあるアイテムのキューに拡張される．例えば，頻出アイテム射影 $acde$ は c キューに挿入され，$adeg$ は d キューに挿入される (図 2.9)．

c が射影されたデータベース (c キュー) を再帰的にマイニングすることによって，a を含まず c を含む頻出パターンを抽出することができる．a を持つ全頻出パターンはすでに

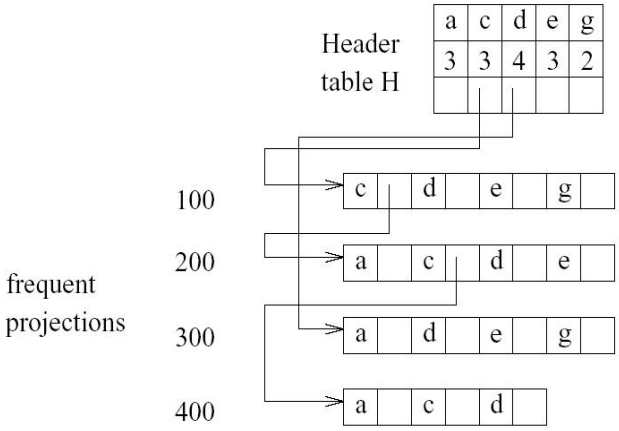

図 2.9: a が射影されたデータベースを処理した後のハイパーリンクの状態 ([11] より引用)

発見されているため，アイテム a は c が射影されたデータベースには含まれない．a を含む頻出パターンを抽出した時と同様に，残りの 4 つのパターンに対してもマイニングが行われ，全ての頻出パターンが抽出される．

2.3.8　Pattern Decomposition

2001 年に Zou らによって提案された Pattern Decompostition(PD)[15] は，FP-tree アルゴリズムと同様に，トランザクションデータベースを別のデータ構造に変換し，頻出パターンを抽出する手法である．FP-growth と異なる点は，PD は新しいデータ構造を前もって構築しないという点である．その代わりに，パスを経るごとにデータセットが変換される．PD の基本的な考え方は，データセットのサイズを小さくしていくことである．PD は新しい非頻出パターンが発見されると，データセットを小さくする．

従来の候補アイテムセットを生成する手法は，アイテムをリテラル文字，トランザクションを一つのバスケットに存在するリテラル文字の集合として定義していた．この論文では，新たなデータ構造を用いるために，以下のように定義する．

1. パターン p とは，アイテムセットの集合とその頻度の組み合わせであり，$< p.IS, p.Occ >$ で表現される．アイテムセットの集合 $p.IS$ の要素であるアイテムセットは，その他のアイテムセットの部分集合にならない．例えば，$p =< abcd,\ cde,\ 3 >$, $p.IS = abcd,\ cde$, $p.Occ = 3$. のように表す．
2. データセット D とはパターンの組み合わせである．例えば，

$$D_1 = \{abc : 1,\ abd : 2,\ abe : 1,\ ace : 1,\ ade : 1,\ bce : 1\}.$$

のように表す．ここで定義するデータセットとは，パターンとともにその頻度を含む．

3. データセット D におけるアイテムセット I のサポートとは

$$Sup(I|D) = \sum p.occ,\ if\ p \in D\ and\ (\exists R \in p.IS\ and\ I \subseteq R).$$

上の D_1 の例では，$Sup(abd|D_1) = 2$, $Sup(ab|D_1) = 4$ である．

4. アイテムセット I の分解 (decomposition) とは，$\sim L_k$ に含まれる非頻出アイテムセットを含まない，I の最も大きい部分集合 S を見つけることである．つまり，S に含まれる k-アイテムセットはすべて L_k で頻出である．

PD は，頻出パターンを発見するためにボトムアップ探索を行う．データセット D_1 に対して，パス 1 から開始する．以下に，パス k における処理を示す．

1. D_k にある全ての k アイテムセットを数え，頻出アイテムセット L_k と非頻出アイテムセット $\sim L_k$ を生成する．
2. $\sim L_k$ にあるアイテムセットを含まない D_{k+1} を得るために，D_k を分解する．

以上の処理を繰り返し，パス k で D_k が空になれば PD は終了する． PD の動きを [15] にならって図 2.10 に示す．

オリジナルデータセットを D_1，最小サポートを 2 とする．パス 1 では，L_1 と $\sim L_1$ を決定するために，D_1 の全アイテムのサポートを数える．この例では，頻出 1-item は $L_1 = \{a, b, c, d, e\}$ であり，非頻出 1-item は $\sim L_1 = \{f, g, h, k\}$ である．その後，D_2 を得るために，$\sim L_1$ を用いて D_1 にある各パターンを分解する． 例えば，D_1 で最初に出現するパターン $p = abcdef : 1$ に注目すると，パターン p から f を取り除き，新しいパターンである $abcde$ を D_2 に生成する．また，2 番目のパターンと 5 番目のパターンを分解すると，同じパターン abc が生成されるため，サポートをマージする．その結果，D_2 に $abc : 2$ というパターンが生成される．

パス 2 では，L_2 と $\sim L_2$ を決定するために，D_2 にある 2-item のサポートを数える．その後，D_3 を得るた

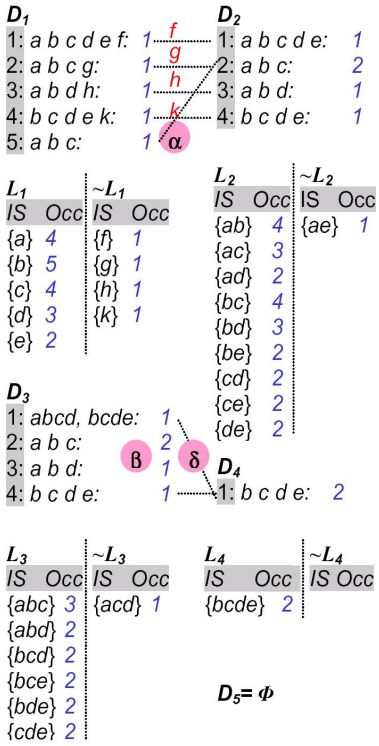

図 2.10: PD の例 (参考文献 [15] より引用)

めに，D_2 にあるパターンを分解する．非頻出 2-item は $L_2 = \{ae\}$ である．D_2 で最初に出現するパターン $abcde : 1$ に注目すると，$\{ae\}$ は頻出でないため，$p = abcde : 1$ は $q = abcd, bcde : 1$ に分解される．

パス 3 では，L_3 と $\sim L_3$ を決定するために，D_3 にある 3-item のサポートを数える．その後，D_4 を得るために，D_3 にあるパターンを分解する．$acd \in \sim L_3$ であり，$acd \subseteq abcd$ であるため, D_3 の最初のパターンに $abcd$ は abc, abd, bcd に分解される．サイズが 4(次のパス) より小さいため，新しいパターンとして D_4 に生成されない．アイテムセット $bcde$ は acd を含まないため，そのまま D_4 に含まれる．

空のアイテムセットである D_5 を決定するまで，この処理が続く．最終的な頻出アイテムセットは，L_1 から L_4 の和集合をとったものである．

2.3.9　Max Miner

98 年に IBM Almaden 研究所の Roberto らによって提案された Max-Miner は，MaximalPattern Mining として初めて提案されたアルゴリズムである．Max-Miner は，prefix tree を用いて，MFI を抽出するアルゴリズムである [4]．prefix tree の全てのノード g に対し，以下に示す 3 つのアイテムセットで構成される候補グループを生成し，マイニングを実行する．

- h(g)　root ノードからノード g までたどることによって得られるアイテムセット（head）
- t(g)　ノード g の下層に現れる全てのアイテム集合 (tail)
- i　アイテム集合 t(g) に含まれる任意の 1-アイテム

マイニングを実行するに当たって，ノード g とノード g の下に配置されているノードを対象に，候補グループ G を定義する．候補グループ G は，$h(g)$，$h(g) \cup i$，$h(g) \cup t(g)$ の 3 つのアイテム集合で構成される．例として，$1, 2, 3, 4$ で構成される TDB に対する prefix tree は，図 2.11 のように表現され，図 2.11 中の候補グループ $\{1\}$ の場合，$h(g) = \{1\}$，$t(g) = \{2, 3, 4\}$，$h(g) \cup t(g) = \{1, 2, 3, 4\}$，$h(g) \cup i = \{1, 2\}; \{1, 3\}; \{1, 4\}$ となる．

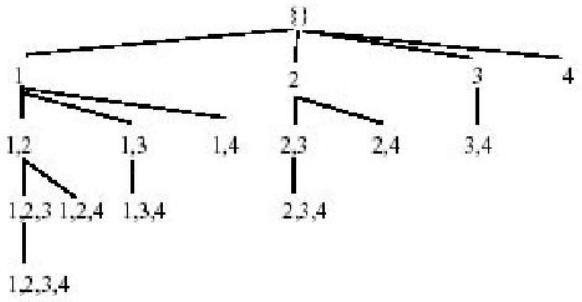

図 2.11: Prefix Tree の例（文献 [4] より引用）

2.3.10　FPclose

FP-growth によって生成されるパターンの数は莫大である．この問題を解決するために，2003 年に Grahne らによって FPclose アルゴリズムが提案された [6]．FPclose は，FPgrowth をベースにして飽和頻出パターンを抽出するアルゴリズムである．FPclose は，2003 年 11 月時点で，Closed Pattern Mining アルゴリズムの中で最速と判定されている [5]．FPclose では，FPgrowth 同様に FP-tree を構築し，構築した FP-tree から頻出パターン生成を行う．生成された頻出パターンを CFI-tree(Closed Frequent Itemset tree) に挿入し CFI を抽出する．

CFI-tree

CFI-tree は，FP-tree と似たデータ構造である．CFI-tree は，ヘッダテーブルと木構造で構成されている．CFI-tree のヘッダテーブルは，アイテム名とノードリンクのヘッドによって構成されている．CFI-tree のヘッダテーブル中のアイテムの順番は，FP-tree のヘッダテーブルのアイテムの順番と同じである．また，CFI-tree の各ノードは

- アイテム名
- カウント値
- レベル (root ノードから該当ノードへたどり着くため通過したエッジの数)
- ノードリンク

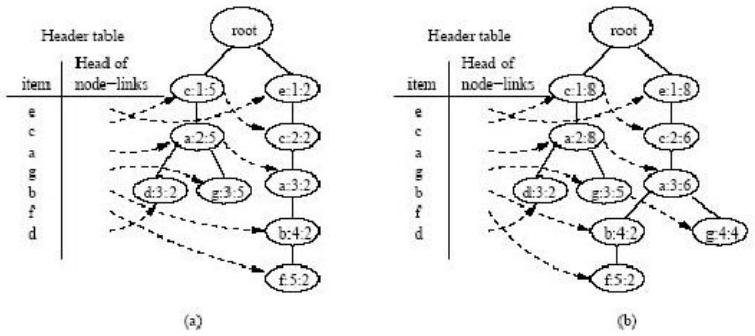

図 2.12: CFI-tree 構築の方法（文献 [6] より引用）

T_{Head} の 4 つのエントリによって構成されている．CFI-tree は，FP-tree と同じ数だけ構築される．ここで，あるアイテムセット X とすると，X 条件付 FP-tree から生成されたアイテムを挿入する CFI-tree（X 条件付 CFI-tree）を C_X と置く．C_X には，アイテムセット X を含み，かつ既に CFI であると判断されたアイテムセットが格納されている．X 条件付 FP-tree（T_X）より新しく抽出されたパターン Y が抽出された場合，C_X に格納されている CFI と比較する．Y が，C_X に格納されている CFI と同じカウント値を持ち，かつ Y のサブセットでないアイテムセットが存在しない場合に，Y は C_X に挿入される．図 2.12 は，図 2.12 に示している FP-tree から生成されるパターンの挿入例を示している．図 2.12 においてのノード x : l : c は，アイテム x のノードで root ノードからのレベルが 1 であり，カウント値が c であるノードを指す．図 2.12(a) では，カウント値が 2 である (c; a; d) と (e; c; a; b; f) が挿入された後，カウント値が 5 である (c; a; g) を挿入した CFI-tree である．(c; a; g) が，prefix 部分 (c; a) を (c; a; d) と共有しているので，ノード g のみが追加される．同時に，共有部分であるノード c とノード a のカウント値を 5 に変更する．

図 2.12(b) では，(e; c; a; g) : 4，(c; a) : 8，(c; a; e) : 6，(e) : 8 が挿入された後の CFI-tree である．この時点で，図 2.12 の FP-tree から全ての CFI が CFI-tree に格納されている．FPclose アルゴリズム は，TDB から FP-tree を構築する．ここで，アイテムセット Head 条件付 FP-tree T_{Head} とおき，T_{Head} から抽出されたパターンを格納する CFI-tree を C_{Head} とおく．さらに，T_{Head} から CFI を抽出し C_{Head} に格納する関数を FPclose(T_{Head},C_{Head}) とおく．FP-close は，FP-tree を構築した後に，関数 FPclose(T,C) を再帰的に実行することによって，CFI を抽出する．以下では，関数 FPclose(T,C) について説明する．関数 FPclose(T_{Head},C_{Head}) を実行する前の段階で，Head と Head 条件

付 FP-tree に存在するアイテムセットの組み合わせたアイテムセットが，既に抽出された CFI であり，かつ同じカウント値でないことを確認する．

1. T_{Head} が単一パス P で構成されているかどうかを調べる．
2. T_{Head} が単一パスで構成されていた場合
   (a) P から全ての候補 CFI を生成する．
   (b) 生成した全ての候 (a) T_{Head} のヘッダテーブル中の任意の 1-itemset{i} を Head に追加する．
3. T_{Head} が単一パスで構成されていなかった場合
   (a) T_{Head} のヘッダテーブル中の任意の 1-itemset{i} を Head に追加する．
   (b) 配列 A_{Head} から，Head 条件付パターンベース中の頻出アイテムセット全てで構成されるアイテムセットを Tail と定義する．
   (c) Tail の中のアイテムを，サポート値降順に並び替える．
   (d) $T_{Head\cup i}$ と $A_{Head\cup i}$ を構築する．
   (e) $C_{Head\cup i}$ を初期化する．
   (f) 関数 FPclose($T_{Head\cup i}$,$C_{Head\cup i}$) を実行する．
   (g) $C_{Head\cup i}$ を C にマージする

2.3.11　逐次頻出パターン抽出アルゴリズムのまとめ

前節までで述べたように，Apriori アルゴリズムをベースとして，スキャン数の削減や候補アイテムセット数の削減を行う研究がなされていた．一方，2000 年以降，Apriori とは異なるアプローチを用いて，候補アイテムセットを生成しない FP-growth，H-Mine，PD といったアルゴリズムが提案されている．これら 3 つのアルゴリズムは，トランザクションデータベースを，頻出アイテムセットを抽出するために必要な情報のみを格納したデータ構造に変換する．そして，変換したデータ構造に対してマイニングを行う．これら 3 つのアルゴリズムは Apriori アルゴリズムよりも高速であり，最小サポートが小さくなるにつれて実行時間の差が広がってくる．

さらに，ユーザにとって冗長である莫大な数のパターンを結果として抽出する Apriori や FP-growth とは異なり，冗長性が削減されたパターン（CFI）を抽出する手法も提案されている．逐次頻出パターン抽出アルゴリズムの特徴を表 2.5 にまとめた．

表 2.5: 逐次頻出パターン抽出アルゴリズムの特徴

アルゴリズム	レイアウト	データ構造	スキャン回数	特徴
Apriori ('94)	horizontal	ハッシュ木	k	ベーシックなアルゴリズム，候補アイテムセット数を格納するためのメモリ容量とスキャンの繰り返しが欠点
DHP ('97)	horizontal	ハッシュ木	k	Apriori ベース，候補アイテムセット数削減
DIC ('97)	horizontal	prefix-tree	k	Apriori ベース，スキャン数削減
Partition ('95)	vertical	none	2	水平データレイアウトを利用，トランザクションを分割する
SEAR ('95)	horizontal	prefix-tree	k	Apriori と原理は同じ，hash-tree ではなく prefix-tree に候補アイテムセットを格納
SPEAR ('95)	horizontal	prefix-tree	2	SEAR に Partition を適用
FP-growth ('00)	horizontal	FPtree	2	FP-tree 構造を利用，候補アイテムセットを生成せず Apriori よりも高速
H-Mine ('01)	horizontal	H-struct	2	ハイパーリンクを利用した H-struct を利用，候補アイテムセットを生成しない
PD ('01)	horizontal	dataset	1	候補アイテムセットを生成しない，スキャン回数が 1 回のみ，パターンを分割してデータセットを小さくする
Max-Miner ('98)	horizontal	prefix-tree	$l+1$(l:最大アイテムセット長)	深さ優先探索で極大頻出集合を求める．
FPclose ('03)	horizontal	FPtree と CFItree	2	FP-growth ベースのアルゴリズム．頻出パターンを求めたのち，CFI を抽出する．

2.4　並列頻出パターン抽出アルゴリズム

頻出パターン抽出高速化のアプローチとして並列化がある．並列化のターゲットとしては共有メモリ型並列計算機と分散メモリ型並列計算機がある．本節では，PC クラスタがスケーラビリティやコストパフォーマンスの面から次世代のデータベースプラットフォームとして注目されている点を考慮して，分散メモリ型並列計算機をターゲットとした並列アルゴリズムについて述べる．

2.4.1　分散カウント

Apriori の最も簡単な並列化手法は，1996 年に Agrawl らによって提案された分散カウント法 (CD:Count Distribution) である [17]．CD では，あらかじめトランザクションデータベースをノード数で分割し，各ノードに分配しておく．各ノードは，割り当てられたトランザクションデータに対して，独立に候補アイテムセットの出現頻度を求める．CD では，Apriori アルゴリズムの k 回目のパスが以下のように並列化される．

1. マスターノード M が，要素数 k の候補アイテムセットを全ノードに送信する．
2. 各ノードで，割り当てられたトランザクションデータをスキャンして，候補アイテムセットの出現頻度を数える．
3. ノード M で手順 2 の結果を各候補アイテムセットごとにマージして，データベース全体での出現頻度（グローバルサポート）をもとめる．
4. ノード M が要素数 $k+1$ の候補アイテムセットを生成する．

この処理を候補アイテムセットがなくなるまで繰り返す．図 2.13 に CD の動きを示す．各ノードに {A, B}，{C, D}，{E, F} という候補アイテムセットが複製されている．各ノードがローカルディスクから候補アイテムセットのサポートを数えたら，グローバルサポートを得るためにローカルサポートをマージする．

CD ではローカルディスク中の候補アイテムを数えている間は，各ノードが独立して動作できるので，候補アイテムセットが一つの主記憶に収まる場合は，高い台数効果が期待できる．しかし，候補アイテムセットが全ノードに複製されるためメモリ利用効率が低くなる．

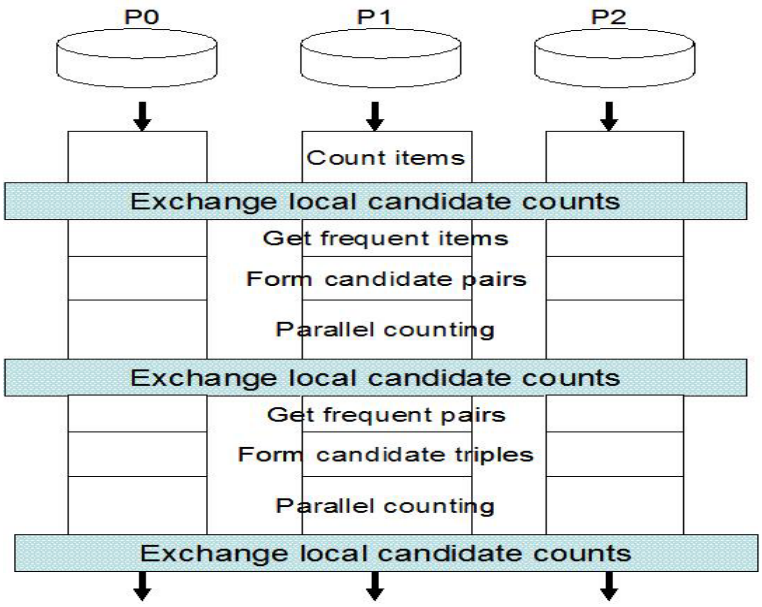

図 2.13: Count Distribution

2.4.2　Data Distribution

1996 年に Agrawal によって提案された Data Distribution(DD) は，各ノードでばらばらの候補アイテムセットを生成する [18]．DD では，グローバルなサポートを得るために，各ノードが全てのノードのデータベースに対してスキャンを行う必要がある．つまり，各ノードのトランザクションを他のノードにブロードキャストしなければならない．そのため，DD は CD と比較すると通信コストが高くなり，全体の処理効率も低下してしまう．

図 2.14 に DD の動きを示す．3 つの候補アイテムセット {A, B}，{C, D}，{E, F} が，各ノードへ分配される．各ノードは，割り当てられた候補アイテムセットのグローバルサポートを得るために，ローカルディスクにあるアイテムセットのサポートと，別ノードから送られたアイテムセットのサポートを数える．

2.4.3　ハッシュ分割アプリオリ

CD では候補アイテムセットが主記憶に収まらない場合，並列に処理を行っても主記憶の不足する状況が改善されなかった．1996 年に東京大学の Shintani らによって提案されたハッシュ分割アプリオリ (HPA:Hash Prtitioned Apriori)[20] は，ハッシュ関数を用いて

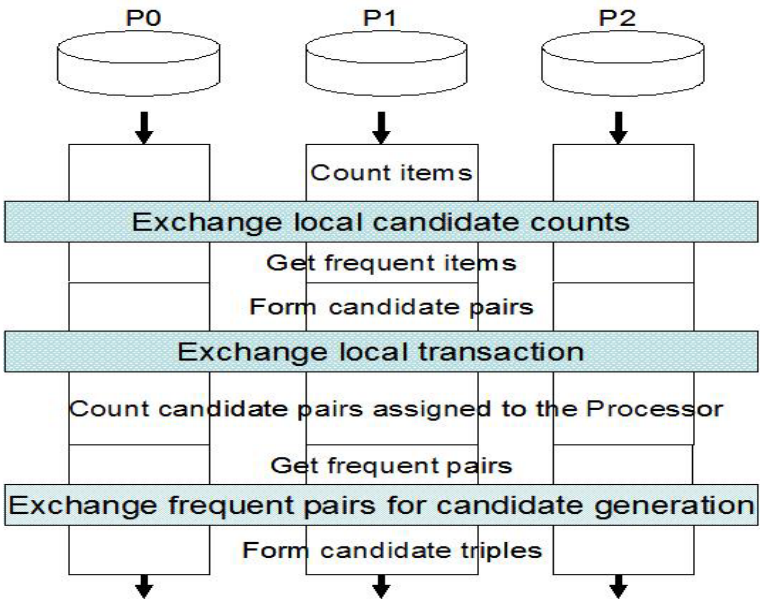

図 2.14: Data Distribution

候補アイテムセットをノードごとに分割し，記憶効率を高めることができる手法である．HPA の手順は以下のとおりである．

1. 要素数 k の各候補アイテムセットをハッシュ関数によって決定されるノードに送信する．
2. 各ノードで，割り当てられたデータベースの一部の各トランザクション t に対して
   a) t から要素数 k の部分集合をすべて作り，各々について Step1 のハッシュ関数を適用して決定されるノードに送る．
   b) 要素数 k のトランザクションを受信したノードは，それと一致する候補アイテムセットを探し，その出現数を 1 増やす．
3. 各ノードが候補アイテムセットが頻出か否かを決定し，結果を一つのノードに集める．
4. 結果が集められたノードで，要素数 $k+1$ の候補アイテムセットを生成する．

図 2.15 に HPA の動きを示す．各候補アイテムセット {A, B}，{C, D}，{E, F} がハッシュ関数によって決定されるノードへ分配される．各ノードのトランザクションから生成される要素数 2 のアイテムセットも，候補アイテムセットを分配した時と同じハッシュ関数を適用して決定されるノードへ送られる．つまり，P1 が数える {A, B} というアイテムセットは，P1 にしか送信されない．この点で，各ノードのトランザクションをブロード

キャストしていた DD とは異なる．

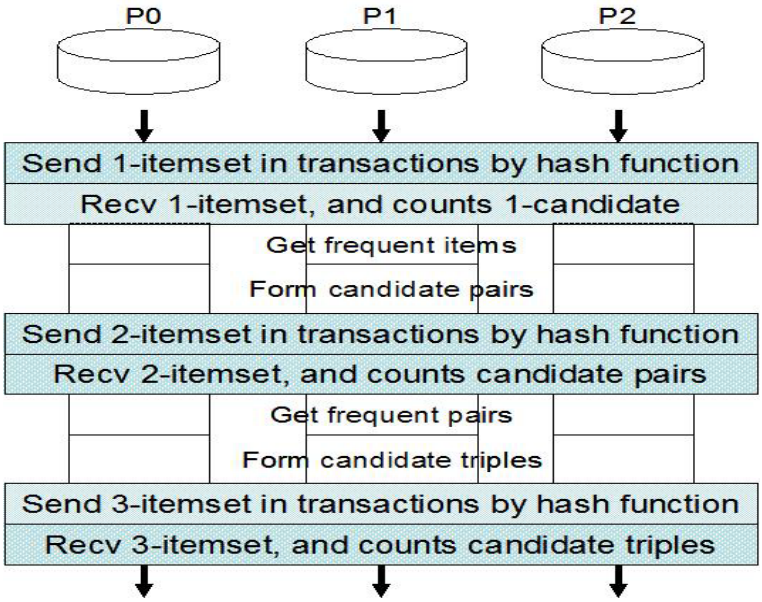

図 2.15: Hash Partitioned Apriori

HPA では記憶効率を高めることができる代わりに，ノード間の通信やハッシュ関数の計算を繰り返す必要がある．この問題を緩和するために提案されたのが HPA-ELD(HPA with Extremely Large itemset Duplication) アルゴリズム [20] である．HPA-ELD では頻度の高い候補アイテムセットを各ノードで複製して保持し，複製された候補は CD と同様に処理される．この方法によって，アイテムセットの通信量を減らすことができる．

さらに HPA-ELD は負荷の分散を助ける効果も持っている．HPA ではデータスキューが極端な場合にあるノードに負荷が集中してしまう．例えば，POS データ[3]には非常な偏りがあり，牛乳や卵といった商品は他のアイテムよりも出現頻度が高い．そういったアイテムが割り当てられたノードには，多くのアイテムセットが送信され負荷が集中してしまう．HPA-ELD では頻度の高いアイテムは各ノードに複製されているので，負荷を分散して処理を行うことができる．

2.4.4 Parallel Data Mining

1995 年に Jong Soo Park らによって提案された Parallel Data Mining(PDM) アルゴリズムは，DHP を並列化した手法である [13]．PDM ではまず DHP と同様に各ノードが要

[3]Point of Sales：バーコードなどと連動させて，店頭での販売情報をリアルタイム管理するシステム

素数 1 のアイテムを数え，同時にハッシュテーブルを用いて要素数 2 のアイテムセットのサポートを概算する．各ノードがローカルなサポートをブロードキャストすることによって，グローバルなサポートが計算される．しかし，要素数 2 のアイテムセットを格納したハッシュテーブルは大きいため，ブロードキャスト通信にかかるコストが大きい．

そこで，PDM では頻出することが保証されている，つまり，バケットに格納されているサポート値が最小サポートを満たすセルのみを通信する．しかし，この方法では一つのパスで 2 段階の通信を必要とする．PDM では 2 回目のパスでのみ，候補アイテムセットを生成するためにグローバルハッシュテーブルを使う．3 回目以降のパスでは，Apriori と同様に一つ前のパスでの頻出アイテムセットから候補アイテムセットを生成する．これは DHP が要素数 2 の候補アイテムセット数を削減するという特徴を踏まえている．

PDM では候補アイテムセットが各ノードに複製される．各ノードがローカルサポートを求め，グローバルサポートを得るために，ローカルサポートが all-to-all ブロードキャストされる．その結果，通信コストが高くなり，並列処理効率が低下してしまうケースがある．

2.4.5　Parallel Efficient Association Rules

1995 年に提案された Parallel Efficient Association Rules(PEAR) は，SEAR を並列化した手法である [2]．SEAR の原理は Apriori と同じであり，SEAR を並列化した PEAR の原理は Apriori を並列化した CD(2.4.1 節参照) と同じである．各パスにおいて，全てのノードがそれ以前のパスの頻出アイテムセットをもとに，候補アイテムセットを格納した prefix-tree を生成する．各ノードにおいて，同じ候補アイテムセットが複製されている．各ノードはローカルなサポートを計算し，それらを合計することによってグローバルなサポートを得る．

2.4.6　Partitioned Parallel Association Rules

1995 年に提案された Partitioned Parallel Association Rules(PPAR) は，SPEAR を並列化した手法である [2]．PPAR は以下のように処理される．まず，各ノードはローカルディスク内のローカル頻出アイテムセットを数える．ローカル頻出アイテムセットは，他のノードにブロードキャストされる．各ノードはローカルディスクを 2 回目にスキャンするときに，これらのグローバルな候補アイテムセットを数え上げる．この結果をブロードキャストすることによってグローバルな頻出アイテムセットのサポートを得ることができ

る．PPAR は，結果的には頻出アイテムセットではない不必要な候補アイテムセットを多く生成するため，PEAR に比べると性能が劣ることが示されている．

2.4.7　FP-growth 無共有並列実行

2003 年には，Iko らによって PC クラスタ上で FP-growth を並列実行する手法が提案された [8]．この手法では，条件付きパターンベース処理が他のアイテムの条件付きパターンベース処理と独立して行える点に注目している．

手順としては，まず，1 回目のスキャンでは各ノードがローカルトランザクションからアイテムの数え上げを行い，マージしてグローバルな F-list を得る．2 回目のスキャンでは，各ノードがローカルトランザクションから FP-tree を構築する．ローカル FP-tree が構築されると，各ノードは条件付きパターンベースを生成する．生成されたローカル条件付きパターンベースは，割り付けられたアイテムを処理するノードに集約される．それぞれのノードが完全な条件つきパターンベースを受けてから，独立してその条件つきパターンベースの処理を完成させる．

さらに，[8] で提案された手法では，十分な台数効果を得るために「パス深さ」と呼ばれるパラメータを活用して，実行ノード間の負荷を動的に均等化するメカニズムを提案している．

2.4.8　並列頻出パターン抽出手法のまとめ

分散メモリ型並列計算機をターゲットとした，並列頻出パターン抽出アルゴリズムの特徴を表 2.6 にまとめる．Apriori をベースとした手法として，各ノードへ候補アイテムセットを複製する手法，候補アイテムセットを各ノードへ分配する手法，これら二つを織り交ぜた手法の 3 種類がある．近年では，FP-growth をベースとした手法も提案されている．

表 2.6: 並列頻出パターン抽出アルゴリズムの特徴

アルゴリズム	手法	利点	欠点
CD ('95)	Apriori ベース．候補アイテムセットをブロードキャストし，各ノードが同じ候補アイテムセットを数える．	候補アイテムセットが一つの主記憶におさまる場合，台数効果が高い．	候補アイテムセットが一つの主記憶におさまらない場合，主記憶の容量不足が解消されない．
PDM ('95)	DHP(Apriori) ベース．原理は CD と同様．DHP の利点である要素数 2 の候補アイテムセットを削減できることを利用する．	要素数 2 の候補アイテムセットの通信量を削減する．	CD と同様．
PEAR ('95)	SEAR ベース．原理は CD と同様，候補アイテムセットを prefix-tree に格納する	CD と同様．	CD と同様．
PPAR ('95)	SPEAR ベース．原理は CD と同様．PEAR に Partition を適用する	CD と同様．	CD と同様．
DD ('96)	Apriori ベース．トランザクションをブロードキャストし，ノード毎に異なる候補アイテムセットを数える．	全ての候補アイテムセットが一つの主記憶に収まる必要がないため，ノード数に応じて利用できる主記憶の容量が増加する．	トランザクションをブロードキャストするため，CD よりも通信コストが高い．
HPA ('96)	Apriori ベース．ハッシュ関数で候補アイテムセットを各ノードへ分割する．同様に，トランザクションも同じハッシュ関数で各ノードへ分割する． 各ノードは，割り当てられたトランザクションに対して，割り当てられた候補アイテムセットを数える．	DD と同様に，全ての候補アイテムセットが一つの主記憶におさまらない場合に生じる容量不足を緩和する．さらに，送信するトランザクションをハッシュで分割するため，DD よりも通信コストが抑えられる．	ハッシュ計算に時間が掛かる．CD と比較すると通信コストが掛かる．
Parallel FP-growth('03)	FPgrowth ベース．通信を行ってグローバルな条件付きパターンベースを生成した後，各ノードが独立して条件付き FP-tree を再帰的に処理する．処理時間の掛かるアイテムが割り当てられたノードは，アイドルノードに処理を再割り当てする．	条件付き FP-tree の処理を動的に再割り当てすることで， 負荷分散が施される．	条件付きパターンベースを交換するために，通信コストが高くなる．

第3章　FPclose並列化手法の提案

第2章で述べたように，頻出パターン抽出並列アルゴリズムの多くが，Apriori をベースとする手法である．近年は，FP-growth をベースとする並列化手法も提案されている．しかし，これらの並列化手法では，頻出パターンは抽出できるが，飽和頻出パターン（CFI）は抽出されない．

第3章では，冗長性の少ない有益なパターンをより高速に抽出するために，FPclose をベースとした並列化手法を提案する．FPclose は，CFI 抽出アルゴリズムの中で最速のアルゴリズムである（2003 年 11 月時点）．この FPclose を並列化することで，冗長性の少ないパターン（CFI）を従来よりも高速に抽出することができると考えられる．

FPclose は，FPgrowth と同様に頻出パターンを生成したのち，生成されたパターンを検証し CFI を抽出する手法である．よって，提案する FPclose 並列化手法における頻出パターン生成処理でも，FP-growth 並列化手法 [8] に基づいて拡張を行う．生成された頻出パターンから CFI を生成する過程では，さらなる通信を必要とせずに完全な CFI-tree を構築できることに着目して，並列処理効率を高めることを図る．具体的な並列化の手順を以下で述べる．

3.1　FP-tree 構築並列化

並列化によって得られる大きな利点の一つとして考えられるのが，トランザクションデータに対するスキャンを並列化できることである．例えば，p 台のノードのディスクにデータが格納されていれば，シーケンシャルにデータを読む場合に比べて，単位時間あたり p 倍のデータ量を読むことができる．この利点を生かして，FP-tree 構築を並列化する．

まず，従来手法 [17][8] と同様に，あらかじめトランザクションデータ TDB を先頭から順に均等に各ノード p_i へ分配しておく．ここで，本プログラムは，n 個のプロセスからなり，$0 \leq i < n$ であるとする．各ノード p_i は，あらかじめ分配されたローカルトランザクション TDB_i からローカルな FP-tree 構造 FPT_i を生成する．

3.1.1　ローカルサポートの数え上げ

1 回目の TDB に対するスキャンでは，頻出 1-item を数える．各ノード p_i には，同数のトランザクションが分配されている．各ノード p_i は，分配されたトランザクション TDB_i に出現するローカルなアイテムを数え上げる．ローカルなアイテムを数え上げている間，各ノードは独立して動作する．

3.1.2　グローバルサポートの数え上げ

各ノードにおいてローカルなアイテムの数え上げが終了したら，入力データ全体でのサポートを得るために，集団通信を行ってローカルサポートをマージする．マージする際の通信としては，各ノードが数え上げたローカルサポートを他ノードに送信することによって，グローバルサポートが数えられ，頻出でないアイテムが除かれる．見つかった頻出アイテムは，サポートが降順に並ぶようにソートされる．このリストが FP-growth における F-list である．本手法においては，この時点で全てのノードが同じ F-list を持つことになる．

3.1.3　ローカル FP-tree 構築

ローカル FP-tree を構築するために，2 回目の TDB に対するスキャンを行う．各ノードはローカルなトランザクションからローカル FP-tree を構築する．各トランザクションに対して F-list にある頻出アイテムだけが集められる．F-list の順序にしたがって各アイテムの頻度が降順に並ぶようにソートされる．各トランザクション内でソートされたアイテムは，ローカル FP-tree を構築するために，以下のように使われる．

1. トランザクションに出現するアイテムに対して，root に子ノードが存在するかどうかをチェックする．
2. 子ノードが存在すれば，子ノードのサポートをインクリメントする．子ノードが存在しなければ，出現したアイテムのために，新しいノードを追加し，サポートを 1 とする．
3. 読み込んだアイテムを新しい root として，トランザクションに出現する次のアイテムに対しても手順 1. を繰り返す．

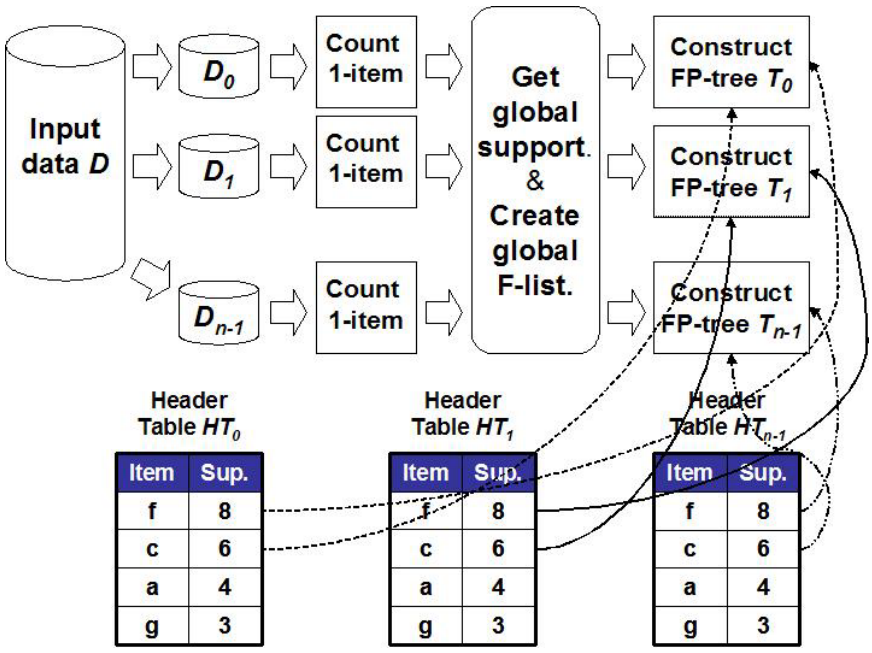

図 3.1: 並列構築された FP-tree とヘッダテーブルの様子

以上の手順で，各ノードのローカルディスクからローカル FP-tree を構築する．ヘッダテーブルには，データベース全体で頻出するアイテムと，そのサポートが格納されている．また，ヘッダテーブルは，各ノードのローカル FP-tree で最初に出現する各アイテムへのリンクを持つ．図 3.1 に，並列構築された FP-tree とヘッダテーブルの例を示す．まず，1 回目のスキャンを行った後，通信を行いグローバルなサポートを得た結果，F-list ($f:8$, $c:6$, $a:4$, $g:3$) が得られたとする．F-list が生成されると，各ノードは 2 回目のスキャンを行い，ローカル FP-tree を生成する．各ノードにあるヘッダテーブルにおいて，アイテムと対応するサポート値は同じである．異なるのは最初に出現するノードへのポインタである．つまり，各ノードのヘッダテーブル HT_i は，各ノード内の FP-tree 構造 T_i で初めに出現するノードへのポインタを持つ．

3.2　パス深さを考慮した負荷分散

生成された FPT_i からローカルな条件付きパターンベース CPB_i を生成する．しかし，CPB_i だけでは TDB 全体に頻出するパターンを抽出できないため，通信を行ってグローバルな条件付きパターンベースを求める必要がある．そこで，各ノード p_i が CPB_i を生成したら，どのノードがどのアイテムに基づくグローバル条件付きパターンベースを処理するかを決定し，CPB_i をマージする．

ここで，[8] では，処理ノードをハッシュ関数によって決定している．しかし，ノード毎のタスク負荷の偏りが問題となる並列マイニングにおいては，負荷をより平坦化する処理ノード決定法が必要である．

そこで，本手法では，負荷をより平坦化するために，FPT_i に含まれる情報を利用し，処理ノードを決定する．条件付きパターンベースの処理ノードを決定する時点では，各ノードには TDB_i を読み込んで構築した FPT_i が既に存在する．つまり，この時点で TDB における出現アイテムの特性をある程度得ることができる．本手法では，[8] のタスク分割の際に用いられるパス深さ[1]を用いることで，条件付きパターンベース処理時間を予測し，タスク負荷の平坦化を図る．つまり，条件付きパターンベースを処理するノードを決定する前に，各ノードの FPT_i から各条件付きパターンベースのパス深さの値を予測し，その値をもとに条件付パターンベースを処理するノードを決定する．これによって，条件付きパターンベース処理に必要なタスクが平坦化され，全体としての処理の高速化が期待できる．

[1]パス深さとは，条件付きパターンベース内の最小サポート値を満たす最長パターンの長さである．条件付きパターンベースの処理時間は，この「パス深さ」に比例することが知られている．

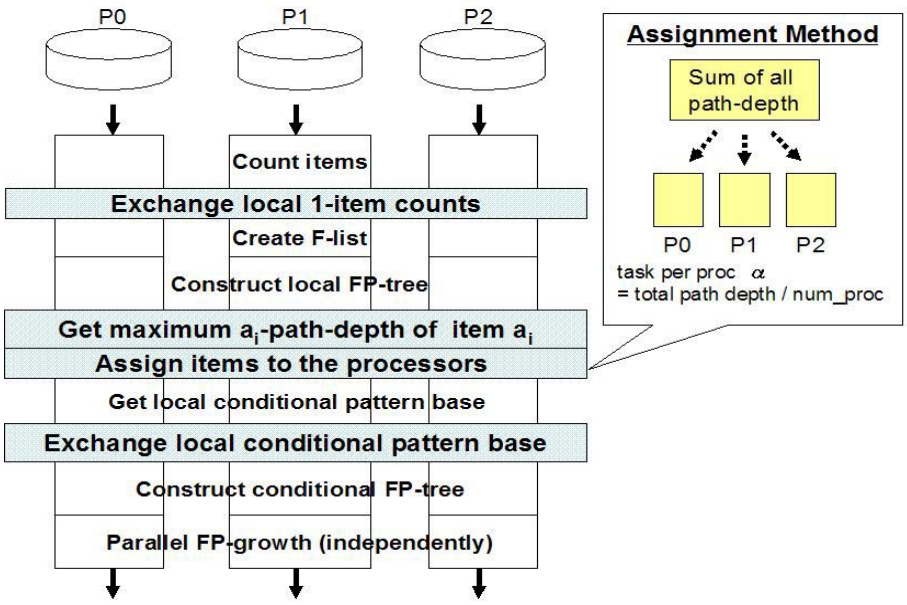

図 3.2: 頻出パターン抽出までの流れ

具体的には以下の手順をとる（図 3.2 参照）．まず，各ノードはローカルな FP-tree から，パス深さを得る．ローカルなパス深さを得たら，トランザクションデータベース全体でのパス深さを得るために，通信を行って全ノードで各アイテムの最大パス深さを得る．続いて，全アイテムの最大パス深さを合計し，これを総タスク量とする．この総タスク量をノード数（PU 台数）で割った値が，１つのノードあたりの処理すべきタスク量 α である．処理ノードを決定する際は，1 つのノードの処理タスク量が α を超えないように，Flist の先頭から順に割り振っていく．これによって，各ノードの処理タスク量がある程度均等化される．

3.3　CFI 抽出並列化

ローカルな条件付きパターンベースをマージした結果，各ノードがグローバルな条件付きパターンベースを取得したら，各ノードは独立して頻出パターンを生成する．生成された頻出パターンは，各ノードの CFI-tree に挿入され，CFI であるか否かのチェックが行われる．ここで，CFI-tree 挿入に関する処理は，各ノードで独立して実行することができる．なぜならば，X 条件付き CFI-tree に必要な情報は，すべて X 条件付きパターンベースに含まれるからである．つまり，各ノードに割り当てられたアイテム X を含むパターンが CFI であるか否かをチェックするために必要な情報は，X が割り当てられたノードが持つ，グローバルな X 条件付きパターンベースに格納されているからである[2]．以上の手続きを踏むことで，あらかじめ負荷を平坦化した並列化手法を用いつつ，冗長性の少ないパターンである CFI を抽出することができる．図 3.4 に一連の手順を擬似コード化したものを示す．

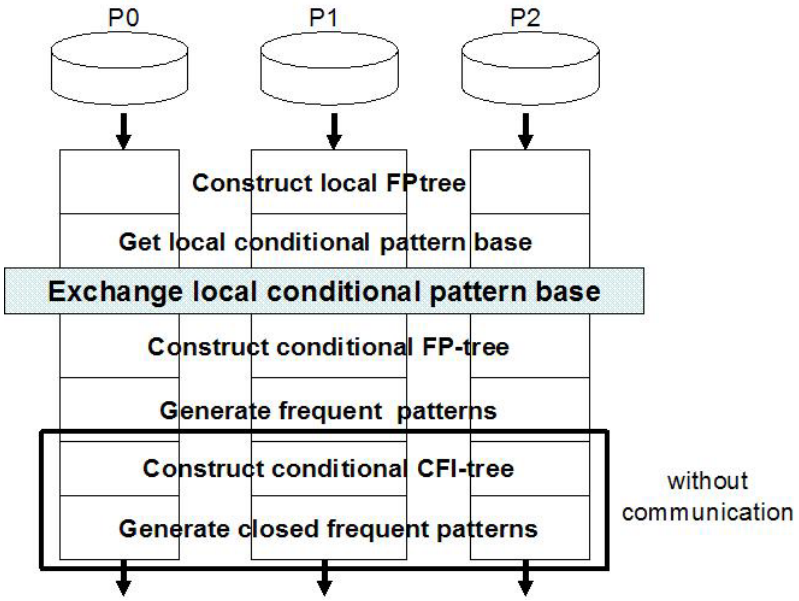

図 3.3: CFI 抽出の並列化

[2]FPclose アルゴリズムでは，CFI-tree は FP-tree と同じ数だけ構築される．

```
input : database TDB, Items I, minimum support min_sup

SEND Process:
{
    sum_path_depth = 0;
    dest_node = 0;
    local_support = count_support(TDB, I);
    global_support = merge_support(local_support);
    FList = create_FList(global_support, min_sup);
    FPtree = construct_FPtree(FList,TDB);
    /* get total_path_depth */
    local_path_depth = get_path_depth(FList, I);
    approximate_path_depth = max(local_path_depth);
    total_path_depth = sum(approximate_path_depth);
    foreach item in FList
    {
        cond_pbase = create_cond_pbase(FPtree, item);
        path_depth = get_path_depth(FPtree, item);
        sum_path_depth = sum_path_depth + get_path_depth(FList, item);
        if(sum_path_depth > (dest_node + 1) * total_path_depth / num_node)
            dest_node++;
        send_cond_pbase(dest_node, cond_pbase);
    }
}
RECV Process:
{
    cond_pbase = collect_cond_pbase();
    cond_FPtree = construct_fptree(cond_pbase, FList);
    FPclose(cond_FPtree, NULL);
}
```

図 3.4: Parallel FPclose 擬似コード

第4章　性能評価

第3章で述べた手法をMPIを用いて実装し，PCクラスタ上で性能評価を行った．ただし，3.2で述べた負荷分散機構に関してはまだ実装を終えておらず，現時点ではCFI抽出単純並列化（ハッシュによる処理ノード決定法）に関するデータを測定した．今回の実験では，並列化を施していないFPcloseと比較した速度向上率と，大規模データに対するスケーラビリティを評価した．

4.1　実験環境

本節では，実験を行ったクラスタ環境について述べる．今回の実験では，表4.1に示すスペックからなる計算機32台をクラスタとして構成した．各計算機は，Gigabitイーサネットによって図4.1のように接続される．なお，コンパイラはgcc2.96を用い，コンパイルオプションは-O2とした．

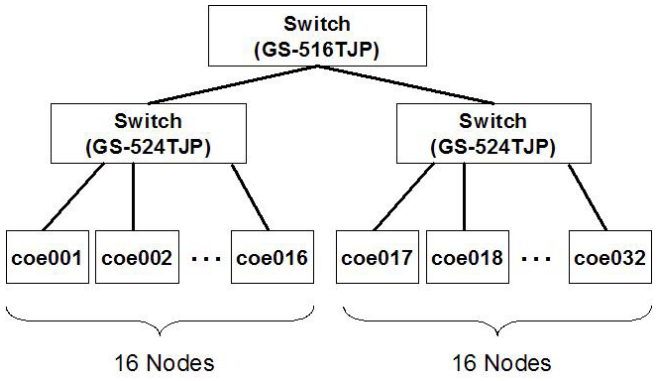

図 4.1: クラスタ構成図

表 4.1: 各PCのスペック

CPU	Pentium 4
Clock	2.40 GHz
L2cache	512 KB
Memory	512 MB×2
Network	1000BaseTX Ethernet
OS	RedHat 7.3

4.2　MPI

FPclose の並列化を実装する際に MPI(Message Passing Interface) を用いた．MPI は世界の多くのベンダと，米国アルゴンヌ国立研究所やミシシッピ大学等の研究機関・大学が参加している MPI フォーラムで開発されたメッセージ通信の API 仕様である．MPI は C，C++プログラム中から関数，サブルーチンとして使用する． 今回はフリーな通信ライブラリとしてポピュラーである MPICH[21](Ver 1.2.5) を用いた[1]．

4.3　データセット

実装した手法を評価するデータセットとして，IBM の人工データ生成プログラム [7] を用いて，データセットを生成した．このプログラムではオプションで与えるパラメータによって，異なるデータセットが生成される．以下で，各パラメータについて述べる．

- **-ntrans**　トランザクション数 (x1000)
- **-tlen**　1 トランザクションあたりのアイテム数
- **-nitems**　トランザクションデータベース中のアイテム数
- **-npats**　パターン数
- **-patlen**　パターンの平均最大長

実験で使用するデータセットを T25I15D100k のように表記する．T25I15D100k はトランザクション総数が 100000(100k)，トランザクションあたりのアイテム数が 25，パターンの平均最大長が 15 というパラメータを持つデータセットである．その他のパラメータは，アイテム総数 1000，パターン数 5000 として実験を行った．

4.4　性能評価指標

実験の結果を評価する指標として，台数効果と並列処理効率を用いる．本節では，二つの評価指標について述べる．

[1]フリーな通信ライブラリとしては，MPICH のほかに LAM[22] がある．

4.4.1　速度向上率

並列処理の性能評価で最もよく用いられる指標は速度向上率である．速度向上率とは，p 台のノードを用いた場合，1 台のノードを用いた場合に比べてどれくらい速くなっているかを示す指標である．本手法の実験では，並列化を施していないプログラムの実行時間を T_1（ノード 1 台），p 台のノードを用いた並列プログラムの実行時間を T_p とする．このとき速度向上率 S_p は

$$S_p = \frac{T_1}{T_p} \tag{4.1}$$

として表すことができる．並列処理が効率よく行われるほど，T_p が小さくなるため，S_p の値が大きければ速度向上率が高いということである．

4.4.2　並列処理効率

並列処理を行うことによる速度向上の理想は，p 台のノードを用いた場合に速度も p 倍速くなるということである．つまり，$S_p = p$ を満たすことが理想である．しかし，現実的にはこの条件を満たすことは難しく，$S_P < p$ となることが多い．そこで，投入したノード台数に対して，どの程度速度が向上したかを示す指標として，並列処理効率 E_p を以下のように定義する．

$$E_p = \frac{S_p}{p} \tag{4.2}$$

仮に $S_p = p$ であれば $E_p = 1$ (100%) であり $S_p < p$ であれば $E_p < 1$ となる．

4.5　実験結果と考察

4.5.1　逐次 FPclose と比較した速度向上率

今回の実験では，T10I4D100k，T10I4D500k，T10I4D1000k としてデータセットを生成した．ノード台数を 1 台から 32 台まで変化させて，最小サポート値 2.0%，1.5%，1.0%，0.5% の場合について実験を行った．ノード台数が 1 台の場合は，並列化を施していないプログラムを実行し，実行時間を測定した．実験結果を，PU 台数と速度向上率の関係でグラフに表したのが図 4.2，図 4.3，図 4.4 である．図 4.2 は T10I4D100k に対する実験結果，図 4.3 は T10I4D500k に対する実験結果，図 4.4 は T10I4D1000k に対する実験結果を表している．

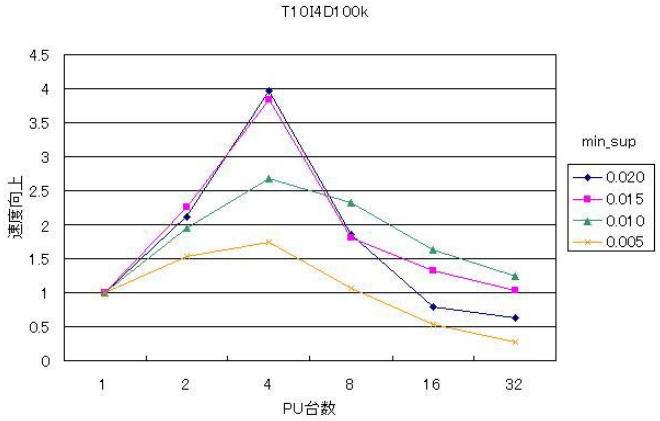

Figure 4.2: PU 台数と速度向上（トランザクション数 100k）

図 4.2 を見ると，どのサポート値に対しても，4PU 実行時に速度向上の最高値を得ることが分かる．また，最小サポートが小さくなるにつれて，同数の PU を投入したとしても，速度向上が低下してゆく．これらの原因は，最小サポートが小さい場合，抽出される頻出アイテムや，計算すべき条件付きパターンが増加し，それに伴って通信量も増加する[2]ためである．通信量が増加することによって，全体としての実行時間が増加してしまう．

[2]現時点の実装では，条件付きパターン収集処理において通信が多く発生するため，改善の余地はある．

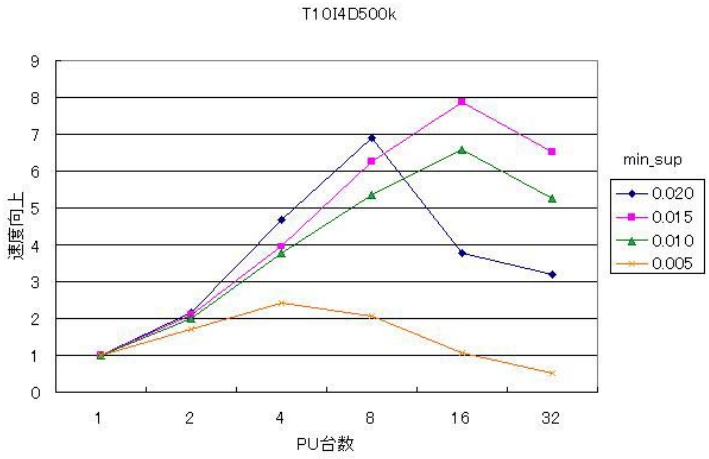

Figure 4.3: PU 台数と速度向上（トランザクション数 500k）

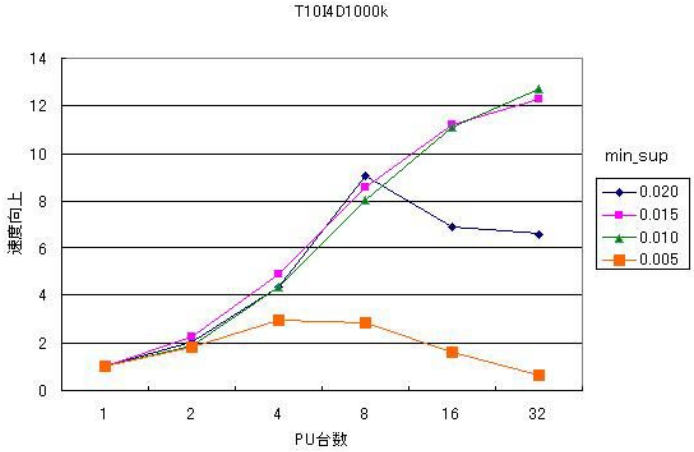

Figure 4.4: PU 台数と速度向上（トランザクション数 1000k）

図 4.3，図 4.4 に注目すると，トランザクション数が増加するにつれて，速度向上の最大値が大きくなるということと，そのときの PU 台数の値が大きくなることが分かる．具体的には，図 4.3（トランザクション数 500k）では 16PU 投入時に約 8 倍の速度向上を得るのに対して，図 4.4（トランザクション数 1000k）では 32PU 投入時に約 12 倍の速度向上を得る．これは，トランザクション数が大きくなるにつれて，全体の実行時間に対してディスクアクセス時間が占める割合が大きくなるためである．

また，図 4.3 と図 4.4 を見ると，最小サポートが最大（0.020）となる場合に，速度向上

の最大値が得られていないことが分かる．これは，処理ノード決定法がハッシュに基づくためであると考えられる．現時点では，ハッシュに基づいて処理ノードを決定しているため，処理時間の大きいパターンをあるノードが他ノードよりも多く割り当てられる可能性がある．結果的に，ノード毎でタスク負荷の偏りが発生し，処理時間の掛かるノードがボトルネックとなり，速度向上が低下してしまう．

4.5.2　大規模データに対するスケーラビリティ評価

本手法の大規模データに対するスケーラビリティを評価するために，同じく IBM の実行データ生成プログラム [7] を用いて，トランザクション数の違う新たなデータセット T10I4D100k，T10I4D500k，T10I4D1000k，T10I4D5000k，T10I4D10000k を生成した．これら 5 つのデータセットを対象に，本手法の実行時間を測定した．最小サポートを 2%，PU 台数は 8 台として実験を行った．図 4.5 にトランザクション数と実行時間の関係を示す．

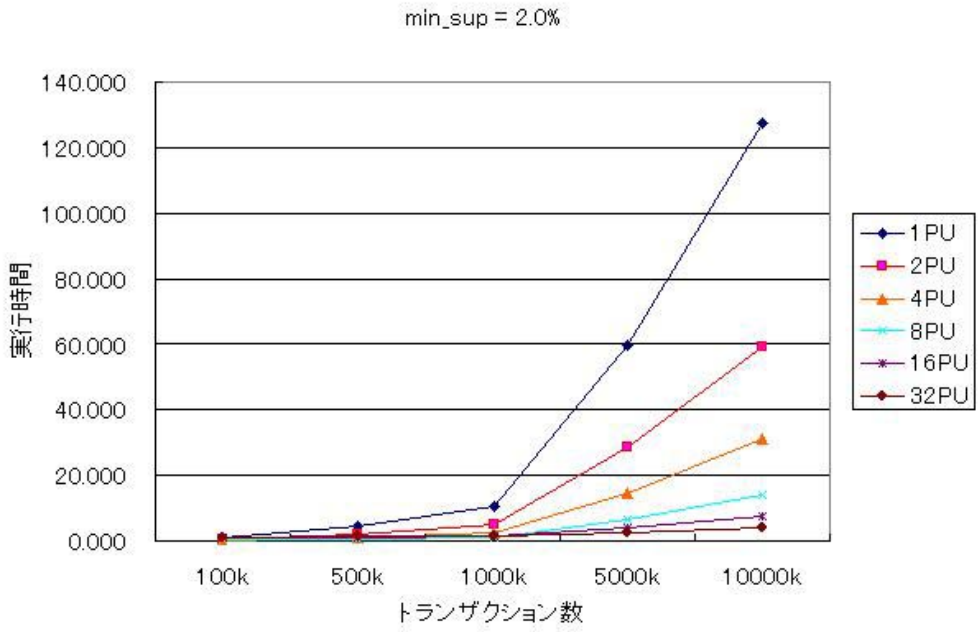

Figure 4.5: スケーラビリティの評価

図 4.5 が示すように，本手法を 8PU で並列実行した場合の実行時間は，単一ノードで CFI 抽出を実行した場合の実行時間よりも，トランザクション数，つまりデータの規模に影響を受けることが少なかった．具体的には，単一ノードで実行した場合，トランザクションが 1000k から 5000k に増えると，実行時間は 50 秒程度余計に掛かった．一方，本手法により，8PU で並列実行した場合，トランザクションが 1000k から 5000k に増えても，実行時間は 5 秒程度しか増加しなかった．これは，データが大規模になるにつれて，ディスクアクセスに必要な時間が増加するためであると考えられる．PC クラスタなどの分散並列計算環境においては，ディスクアクセスを並列化することによって，総実行時間を大幅に短縮することができるため，大規模なデータを対象にしても，パターン抽出を高速に実行することができる．最小サポートを 2%とした今回の実験では，データが大規模になる

につれて，PC 台数に対してリニアに計算時間を速めることができた.

第5章　おわりに

近年，ネットワーク環境の整備や，記憶装置の低価格化・大容量化にともなって，データの洪水化がすすんでいる．大規模なデータから有用な知識を抽出するために，データマイニング技術が注目を集めている．データマイニング分野の重要な問題として，頻出パターン抽出がある．

頻出パターン抽出は，バスケット解析や DNA 解析などに適用される重要な技術である．しかし，頻出パターン抽出は大規模なデータを対象に処理を行うため，高速化の研究が不可欠である．頻出パターン抽出高速化に関する研究の中でも，メモリ容量不足やディスクアクセス増加といったリソース面での制約を緩めるために，PC クラスタなどをターゲットとした並列化手法が提案されている．従来提案されてきた並列化手法の多くは，Apriori や FP-growth をベースとする手法である．これらの並列化手法では，全ての頻出パターンを抽出するため，結果として莫大な数のパターンが抽出され，ユーザに負担が掛かるという問題がある．また，頻出パターン並列抽出処理においては，トランザクションデータベース中に出現するアイテムの特性により，ノード毎でタスク負荷が偏ってしまい，結果として並列処理効率が低下するという問題があった．

本論文では，ユーザにとって解析する負担が少ないパターンを高速に提示するために，FPclose をベースとして飽和頻出パターンを並列抽出する手法を提案した．さらに，頻出パターン並列抽出において問題となる，タスク負荷の偏りを平坦化する手法を提案した．この提案した手法を PC クラスタ上で実装し，性能評価を行なった結果，32 ノード PC クラスタ上で最大 30.9 倍の速度向上を得ることができた．また，本手法がデータの大規模化に対応しうることも確認できた．

5.1　今後の課題

今後の課題としては，現時点の実装では条件付きパターンベース収集処理において，冗長な通信を行っているため，並列処理効率が低下しているという問題がある．また，提案した負荷分散機構を実装し，その有用性を検証する必要がある．現在，DEWS2005 発表

当日までに結果を出すべく，これらの問題に取り組んでいる最中である．

謝辞

本研究を行うにあたり数々のご指導を頂いた山名早人助教授に深く感謝致します．また，
私を助けてくれた山名研究室の先輩の方々，同輩の方々に厚く御礼申し上げます．

参考文献

[1] 石川慎也, ”データマイニングの宝箱”, http://www5.ocn.ne.jp/ shinya91/index.html

[2] A. Mueller, ”Fast Sequential and Parallel Algorithms for Assosiation Rule Mining,” A Comparison, Tech, Report CS-TR-3515, 1995.

[3] A. Savasere, E. Omiecinski, and S. Navathe, ”An Effiecient Algorithm for Mining Association Rules in Large Databases,” In Proceedings 21st International Conference on Very Large Databases, pp.423-444, 1995.

[4] Bayard, R.J. ”E ± ciently mining long patterns from databases ”, In Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 85-93, 1998.

[5] Goethals, M. J. Zaki, ”FIMI ’03: Workshop on Frequent Itemset Mining Implementations, ” In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Inplementations, 2003.

[6] Gosta Grahne and Jianfei Zhu, Efficiently Using Prefix-trees in Mining Frequent Itemsets, Proceeding of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI’03), 2003

[7] IBM Quest Data Mining Project. Quest synthetic data generation code. http://www.almaden.ibm.com/cs/quest/syndata.html

[8] Iko Pramudiono, Masaru Kitsuregawa, ”Tree Structure based Parallel Frequent Pattern Mining on PC Cluster ,” In Proceedings of 14th International Conference on Database and Expert Systems Applications (DEXA’2003), pp.537-547, 2003.

[9] J. Han, J. Pei, and Y. Yin, ”Mining frequent patterns without candidate generation,” In Proceedings of the ACM SIGMOD Conference on Management of Data, pp.1-12, 2000.

[10] J.Pei, J.Han, and R.Mao, ”CLOSET: An e ± cient algorithm for mining frequent closed itemsets, ” In DMKD ’00, 2000.

[11] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, ”H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases,” In Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM’01), pp.441-448, 2001.

[12] J.S. Park, M. Chen, and P.S. Yu, ”An effective hash-based algorithm for mining association rules,” In Proceedings of the ACM SIGMOD Conference on Management of Data, pp.175-186, 1995.

[13] J.S. Park, M. Chen, and P.S. Yu, ”Efficient Parallel Data Mining for Association Rules,” In Proceedings of the ACM International Conference on Information and Knowledge Management, pp.31-36, 1995.

[14] J.Wang, J. Han, and J. Pei, ”CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets, ” In Proceedings of the ACM SIGKDD Conderence, Aug. 2003.

[15] Q. Zou, W. Chu, D. Johnson, and H. Chiu, ”A Pattern Decomposition (PD) Algorithm for Finding All Frequent Patterns in Large Datasets ,” In Proceedings of the 2001 IEEE International Conference on Data Mining(ICDM’01), pp.673-674, 2001.

[16] R. Agrawal and R. Srikant, ”Fast Algorithms for Mining Association Rules,” In Proceedings of the International Conference on Very Large Data Bases, pp. 487-499, 1994.

[17] R. Agrawal and R. Srikant, ”Parallel mining of association rules,” IEEE Transactions on Knowledge and Data Engineering, 8(6), 1996.

[18] R. Agrawal and J.C.Shafer, ”Parallel mining of association rules:Design, implementation and experience,”IBM Research Report RJ10004, 1996

[19] S. Brin, R. Motowani, J. Ullman, and S. Tsur, ”Dynamic itemset counting and implication rules for market basket data,” In Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 255-264, 1997.

[20] T. Shintani and M. Kitsuregawa, "Hash based parallel algorithms for mining association rules," In Proceeding International Conference on Parallel and Distributed Information Systems, pp.19-30, 1996.

[21] -:MPICH Home Page. http://phase.etl.go.jp/mirrors/mpi/mpich/

[22] -:LAM HOME. http://www.lam-mpi.org/

研究業績

講演

1. 岩橋永悟，山名早人:”FP-growth の並列化による頻出パターン抽出の高速化”, 情処研報 (DBS),Vol.2003,No.71,pp.327-334 (2003.7)
2. 岩橋永悟，平手勇宇，山名早人:”PC クラスタ上における頻出飽和パターン抽出並列化手法の提案”，DEWS2005(掲載予定)

その他（コンテスト等）

1. 平成 15 年度データ解析コンペティション・世帯別電力消費データ（学生部門）優秀賞：チーム早稲田大学山名研究室（岩橋永悟，岩渕寿寛，平手勇宇，山名早人）
2. Imagine Cup 2004 ソフトウェアデザイン部門日本大会準優勝：チーム早稲田大学大学院（益子理絵，岩橋永悟，仲沢由香里，本田 大）

付 録A　実行時間表

第 4 章の実験で FPclose 並列化プログラムを実行したときの総実行時間を表 A.1，表 A.2，表 A.3 に示す.

表 A.1: 総実行時間 (秒):T10I4D100k

PU数 / min sup	1	2	4	8	16	32
0.020	0.770	0.362	0.194	0.416	0.968	1.230
0.015	1.070	0.471	0.279	0.592	0.810	1.040
0.010	1.340	0.687	0.500	0.576	0.826	1.078
0.005	1.660	1.091	0.953	1.570	3.116	5.962
0.002	2.000	71.223	126.943	287.028	685.311	1613.820

表 A.2: 総実行時間 (秒):T10I4D500k

PU数 / min sup	1	2	4	8	16	32
0.020	4.610	2.138	0.984	0.669	1.226	1.449
0.015	5.810	2.801	1.475	0.929	0.738	0.889
0.010	7.340	3.659	1.944	1.365	1.113	1.388
0.005	8.850	5.166	3.667	4.274	8.234	17.424

表 A.3: 総実行時間 (秒):T10I4D1000k

PU数 / min sup	1	2	4	8	16	32
0.020	10.260	5.078	2.372	1.132	1.485	1.557
0.015	14.120	6.222	2.909	1.646	1.259	1.152
0.010	16.150	8.716	3.725	2.005	1.453	1.268
0.005	21.130	11.628	7.170	7.483	13.214	31.533