6.6 FP 增长算法

本节介绍另一种称作 FP 增长的算法。该算法采用完全不同的方法来发现频繁项集。该算法不同于 *Apriori* 算法的“产生-测试”范型，而是使用一种称作 FP 树的紧凑数据结构组织数据，并直接从该结构中提取频繁项集。下面详细说明该方法。

6.6.1 FP 树表示法

FP 树是一种输入数据的压缩表示，它通过逐个读入事务，并把每个事务映射到 FP 树中的一条路径来构造。由于不同的事务可能会有若干个相同的项，因此它们的路径可能部分重叠。路径相互重叠越多，使用 FP 树结构获得的压缩的效果越好。如果 FP 树足够小，能够存放在内存中，就可以直接从这个内存中的结构提取频繁项集，而不必重复地扫描存放在硬盘上的数据。

图6-24显示了一个数据集，它包含10个事务和5个项。图中还绘制了读入前3个事务之后FP树的结构。树中每一个结点都包括一个项的标记和一个计数，计数显示映射到给定路径的事务个数。初始，FP树仅包含一个根结点，用符号null标记。随后，用如下方法扩充FP树：

(1) 扫描一次数据集，确定每个项的支持度计数。丢弃非频繁项，而将频繁项按照支持度的递减排序。对于图 6-24 中的数据集，a 是最频繁的项，接下来依次是 b, c, d 和 e。

(2) 算法第二次扫描数据集，构建 FP 树。读入第一个事务$\{a, b\}$之后，创建标记为 a 和 b 的结点。然后形成 null→a→b 路径，对该事务编码。该路径上的所有结点的频度计数为 1。

(3) 读入第二个事务$\{b, c, d\}$之后，为项 b, c 和 d 创建新的结点集。然后，连接结点 null→b→c→d，形成一条代表该事务的路径。该路径上的每个结点的频度计数也等于 1。尽管前两个事务具有一个共同项 b，但是它们的路径不相交，因为这两个事务没有共同的前缀。

(4) 第三个事务$\{a, c, d, e\}$与第一个事务共享一个共同前缀项 a，所以第三个事务的路径 null→a→c→d→e 与第一个事务的路径 null→a→b 部分重叠。因为它们的路径重叠，所以结点 a 的频度计数增加为 2，而新创建的结点 c, d 和 e 的频度计数等于 1。

(5) 继续该过程，直到每个事务都映射到 FP 树的一条路径。读入所有的事务后形成的 FP 树显示在图 6-24 的底部。

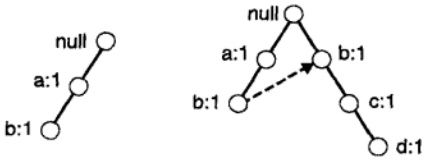
通常，FP树的大小比未压缩的数据小，因为购物篮数据的事务常常共享一些共同项。在最好情况下，所有的事务都具有相同的项集，FP树只包含一条结点路径。当每个事务都具有唯一项集时，导致最坏情况发生，由于事务不包含任何共同项，FP树的大小实际上与原数据的大小一样，然而，由于需要附加的空间为每个项存放结点间的指针和计数，FP树的存储需求增大。

FP 树的大小也取决于项的排序方式。如果颠倒前面例子的序，即项按照支持度由小到大排列，则结果 FP 树显示在图 6-25 中。该树显得更加茂盛，因为根结点上的分支数由 2 增加到 5，并且包含了高支持度项 a 和 b 的结点数由 3 增加到 12。尽管如此，支持度计数递减序并非总是导致最小的树。例如，假设加大图 6-24 给定的数据集，增加 100 个事务包含$\{e\}$、80 个事务包含$\{d\}$、60 个事务包含$\{c\}$、40 个事务包含$\{b\}$。现在，项 e 是最频繁的，接下来依次是 d, c, b 和 a。使用加大的事务数据，支持度计数递减序将导致类似于图 6-25 中的 FP 树，而基于支持度计数递增序将产生一棵类似于图 6-24(iv)的较小的 FP 树。

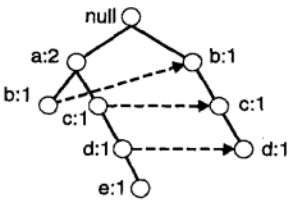
FP 树还包含一个连接具有相同项的结点的指针列表。这些指针在图 6-24 和图 6-25 中用虚线表示，有助于方便快速地访问树中的项。下一节，解释如何使用 FP 树和它的相应指针产生频繁项集。

事务数据集

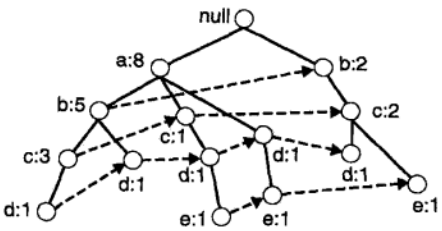
TID	项
1	{a,b}
2	{b,c,d}
3	{a,c,d,e}
4	{a,d,e}
5	{a,b,c}
6	{a,b,c,d}
7	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}



(i) 读入 TID = 1 之后　(ii) 读入 TID = 2 之后



(iii) 读入 TID = 3 之后



(iv) 读入 TID = 10 之后

图 6-24　构造 FP 树

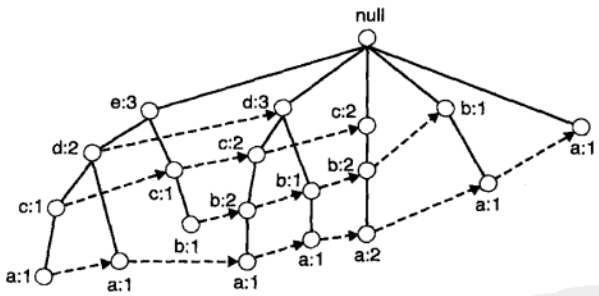
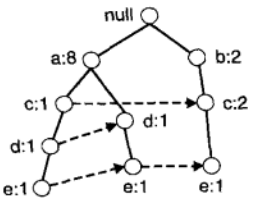


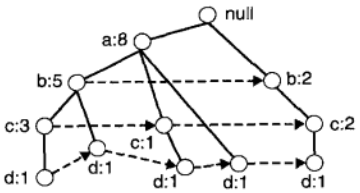
图 6-25　对图 6-24 所示数据集使用不同的项序方案的 FP 树表示

6.6.2　FP 增长算法的频繁项集产生

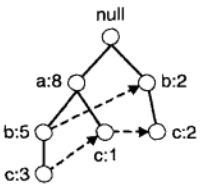
FP 增长（FP-growth）是一种以自底向上方式探索树，由 FP 树产生频繁项集的算法。给定图 6-24 所示的树，算法首先查找以 *e* 结尾的频繁项集，接下来依次是 *d*, *c*, *b*，最后是 *a*。这种用于发现以某一个特定项结尾的频繁项集的自底向上策略等价于 6.5 节介绍的基于后缀的方法。由于每一个事务都映射到 FP 树中的一条路径，因而通过仅考察包含特定结点（例如 *e*）的路径，就可以发现以 *e* 结尾的频繁项集。使用与结点 *e* 相关联的指针，可以快速访问这些路径。图 6-26a 显示了所提取的路径。稍后详细解释如何处理这些路径，以得到频繁项集。



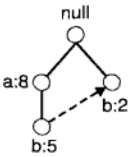
(a) 包含结点 *e* 的路径



(b) 包含结点 *d* 的路径



(c) 包含结点 *c* 的路径



(d) 包含结点 *b* 的路径

(e) 包含结点 *a* 的路径

图 6-26　将频繁项集产生的问题分解成多个子问题，其中每个子问题分别涉及发现以 *e*, *d*, *c*, *b* 和 *a* 结尾的频繁项集

发现以 *e* 结尾的频繁项集之后，算法通过处理与结点 *d* 相关联的路径，进一步寻找以 *d* 结尾的频繁项集。图 6-26b 显示了对应的路径。继续该过程，直到处理了所有与结点 *c*, *b* 和 *a* 相关联的路径为止。图 6-26c、图 6-26d、图 6-26e 分别显示了这些项的路径，而它们对应的频繁项集汇总在表 6-6 中。

表 6-6　依据相应的后缀排序的频繁项集

后　缀	频 繁 项 集
e	{*e*},{*d*,*e*},{*a*,*d*,*e*},{*c*,*e*},{*a*,*e*}
d	{*d*},{*c*,*d*},{*b*,*c*,*d*},{*a*,*c*,*d*},{*b*,*d*},{*a*,*b*,*d*},{*a*,*d*}
c	{*c*},{*b*,*c*},{*a*,*b*,*c*},{*a*,*c*}
b	{*b*},{*a*,*b*}
a	{*a*}

FP增长采用分治策略将一个问题分解为较小的子问题，从而发现以某个特定后缀结尾的所有频繁项集。例如，假设对发现所有以*e*结尾的频繁项集感兴趣。为了实现这个目的，必须首先检查项集{*e*}本身是否频繁。如果它是频繁的，则考虑发现以*de*结尾的频繁项集子问题，接下来是*ce*和*ae*。依次，每一个子问题可以进一步划分为更小的子问题。通过合并这些子问题得到的结果，就可以找到所有以*e*结尾的频繁项集。这种分治策略是FP增长算法采用的关键策略。

为了更具体地说明如何解决这些子问题，考虑发现所有以 *e* 结尾的频繁项集的任务。

(1) 第一步收集包含 *e* 结点的所有路径。这些初始的路径称为**前缀路径**（prefix path），如图 6-27a 所示。

(2) 由图 6-27a 中所显示的前缀路径，通过把与结点 *e* 相关联的支持度计数相加得到 *e* 的支持度计数。假定最小支持度为 2，因为{*e*}的支持度是 3 所以它是频繁项集。

(3) 由于{*e*}是频繁的，因此算法必须解决发现以 *de*、*ce*、*be* 和 *ae* 结尾的频繁项集的子问题。在解决这些子问题之前，必须先将前缀路径转化为**条件 FP 树**（conditional FP-tree）。除了用于发现以某特定后缀结尾的频繁项集之外，条件 FP 树的结构与 FP 树类似。条件 FP 树通过以下步骤得到。

(a) 首先，必须更新前缀路径上的支持度计数，因为某些计数包括那些不含项 *e* 的事务。例

如，图 6-27a 中的最右边路径 null→*b*:2→*c*:2→*e*:1，包括并不含项 *e* 的事务{*b*, *c*}。因此，必须将该前缀路径上的计数调整为 1，以反映包含{*b*, *c*, *e*}的事务的实际个数。

(b) 删除 *e* 的结点，修剪前缀路径。删除这些结点是因为，沿这些前缀路径的支持度计数已经更新，以反映包含 *e* 的那些事务，并且发现以 *de*, *ce*, *be* 和 *ae* 结尾的频繁项集的子问题不再需要结点 *e* 的信息。

(c) 更新沿前缀路径上的支持度计数之后，某些项可能不再是频繁的。例如，结点 *b* 只出现了 1 次，它的支持度计数等于 1，这就意味着只有一个事务同时包含 *b* 和 *e*。因为所有以 *be* 结尾的项集一定都是非频繁的，所以在其后的分析中可以安全地忽略 *b*。

e 的条件 FP 树显示在图 6-27b 中。该树看上去与原来的前缀路径不同，因为频度计数已经更新，并且结点 *b* 和 *e* 已被删除。

(4) FP 增长使用 *e* 的条件 FP 树来解决发现以 *de*, *ce*, *be* 和 *ae* 结尾的频繁项集的子问题。为了发现以 *de* 结尾的频繁项集，从项 *e* 的条件 FP 树收集 *d* 的所有前缀路径（图 6-27c）。通过将与结点 *d* 相关联的频度计数求和，得到项集{*d*, *e*}的支持度计数。因为项集{*d*, *e*}的支持度计数等于 2，所以它是频繁项集。接下来，算法采用第 3 步介绍的方法构建 *de* 的条件 FP 树。更新了支持度计数并删除了非频繁项 *c* 之后，*de* 的条件 FP 树显示在图 6-27d 中。因为该条件 FP 树只包含一个支持度等于最小支持度的项 *a*，算法提取出频繁项集{*a*, *d*, *e*}并转到下一个子问题，产生以 *ce* 结尾的频繁项集。处理 *c* 的前缀路径后，只发现项集{*c*, *e*}是频繁的。接下来，算法继续解决下一个子问题并发现项集{*a*, *e*}是剩下唯一的频繁项集。

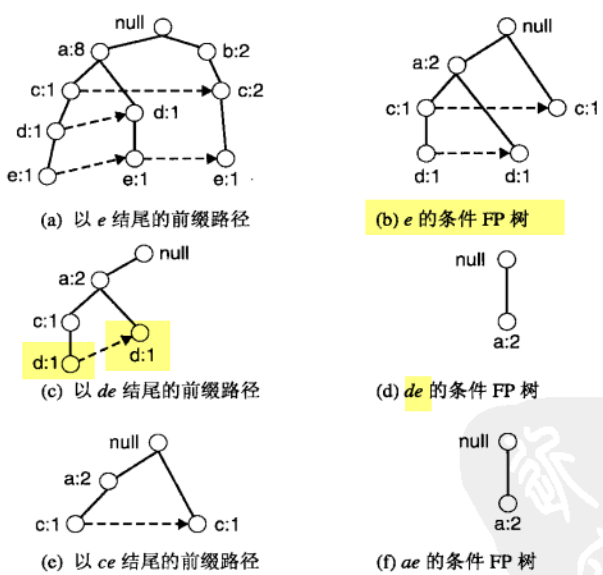


图 6-27 使用 FP 增长算法发现以 *e* 结尾的频繁项集的例子

这个例子解释了FP增长算法中使用的分治方法。每一次递归，都要通过更新前缀路径中的支持度计数和删除非频繁的项来构建条件FP树。由于子问题是不相交的，因此FP增长不会产生任何重复的项集。此外，与结点相关联的支持度计数允许算法在产生相同的后缀项时进行支持度计数。

FP增长是一个有趣的算法，它展示了如何使用事务数据集的压缩表示来有效地产生频繁项集。此外，对于某些事务数据集，FP增长算法比标准的*Apriori*算法要快几个数量级。FP增长算法的运行性能依赖于数据集的压缩因子（compaction factor）。如果生成的条件FP树非常茂盛（在最坏情形下，是一棵满前缀树），则算法的性能显著下降，因为算法必须产生大量的子问题，并且需要合并每个子问题返回的结果。