深入剖析FP-Growth原理

原创 海涛1992 海涛技术漫谈 2019-04-21

> 同步更新公众号：https://blog.csdn.net/zhanht/article/details/89436460

频繁项挖掘广泛的应用于寻找关联的事物。最经典的就是，电商企业通过分析用户的订单，挖掘出经常被共同购买的商品，用于推荐。

本文首先介绍频繁项挖掘技术的演进，从暴力求解到Aprioir算法。然后，通过一个案例详细的讲解FP-Growth的原理。接下来介绍并行FP-Growth算法怎么通过3次map-reduce实现了并行化。最后通过分析spark mlib包中PFP-Growth的核心实现代码，进一步加深理解。

假设我们的Transaction数据库有5条交易数据，如下表，其中abcde为5个商品。假设设定minSupport = 0.4，即要求至少共同出现2次。

id	购买的商品
1	a b d
2	b c d
3	a b e
4	a b c
5	b c d

表1：交易数据

一：频繁项挖掘的技术演进

1．暴力求解

5个sku，一共有2的5次方种购买可能，如下图所示。暴力求解循环每种可能，全量扫描交易数据库计算购买次数，看其是否超过设定的minSupport，进而判断是否是频繁项。

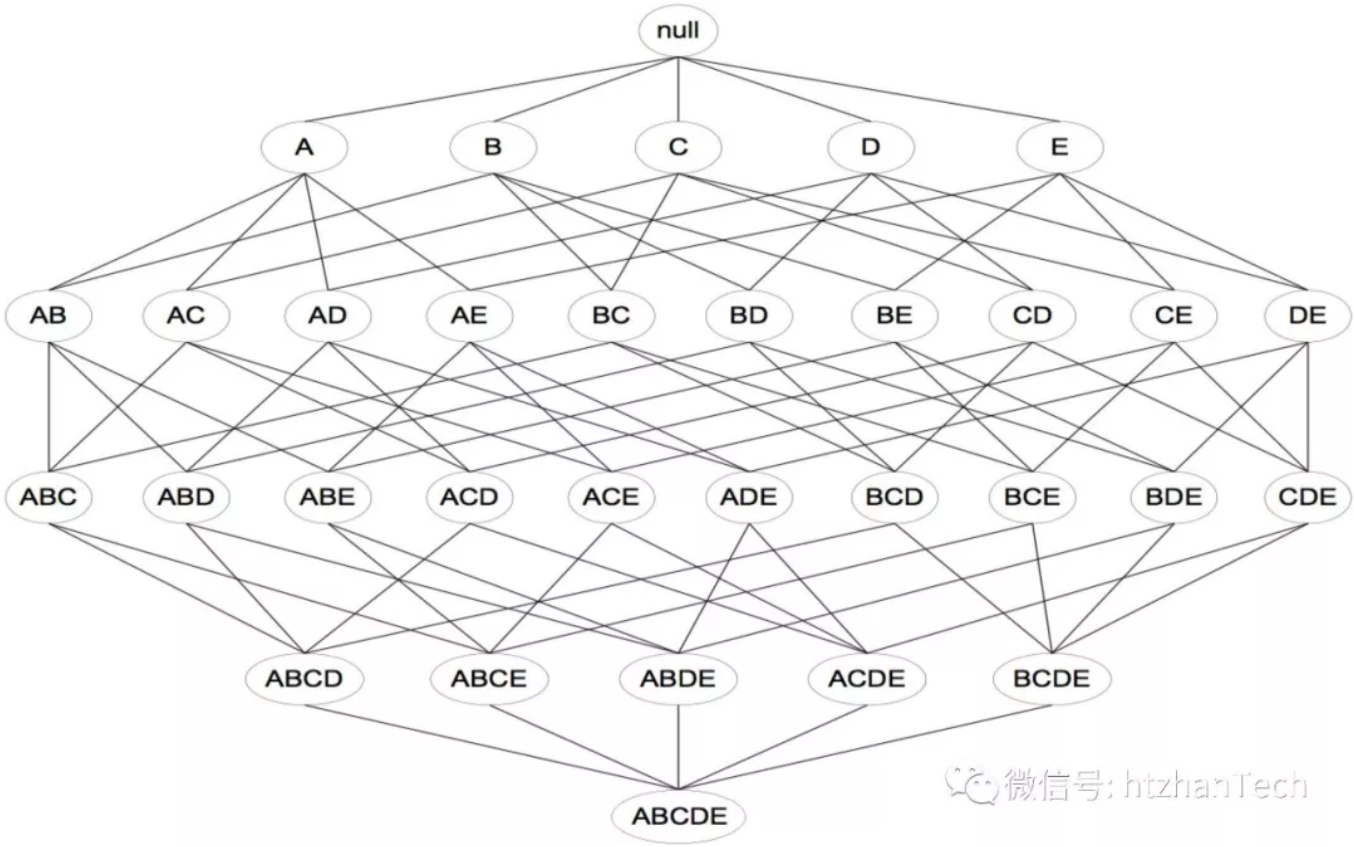

图1：暴力求解

2. Apriori算法

上述的暴力求解方式对每种可能都会进行数据库的全表扫描，效率低下。因此有学者提出了Apriori算法。Apriori中文含义是先验的，因为算法基于这么一个先验知识：当购买组合A不是频繁项时，那么包含A的任何超集也必然不是频繁项。

通过这个先验知识，可以避免大量的无效数据库扫描，提高效率，提高的程度取决于交易数据和设置的minsupport。示意图如下所示：

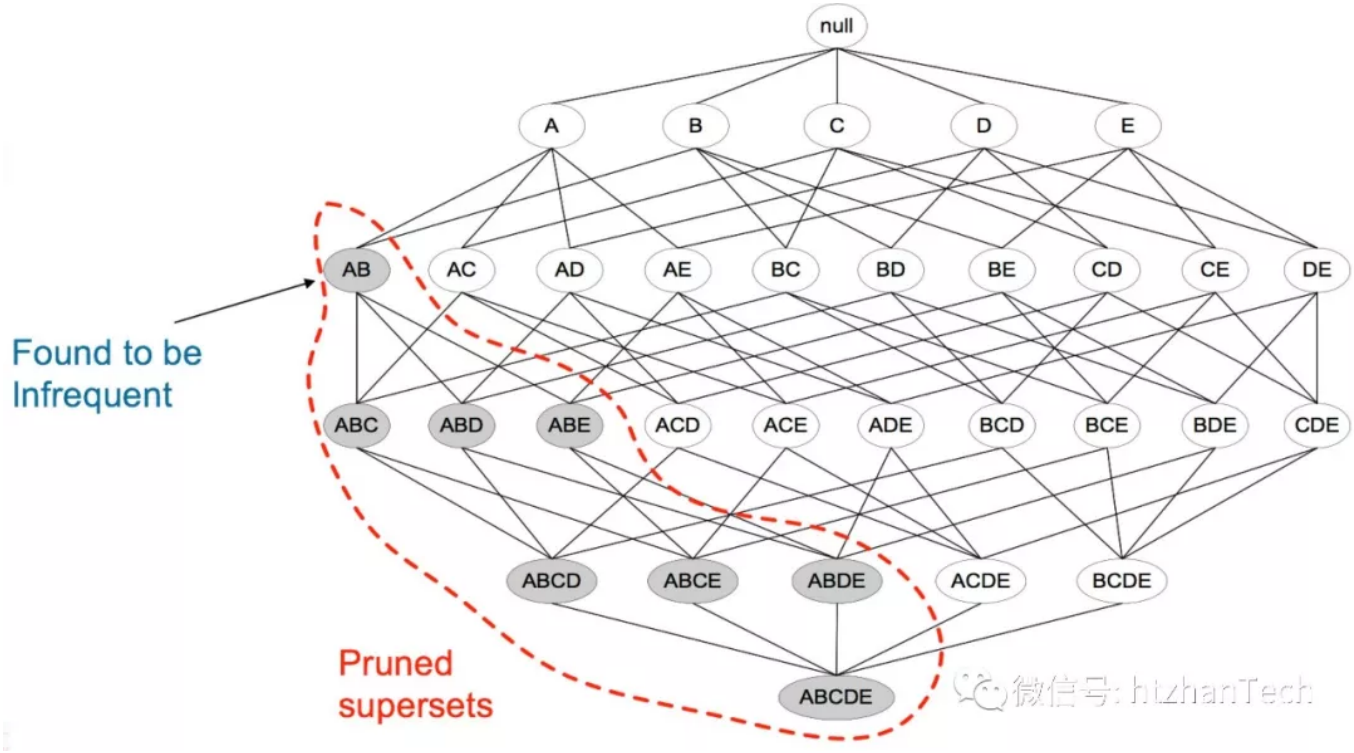

图2: Apriori算法示意图

二：FP-Growth算法

FP-Growth算法更进一步，通过将交易数据巧妙的构建出一颗FP树，然后在FP树中递归的对频繁项进行挖掘。

FP-Growth算法仅仅需要两次扫描数据库，第一次是统计每个商品的频次，用于剔除不满足最低支持度的商品，然后排序得到FreqItems。第二次，扫描数据库构建FP树。

还是以上面的交易数据作为例子，接下来一步步的详细分析FP树的构建，和频繁项的递归挖掘。

2.1 统计频次

第一步，扫描数据库，统计每个商品的频次，并进行排序，显然商品e仅仅出现了一次，不符合minSupport,剔除。最终得到的结果如下表：

商品	频次
b	5
a	3

商品	频次
c	3
d	3

表2：商品的出现频次

2.2 构建FP树

第二步，扫描数据库，进行FP树的构建。FP树以root节点为起始，节点包含自身的item和count，以及父节点和子节点。

首先是第一条交易数据，a b d，结合第一步商品顺序，排序后为b a d，依次在树中添加节点b，父节点为root，最新的的频次为1，然后节点a，父节点为a，频次为1，最后节点d，父节点为b，频次为1。如下图所示：

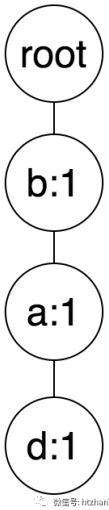

图3：第一条交易后的FP树

然后是第二条交易数据，排序后为：b c d。依次添加b，树中已经有节点b，因此更新频次加1，然后是节点c，b节点当前只有子节点d，因此新建节点c，父节点为b，频次为1，最后是d，父节点为c，频次为1。

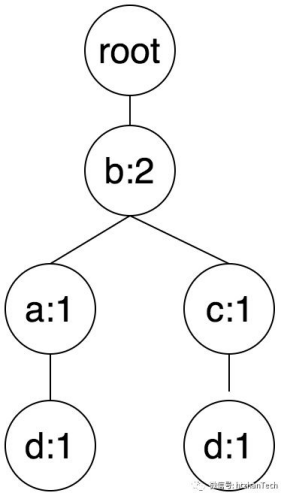

图4：第二条交易后的FP树

后面三条交易数据的处理和前两条一样，就不详细阐述了，直接画出每次处理完的FP树示意图。

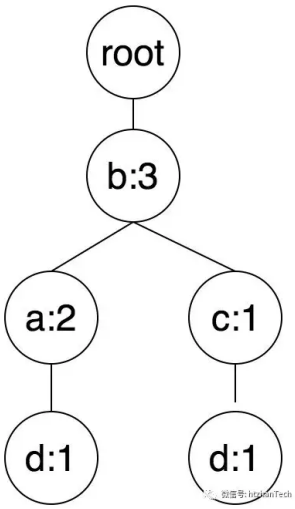

图5：第三条交易后的FP树

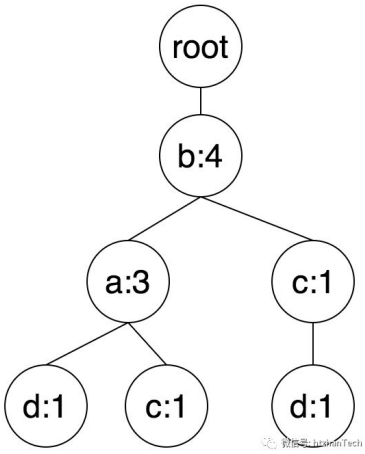

图6：第四条交易后的FP树

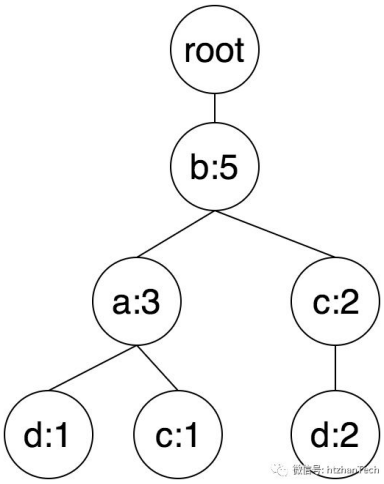

图7：第五条交易后的FP树，也即最终的FP树

2.3 频繁项的挖掘

2.3.1 商品b频繁项的挖掘

首先是商品b，首先b节点本身的频次符合minSupport，所以是一个频繁项(b：5)，然后b节点往上找subTree，只有根节点，所以结束，b为前缀的频繁项只有一个：（b：5）。

2.3.2 商品a频繁项的挖掘

然后是a，显然a本身是个频繁项(a：3)，然后递归的获取a的子树，进行挖掘。子树构建方式如下：新建一个新的FP树，然后遍历树中所有的a节点，往上找，直到root节点，然后把当前路径上的非根节点添加到subTree中，每个节点的频次为当前遍历节点的频次。

因为a只有一个节点(a, 3)，所以往上遍历得到节点b，因此把b加入subTree中，频次为节点(a, 3)的频次3。得到如下subTree，显然在这个subTree中只能挖掘出频繁项(b：3)，然后别忘了这是a递归得到的子树，得拼上前缀a，所以得到频繁项为(ab：3)

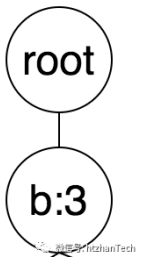

图8：a的子树

此时的subTree只有一个节点(b, 3)，不用进一步递归，因此商品a的频繁项挖掘结束，有两个频繁项为：(a：3), (ab：3)。

2.3.3 商品c频繁项的挖掘

商品c在FP树中包含两个节点，分别为: (c, 1), (c, 2)。显然c自身是个频繁项(c：3)，然后进行递归。(c, 1)节点往上路径得到如下节点：(a, 1), (b, 1)。节点(c, 2)往上得到(b, 2)，上述三个节点可以构造出如下的subTree:

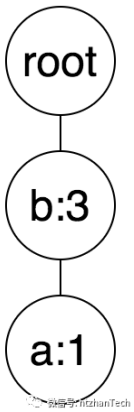

图9 ： c的子树

subTree中的节点(b, 3)符合minsupport,拼上前缀c得到频繁项(bc：3)。节点(a, 1)不满足要求，丢弃。

因此，c挖掘出的频繁项为：(c:3), (cb：3)

2.3.4 商品d频繁项的挖掘

同理，(d：3)是一个频繁项，d的subTree为:

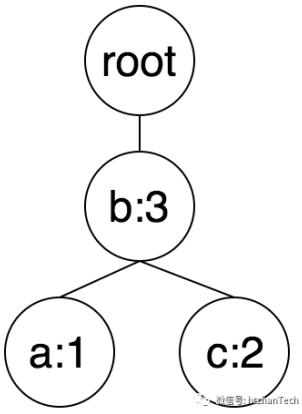

图10 ： d的子树

子树首先挖出(c：2)，(b：3)，拼上前缀d得到(dc：2),(db：3)，然后subTree中的节点c的subTree仅仅有根节点和节点(b, 2)，拼上两个前缀得到(dcb：2)

2.3.5 最终结果和验证

通过上述的挖掘过程，我们依次挖出了如下9个频繁项：(b：5), (a：3), (ab：3), (c:3), (cb：3), (d：3), (dc：2),(db：3), (dcb：2)

我们通过pyspark进行下验证，得到的结果和我们一步步推算的结果丝毫不差。

```
#!/usr/bin/env python3
#encoding:utf-8

from pyspark.sql import SparkSession

ss = SparkSession.builder \
        .appName("test_fp") \
        .config("spark.executor.memory", "32G") \
        .config("spark.driver.memory", "32G") \
        .config("spark.python.worker.memory", "32G") \
        .config("spark.default.parallelism", "4") \
        .config("spark.executor.cores", "8") \
        .config("spark.sql.shuffle.partitions", "500") \
        .config("spark.sql.crossJoin.enabled", "true")\
        .config("spark.sql.broadcastTimeout","36000") \
        .enableHiveSupport() \
        .getOrCreate()

data = [["a", "b", "d"], ["b", "c", "d"], ["a", "b", "e"], ["a", "b", "c"],["b", "c", "d
rdd = ss.sparkContext.parallelize(data, 2)
from pyspark.mllib.fpm import FPGrowth
model = FPGrowth.train(rdd, 0.4, 2)
model.freqItemsets().collect()

# 结果如下：
#[FreqItemset(items=['a'], freq=3),
 #FreqItemset(items=['a', 'b'], freq=3),
 #FreqItemset(items=['b'], freq=5),
 #FreqItemset(items=['d'], freq=3),
 #FreqItemset(items=['d', 'b'], freq=3),
 #FreqItemset(items=['c'], freq=3),
 #FreqItemset(items=['c', 'd'], freq=2),
 #FreqItemset(items=['c', 'd', 'b'], freq=2),
 #FreqItemset(items=['c', 'b'], freq=3)]
```

三：Parallel FP-Growth

当交易的数据量级太大时，单机版本的FP-Growth内存将无法放下完整的FP树，并且单机效率较慢。如果能够实现并行化那就再好不过了，但是怎么将一颗完整的FP树拆分成多棵小树，分别在不同的机器节点上运行FP-Growth，这些小树得出的结果汇总后不会丢失频繁项，这是关键所在。

在这样的背景下，Google北京研究院相关人员提出了PFP(Parallel FP-Growth)[1]，通过3次Map-Reduce操作对FP-Growth进行了并行化运行。目前各大开发包的实现也都是基于这一篇文章实现的，包括spark和mahout。

通过前面一步步的构建FP树和频繁项挖掘过程，大家应该发现了，挖掘某一个商品的频繁项时，并不是所有的交易数据都是有用的，显然不包含此商品的交易数据肯定是冗余的。PFP的关键就是将商品进行分堆到不同的机器节点上运行，每个节点获取和自己负责的节点相关的部分交易数据，然后后续的构建树和挖掘频繁项工作则是一样的。

第一个map-reduce用于统计每个item的频次，得到frequentSets，功能类似于wordCount。

第二个map-reduce实现分布式的FP-Growth，是关键所在。mapper负责生成group-dependent transactions。首先将frequentSets进行分堆，假设分为两堆，得到如下表的G_list。第一台机器负责b和a的频繁项挖掘，第二台机器负责c和d频繁项的挖掘。

商品	频次	group_id(后面简写为gid)
b	5	1
a	3	1
c	3	2
d	3	2

表3：G_list

然后开始划分交易数据，以第一条a b d为例，排序后为b a d。从后往前遍历，首先是d，gid为2，所以得到如下交易数据<2, (b a d)>，第一项为gid，第二个为对应的交易数据。第二个为a，gid为1，得到<1, (b a)>，因为d的频次小于b和a，所以d的存在与否对以b和a为前缀的频繁项挖掘没影响，只对d为前缀的频繁项有作用。最后是d，gid=2,因为已经有了gid=2的交易数据，因此不用处理。

通过上述步骤，可以生成各个group_id需要的交易数据，然后送个reducer。Reducer负责在这些交易数据上进行普通的FP-Growth操作，进行自己负责的item的频繁项的挖掘。

第三次map-reduce对上述不同分区产生的频繁项进行聚合得到最终结果。总体示意图如下：

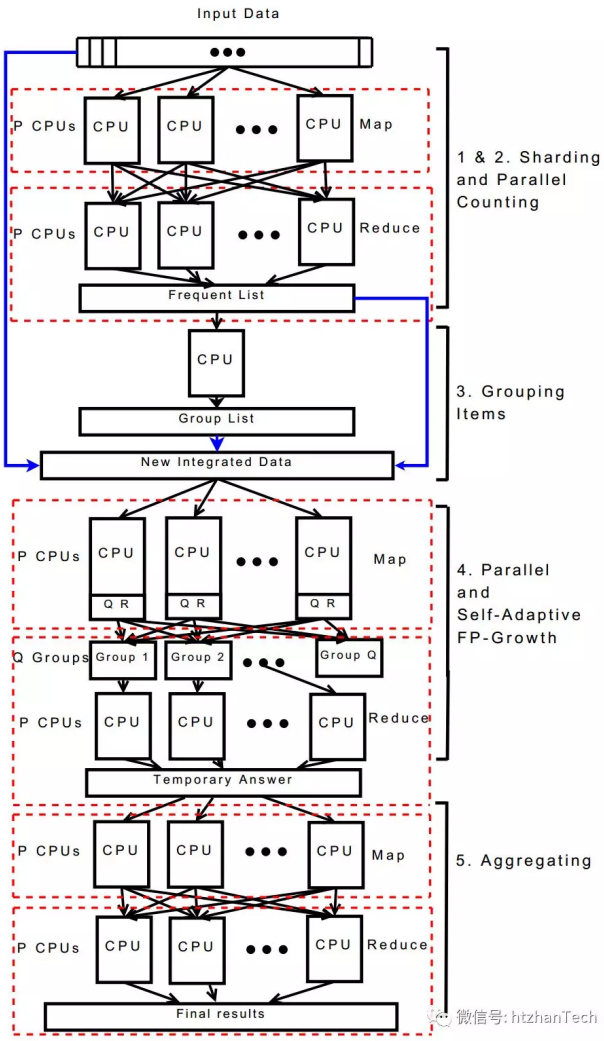

图11：PFP运行示意图

四：Spark mlib实现核心方法解析

spark中mlib包对上述的PFP进行了实现，在FPGrowth类的run方法中，代码如下：

```
def run[Item: ClassTag](data: RDD[Array[Item]]): FPGrowthModel[Item] = {
    if (data.getStorageLevel == StorageLevel.NONE) {
      logWarning("Input data is not cached.")
    }
    val count = data.count()
    // 计算最小的频次
    val minCount = math.ceil(minSupport * count).toLong
    // 如果没设置分区数量，就使用训练数据的分区数
    val numParts = if (numPartitions > 0) numPartitions else data.partitions.length
    val partitioner = new HashPartitioner(numParts)
    // 类似wordCount 实现每个item 频次的统计
    val freqItems = genFreqItems(data, minCount, partitioner)
    // 频繁项的挖掘
    val freqItemsets = genFreqItemsets(data, minCount, freqItems, partitioner)
    new FPGrowthModel(freqItemsets)
  }
```

考虑篇幅有限，统计item的频次比较简单，就不阐述了。下面介绍生成频繁项的方法：

```
private def genFreqItemsets[Item: ClassTag](
      data: RDD[Array[Item]],
      minCount: Long,
      freqItems: Array[Item],
      partitioner: Partitioner): RDD[FreqItemset[Item]] = {
    val itemToRank = freqItems.zipWithIndex.toMap
    data.flatMap { transaction =>
      // 得到每个分区相关的交易数据
      genCondTransactions(transaction, itemToRank, partitioner)
      // 按照key进行聚合，key为group_id
    }.aggregateByKey(new FPTree[Int], partitioner.numPartitions)(
      // 用交易数据对树进行构建
      (tree, transaction) => tree.add(transaction, 1L),
      // 因为是key是group_id，所以不同item树可能不再一个分区中，
      // 所以不同分区，同一个group_id的树进行合并
      (tree1, tree2) => tree1.merge(tree2))
    .flatMap { case (part, tree) =>
      // 对不同gid下的树，进行频繁项的挖掘,第二个参数控制只挖自己gid负责的item
      tree.extract(minCount, x => partitioner.getPartition(x) == part)
    }.map { case (ranks, count) =>
      new FreqItemset(ranks.map(i => freqItems(i)).toArray, count)
    }
  }
```

然后是看怎么生成各个group相关的交易数据的。

```
private def genCondTransactions[Item: ClassTag](
      transaction: Array[Item],
      itemToRank: Map[Item, Int],
      partitioner: Partitioner): mutable.Map[Int, Array[Int]] = {
    val output = mutable.Map.empty[Int, Array[Int]]
    val filtered = transaction.flatMap(itemToRank.get)
    ju.Arrays.sort(filtered)
    val n = filtered.length
    var i = n - 1
    while (i >= 0) {
      val item = filtered(i)
      // 从后往前。得到item对应的group_id
      val part = partitioner.getPartition(item)
      if (!output.contains(part)) {
        // 如果输出没有这个group_id的数据，加进输出。key为group_id
        output(part) = filtered.slice(0, i + 1)
      }
      i -= 1
    }
    output
  }
```

FP树的构建

```
def add(t: Iterable[T], count: Long = 1L): this.type = {
    require(count > 0)
    var curr = root
    curr.count += count
    t.foreach { item =>
      // summaries用于维护每个item的频次和关联的节点
      val summary = summaries.getOrElseUpdate(item, new Summary)
      summary.count += count
      // 如果当前curr指向节点的子节点没有item，则新建节点，父节点为curr，更新summary
      val child = curr.children.getOrElseUpdate(item, {
        val newNode = new Node(curr)
        newNode.item = item
        summary.nodes += newNode
        newNode
      })
      child.count += count
      // curr指向新加的节点
      curr = child
    }
    this
  }
```

频繁项的挖掘代码

```
def extract(
      minCount: Long,
      validateSuffix: T => Boolean = _ => true): Iterator[(List[T], Long)] = {
    summaries.iterator.flatMap { case (item, summary) =>
      // summaries维护了所有item的count和节点，拿到前缀符合，频次也符合的节点
      if (validateSuffix(item) && summary.count >= minCount) {
        // 首先单节点自身是个频繁项，然后拼接递归挖掘出的频繁项
        Iterator.single((item :: Nil, summary.count)) ++
          // 先生成item对应的子树，然后递归的进行挖掘
          project(item).extract(minCount).map { case (t, c) =>
             // 子树挖掘的频繁项记得拼上递归前的前缀
            (item :: t, c)
          }
      } else {
        Iterator.empty
      }
    }
  }
```

子树的生成

```
private def project(suffix: T): FPTree[T] = {
    // 新生成一棵树
    val tree = new FPTree[T]
```

```
    if (summaries.contains(suffix)) {
      // 得到item的所有节点
      val summary = summaries(suffix)
      summary.nodes.foreach { node =>
        var t = List.empty[T]
        var curr = node.parent
        // 往上获取到root路径上的所有节点
        while (!curr.isRoot) {
          t = curr.item :: t
          curr = curr.parent
        }
        // 节点的频次都设置为当前node的频次
        tree.add(t, node.count)
      }
    }
    tree
  }
```

五：总结

本文通过简单案例入手，一步步的分析FP-Growth的原理，包括FP树的构建，频繁项的挖掘。然后分析并行FP-Growth以及其在spark中的实现。

认真看完上述的分析，还不懂FP-Growth的算我输！！

六：参考文献

[1] : PFP: Parallel FP-Growth for Query Recommendation