12.12 Data Mining: An Introduction

- What is data mining?
  - **Def**: Data mining is the process of extracting hidden patterns or trends in large data sets for the purpose of prediction.
  - The basic idea: comb through available data, looking for unusual unobvious patterns and report them.
- Why is data mining important?
  - Businesses are interested in exploiting knowledge about patterns.
  - Standard statistical techniques (multivariate analysis) work only on numeric data and with few variables.
- Examples of applications:
  - *Banking.*
    - Suppose a bank sifted through its archives and discovered the following statistic:
      "77 percent of loan defaults involved (1) a customer in the age group 18-21, (2) a car loan for a red sportscar and (3) income group $15,000-$20,000".
    - The bank can use this pattern to avoid giving loans.
    - Similarly, some unobvious patterns can indicate likely attributes of a "good" customer.
    - Today, many banks (e.g., Citibank, Signet) use information extracted by data mining algorithms.

- *Sports.*
  - Suppose a basketball team (e.g., Chicago Bulls) sifted through their records and discovered the following: "On 60 percent of plays in which Scottie Pippen is defended by the opposing guard, the Bulls eventually win the possession."
  - The coach can use this pattern to improve chances of winning.
  - Today, several NBA teams use Advanced Scout, a data mining package to produce such statistics.
- *Retail industry.*
  - By sifting through customer purchase data, a grocery store discovers that "40 percent of customers that buy wine also buy a specialty cheese".
  - The store can use the information in marketing and display strategies.

- Several types of data mining:
  - *Association rule mining*: Find "rules" of the sort "77 percent of loan defaults with attributes A,B,C also have attributes X and Y".
  - *Clustering*: Find groupings of data based on available attributes based on the structure of the data, e.g., "90 percent of renters in the Williamsburg area fall into either the 18-25 or 65-75 age groups".
  - *Classification*: Find natural groupings of data based on available attributes that seek to predict an outcome. e.g. group bank customers into three groups: (1) "most-likely to repay"; (2) "most-likely to default" and (3) "don't know".
  - *Other*: finding patterns in sequences (Stock Market application), deviation detection (Fraud detection application).

- Types of data sets:
  - Most data sets are large relational tables, with many attributes, e.g., bank customers may collect 50-100 attributes on a loan application.
  - Some data sets are unnormalized "basket" data, such as the list of items checked out by each customer at a grocery store.
- Why data mining is an interesting problem:
  - Typical data sets are very large with many attributes, e.g.,
    * Census data: about 400 attributes per individual.
    * Retail store data: millions of transactions, thousands of attribute types.
  - A naive approach of trying all possible rules causes a combinatorial explosion, e.g,
    * Consider a relation $R(A_1, A_2, \ldots, A_{100})$ where each attribute value is boolean.
    * Suppose we are interested in generating rules of the sort $A_{i_1} A_{i_2} \ldots A_{i_k} \rightarrow A_{j_1} A_{j_2} \ldots A_{j_m}$
      e.g., of the 100 records with $A_3 A_4 A_7$ true, 68 of them also have $A_1 A_8$ true.
    * Consider all possible combinations of $A_{i_1} A_{i_2} \ldots A_{i_k} \rightarrow A_{j_1} A_{j_2} \ldots A_{j_m}$.
    * For each such combination, scan relation R to count percentages.

12.13 Association Rule Mining: Introduction

- Consider data collected at a supermarket checkout counter:
  - The system records customer purchases in a variable-size record (unnormalized), e.g.

    r_{257}=<Eggs, bread, pasta, milk, cheese, beer, soap>.

    (Customer 257 bought eggs, bread etc).
  - The system has thousands of such customer purchase-records each day.
  - An *association rule* seeks to answer questions like:
    when pasta and pasta sauce are bought, what is the probability that mushrooms are also purchased?
  - In terms of available data, this question can be rephrased as:
    among those records that contain both pasta and pasta sauce, how many also contain mushrooms?
  - Why is this question useful?
    The answer (if high) can drive pricing and display strategies
    $\Rightarrow$ package discount for the combination of pasta and mushrooms.
  - Suppose our data has 100,000 records, of which
    * 30,000 records contain pasta and pasta sauce;
    * 22,500 of these 30,000 records contain mushrooms.

    Then, we have the *association rule*

    $$\{\text{pasta, pasta sauce}\} \rightarrow \{\text{mushrooms}\}$$

    with

    $$\begin{array}{lcccc} \text{support} & = & 22{,}500/100{,}000 & = & 0.225 \\ \text{confidence} & = & 22{,}500/30{,}000 & = & 0.75 \end{array}$$

  - Intuition: 75 percent of the time when pasta and pasta sauce are bought, mushrooms are also bought.

- Both *support* and *confidence* are important:
  - Consider the rule
    $$\{\text{Non-alcoholic beer}\} \rightarrow \{\text{chips}\}.$$
  - Suppose
    $$\begin{aligned} \text{confidence} &= 0.9 \\ \text{support} &= 0.001 \end{aligned}$$
  - Thus, 90 percent of customers that buy non-alcoholic beer also buy chips.
  - But, this pattern occurs only in 0.1 percent of the data
    $\Rightarrow$ not an important rule.

- **Def**: an association rule $X \rightarrow Y$, where X and Y are sets of attributes, satisfies confidence level c and support s if:
  1. the actual confidence is at least c and
  2. the actual support is at least s.

- The **association rule mining problem**: *given a confidence level* c *and a support level* s, *find* **all** *rules that satisfy* c *and* s.

12.14 Association Rule Mining: Problem Formulation

- Notation:
  - Let $I = \{I_1, I_2, \ldots, I_m\}$ be a set of items.
    Think of I as {eggs,cheese,pasta,...} in the supermarket example. (Set of all possible supermarket products).
  - A subset of items $X \subseteq I$ will sometimes be called an itemgroup.
  - We will use letters like X and Y to denote itemgroups.
  - Let $R = \{r_1, r_2, \ldots, r_n\}$ be a set of unnormalized records (basket data).
    Here, r_i is the set of items bought by customer i,
    e.g. r_{257} ={pasta, pasta sauce, tomatoes, beef, soap}
    Thus, $\forall i : r_i \subseteq I$.
  - **Def**: a record $r \in R$ *contains* itemgroup X if $X \subseteq r$.
  - Let $R(X) = \{r \in R : r \text{ contains } X\}$.
  - For any itemgroup X, let $\alpha(X) = |R(X)|$, the number of records that contain X.
  - For any itemgroup X, define the support of X to be

$$\beta(X) = \frac{\alpha(X)}{|R|}.$$

  - Define

$$F_s(R) = \{X \subseteq I : \beta(X) = \frac{\alpha(X)}{|R|} \geq s\}.$$

    (All the itemgroups satisfying support s).

– **Def**: A rule $X \rightarrow Y$ is an *association rule* satisfying support s and confidence c if

1. X and Y are itemgroups (i.e., $X, Y \subseteq I$).
2. X and Y are disjoint (i.e., $X \cap Y = \emptyset$).
3. At least s fraction of records contain both X and Y, i.e.,

$$\beta(X \cup Y) = \frac{\alpha(X \cup Y)}{|R|} \geq s.$$

4. Of those records containing X, at least c fraction contain Y, i.e.,

$$\frac{\alpha(X \cup Y)}{\alpha(X)} \geq c.$$

- Example: I = {milk, eggs, pasta, pasta sauce, cheese}
  R is given by:

r_1	=	<milk, eggs>
r_2	=	<milk, eggs>
r_3	=	<milk, eggs, cheese>
r_4	=	<milk, pasta, cheese>
r_5	=	<eggs, pasta sauce, cheese>
r_6	=	<pasta, pasta sauce>
r_7	=	<pasta, pasta sauce, cheese>
r_8	=	<pasta, pasta sauce, cheese>
r_9	=	<milk, eggs, pasta, pasta sauce, cheese>
r_{10}	=	<milk, pasta, pasta sauce, cheese>

– Consider X={eggs, pasta sauce}. Then,

$$\begin{aligned} R(X) &= \{r_5, r_9\} \\ \alpha(X) &= 2 \\ \beta(X) &= \frac{2}{10} = 0.2 \end{aligned}$$

– Consider X={milk, eggs}.

$$\begin{aligned} R(X) &= \{r_1, r_2, r_3, r_9\} \\ \alpha(X) &= 4 \\ \beta(X) &= \frac{4}{10} = 0.4 \end{aligned}$$

– Consider X={milk,eggs} and Y={eggs}
⇒ not a valid rule since $X \cap Y \neq \emptyset$.

– $X \rightarrow Y$ is a potential association rule where X={milk} and Y={eggs,pasta,cheese}.

– Consider X={milk,eggs} and Y={cheese}. Then

$$\begin{aligned} |R| &= 10 \\ \alpha(X) &= 4 \\ \alpha(X \cup Y) &= 2 \\ \beta(X \cup Y) &= \frac{2}{10} = 0.2 \end{aligned}$$

Hence

$$\begin{aligned} \text{support} &= \beta(X \cup Y) = \frac{\alpha(X \cup Y)}{|R|} = \frac{2}{10} = 0.2 \\ \text{confidence} &= \frac{\alpha(X \cup Y)}{\alpha(X)} = \frac{2}{4} = 0.5 \end{aligned}$$

Thus, $X \rightarrow Y$ with support 0.2 and confidence 0.5.

– Suppose $s = 0.3$. Since $|R| = 10$, we want all itemgroups that appear in at least 3 records, i.e.,

$$F_{0.3}(R) = \{X \subseteq I : \beta(X) = \frac{\alpha(X)}{|R|} \geq 0.3\}.$$

Here, $F_{0.3}(R)$ ={ {milk}, {eggs}, {pasta}, {pasta sauce}, {cheese}, {milk,eggs}, {pasta, pasta sauce}, {milk,pasta}, {milk,cheese}, {pasta,cheese}, {pasta,pasta sauce,cheese}, {milk,eggs,cheese}, {milk,pasta,cheese} }.

- **Def**: A rule $X \rightarrow Y$ is a 1-RHS rule if $|Y| = 1$.
(Right-hand side has only one item).

- Typical restriction on problem: find all 1-RHS association rules (satisfying given s and c).

  Examples of 1-RHS rules from above:

{milk}	$\rightarrow$	{eggs}
{eggs}	$\rightarrow$	{milk}
{pasta}	$\rightarrow$	{cheese}
{pasta,cheese}	$\rightarrow$	{milk}
{milk,eggs}	$\rightarrow$	{cheese}

  Note that

$$\{\text{milk}\} \rightarrow \{\text{eggs,cheese}\}$$

  is a rule but not a 1-RHS rule.

- An observation:
  - Suppose we have computed $\beta(X)$ (support) for each possible itemgroup X.
  - Consider a rule $X \rightarrow Y$. Then,

$$\begin{aligned} \frac{\beta(X \cup Y)}{\beta(X)} &= \frac{\alpha(X \cup Y)/|R|}{\alpha(X)/|R|} \\ &= \frac{\alpha(X \cup Y)}{\alpha(X)} \\ &= \text{confidence of rule } X \rightarrow Y \end{aligned}$$

  - Thus, given only support numbers for itemgroups, we can compute rule confidences.
  - Also, for a rule $X \rightarrow Y$ to meet the required support s, we must have $\beta(X \cup Y) \geq s$
    $\Rightarrow X \cup Y \in F_s(R)$.
  - Note that $\beta(X \cup Y) \geq s \Rightarrow \beta(X) \geq s$.
    $\Rightarrow X, X \cup Y \in F_s(R)$.
    $\Rightarrow$ The association rule mining problem reduces to finding itemgroups with large enough support, i.e., computing $F_s(R)$.
  - Thus, for the remainder we will focus on simply identifying $F_s(R)$, the set of itemgroups with large enough support.

    Typically, we will want to output each itemgroup and its actual support.

12.15 Two Naive Algorithms

- **Algorithm**: NAIVE-1 (R, I, s)
  - Generate all possible itemgroups and initialize a counter for each.

    Note: All possible itemgroups = 2^I = all possible subsets of I.
  - Scan R once and count support for each itemgroup.
  - Output those itemgroups satisfying s.
- Analysis of NAIVE-1:
  - How many possible itemgroups with $I = \{I_1, \ldots, I_m\}$?
    $\Rightarrow \left|2^I\right| = 2^{|I|} = 2^m$
  - If m is large (say, $m > 100$), 2^m is too big for main memory.
  - Also, if $|F_s(R)|$ is small, we waste time updating counts for itemgroups not in $F_s(R)$.
- **Algorithm**: NAIVE-2 (R, I, s)
  - **while** not over **do**
    - Generate a new itemgroup.
    - Scan R to obtain support.
    - **if** support $\geq s$, retain itemgroup.
  - **endwhile**
  - Output all itemgroups retained.
- Analysis of NAIVE-2:
  - Too many scans of the data.

- **Key observation**: $X \notin F_s(R) \Rightarrow X \cup Y \notin F_s(R)$ for any Y.
  (If itemgroup X does not satisfy s, neither will any extension of X such as $X \cup Y$)

  Example: if {milk} occurs in only 0.1 fraction of records, then {milk,eggs} occurs in no more than 0.1 fraction of records.

  This observation is used in better algorithms.

- We will use the following example for illustration:

$$
\begin{aligned}
I &= \{A, B, C, D, E\} \\
r_1 &= < A, B > \\
r_2 &= < A, B > \\
r_3 &= < A, B, E > \\
r_4 &= < A, C, E > \\
r_5 &= < B, D, E > \\
r_6 &= < C, D > \\
r_7 &= < C, D, E > \\
r_8 &= < C, D, E > \\
r_9 &= < A, B, C, D, E > \\
r_{10} &= < A, C, D, E >
\end{aligned}
$$

12.16 Algorithm Record-Derived-Itemgroups

- Key ideas:
  - Consider the itemgroup ABD and the item E. If ABD has poor support then so does $ABDE$, the extension of ABD to E.
    Thus, $\beta(ABD) < s \Rightarrow \beta(ABDE) < s$.
  - We will assume the items are lexicographically ordered.
    Thus, we will extend ACF to $ACFG$ but not $ACDF$.
    (Because $ACDF$ will be considered when ACD is extended).
  - The support for a tentative extension can be estimated using independence:
    $$\hat{\beta}(ABDE) = \beta(ABD)\beta(E).$$
    For example, if $\beta(ABD) = 0.4$ (40 percent of R) and $\beta(E) = 0.6$ is known from previous iterations, then
    $$\hat{\beta}(ABDE) = 0.4 \times 0.6 = 0.24.$$
    Of course, independence may turn out to be a poor approximation.
  - The algorithm makes multiple scans of data. In each scan:
    * Counts are maintained for various itemgroups.
    * At the end of each scan, itemgroups with low support are discarded.
    * As each record is encountered, the items within it are used to create new potential itemgroups.
    * If the estimated support is high, additional extensions are considered.
    * If the estimated support is low, an itemgroup is placed in a Next_Frontier set (to be re-examined at the end of the pass).

- Generally, if an itemgroup was mistakenly placed in the Next_Frontier set (support underestimated), it's count will actually be high, and therefore is considered for extension later.
- If an itemgroup was mistakenly extended too much (support overestimated), it will be discarded at the end of the scan.
- Example: consider the itemgroup A, B and the record $\{A, B, D, E, F\}$.
  * Itemgroup AB can be extended to create the following potential itemgroups: ABD, ABE, ABF, $ABDE$, $ABDF$, $ABEF$ and $ABDEF$.
  * Suppose it turns out

$$\begin{array}{lcl} \hat{\beta}(ABD) & \geq & s \\ \hat{\beta}(ABE) & < & s \\ \hat{\beta}(ABF) & \geq & s \end{array}$$

    Then,
    - ABD can be extended to the next size ($ABDE$ or $ABDF$) lexicographically.
    - ABE is not extended (and placed in Next_Frontier).
    - ABF cannot be extended because the record has nothing beyond F.
    - Since ABD got extended to $ABDE$, we consider expanding $ABDE$ to $ABDEF$ (if the estimated support is good).
- Why are low-estimate itemgroups kept around in Next_Frontier?
  $\Rightarrow$ need to compute counts in case estimate was bad
  $\Rightarrow$ they may still satisfy s.

- Pseudocode:
  - Note: A first pass is done separately to initialize counts for the 1-item itemgroups.
  - Assume that the set of items is $I = \{I_1, \ldots, I_m\}$.

Algorithm: RECORD-DERIVED-ITEMSETS (R, I, s)

Input: Set of records R, set of items I, support s.
Output: Collection of itemgroups with large enough support.

```
1.  Large := ∅;
    // First pass
2.  ∀k : α[I_k] := 0 // Initialize counts
3.  for j := 1 to |R| do
4.    for k := 1 to m do
5.      if I_k ∈ r_j
6.        α[I_k] := α[I_k] + 1;
7.  for k := 1 to m
8.    β[I_k] := α[I_k]/|R|;
9.    if β[I_k] ≥ s
10.     Large := Large ∪ {I_k};
11. endfor;
    // All other passes.
12. Frontier := I; // Keep Frontier sorted by size.
13. while Frontier ≠ ∅
14.   H := ∅;
15.   for j := 1 to |R| // Scan data.
16.     for each itemgroup X ∈ Frontier
17.       if X ∈ record r_j
18.         G := COMPUTE-EXTENSIONS (X, I, r);
19.       for each Y ∈ G
20.         if Y ∈ H
21.           Y.count := Y.count + 1;
22.         else
23.           H := H ∪ {Y};
24.           Y.count := 1;
25.         endif;
26.       endfor;
27.     endfor;
28.   endfor;
```

... continued

Algorithm: RECORD-DERIVED-ITEMSETS ... continued

```
        // Identify itemgroups that satisfy s.
29.     for each Y ∈ H
30.       if Y.count/|R| ≥ s
31.         Large := Large ∪ {Y};
32.   // Set next frontier to be considered for extension
33.     Frontier := Next_Frontier ∩ Large;
34.   endwhile;
35.   return Large;
```

Algorithm: COMPUTE-EXTENSIONS (X, I, r)

Input: an itemgroup X, the set of items I, record r.
Output: Extensions of X.

```
1.  k := |X|;
2.  G' := {X};
3.  repeat
4.    No_change := true;
      // Compute extensions for size k.
5.    for each Y ∈ G' such that |Y| = k
        // Suppose Y = I_{j1} I_{j2} ... I_{jk}.
6.      for l := j_k + 1 to m
7.        if I_l ∈ record r
8.          Z := Y ∪ {I_l};
            // See if Z is worth the trouble.
9.          β̂[Z] := β̂[Y] * β[I_l];
10.         if β̂[Z] ≥ s
11.           G' := G' ∪ {Y};
12.           No_change := false;
13.         else
14.           Next_Frontier := Next_Frontier ∪ Y;
15.         endif;
16.       endif;
17.     endfor;
18.   endfor;
19.   k := k + 1;
20. until No_change or k = m;
21. return G := G'∪Next_Frontier;
```

- Example: $s = 0.3$
  - First pass: $\alpha(A) = 6$, $\alpha(B) = 5$, $\alpha(C) = 6$, $\alpha(D) = 6$, $\alpha(E) = 7$.

$\beta(A) = 0.6$, $\beta(B) = 0.5$, $\beta(C) = 0.6$, $\beta(D) = 0.6$, $\beta(E) = 0.7$.
Frontier $= \{A, B, C, D, E\}$

– Second pass:

1. When $r_1 =< A, B >$ is scanned:
   * The only possible extension is AB (BA is not considered because it is not a lexicographic extension).
   * $\hat{\beta}(AB) = \hat{\beta}(A)\hat{\beta}(B) = 0.3$.
     $\Rightarrow AB.\text{count} := 1$.
   * $H = \{AB\}$.
2. When $r_2 =< A, B >$ is scanned:
   * Only extension possible is AB.
   * AB.count $= 2$, $H = \{AB\}$.
3. When $r_3 =< A, B, E >$ is scanned:
   * Extensions (with support):
     $G = \{AB(0.3), AE(0.42), BE(0.35), ABE(0.21)\}$.
   * AB.count=3, BE.count=1, ABE.count=1.
   * $H = \{AB, AE, BE, ABE\}$.
   * Next_Frontier=$\{ABE\}$ (it's estimate was not high enough).
4. When $r_4 =< A, C, E >$ is scanned:
   * $G = \{AC(0.36), AE(0.42), ACE(0.252), CE(0.42)\}$.
   * Counts: $\boldsymbol{AB(3)}, \boldsymbol{AC(1)}, \boldsymbol{AE(2)}, ABE(1), \boldsymbol{ACE(1)}, BE(1), \boldsymbol{CE(1)}$.
   * Next_Frontier=$\{ABE, ACE\}$.
5. When $r_5 =< B, D, E >$ is scanned:
   * $G = \{BD(0.3), BDE(0.21),$
     $BE(0.35), DE(0.42)\}$.
   * Counts: $AB(3), AC(1), AE(2), ABE(1), ACE(1), \boldsymbol{BD(1)}, \boldsymbol{BE(2)}, \boldsymbol{BDE(1)}, CE(1), \boldsymbol{DE(1)}$.
   * Next_Frontier=$\{ABE, ACE, ADE, BDE\}$.
6. When $r_6 =< C, D >$ is scanned:

* $G = \{CD(0.36)\}$.
* Counts: $AB(3), AC(1), AE(2), ABE(1), ACE(1), BD(1), BE(2), BDE(1), \boldsymbol{CD(1)}, \boldsymbol{CE(1)}, DE(1)$.
* Next_Frontier=$\{ABE, ACE, ADE, BDE\}$. (Unchanged).

7. When $r_7 =< C, D, E >$ is scanned:
   * $G = \{CD(0.36), CE(0.42), CDE(0.252), DE(0.42)\}$.
   * Counts: $AB(3), AC(1), AE(2), ABE(1), ACE(1), BD(1), BE(2), BDE(1), \boldsymbol{CD(2)}, \boldsymbol{CE(2)}, \boldsymbol{CDE(1)}, \boldsymbol{DE(2)}\}$.
   * Next_Frontier=$\{ABE, ACE, ADE, BDE, CDE\}$.
8. When $r_8 =< C, D, E >$ is scanned:
   * $G = \{CD(0.36), CE(0.42), CDE(0.252), DE(0.42)\}$.
   * Counts: $AB(3), AC(1), AE(2), ABE(1), ACE(1), BD(1), BE(2), BDE(1), \boldsymbol{CD(3)}, \boldsymbol{CE(3)}, \boldsymbol{CDE(2)}, \boldsymbol{DE(3)}\}$.

Continuing, the large itemgroups turn out to be:

Large = $\{A, B, C, D, E, AB, AC, AE, ACE, BE, CD, CE, CDE, DE\}$.

– Third pass:

* Here, the size 3 itemgroups are $\{ACE, CDE\}$.
  They were expected-small but turned out to have enough support.
* These can't be expanded (they end in E)
  $\Rightarrow$ we're done.

– Final result:

$$A, B, C, D, E, AB, AC, AE, ACE, BE, CD, CE, CDE, DE.$$

12.17 Algorithm Pass-Derived-Itemgroups

- In the previous algorithm, the following problem arises:
  - Suppose a record has items $< A, B, C, D, E, F >$ and suppose that $\{AC, AD, CD, CE\}$ are in the current Frontier.
  - Potential extensions include $\{ACF, ADF, CDF, CEF\}$.
  - If all of them have small expectations, we're still going to maintain counts for them
    $\Rightarrow$ the algorithm wastes time counting useless itemgroups.
  - We will try to minimize this problem in the next algorithm.
- Key ideas in Algorithm Pass-Derived-Itemgroups:
  - In pass k only itemgroups of size k are considered.
  - At the end of pass $k-1$, we will have counts for the large itemgroups of size $k-1$
    $\Rightarrow$ we know L_{k-1}=$\{$large itemgroups of size $k-1\}$.
  - Before starting pass k, we compute all possible itemgroups of size k.
  - Example:
    * Suppose in pass $k = 5$ we generate $ACDEF$ has a potential itemgroup.
    * We consider all possible $(k-1)$-size subgroups, such as $ACEF$ and $ADEF$.
    * If any of these subgroups is not in L_{k-1}, we can reject $ACDEF$ immediately.
  - Another idea used is to generate potential groups in an intelligent way (exploiting lexicographic order):
    * Suppose $k = 5$ and $L_4 = \{ABCD, ABCE, BCDE, BCEF\}$.

* The naive way to generate size-5 itemgroups would be to consider all possible extensions of the above four itemgroups.
* Note instead, that the itemgroup $ABCDE$ will be large only if $ABCD$ and $ABCE$ are already large,
  i.e., only if both $ABCD$ and $ABCE$ are in L_4.
  $\Rightarrow$ we will allow $ABCDE$ to be generated only from $ABCD$ and $ABCE$.
* In general, we will generate $X_1X_2 \ldots X_{k-1}X_k$ from $X_1X_2 \ldots X_{k-2}X_{k-1}$ and $X_1X_2 \ldots X_{k-2}X_k$.
* Interestingly, this "combining" can be stated as a join:
  - Note that L_{k-1} is a collection of size-$(k-1)$ itemgroups.
  - Suppose that each itemgroup is considered a tuple in a relation called L_{k-1}, where the i-th attribute in a tuple is the i-th item in the itemgroup.
  - Example ($k = 5$): the tuple for $ABCE$ will be the tuple $< A, B, C, E >$.
  - Suppose the attribute names are item$_1$,..., item$_{k-1}$.
  - The join statement is:

```
select   p.item_1, ..., p.item_{k-1}, q.item_{k-1}
from     L_{k-1} as p, L_{k-1} as q
where    p.item_1 = q.item_1
         ...
         and p.item_{k-2} = q.item_{k-2}
         and p.item_{k-1} < q.item_{k-1}
```

– One additional observation:
  * Consider $k = 6$ and suppose the join resulted in $ABCDEF$.
  * We now need to look at all possible subgroups (of size $k-1$) of $ABCDEF$.
  * How many possible subgroups are there?
    $\Rightarrow$ at most 6 (drop one letter at a time for each size 5 string).

- Pseudocode:

Algorithm: PASS-DERIVED-ITEMGROUPS (R, I, s)

Input: Set of records R, set of items I, support s.
Output: Collection of itemgroups with large enough support.

```
        // First pass
 1.     ∀k :  α[I_k] := 0 // Initialize counts
 2.     for j := 1 to |R| do
 3.       for k := 1 to m do
 4.         if I_k ∈ r_j
 5.           α[I_k] := α[I_k] + 1;
        // L[1] := Build-Set (∅);
 6.     for k := 1 to m
 7.       if α[I_k]/|R| ≥ s
 8.         L_1 := L_1 ∪ {I_k}; // Add-Set (L[1], I_k);
        // All other passes.
 9.     k := 2;
 10.    while L_{k-1} ≠ ∅ // Set-Not-Empty (L[1]);
 11.      C := Compute-Join (L_{k-1}, L_{k-1});
 12.      // C := Build-Set (Compute-Join (L_{k-1}, L_{k-1}));
 13.      for each itemgroup X = I_{j_1} ... I_{j_k} ∈ C // Winnowing
 14.        l := 1;       over := false;
 15.        while l ≤ k − 2 and not over
 16.          if Y = I_{j_1} ... I_{j_{j-1}} I_{j_{l+1}} ... I_{j_k} ∉ L_{k-1} // Not-In-Set (L[k − 1], Y)
 17.            C := C − {Y}; // Remove-Element (C, Y)
 18.            over := true;
 19.          else
 20.            l := l + 1;
 21.          endif;
 22.        endwhile;
 23.      endfor;
        ... continued
```

Algorithm: PASS-DERIVED-ITEMGROUPS ... continued

```
24.    for i := 1 to R do // Check counts
25.      for each k-sized itemgroup X ∈ r_i do
26.        if X ∈ C // SET-MEMBER-OF (X, C)
27.          X.count := X.count + 1;
28.      endfor;
29.    endfor;
30.    L_k := {X ∈ C : X.count≥ s};
31.    // Use BUILD-SET and ADD-SET here.
32.    k := k + 1;
33.  endwhile;
34.  return ∪_{k≥1} L_k;
```

NOTE:

- The join computation is not shown.
- Some set operations are shown mathematically, with comments indicating the kinds of set-manipulation functions needed.

- Example: (same example as before with $s = 0.3$)
  - First pass: $\alpha(A) = 6$, $\alpha(B) = 5$, $\alpha(C) = 6$, $\alpha(D) = 6$, $\alpha(E) = 7$.
    $L_1 := \{A, B, C, D, E\}$.
  - Second pass ($k = 2$):
    * The (L_1, L_1)-join gives $C = \{AB, AC, AD, BC, BD, BE, CD, CE, DE\}$.
    * Since each 1-size subset of each of these is in L_1, the winnowing does not remove anything from C.
    * After a scan, the counts obtained are:

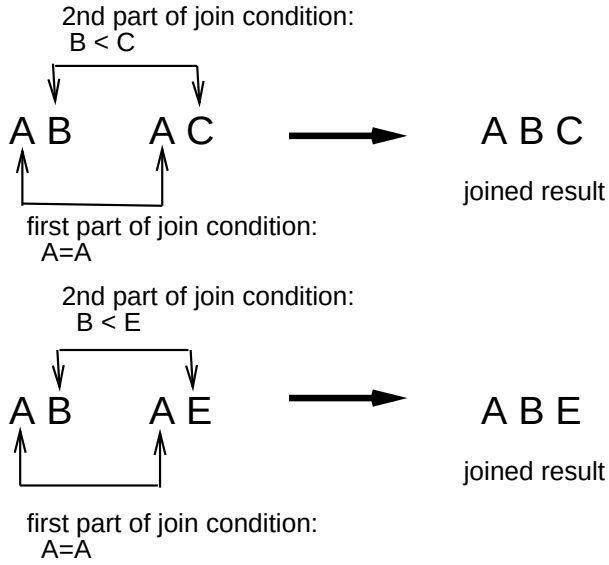
      $C = \{AB(4), AC(3), AD(2), BC(1), BD(2), BE(3), CD(5), CE(5), DE(5)\}$.

* Those with high enough count (3 or more) are retained:

$$L_2 \leftarrow \{AB, AC, AE, BE, CD, CE, DE\}$$

– Third pass ($k = 3$):

* The (L_2, L_2)-join gives $C = \{ABC, ABE, ACE, CDE\}$, e.g.,



* *Winnowing*:
  · For ABC, we need to check whether $BC \in L_2$
    $\Rightarrow$ not in L_2
    $\Rightarrow$ discard ABC.
  · For ABE, we need to check whether $BE \in L_2$
    $\Rightarrow$ $BE \in L_2$
    $\Rightarrow$ retain ABE.
  · Continuing, we find that ABE, ACE, CDE are retained.
* After a scan, the counts are:

$$ABE(2), ACE(3), CDE(4).$$

* Those with high enough count (3 or more) are retained:

$$L_3 = \{ACE, CDE\}.$$

– Fourth pass ($k = 4$):

  * The (L_3, L_3)-join is empty.

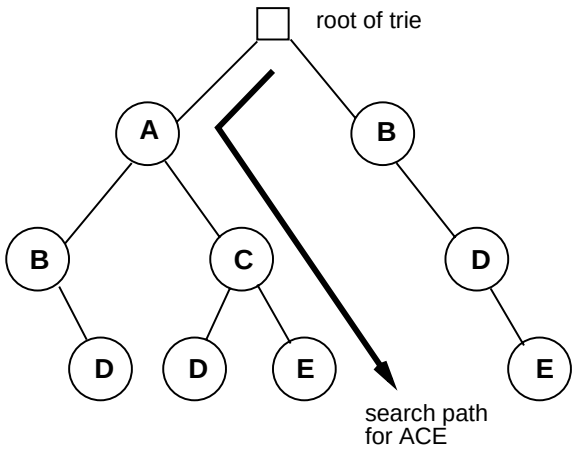
– Final result:

$$A, B, C, D, E, AB, AC, AE, BE, CD, CE, DE, ACE, CDE.$$

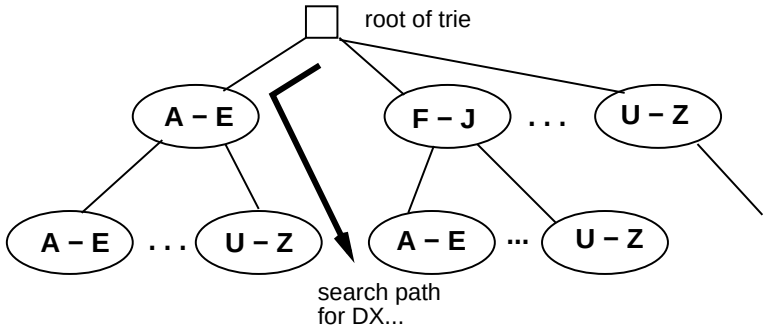
12.18 Using Hashtrees: The Apriori Algorithm

- Recall that in the k-th pass of PASS-DERIVED-ITEMGROUPS:
  - A set of size-k itemgroups is computed via a join.
  - The set is pruned using the size-$(k-1)$ itemgroups.
  - A scan is made to generate the count for each itemgroup.
- A run-time profile of the previous algorithm shows that a lot of time is spent in generating counts:
  - For each record, we need to figure out which counts should be updated.
  - Example:
    * Suppose $C = \{ABC, ABD, ABE, ACE, BCE, BDE, CDE\}$.
    * Consider a record $< A, B, D, E, F >$.
      Which of the above itemgroups occur in the record?
    * One way of checking: **Record Subset method**
      1. generate all possible size-3 itemgroups in the record:
         $ABD, ABE, ABF, ADE, ADF, AEF, BDE, BDF, BEF, DEF$
         (10 itemgroups).
      2. For each such itemgroup, check whether it is in C.
    * For a record with n items and size-k itemgroups: $\begin{pmatrix} n \\ k \end{pmatrix}$.
      $\Rightarrow$ very large for even moderate sizes (e.g.,$n = 20, k = 10$).
    * Another approach: **Itemgroup Scan method**
      1. Scan each itemgroup in C.
      2. Test whether each itemgroup is in the given record.
         $\Rightarrow$ will be slow if number of itemgroups is large.

- Using a trie in the Record Subset method:
  - To test whether whether ABC is in C, one approach is to test the string ABC against each itemgroup in C.
  - A faster approach is to use a suitable data structure, such as a trie.
  - Example: suppose $C = \{ABD, ACD, ACE, BDE\}$



  - However, if the number of items is large (.e.g, 10^5), the trie could be very wide
    $\Rightarrow$ may not fit in memory.
  - To reduce branching factor, some paths can be coalesced:



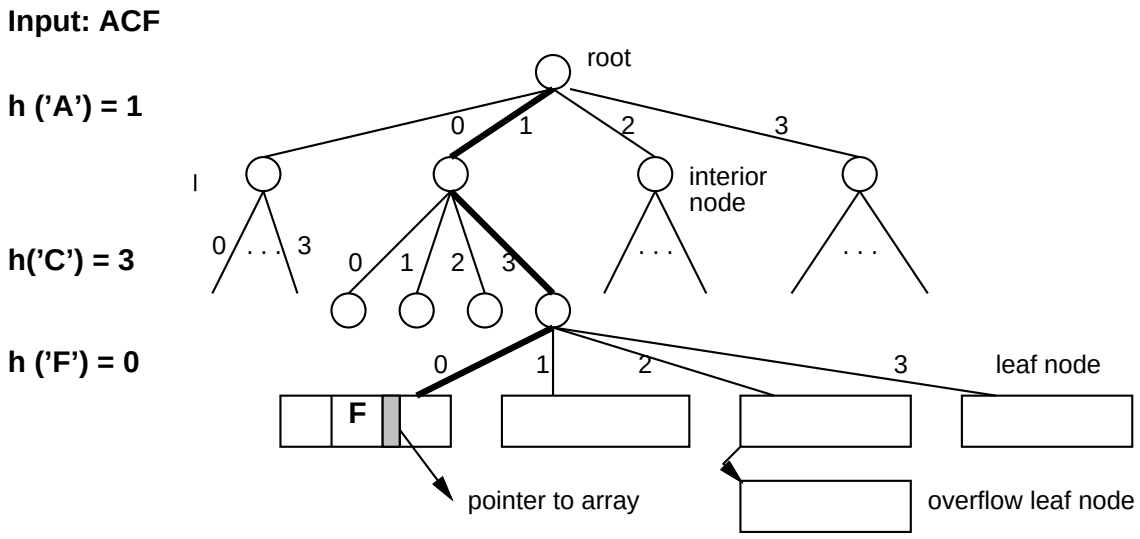
    This is the basic idea used in the hashtree.

- Hashtrees: key ideas
  - Recall: in pass k, C is a list of size-k itemgroups.
  - In pass k, a fresh hashtree is constructed for size-k itemgroups.
  - The list of itemgroups is stored in an array, e.g.

```
typedef struct itemgroup_type {
  char *itemgroup;   // The actual itemgroup
  int count;         // The count, initially set to zero
} itemgroup_type;
itemgroup_type *itemgroup_array;
  ...
// Later
   itemgroup_array[i].count ++; // Incrementing the count
```

  - The hashtree is like a B+-tree in some ways:
    - The tree stores pointers to the actual data.
    - In this case, the correct index into the `itemgroup_array` is stored.
    - The hashtree has *internal* and *leaf* nodes.
    - Internal nodes are used for navigation.
    - Leaf nodes contain pointers (offsets) to the itemgroup array.
  - The hashtree is also different in many ways:
    - The leaf nodes are not linked.
    - The search is not in-order: which branch to take depends on a hashing function.

- Example:
  - Consider a hashtree with *branching factor* = 4.
    (Typically, branching factor is higher).
  - Suppose we want to check whether the itemgroup ACF is in the itemgroup array:

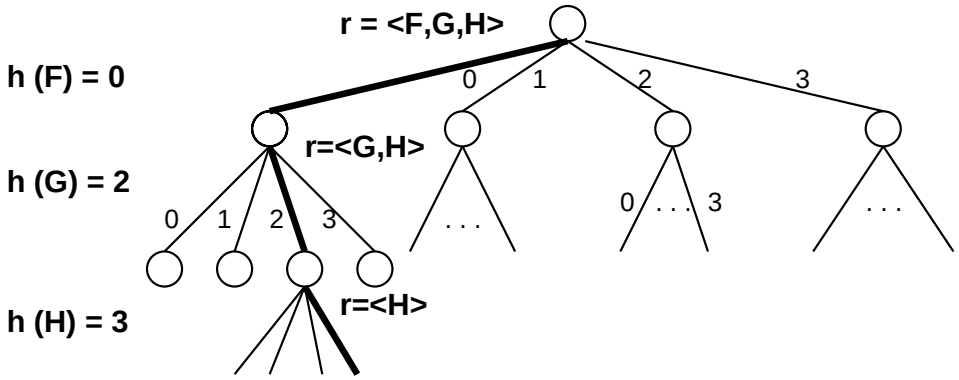


    * Apply the hashing function to A at the first level, to C at the next level and F at the third level.
    * Once at the leaf, search for F in the leaf and follow the pointer to the array.
    * Note: the depth of the hashtree is always the itemgroup size
      $\Rightarrow$ we will always stop at the leaf level with the last item.

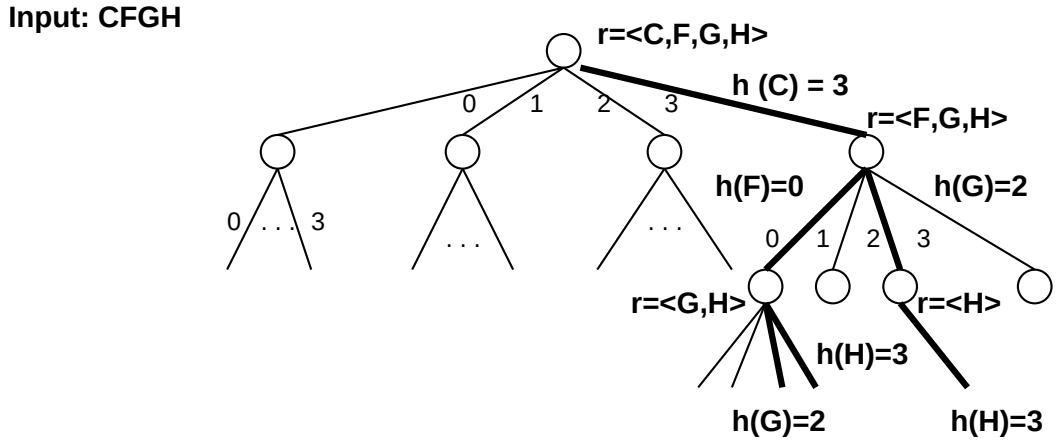
- Insertion:
  - First insert the itemgroup in the itemgroup array, and note the array offset (pointer).
  - Then find the appropriate leaf by doing a regular search.
  - Insert in sorted order in the leaf, along with pointer to the itemgroup array.
  - If leaf is full, extend by adding an overflow leaf node.

- Checking which subsets are in a record:
  - In a search, we are given a record (e.g., $< A, C, F, G, H >$) and we want to know which itemgroups are in the record
    $\Rightarrow$ need to increment their counters.
  - One approach:
    * Generate all possible size-k itemgroups of the record.
    * For each such itemgroup, traverse the hashtree and see if it exists in the itemgroup array.
    * For every itemgroup that is found, increment the counter.
  - Recursive approach:
    * Rather than generate all possible size-k itemgroups, a simple recursive method can be used.
    * Observation: each subtree of the root is a size-$(k-1)$ hashtree.
    * Thus, to check all itemgroups beginning with A:
      · Hash $A \Rightarrow$ say, we get the '1' branch.
      · The remainder of the record is $< C, F, G, H >$.
      · Now apply the function recursively to the '1' subtree with record $< C, F, G, H >$.
    * Now, size-3 itemgroups in the record $< A, C, F, G, H >$ can start with any one of the items A, C or F.
    * Thus, we do hash-search at the root for each of these (and the reminder of the record).
    * In the case of starting with F, the only possibility is to hash G at the next level, then H at the third level.

Input: FGH



* But if we start with C at the root:
  - At the next level we have $< F, G, H >$.
  - Both F and G are hashed separately because we have three possible size-2 itemgroups: FG, FH and GH.
    $\Rightarrow$ the first items are F and G.



- Pseudocode:

  The algorithm has been called the Apriori Algorithm in the literature (because "checking for low-support subgroups" is done *prior* to a scan).

Algorithm: APRIORI (R, I, s)

Input: Set of records R, set of items I, support s.
Output: Collection of itemgroups with large enough support.

```
        // First pass
 1.     ∀k :  α[I_k] := 0 // Initialize counts
 2.     for j := 1 to |R| do
 3.       for k := 1 to m do
 4.         if I_k ∈ r_j
 5.           α[I_k] := α[I_k] + 1;
 6.     h := Hashtree-Create (1) // Size = 1.
 7.     for k := 1 to m
 8.       if α[I_k]/|R| ≥ s
 9.         Hashtree-Insert (I_k);
        // All other passes.
10.     k := 2;
11.     while Hashtree-Not-Empty (h)
12.       C := Compute-Join (L_{k-1}, L_{k-1});
13.       h' := Hashtree-Create (k);
14.       for each itemgroup X = I_{j_1} ... I_{j_k} ∈ C // Winnowing
15.         l := 1;       valid := true;
16.         while l ≤ k − 2 and valid
17.           // Y = I_{j_1} ... I_{j_{j-1}} I_{j_{l+1}} ... I_{j_k}.
18.           if not Hashtree-Recursive-Search (h, Y)
19.             valid := false;
20.           else
21.             l := l + 1;
22.           endif;
23.         endwhile;
24.         if valid // All subgroups checked out
25.           Hashtree-Insert (h', X);
26.       endfor;
        ... continued
```

Algorithm: APRIORI ... continued

```
27.    HASHTREE-DESTROY (h);
28.    h := h';
29.    for i := 1 to |R| do
30.       HASHTREE-UPDATE-COUNTS (h, r_i, k);
       // Remove all itemgroups with low counts.
31.    for each itemgroup X ∈ h do // X is in array
32.       if X.count < s
33.          HASHTREE-REMOVE-ITEMGROUP (h, X);
34.    k := k + 1;
35. endwhile;
36. return ∪_{k≥1} L_k;
```

Algorithm: HASHTREE-UPDATE-COUNTS (h, r, k)

Input: hashtree id h, record r, size k.
Output: Counts are updated.

```
       // Suppose r =< I_{j_1}, I_{j_2}, ..., I_{j_l} >.
1.     for p := 1 to l − k
2.        HASHTREE-RECURSIVE-UPDATE (r, p, k, h.root);
3.     return;
```

Algorithm: HASHTREE-RECURSIVE-UPDATE (r, p, k, node)

Input: record r, offset p, size k, hashtree node.
Output: Counts are updated.

```
        // Suppose r =< I_{j_1}, I_{j_2}, ..., I_{j_l} >.
1.      if node.leaf = true // Bottom out of recursion.
2.        if I_{j_p} ∈ node
3.          follow pointer to itemgroup array and increment count;
            // Note: count should be incremented only once
            // for each record.
4.          return;
5.        endif;
6.      endif;
7.      c := hashfunction (I_{j_p});
8.      node2 := node.child[c];
9.      for q := p + 1 to l − k
10.       HASHTREE-RECURSIVE-UPDATE (r, q, k − 1, node2);
11.     return;
```

Algorithm: HASHTREE-RECURSIVE-SEARCH (node, Y, i)

Input: hashtree node, itemgroup Y, offset i.
Output: true if itemgroup is in tree, false otherwise.

```
        // Assume Y = I_{j_1} ... I_{j_k}.
  1.    if node.leaf = true
  2.       if I_{j_k} ∈ node
  3.          return pointer to location in itemgroup array;
  4.       else
  5.          return NULL; // false
  6.    else
  7.       c := hashfunction (I_{j_1});
  8.       node2 := node.child[c];
  9.       return HASHTREE-RECURSIVE-SEARCH (node2, Y, i + 1);
  10.   endif;
```

12.19 On-Line Analytical Processing (OLAP): Introduction

- On-Line Analytical Processing (OLAP) is the term used for a class of *aggregate* queries.
- Consider an airline (McValue Airlines) with the following data

  SALES (YEAR, CONTINENT, FLT_TYPE, REVENUE).

  where the FLT_TYPE is given by

FLT_TYPE	DESCRIPTION
1	Short-domestic
2	Long-domestic
3	International

  and where the data in SALES is: (revenue in millions)

SALES	YEAR	CONTINENT	FLT_TYPE	REVENUE
	1997	Europe	1	125
	1997	Europe	2	50
	1997	Europe	3	225
	1997	Asia	1	25
	1997	Asia	2	75
	1997	Asia	3	100
	1997	N.America	1	325
	1997	N.America	2	450
	1997	N.America	3	75
	1998	Europe	1	110
	1998	Europe	2	40
	1998	Europe	3	200
	1998	Asia	1	20
	1998	Asia	2	130
	1998	Asia	3	50
	1998	N.America	1	460
	1998	N.America	2	170
	1998	N.America	3	30

Note: number of tuples = 2 YEARs × 3 CONTINENTS × 3 FLT_TYPEs = 18.

- Typical queries:
  - What is the total 1997 revenue?
  - Output the total revenue in each continent year-by-year.
  - What is the total 1997 revenue for International flights?
  - What is the total revenue on European domestic (long and short) flights across all years?
  - What is the maximum revenue in any European flight category in any year?

  NOTE:
  - All the queries involve aggregate functions (**sum** and **max** above).
  - The queries involve aggregates across various subsets of the attributes.

  The answers:
  - What is the total 1997 revenue?
    ⇒ $1,450 million.
  - Output the total revenue in each continent year-by-year.

1997	Europe	400
1998	Europe	350
1997	Asia	200
1998	Asia	200
1997	N.America	850
1998	N.America	660

  - What is the total 1997 revenue for International flights?
    ⇒ $400 million.
  - What is the total revenue on European domestic (long and short) flights across all years?
    ⇒ $325 million.

- What is the maximum revenue in any European flight category in any year?
  ⇒ $225 million (International).

- Computing the queries in SQL:
  - What is the total 1997 revenue?

```
select     S.YEAR, sum(S.REVENUE)
from       SALES S
where      S.YEAR = 1997
group by   S.YEAR
```

  - Output the total revenue in each continent year-by-year.

```
select     S.YEAR, S.CONTINENT, sum(S.REVENUE)
from       SALES S
group by   S.YEAR, S.CONTINENT
order by   S.CONTINENT
```

  - What is the total 1997 revenue for International flights?

```
select     S.FLT_TYPE, sum(S.REVENUE)
from       SALES S
where      S.YEAR=1997 and S.FLT_TYPE=3
group by   S.FLT_TYPE
```

  What is the cost of computation?

  - Consider the query "Output the total revenue in each continent year-by-year."
    * Need to sort data by continent and year.
    * After sort, aggregates can be computed in a single scan.
  - If data was sorted by (CONTINENT, YEAR), then it must be re-sorted for aggregates on (FLT_TYPE).
  - Generally, if the data is already sorted according to the desired output, one scan is required.
  - Otherwise, a sort is also needed.

12.20 OLAP: The CUBE View

- Most OLAP applications consider data with $m + 1$ attributes in which m attributes are "parameter" attributes and the $(m + 1)$-st attribute is the "aggregate" attribute.

  E.g., in

  SALES (YEAR, CONTINENT, FLT_TYPE, REVENUE)

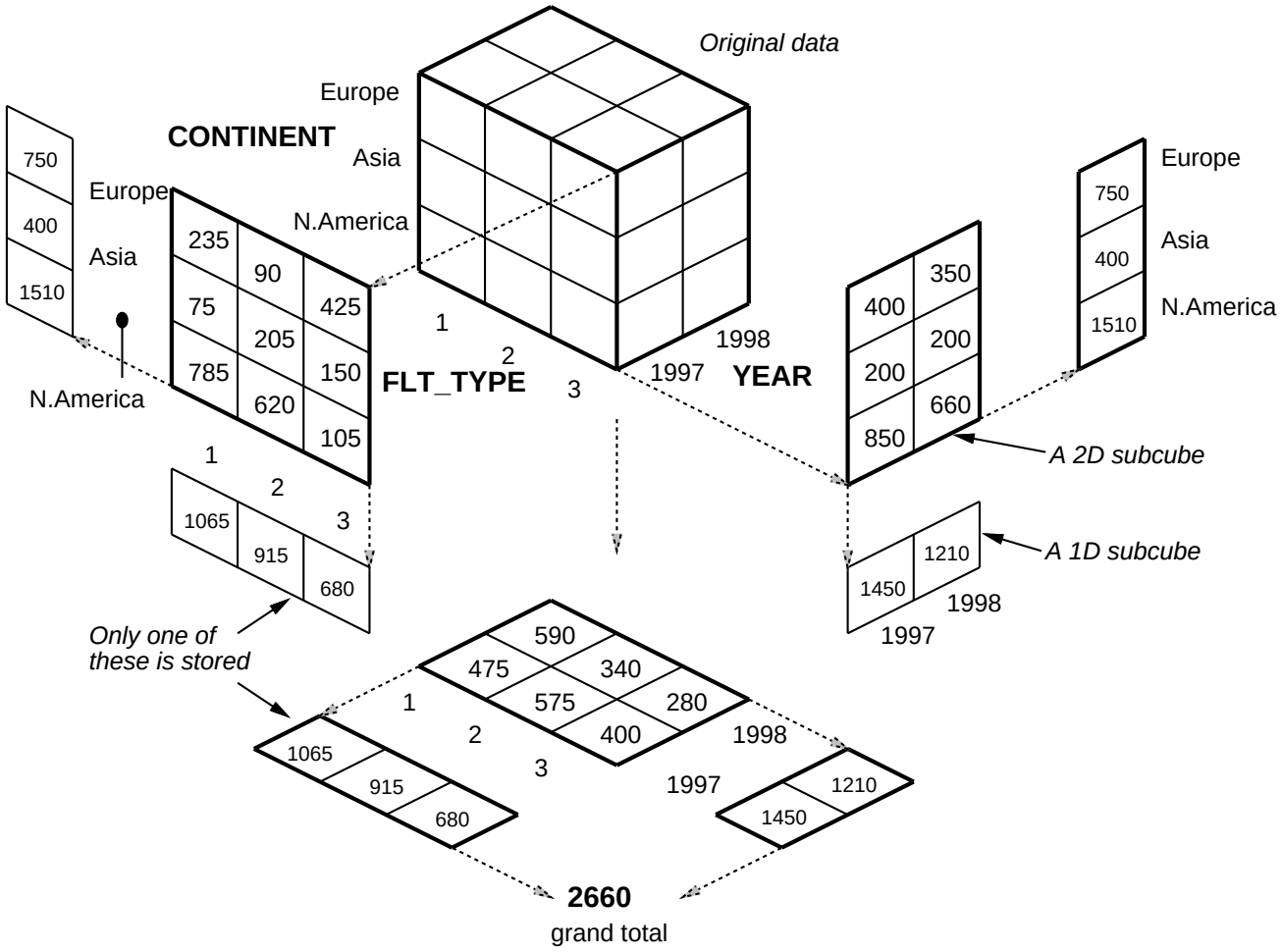
  - REVENUE is the aggregate attribute.
    (Sums are computed over REVENUE values.)
  - YEAR, CONTINENT and FLT_TYPE are parameter attributes.
  - Thus, there are 3 parameter attributes
    $\Rightarrow$ we call this a 3D aggregate problem.
  - For m parameter attributes, it's an m-dimensional aggregate problem.

  In general, the data will be a relation $R(A_1, \ldots, A_m, F)$ where

  - $A_1, \ldots, A_m$ are the parameter attributes.
  - F is the aggregate attribute.

  The subcube with attributes $A_{i_1} \ldots A_{i_k}$ will be denoted by $S(A_{i_1} \ldots A_{i_k})$.

- It is often convenient to view a 3D problem using a cube:
  (Although strictly a cuboid, the term *cube* is used).



For m-dimensional data, there are several subcubes for each dimension $k < m$.

12.21 OLAP: Repeated Queries

- Typically, a manager or accountant sits at a terminal and queries on several attribute subsets repeatedly
  ⇒ multiple views required quickly.

  If queries are generated via SQL statements
  ⇒ could take a long time.

- Prior computation and storage of subcubes:
  - It is better to materialize each subcube and store it.
  - For example, the (YEAR,CONTINENT) subcube is computed as:

1997	Europe	400
1998	Europe	350
1997	Asia	200
1998	Asia	200
1997	N.America	850
1998	N.America	660

  - Storage options:
    1. Store each possible subcube separately:
       * Store the subcube S(YEAR, CONTINENT) in a relation S1 (YEAR, CONTINENT).
       * Store the subcube S(YEAR, FLT_TYPE) in relation S2 (YEAR, FLT_TYPE).
       * Store the subcube S(CONTINENT, FLT_TYPE) in relation S3 (CONTINENT, FLT_TYPE).
       * Store the subcude S(YEAR) in relation S4 (YEAR).
       * ...etc.
    2. Store each subcube within the original relation using **null** values. For example, the subcube S(YEAR, CONTINENT) is

1997	Europe	400
1998	Europe	350
1997	Asia	200
1998	Asia	200
1997	N.America	850
1998	N.America	660

These tuples would be extended with **null**'s and added to the original data:

SALES	YEAR	CONTINENT	FLT_TYPE	REVENUE
	1997	Europe	1	125
	1997	Europe	2	50
	1997	Europe	3	225
	1997	Asia	1	25
	1997	Asia	2	75
	1997	Asia	3	100
	1997	N.America	1	325
	1997	N.America	2	450
	1997	N.America	3	75
	1998	Europe	1	110
	1998	Europe	2	40
	1998	Europe	3	200
	1998	Asia	1	20
	1998	Asia	2	130
	1998	Asia	3	50
	1998	N.America	1	460
	1998	N.America	2	170
	1998	N.America	3	30
	1997	Europe	**null**	400
	1998	Europe	**null**	350
	1997	Asia	**null**	200
	1998	Asia	**null**	200
	1997	N.America	**null**	850
	1998	N.America	**null**	660

NOTE: since **null** already has a use, the use of the keyword **all** has been proposed.

- An observation:
  - The tuples for the subcube S(YEAR, CONTINENT) repeat the strings "1997" and "1998"

⇒ a waste of space.

– The only real information is the collection of aggregates:

1997	Europe	**400**
1998	Europe	**350**
1997	Asia	**200**
1998	Asia	**200**
1997	N.America	**850**
1998	N.America	**660**

Therefore, it is more efficient to directly store the subcubes using an internal representation of a matrix:
⇒ called the *multidimensional approach*.

- Problems with large dimensions:
  - A 20-dimensional data set has 2^{20} subsets of attributes
    ⇒ 2^{20} possible subcubes.
  - Many subcubes are of high dimension
    ⇒ need to store high-dimensional matrices.

12.22 The Multidimensional Approach: Hash-Based Computation

- Some useful observations:

  Computing sizes of subcubes:

  – Consider the subcube S(YEAR, CONTINENT).

    How many entries in the subcube?
    2 YEAR's, 3 CONTINENT's
    $\Rightarrow$ 6 entries

  – In general, for relation $R(A_1, \ldots, A_m)$ let

    $$n(A_i) = \#\text{ values of attribute } A_i \text{ present.}$$

    Then, the size of subcube $A_{i_1}A_{i_2}\ldots A_{i_k}$ is

    $$\text{size}(A_{i_1}\ldots A_{i_k}) = n(A_{i_1})n(A_{i_2})\ldots n(A_{i_k}) = \prod_{j=1}^{k} n(A_{i_j}).$$

  – We have made an implicit assumption: all possible combinations of values exist as tuples in the data
    $\Rightarrow$ the *full-cube* assumption. E.g., if 1997 exists as a YEAR and Europe exists as a CONTINENT then $< 1997, \text{Europe} >$ will exist in some tuple.

  Example:

  – Consider the relation $R(A_1, A_2, A_3, F)$ with

    $$\begin{aligned} n(A_1) &= 4 \\ n(A_2) &= 50 \\ n(A_3) &= 1000 \end{aligned}$$

Thus, there are $4 \times 50 \times 1000 = 200,000$ tuples.

- Which subcubes need to be computed?
  - The subcube $S(A_1A_2A_3)$ is the given data.
  - The three 2-dimensional subcubes $S(A_1A_2)$, $S(A_1A_3)$ and $S(A_2A_3)$.
  - The four 1-dimensional subcubes $S(A_1)$, $S(A_2)$, $S(A_3)$ and $S(A_4)$.
- Suppose 1000 integers fit into a block.
- Sizes:

$$\begin{aligned}
\text{size}(A_1A_2A_3) &= 200,000 \text{ values} = 200 \text{ blocks} \\
\text{size}(A_1A_2) &= 200 \text{ values} = 1 \text{ block} \\
\text{size}(A_1A_3) &= 4000 \text{ values} = 4 \text{ blocks} \\
\text{size}(A_2A_3) &= 50,000 \text{ values} = 50 \text{ blocks} \\
\text{size}(A_1) &= 4 \text{ values} = 1 \text{ block} \\
\text{size}(A_2) &= 500 \text{ values} = 1 \text{ block} \\
\text{size}(A_3) &= 1000 \text{ values} = 1 \text{ block}
\end{aligned}$$

- Computing each subcube via hashing:
  - Scan original file.
  - Hash tuples into hash table containing sums;
  - Update appropriate sum for each tuple scanned.

  For the above data:
  $\Rightarrow$ 6 scans of data
  $\Rightarrow$ 6 scans of 200,000 tuples.

  Observe:
  - Once the subcube $S(A_1A_2)$ is computed, $S(A_1)$ can be computed from a scan of $S(A_1A_2)$
    $\Rightarrow$ only one block needs to be scanned.

- The hash table:
  - Consider computing the subcube A_1A_3.
  - There are 4000 values of A_1A_3
    $\Rightarrow$ need a sum for each of these 4000 values.
  - Create a hashtable with 4000 such sums.
  - Scan data and hash each tuple to discover which sum to update.

  Example:
  - Suppose we are computing the subcube S(YEAR, CONTINENT).
  - We will need a sum for each possible combination of YEAR and CONTINENT:

1997	Europe
1998	Europe
1997	Asia
1998	Asia
1997	N.America
1998	N.America

    In this case, we need 6 sums (the size of the subcube).
  - In practice, the size of the subcube can be large
    $\Rightarrow$ many counters will be needed.
  - As we scan the actual data, we need to add the revenue in a tuple to the appropriate counter.
  - A simple scan or binary search can be used, but hashing is very efficient.
  - If each sum is in a different bucket, a single access is needed for an update.
- Example:

  Consider the relation $R(A_1, A_2, A_3, A_4, F)$ with

$$\begin{aligned} n(A_1) &= 4 \\ n(A_2) &= 50 \end{aligned}$$

$$
\begin{aligned}
n(A_3) &= 1000 \\
n(A_4) &= 200
\end{aligned}
$$

The subcubes are:

- 4-dim: $A_1A_2A_3A_4$.
- 3-dim: $A_1A_2A_3$, $A_1A_2A_4$, $A_1A_3A_4$, $A_2A_3A_4$.
- 2-dim: A_1A_2, A_1A_3, A_1A_4, A_2A_3, A_2A_4, A_3A_4.
- 1-dim: A_1, A_2, A_3, A_4.

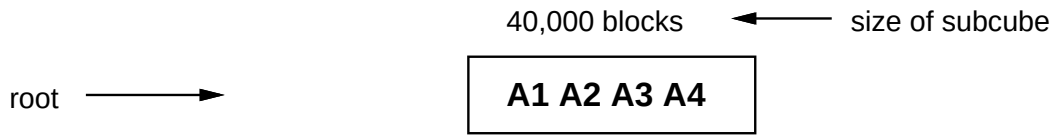
Sizes:

$$
\begin{aligned}
\text{size}(A_1A_2A_3A_4) &= 4 \times 10^7 \text{ values} = 40,000 \text{ blocks} \\
\text{size}(A_1A_2A_3) &= 200,000 \text{ values} = 200 \text{ blocks} \\
\text{size}(A_1A_2A_4) &= 40,000 \text{ values} = 40 \text{ blocks} \\
\text{size}(A_1A_3A_4) &= 800,000 \text{ values} = 800 \text{ blocks} \\
\text{size}(A_2A_3A_4) &= 10^7 \text{ values} = 10,000 \text{ blocks} \\
\text{size}(A_1A_2) &= 200 \text{ values} = 1 \text{ block} \\
\text{size}(A_1A_3) &= 4000 \text{ values} = 4 \text{ blocks} \\
\text{size}(A_1A_4) &= 800 \text{ values} = 1 \text{ block} \\
\text{size}(A_2A_3) &= 50,000 \text{ values} = 50 \text{ blocks} \\
\text{size}(A_2A_4) &= 10,000 \text{ values} = 10 \text{ blocks} \\
\text{size}(A_3A_4) &= 200,000 \text{ values} = 200 \text{ blocks}
\end{aligned}
$$

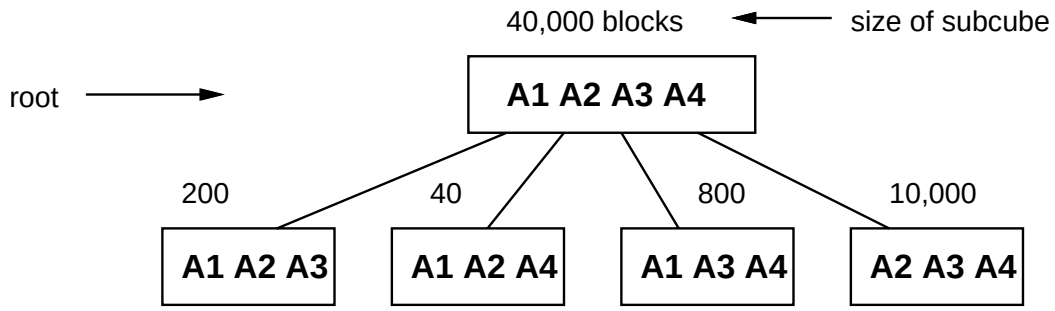
(Sizes of the 1-dim cubes not shown).

Construct a tree:

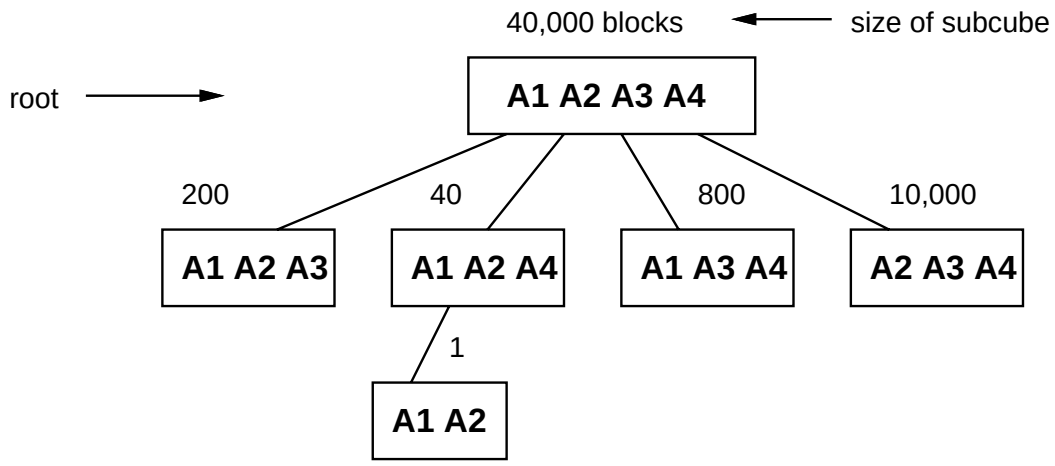
- Step 1: Place the subcube $A_1A_2A_3A_4$ at the root:



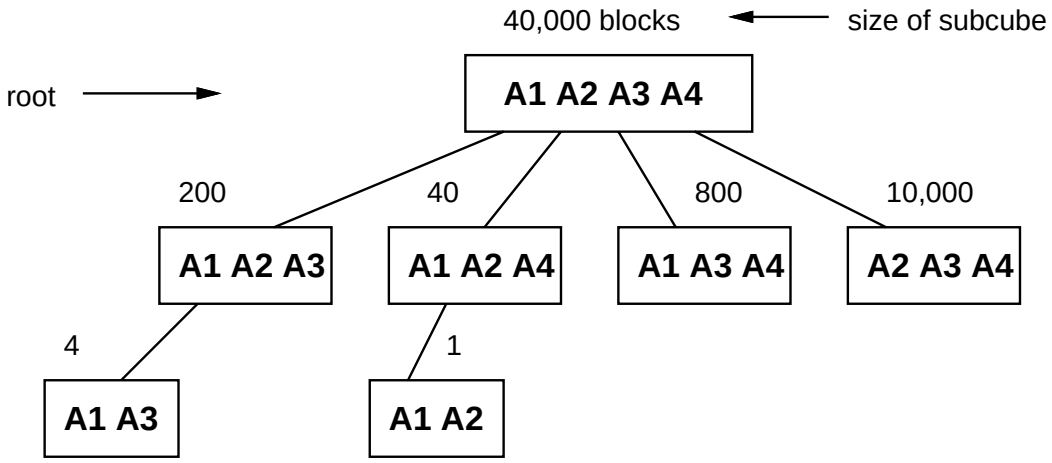
– Step 2: No choice of parent for 3-dim subcubes $A_1A_2A_3$, $A_1A_2A_4$, $A_1A_3A_4$, $A_2A_3A_4$.



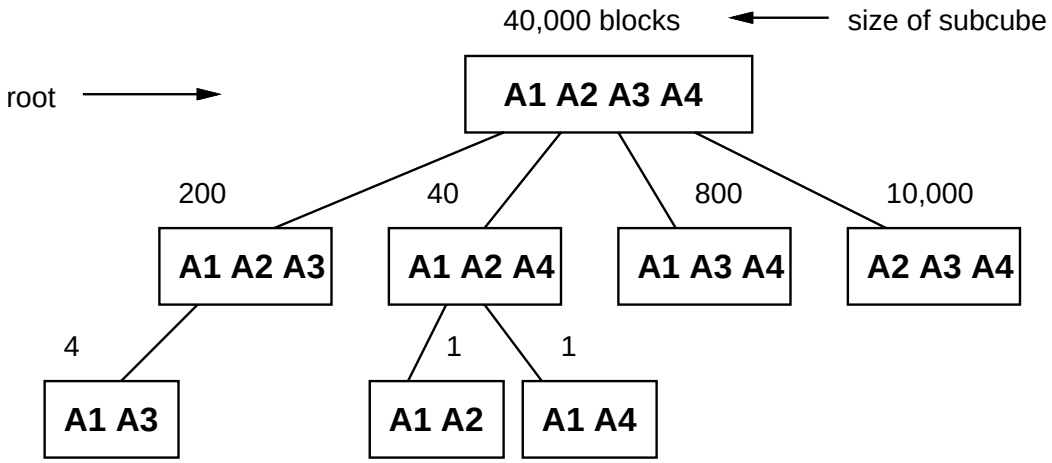
– Step 3: For subcube A_1A_2 pick smallest parent
$\Rightarrow$ 40 blocks of $A_1A_2A_4$.



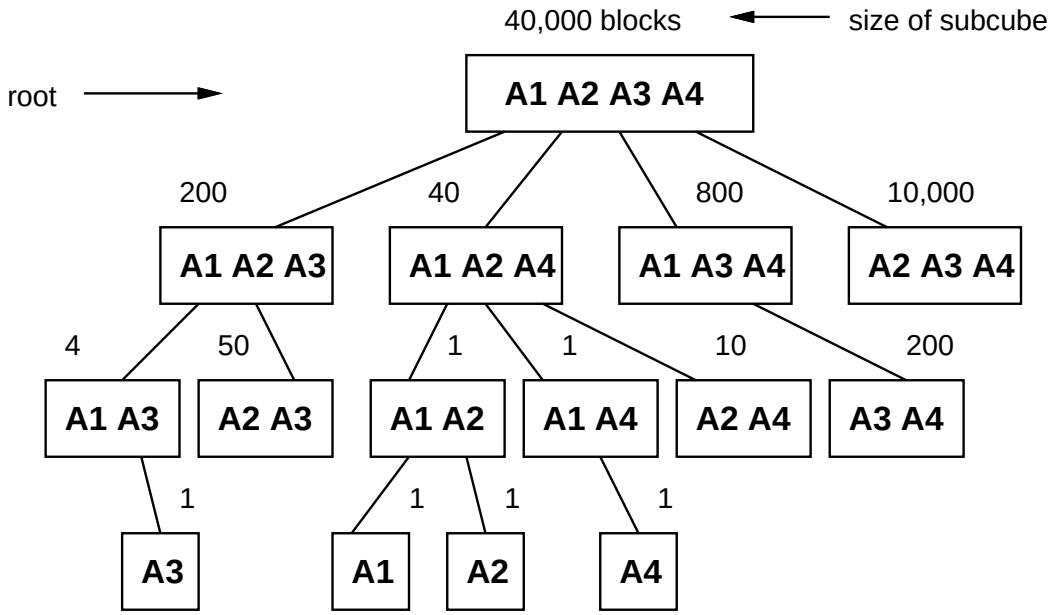
– Step 4: For subcube A_1A_3 pick smallest parent
$\Rightarrow$ 200 blocks of $A_1A_2A_3$.



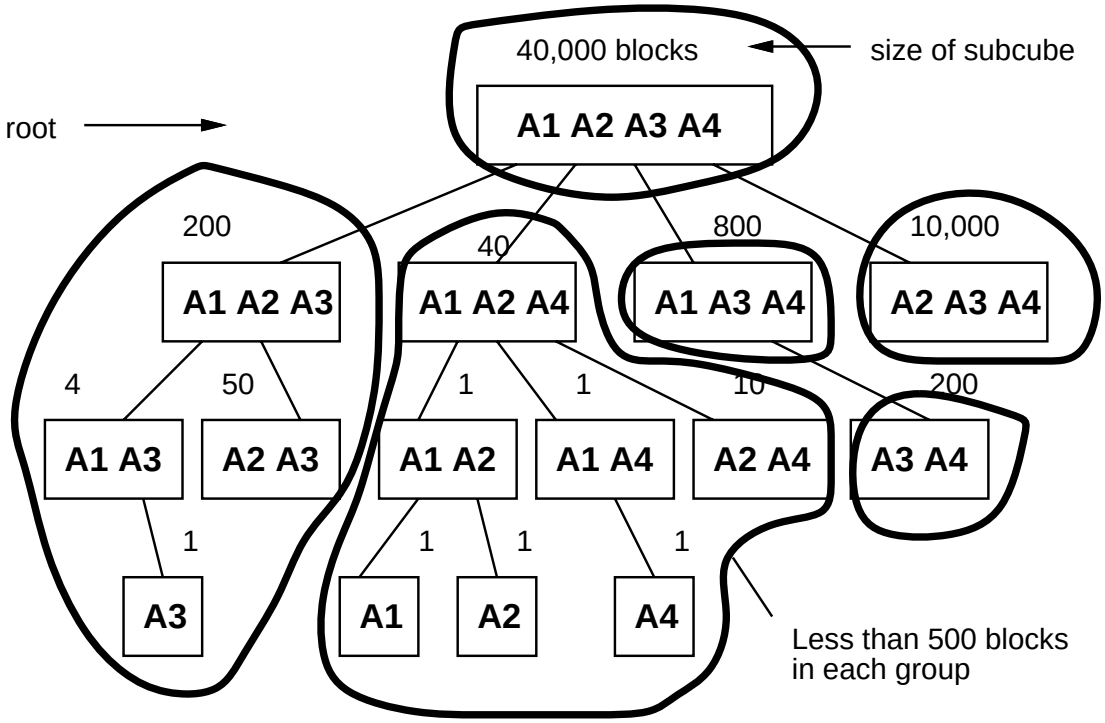
– Step 5: For subcube A_1A_4 pick smallest parent
$\Rightarrow$ 40 blocks of $A_1A_2A_4$.



– ... continuing, we get the final tree:

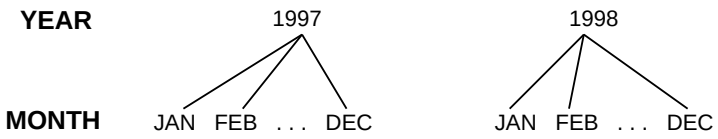


Suppose memory size is 500 blocks. Several subtrees can be computed in parallel:



12.23 Hierarchies on Attributes: Roll-up and Drill-down

- What is a hierarchy on an attribute?
  - Consider the attribute DATE in
    SALES (DATE, CONTINENT, FLT_TYPE, REVENUE).
  - There is a natural division of dates by YEAR and MONTH.
  - YEAR and MONTH form a hierarchical division:



- Why is this important?
  - Queries often use hierarchies.
  - Example:
    - A user requests aggregate revenue by the subcube (YEAR, CONTINENT).
    - Then, if that's interesting, the user wants to look at a breakdown month-by-month
      $\Rightarrow$ a subcube addressed by (MONTH, CONTINENT).
    - This is an example of *drilling down* a hierarchy.
  - Example:
    - A user requests aggregate revenue by (DATE, FLT_TYPE).
    - Then, a broader picture can be obtained by requesting the summary (MONTH, FLT_TYPE)
      $\Rightarrow$ a subcube addressed by (MONTH, CONTINENT).

    - This is an example of *rolling up* a hierarchy.
- Both *drill-down* and *roll-up* are useful OLAP operations.