Big Data Mining

Hashing

LSH(Local Sensitive Hashing,局部敏感哈希)

Shingling

👉 一个文档的k-shingle(也称k-gram):连续k个字符一起出现的序列

$$\mathrm{k}=2 \ \mathrm{doc}=\mathrm{abcab} \ \mathrm{set\ of\ 2\text{-}shingles}=ab,bc,ca$$

👉 通过Shingling技术，可以将文档转换为一个特征向量。将多个文档的特征向量组合即可得到一个矩阵

Minhashing(最小哈希)

👉 将Shingling得到的矩阵进行降维

👉 最小哈希步骤

- 计算签名矩阵

- 通过签名矩阵寻找相似的签名

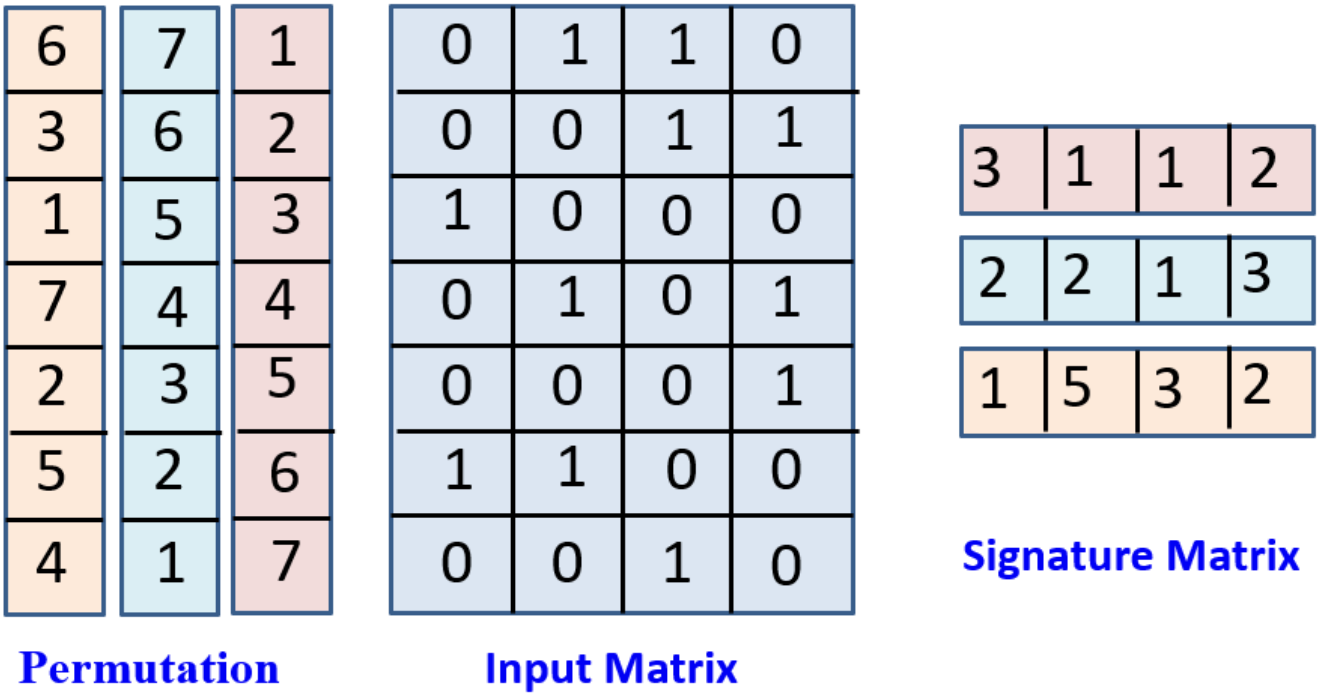
Signature Matrix

👉 将每一列C转换为一个更小的$Sig(C)$，理论上可保证$Sim(C_1,C_2)$与$Sim(Sig(C_1),Sig(C_2))$相等

👉 计算相似度使用的是Jaccard Distance

👉 假设对行号进行随机扰动，定义一个哈希方程$h(C)$，其函数值为扰动过后的某列中第一个值为1的元素的行号(行号也为扰动过的行号)

👉 Case 1



- 未扰动时得到的签名矩阵第一行为$(3,1,1,2)$
- 第一次扰动得到的签名矩阵第二行为$(2,2,1,3)$
- 之后同理
- 将7×4的输入矩阵转换为3×4的签名矩阵

👉 为什么转换后的相似度还是相等

$$Sim(C_1,C_2)=Pr(h(C_1)=h(C_2))$$

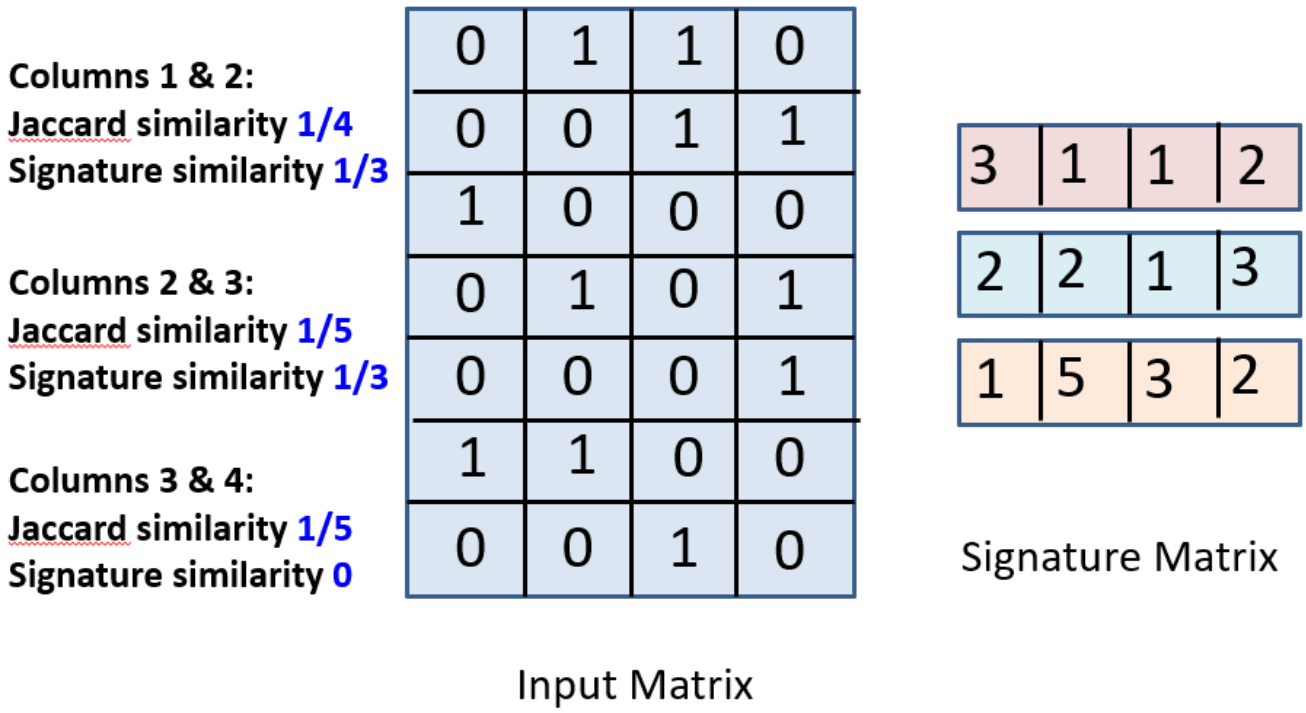
- 因为$h(C)$的值为第一个出现1的行号，故$h(C_1)=h(C_2)$只能为两个列第一个出现1的行号相等,即为$1,1$(相当于计算Jaccard Distance中的a)，而$0,1$和$1,0$可以视为b,c，不考虑$0,0$的情况，所以

$$Pr(h(C_1)=h(C_2))=\frac{a}{a+b+c}=Jaccard(C_1,C_2)$$

- 但是扰动的次数一定要足够多，才能保证转换后的相似度不变，因为上式左边采用的是**极大似然的思想**

👉 Case 1 Continue

- 在Input Matrix中计算每一列雅卡尔相似度
- 在Signature Matrix中计算签名矩阵每一列的相似度



👉 实际实现

伪代码如下，直接进行扰动开销太大，通过**哈希函数**进行扰动

下面的算法只需遍历一遍文档即可实现

每一次扰动即为对行号的改变，扰动之后需要找到元素为1且行号最小的行标

而遍历文档，依次对行号改变，只需不断记录元素为1的最小行号即可，不断更新最小行号，遍历完成后即可得到签名矩阵

Initialize M(i,c) to ∞ for all i and c

for each row *r*

  for each column *c*

    if c has 1 in row *r*

      for each hash function h_i **do**

        if $h_i(r)$ is a smaller value than $M(i, c)$ **then**

          $M(i, c) := h_i(r)$;

- `M(i,c)`是初始化(每个元素都是∞)的矩阵，`i`是扰动的次数,`c`是文档的个数(即输入矩阵的列数)
- Case 2

Row	C1	C2
1	1	0
2	0	1
3	1	1
4	1	0
5	0	1

$h(x) = x \bmod 5$
$g(x) = 2x+1 \bmod 5$

	Sig1	Sig2
$h(1) = 1$	1	-
$g(1) = 3$	3	-
$h(2) = 2$	1	2
$g(2) = 0$	3	0
$h(3) = 3$	1	2
$g(3) = 2$	2	0
$h(4) = 4$	1	2
$g(4) = 4$	2	0
$h(5) = 0$	1	0
$g(5) = 1$	2	0

  - 对每一个数据进行两次扰动

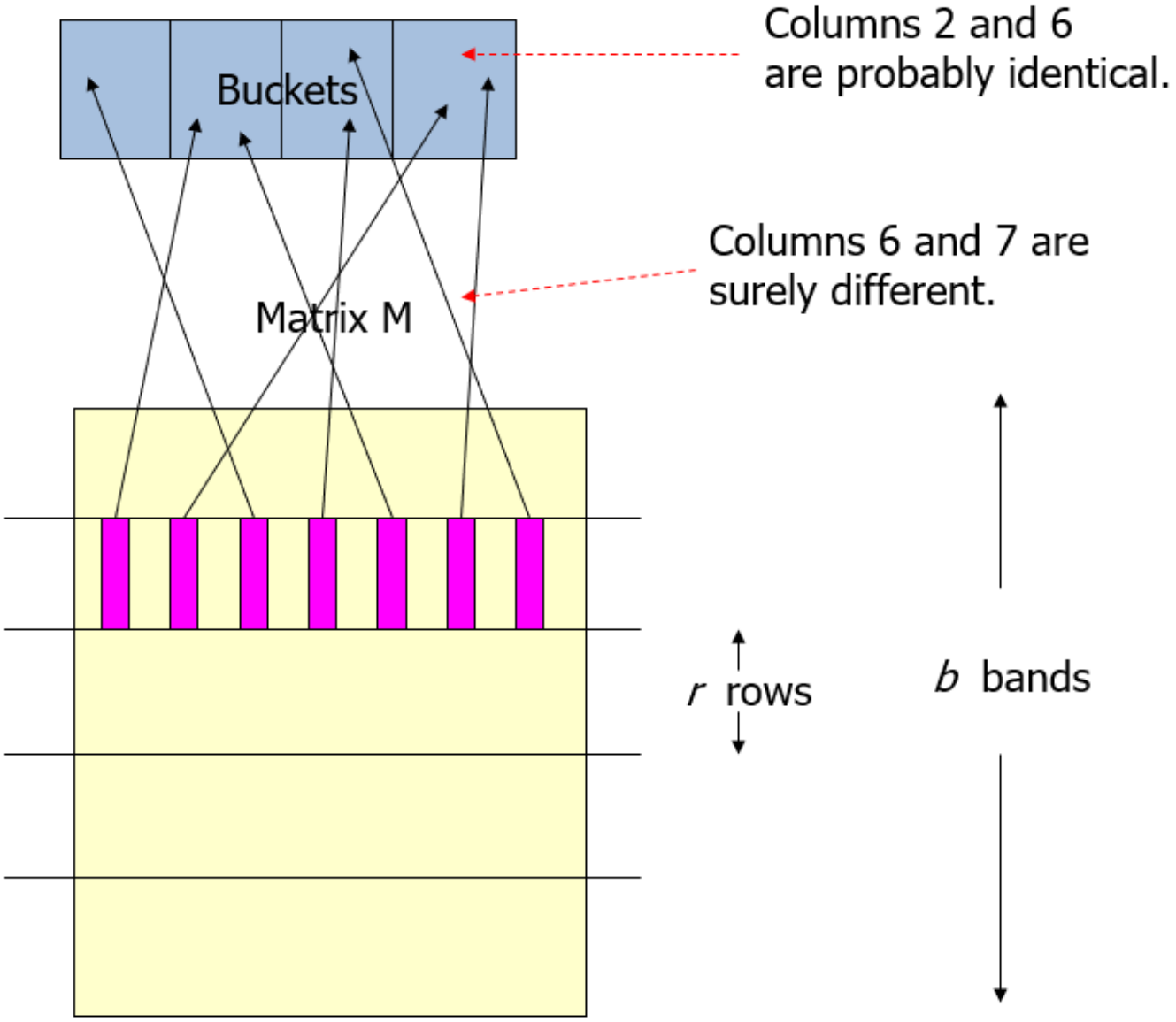
LSH——找相似的文档

👉 即使是签名矩阵，两两计算相似度(枚举)时间开销是很大的

👉 将每一个文档通过哈希映射到不同的桶中(和关联规则挖掘提高Apriori算法的Hashing思想类似)，一个桶中的数据是相似的

👉 但一般不是把每个文档直接映射，而是将签名矩阵划分为几个bands，分别进行映射。**如果两个文档是相似的，则两个文档至少有一个band被映射到同一个桶中**

下图中一个band有r行 一共有b个bands



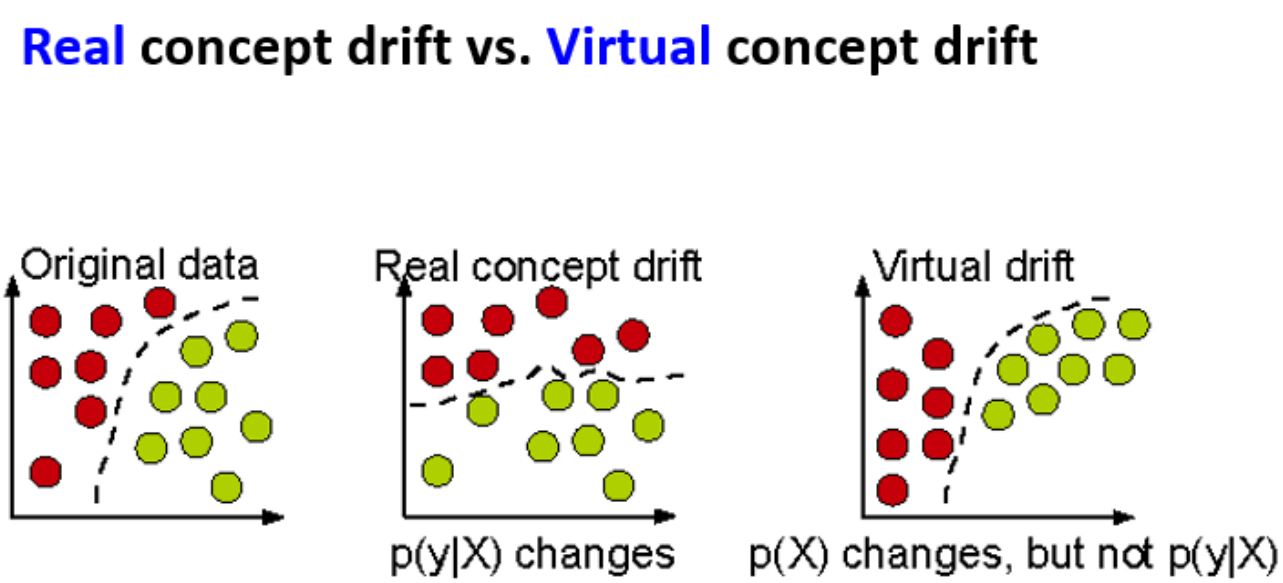
数据流挖掘

👉 数据流具有的特征

- 依次到达(One by One)
- 没有边界(Potentially Unbounded)
- **概念漂移**(Concept Drift)

概念漂移

- 目标变量的统计特征随着时间变化
- 变量的条件概率分布$P(C|X)$改变，C是类别,X是特征向量



$$P(C_i \mid X) = \frac{P(C_i)P(X \mid C_i)}{P(X)}$$

👉 第三幅图为**虚假的概念漂移**，即条件概率分布没有发生变化，只是特征向量的分布发生了变化

检测概念漂移

基于分布的检测(Distribution-based detector)

👉 设置一个窗口大小，窗口截取不同时间段数据快照，检验分布是否发生变化

👉 缺点

- 很难确定窗口大小
- 学习概念漂移缓慢
- 会将虚假概念漂移视为真概念漂移

👉 **Adaptive Windowing**(ADWIN)

begin
  Initialize Window W;
  foreach $(t) > 0$ **do**
    $W \leftarrow W \cup \{x_t\}$ (i.e., add x_t to the head of W);
    repeat
      Drop elements from W
    until $|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| < \epsilon_{cut}$ *holds for every split of* W *into* $W = W_0 \cup W_1$;
  end
  Output $\hat{\mu}_W$
end

👉 不断的丢掉旧的数据(从时间上看，旧的数据)，将一个窗口分为两部分，将分界线进行左右滑动，若无论怎么滑动，被分的两部分都没有显著性差异，则接受新的窗口；否则，一直丢掉旧的数据

基于错误率的检测(Error-rate based detector)

👉 检测**分类错误率**的改变

👉 缺点

- 对噪声敏感
- 难以处理缓慢(渐变)的概念漂移
- 十分依赖于学习模型(模型本身的好坏)

👉 DDM算法

$$p_i + s_i \geq p_{\min} + 3 * s_{\min}$$

- p_i是错误率，s_i是标准差，p_{min}, s_{min}分别是历史最小的错误率和标准差

数据流分类

Decision Tree Learning

VFDT(Very Fast Decision Tree)

👉 算法流程如下所示

- 用数据流最初的数据来挑选根节点的属性
- 要用霍夫丁上界决定每个节点用多少个数据才可以挑选出对应的分裂属性
- X_a是指运用信息增益等指标挑选出的最优的分裂属性
- X_b是指挑选出的次优的分裂属性
- ε是霍夫丁上界
- 如果满足关系式，则进行决策树的分裂迭代，若不满足，则需要收集更多的数据

Calculate the information gain for the attributes and determines the best two attributes

At each node, check for the condition

$$\Delta\overline{G} = \overline{G}(X_a) - \overline{G}(X_b) > \varepsilon$$

If condition satisfied, create child nodes based on the test at the node

If not, stream in more examples and perform calculations till condition satisfied

数据流聚类

👉 **Online**——在内存中会存储每个簇结构的一些统计特征，并会**实时动态更新**

- 内存中存储的是**微簇**(Micro-Cluster)
  - 将相似的点视为一个点
  - 压缩数据，并可以代表这些数据

👉 **Offline**——用K-Means或DBSCAN对数据进行聚类

Cluster Feature

👉 特征簇算法

👉 有三个特征$CF = (N, LS, SS)$

$$N\text{为数据点的个数 } LS = \sum_{i=1}^{N} X_i \; SS = \sum_{i=1}^{N} X_i^2$$

- 可以计算中心和半径

👉 当来了一个新数据X_j

$$CF' = (N+1, LS + X_j, SS + X_j^2)$$

- 可以增量式的计算

CluStream

👉 特征簇算法的经典算法

Hadoop/Spark

Hadoop

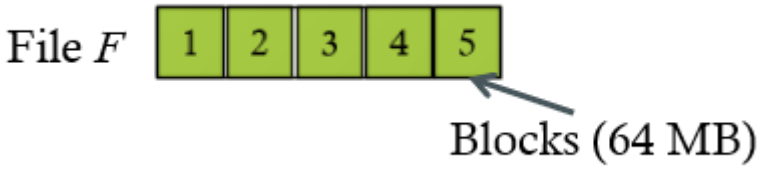
👉 通过分布式处理海量数据，基本思想：**分而治之**

- 自动进行并行化和分布式
- 为用户提供编程接口
- 错误容许和自动恢复
    - 某台机器崩了之后，会自动恢复

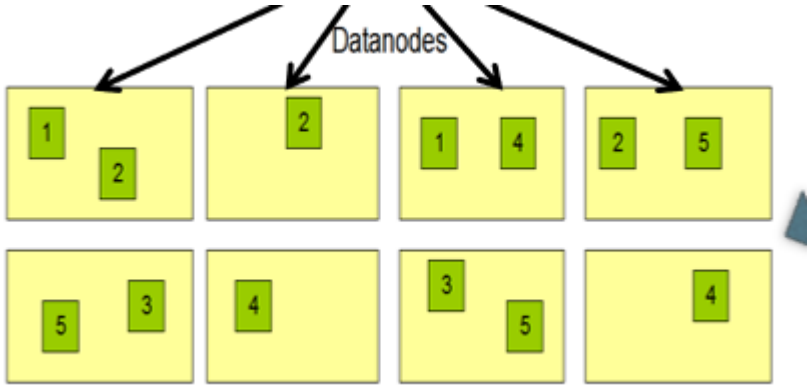
HDFS

👉 Hadoop Distributed File System 用于存储大数据

- **namenode**



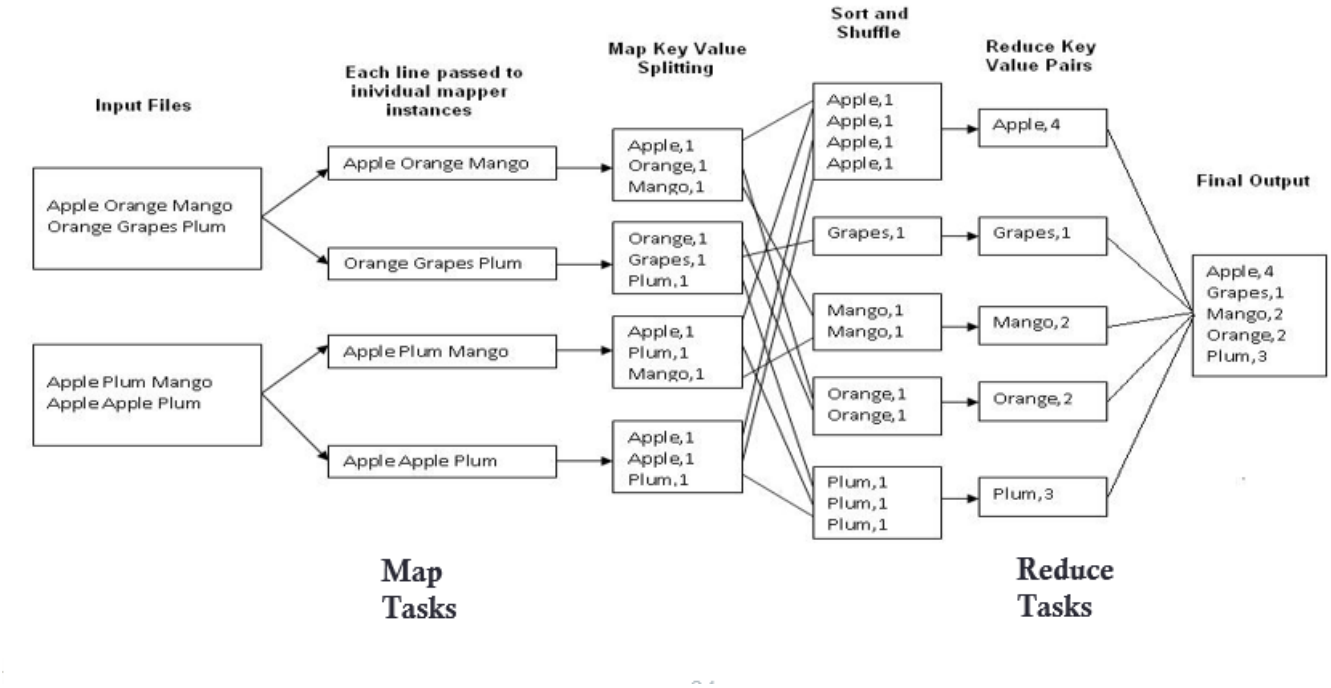
    - 元数据的标记



- **datanode**
    - 存储真实的数据
    - 每个文件被分为很多块
    - 每个块被复制N次(N默认为3)
- 利用namenode检测某个块是否**挂掉**，利用**心跳机制**，复制的每个块定时发送信息

MapReduce

👉 用于计算



👉 对于一次计算速度很快(即单通道)，但是对于多次计算，I/O开销太大

👉 局限性

- 单通道计算有优势，对于多通道计算效率低
- 复制Block和磁盘存储慢
- 文件操作的不同阶段被HDFS的读写间隔 从而造成大量的IO开销

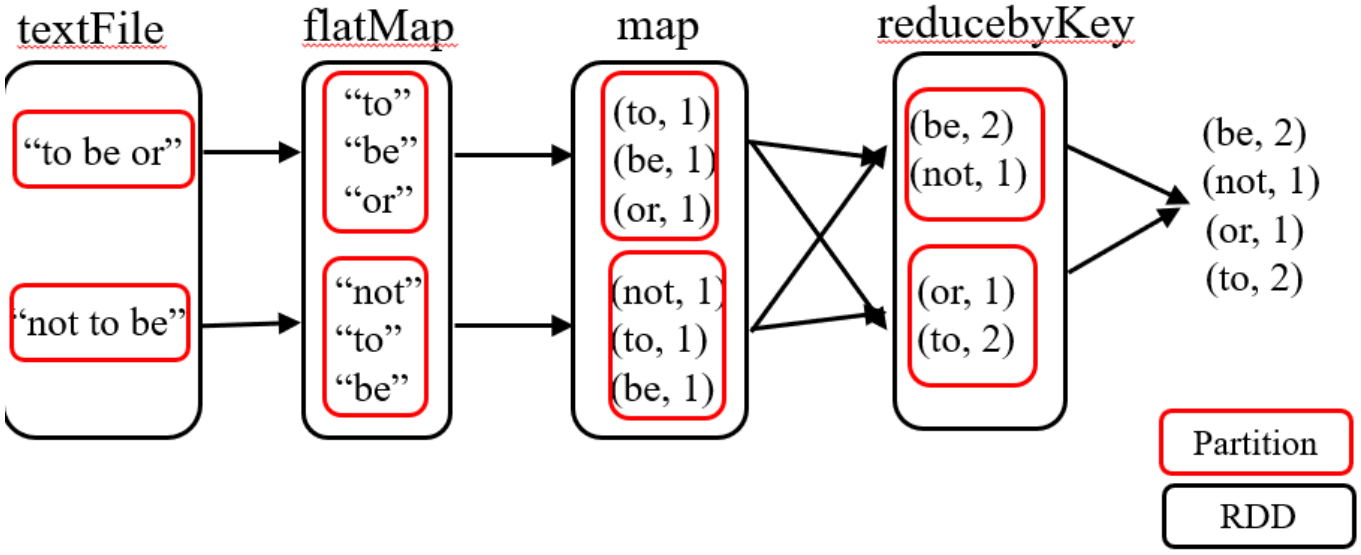
Spark

👉 Spark主要是用于计算，解决了MapReduce存在的问题，发明了RDD

👉 Spark的存储仍然使用HDFS

RDD(resilient distributed dataset)

👉 弹性式分布数据集



Hadoop vs Spark

👉 存储都用HDFS 仅比较计算

- MapReduce
  - 对于单通道计算很有效，对于多通道计算效率低
- Spark
  - RDD 多通道计算
  - 对于别的语言的接口 很简洁
  - 同一台机器可以进行数据抽象，模型训练和交互