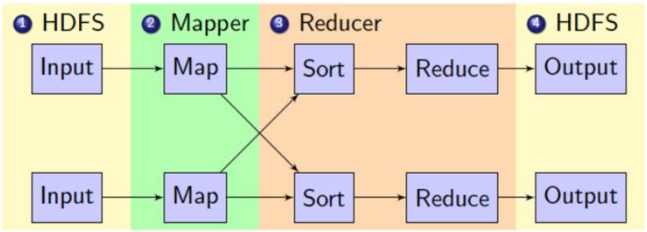
1. Hadoop

(1) Hadoop 的基本概念



(i)Hadoop 的起源

(ii)主要工作流程：

(a) 从 HDFS 上读取文件

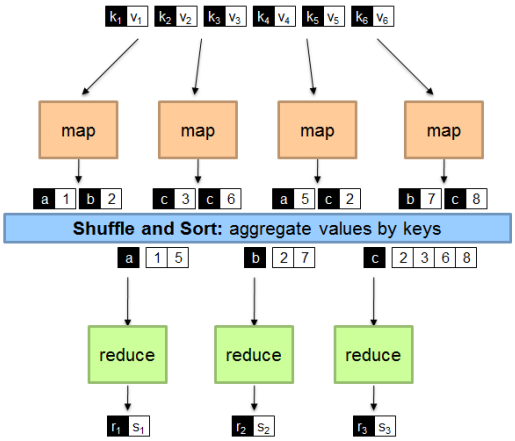
(b*) Mapper 处理读到的文件：将文件内容转化为(k, v)

(c) shuffle and sort：将所有(k, v)重新排序，将相同的 key 分发到同一个 reducer 中。

(d*) Reducer 处理被分发到该 Reducer 的所有 (k, v)

(e) 把结果写到 HDFS 上

注意：



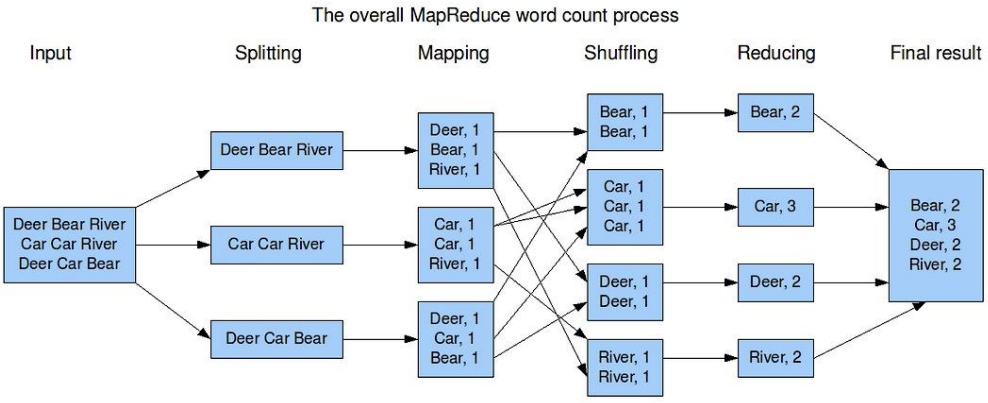
Hdfs 是服务于 hadoop 的一个虚拟文件系统，在一个集群中，文件会被切割成每块 64M 的若干小文件，分别放在不同的主机中，Hadoop 只能在 HDSF 上操作文件；

Hadoop 中 HDFS，Mapper，Reducer 全部用键值对进行通信；

需要编写代码的部分包括 Mapper，Reducer，和 driver，其中 driver 需要写在 main 函数中，主要用于告诉 Hadoop 哪个是 Mapper，哪个是 Reducer，如何启动 Hadoop 项目，指定输入输出；

Shuffle and sort 的中间结果也会被写到硬盘上（不是在 hdfs 上，代码结束后就会被删除），效率较慢，因此出现了 Spark。

Wordcount 实例：



改进：完成以上内容就可以解决问题了，但是效率低下，Hadoop 提供了提高效率的方法

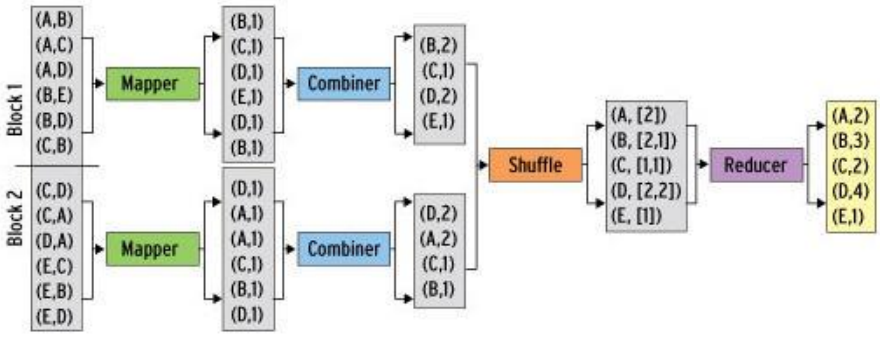
(a) combiner

Combiner 是为了减少 Reducer 的工作量，e.g. 10 X (“are”, 1) vs 1 X (“are”, 10)

注意：可有可无，combiner 只是提高效率，和结果的正确性没关系

combiner 的输入格式必须与 mapper 的输出格式相同，并且与 reducer 的输入格式相同

我称之为“小 reducer”，因为实际上它就是个 reducer，它也继承 Reducer 这样一个父类

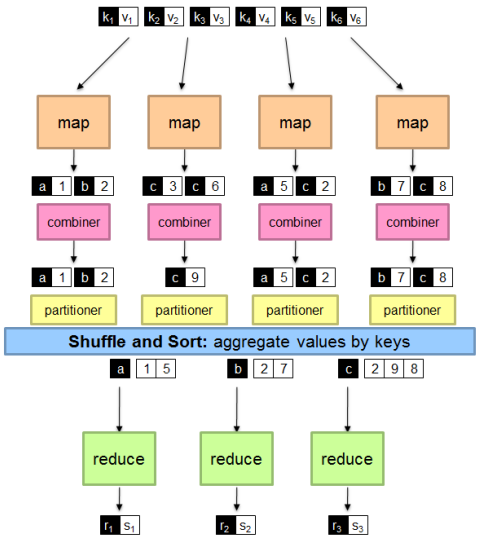


(b) partitioner

多个 reducer 的时候，可以进一步提高计算效率，但是出现一个问题，就是每个(k, v)应该被分发到哪个 reducer 里面，partitioner 就是决定这件事的。

（除此之外，可以通过修改 hashCode()的方法来决定(k,v)被分配给哪个 reducer）

总结：



(2) Hadoop 的四个设计模式

(i) In-mapper Combining：处理 wordcount, computing the mean 问题

Mapper 的 lifecycle：setup() -> map() -> cleanup()

Wordcount: (不用 combiner 依然可以提高 wordcount 的效率)

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, pair (r, 1))
```

```
1: class Combiner
2:     method Combine(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         Emit(string t, pair (sum, cnt))
```

```
1: class Reducer
2:     method Reduce(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         ravg ← sum/cnt
9:         Emit(string t, pair (ravg, cnt))
```

```
1: class Mapper
2:     method Initialize
3:         S ← new AssociativeArray
4:         C ← new AssociativeArray
5:     method Map(string t, integer r)
6:         S{t} ← S{t} + r
7:         C{t} ← C{t} + 1
8:     method Close
9:         for all term t ∈ S do
10:            Emit(term t, pair (S{t}, C{t}))
```

左边是传统的 combiner 统计词长，右边是利用 in-mapper combining 统计词长，都封装在 Pair 对象中进行通信。

(ii) Pairs vs Stripes：处理 Term Co-occurrence Computation 问题

(a)Pair：以 Pair 的方式通信

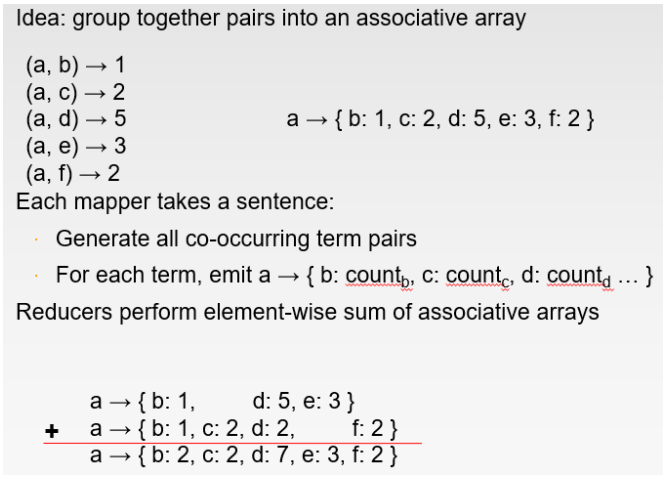
```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ Neighbors(w) do
5:                 Emit(pair (w, u), count 1)        ▷ Emit count for each co-occurrence
```

```
1: class Reducer
2:     method Reduce(pair p, counts [c1, c2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             s ← s + c                              ▷ Sum co-occurrence counts
6:         Emit(pair p, count s)
```

Advantages: Easy to implement, easy to understand

Disadvantages: Lots of pairs to sort and shuffle around; Not many opportunities for combiners to work

(b) Stripe：以 stripe 的方式通信



Advantages: Far less sorting and shuffling of key-value pairs; Can make better use of combiners

Disadvantages: More difficult to implement; Underlying object more heavyweight (maintaining a stripe); Fundamental limitation in terms of size of event space

(iii) Order Inversion: 处理 Computing Relative Frequencies 问题

(a, b)的 relative frequency 是(a, b)共现的次数除以(a, *)出现的次数总和，其中*代表任意字符串。

(a, b)用原始方法就可以得到统计结果，而(a, *)需要得到所有跟 a 有关的共现对以后，进行加和，才能统计得到(a, *)的次数，在此之前所有(a, b), (a, c)…都必须留存在内存中，容易造成内存溢出。Order inversion 就是为解决此问题。

key	values	
$(\text{dog}, *)$	[6327, 8514, …]	compute marginal: $\sum_{w'} N(\text{dog}, w') = 42908$
$(\text{dog}, \text{aardvark})$	[2,1]	$f(\text{aardvark}\|\text{dog}) = 3/42908$
$(\text{dog}, \text{aardwolf})$	[1]	$f(\text{aardwolf}\|\text{dog}) = 1/42908$
…		
$(\text{dog}, \text{zebra})$	[2,1,1,1]	$f(\text{zebra}\|\text{dog}) = 5/42908$
$(\text{doge}, *)$	[682, …]	compute marginal: $\sum_{w'} N(\text{doge}, w') = 1267$
…		

(iv) Value-to-key Conversion: 处理 secondary sort 问题

当我们想让 Reducer 输出结果(k, v1),(k,v2)…中的 vi 也进行排序，如果 Reducer 中排序运行排序算法，会消耗大量内存。因此我们想利用 hadoop 的内部机制帮我们自动排序，即 Reducer 收到的(k,v1),(k,v2)…中的 vi 就是排好序的。该设计模式就是帮我们解决这个问题的。

方法：把将(k, v1), (k, v2)… 转化成 ((k, v1), v1), ((k, v2), v2)…，在 hadoop 运行中的 shuffle and sort 这一步中就会帮我们做好排序，当然我们要编写代码，告诉 hadoop 我们的排序规则是什么。

以上是算法部分，真正写代码的时候会遇到如下问题：

```
(i) in-mapper combining |mapper里构建一个HashMap来替代combiner
(ii) pair/stripe |mapper是发送一个一个的pair呢 还是整体发送一个HashMap呢
(iii) order inversion |虚拟出来一个(a, *)来统计所有第一个元素为a的词对
(iv) value to key conversion (secondary sort) |要将value也进行排序



(ii), (iii), (iv)涉及到pair作为key，要这么写：
class pair
    first, second
    int compareTo(Pair p)
        int ret = this.getFirst.compareTo(p.getFirst)
        if (ret == 0) ret = this.getSecond.compareTo(p.getSecond)

(iii), (iv) 涉及到将pair里的第一个元素分配到同一个partition里执行，要这么写：
class Partitioner
    method int getPartition(key, value, int numPartitions)
        return key.first.hashCode() & Integer.MAX_VALUE % numPartitions

(iv) 涉及到将pair里的第一个元素group到一起，要这么写：
class PairGroupingComparator extends WritableComparator
    method int compare(WritableComparable wc1, WritableComparable wc2)
        return ((Pair) wc1).getFirst.compareTo(((Pair) wc2).getFirst)
```

2. spark

(1) Spark 的基本概念

(i) Spark 介绍： In-memory computing primitives

Spark 的特性：

Expressive computing system, not limited to map-reduce model

Facilitate system memory **(in-memory computing)**

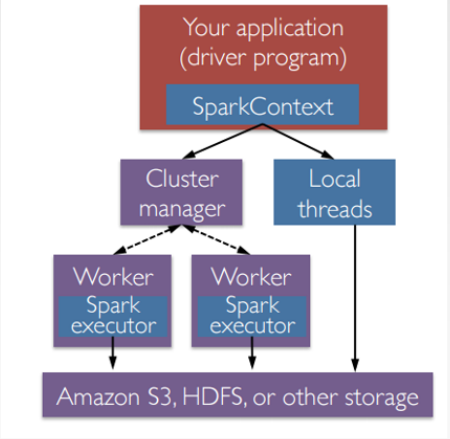
avoid saving intermediate results to disk

cache data for repetitive queries (e.g. for machine learning)

Layer an in-memory system on top of Hadoop.

Achieve fault-tolerance by re-execution instead of replication. (Is able to recompute missing or damaged partitions due to node failures)

Spark 的工作流程：



- A Spark program first creates a SparkContext object
  - Tells Spark how and where to access a cluster
  - Connect to several types of cluster managers (e.g., YARN, Mesos, or its own manager)
- Cluster manager:
  - Allocate resources across applications
- Spark executor:
  - Run computations
  - Access data storage

Wordcount 实例:

```
val file = sc.textFile("hdfs://...")

val counts = file.flatMap(line => line.split(" "))
   .map(word => (word,1))
   .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

(ii) RDD 基础及分类

RDD 的特性:

In-Memory, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.

Immutable or **Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.

Lazy evaluated, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.

Cacheable, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).

Parallel, i.e. process data in parallel.

Typed, i.e. values in a RDD have types, e.g. RDD[Long] or RDD[(Int, String)].

Partitioned, i.e. the data inside a RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node).

RDD 的分类:

Transformation: returns a new RDD.

- Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.
- Transformation functions include *map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, filter, join, etc.*

Action: evaluates and returns a new value.

- When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.
- Action operations include *reduce, collect, count, first, take, countByKey, foreach, saveAsTextFile, etc.*

Transformation 产生新的 RDD

Action 产生新的 value

RDD transformation 的 lazy evaluation：

RDD 当遇到 actions 的时候才真正的开始计算。

如何创建 RDD：

by parallelizing existing collections (lists or arrays) (val inputfile = sc.textFile("...", 4))

by transforming an existing RDDs (val inputVariable = sc.parallelize([1, 2, 3]))

from files in HDFS or any other storage system (val counts = file.map(word => (word, 1)))

Map vs flatMap (这个考试一定会涉及)：

```
comp9313@comp9313-VirtualBox:~$ hdfs dfs -cat inputfile
This is a short sentence.
This is a second sentence.
```

```
scala> val inputfile = sc.textFile("inputfile")
inputfile: org.apache.spark.rdd.RDD[String] = inputfile MapPartitionsRDD[1] at t
extFile at <console>:24
```

```
scala> inputfile.map(x => x.split(" ")).collect()
res3: Array[Array[String]] = Array(Array(This, is, a, short, sentence.), Array(T
his, is, a, second, sentence.))
```

```
scala> inputfile.flatMap(x => x.split(" ")).collect()
res4: Array[String] = Array(This, is, a, short, sentence., This, is, a, second,
sentence.)
```

Write down the output

a) val lines = sc.parallelize(List("hello world", "this is a scala program", "to create a pair RDD", "in spark"))

val pairs = lines.map(x => (x.split(" ")(0), x))

pairs.filter {case (key, value) => key.length <3}.foreach(println)

Output: (“to”, “to create a pair RDD”) (“in”, “in spark”)

b) val pairs = sc.parallelize(List((1, 2), (3, 4), (3, 9), (4,2)))

val pairs1 = pairs.mapValues(x=>(x, 1)).reduceByKey((x,y) => (x._1 + y._1, x._2+y._2)).mapValues(x=>x._2/x._1)

pairs1.foreach(println)

Output: (1, 0) (3, 0) (4, 0) (because no “.toDouble” used)

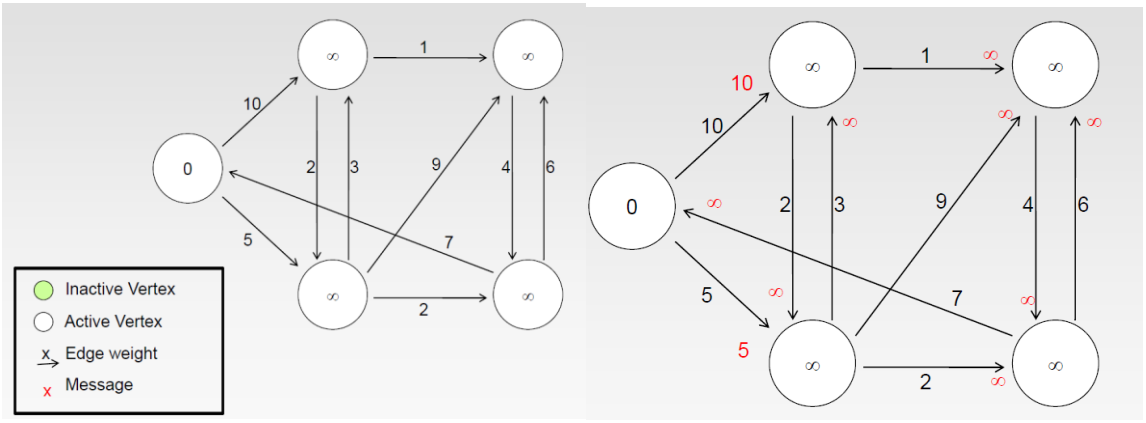
(iii) spark 里面有两种共享变量：broadcast 广播变量，accumulator 累加器

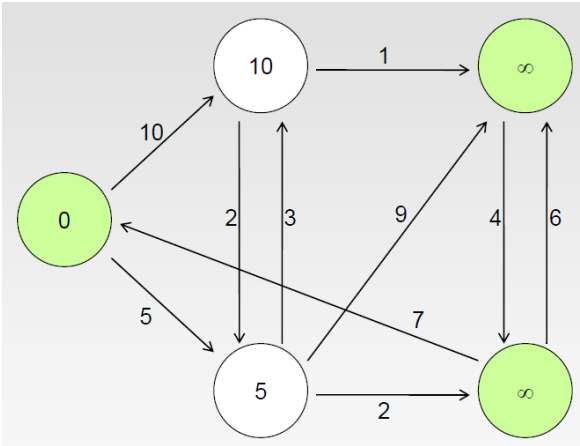
由于在分布式系统中，同一个程序在多台机器上运行，维护不同的数据，如果某一台主机想要频繁的访问另一台主机的数据，每次访问都要进行主机之间的通讯，这样效率很低。Spark 给我们提供了 broadcast variable，把数据放到这里之后，就相当于在每个主机上复制了一份该数据，每次访问该数据就行了，避免了主机之间的通讯。

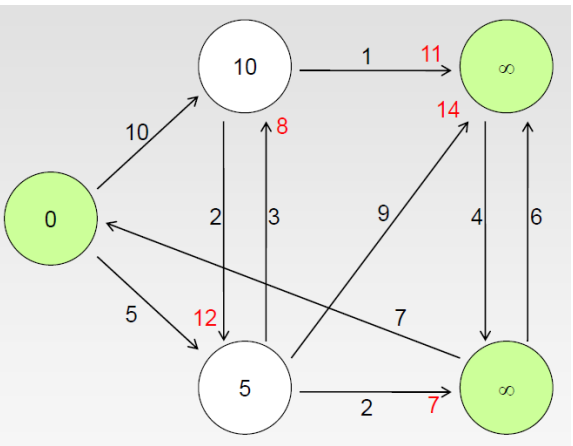
Accumulator 也可以提供一个全局的共享变量，每个主机都能像里面累加值。

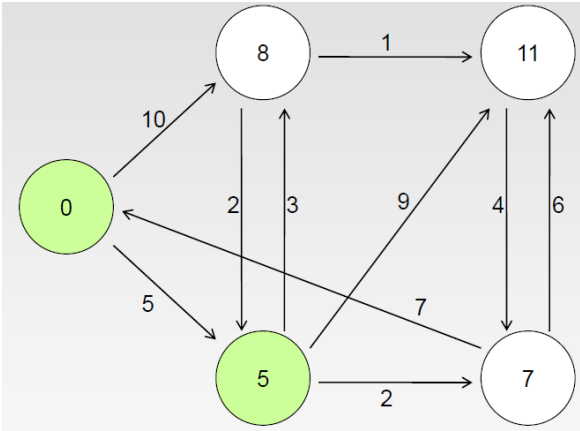
(iv) graphX (pregel operator)

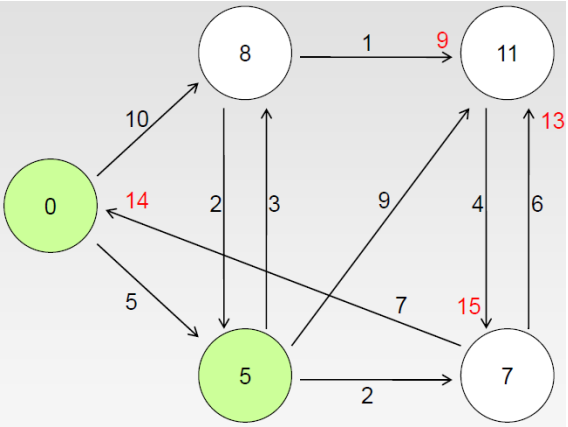
原理：这是一个要进行多个轮次的过程，每个节点都向相邻节点发送 message，如果某一轮有一个节点没有收到消息，那么它将变为 inactive 状态，inactive 状态的节点下一轮将不会向外发送 message，当所有节点都变成 inactive，图计算结束。

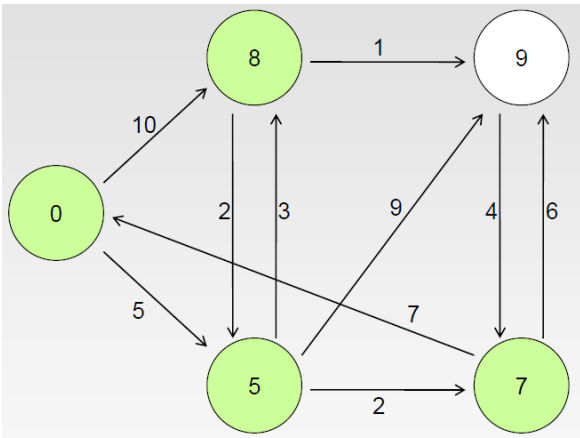


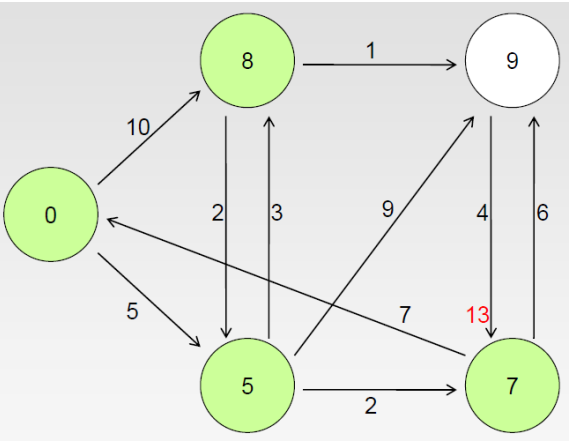

10
1
∞
10
0
2
3
9
4
6
5
7
5
2
∞

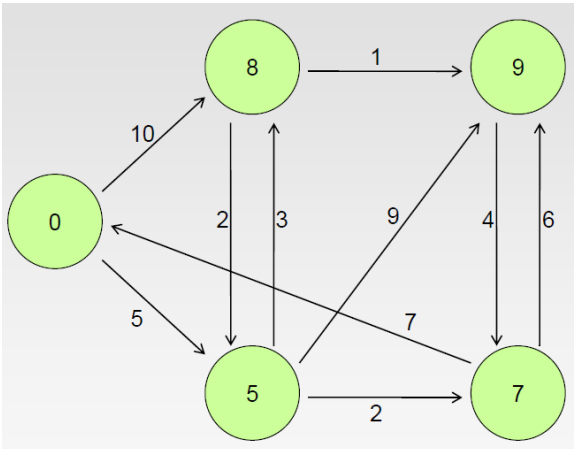

10
1
11
∞
14
10
8
0
2
3
9
4
6
5
12
7
5
2
7
∞


8
1
11
10
0
2
3
9
4
6
5
7
5
2
7


8
1
9
11
10
13
0
14
2
3
9
4
6
5
7
15
5
2
7


8
1
9
10
0
2
3
9
4
6
5
7
5
2
7


8
1
9
10
0
2
3
9
4
6
5
7
13
5
2
7


8
1
9
10
0
2
3
9
4
6
5
7
5
2
7

```
def pregel[A]
    (initialMsg: A,
     maxIter: Int = Int.MaxValue,
     activeDir: EdgeDirection = EdgeDirection.Out)
    (vprog: (VertexId, VD, A) => VD,
     sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
     mergeMsg: (A, A) => A)
  : Graph[VD, ED] = {
          … …
  }
```

注意，pregel 有两个参数列表（graph.pregel(list1)(list2)）。第一个参数列表包含配置参数初始消息（上帝发来的消息）、最大迭代数、发送消息的边的方向（默认是沿边方向出）。第二个参数列表包含用户自定义的函数用来接收消息（vprog）、计算消息（sendMsg）、合并消息（mergeMsg）。

Vprog（接收消息函数）：三个输入参数（id，节点原本的属性值，节点新来的属性值）该函数的意义是这两个属性值你要怎么计算或取舍

sendMsg（发送消息）：一个输入参数（triplet），这个输入参数包含很多信息，triplet.attr 是该 edge 本身的属性，triplet.srcId, triplet.srcAttr 是它源节点的 id 和属性，triplet.dstId, triplet.dstAttr 是它目标节点的 id 和属性。输出值必须是一个 Iterator 对象，第一个参数值是给哪个节点发送，第二个参数是发送什么值

mergeMsg（合并消息）：两个输入参数（属性值 1，属性值 2）返回该类型的值，该函数的意义就是当一个节点获得多个来源的数据的时候，如何计算和取舍

```
val initialGraph = graph.mapVertices((id, _) =>
   if (id == sourceId) 0.0 else Double.PositiveInfinity)
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {  Iterator.empty  }
  },
  (a, b) => math.min(a, b) // Merge Message
)
```

Data streams

7.3 data stream (数据流) 每一时刻接收到的数据组成数据流 （google query）

We can think of the data as infinite and non-stationary (the distribution changes over time)

7.4 data stream 的特性 传统 DBMS: finite (有限数据量), persistent data sets (能够持久保存数据)

Data stream: distributed(多个用户), continuous, unbounded(无法预知最多有多少), rapid, time varying, noisy, . . .

Characteristics

Huge volumes of continuous data, possibly infinite (数据量大)

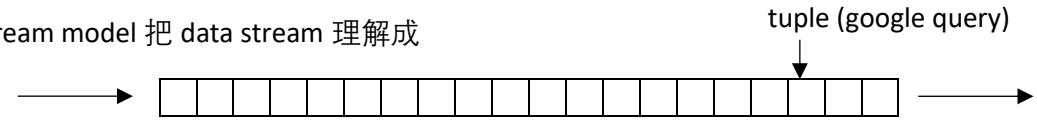
Fast changing and requires fast, real-time response (数据变化迅速)

Random access is expensive—single scan algorithm (can only have one look) (访问开销大)

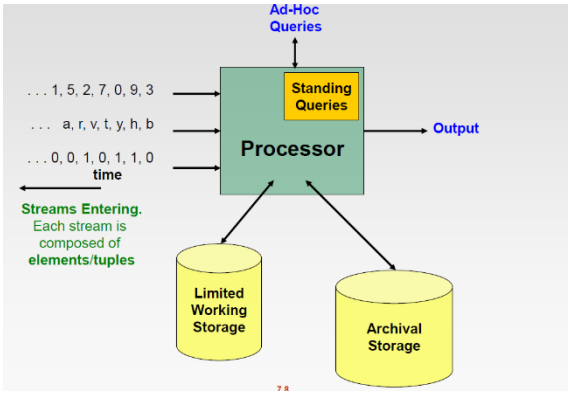
Store only the summary of the data seen thus far (所以我们只存 summary)

7.5 说 data stream 数据量大

7.6 stream model 把 data stream 理解成



7.7 DSMS 架构



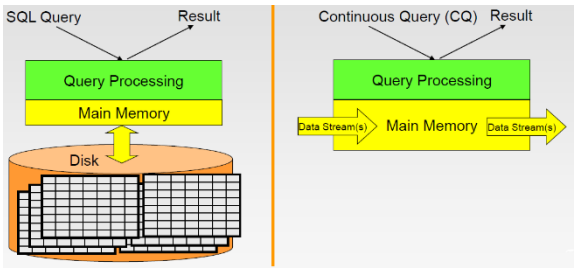
左边：理解成 google query

Ad-hoc query: 对 google query 的 query，比方说查询过去 10 分钟内某个词被查询的次数

Limited working storage: google query 的 summary

Archival storage: 把比较重要的信息存起来

7.8-7.11 DBMS vs DSMS



DBMS：向数据库(disk)查询

DSMS：向内存查询，并且结果随时间变化

其他对比跟之前的一样

7.12 DSMS 现存的问题（后面的内容都会一一解决）

7.13 DSMS 目前有哪些应用

7.14 例子：IP Network Data router 接收 data stream

7.15-7.25 第一个问题：Sampling from a Data Stream

Motivation: 整个 stream 太大了，所以我们只存 sample，那么如何 sample 呢？

思路一：sample 固定比例的 element（问题是总体越大，sample 也就越多）

例如如果只 sample 1/10 的数据，那么给每个 query 标上[0,1,2,3…9]的号，只留下标号为 0 的 tuple。但是这样做有个问题：

Simple question: What fraction of queries by an average search engine user are duplicates?

- Suppose each user issues ***x*** queries once and ***d*** queries twice (total of ***x+2d*** queries)
  - **Correct answer: *d*/(*x*+*d*)**
- **Proposed solution: We keep 10% of the queries**
  - Sample will contain ***x*/10** of the singleton queries and ***2d*/10** of the duplicate queries at least once
  - But only ***d*/100** pairs of duplicates
    - **d/100 = 1/10 · 1/10 · d**
  - Of ***d*** "duplicates" ***18d/100*** appear exactly once
    - **18d/100 = ((1/10 · 9/10)+(9/10 · 1/10)) · d**

So the sample-based answer is $\frac{\frac{d}{100}}{\frac{x}{10}+\frac{d}{100}+\frac{18d}{100}} = \frac{d}{10x+19d}$ **≠*d*/(*x*+*d*)**

想知道重复的占不重复的比例

原本 2d 个重复的 query 中有 2d/10 被 sample 出来，但是这里面有 18d/100 个变成了不重复的，其余 2d/100 个还是重复的

这个比例不是我们想要的比例

怎么办呢？sample user

How to generate a 30% sample?

Hash into b=10 buckets, take the tuple if it hashes to one of the first 3 buckets

思路二：sample 固定长度的 element

我们希望达到的目标是：每个元素被 sample 出来的概率是相等的。

S 是 sample 出来的集合，其长度为 s，n 为时刻，每一时刻来一个数据，那么 n 也可以认为是来了多少数据，我们希望每个元素被 sample 的概率都是 s/n

Stream: a x c y z k c d e g… different timestamps

At **n= 5,** each of the first 5 tuples is included in the sample **S** with equal prob.

At **n= 7,** each of the first 7 tuples is included in the sample **S** with equal prob.

Algorithm (a.k.a. Reservoir Sampling)

- Store all the first ***s*** elements of the stream to ***S***
- Suppose we have seen ***n*** elements, and now the ***(n+1)*th** element arrives (***n+1 > s***)
  - With probability ***s/(n+1)***, keep the ***(n+1)*th** element, else discard it
  - If we picked the ***(n+1)*th** element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random

以上算法我把 n 都换成了 n+1，为与下面公式保持一致

证明：数学归纳法，假如在 n 时刻，每个 element 被 sample 的概率为 s/n, 我们想证明下一时刻每个 element 被 sample 的概率均为 s/(n+1)

对于新来的 element，被保留的概率为 s/(n+1)

对于原本的 element，被保留的概率为

$$\left(1-\frac{s}{n+1}\right)+\left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right)=\frac{n}{n+1}$$

Element **n+1** discarded | Element **n+1** not discarded | Element in the sample not picked

So, at time ***n***, tuples in ***S*** were there with prob. **s/n**

Time ***n*→*n+1***, tuple stayed in ***S*** with prob. **n/(n+1)**

So prob. tuple is in ***S*** at time ***n+1*** = $\frac{s}{n}\cdot\frac{n}{n+1}=\frac{s}{n+1}$

7.26-7.46 第二个问题：query data stream

例如 counting bits

Problem:

Given a stream of **0**s and **1**s

Be prepared to answer queries of the form:
How many 1s are in the last *k* bits? where $k \le N$

Obvious solution:

Store the most recent ***N*** bits

When new bit comes in, discard the ***N*+1st** bit

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 1 1 0 1 1 0

Suppose N=7

Past Future

如果想获得准确答案，就必须存储大小为 N 的 sliding window，但实际上这个 N 是很大的，无法存储。

A simple solution that does not really solve our problem: **Uniformity Assumption**

N

0 1 0 0 1 1 1 0 0 0 1 0 1 0 0 1 0 0 0 1 0 1 1 0 1 1 0 1 1 1 0 0 1 0 1 0 1 1 0 0 1 1 0 1 0

Past Future

- Maintain 2 counters:
  - ***S***: number of 1s from the beginning of the stream
  - ***Z***: number of 0s from the beginning of the stream
- How many 1s are in the last ***N*** bits? $N\cdot\frac{S}{S+Z}$

假设他是均匀分布，但实际上 data stream 经常不是均匀分布。

于是出现了 DGIM 算法（四个人的名字…）

它的主体思想是把 data stream 分成块 (bucket)，每一块有 2 的若干次方的 1

1 0 0 1 0 1 0 1 1 0 0 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 0 1 0 0 0 1 0 1 1 0 0 1 0

N

Each bit in the stream has a timestamp, starting from **1**, **2,** …

Record timestamps modulo ***N*** (**the window size**), so we can represent any **relevant** timestamp in $O(\log_2 N)$ bits

E.g., given the windows size 40 (***N***), timestamp 123 will be recorded as 3, and thus the encoding is on 3 rather than 123

刚进入 data stream 的 1 都有一个 timestamp，它是的最大值为 N，用 O(log2N)个 bits 就可以存储

A bucket in the DGIM method is a record consisting of:

(A) The timestamp of its end [$O(\log N)$ bits]

(B) The number of 1s between its beginning and end [$O(\text{loglog}\, N)$ bits]

Constraint on buckets:

Number of 1s must be a power of 2

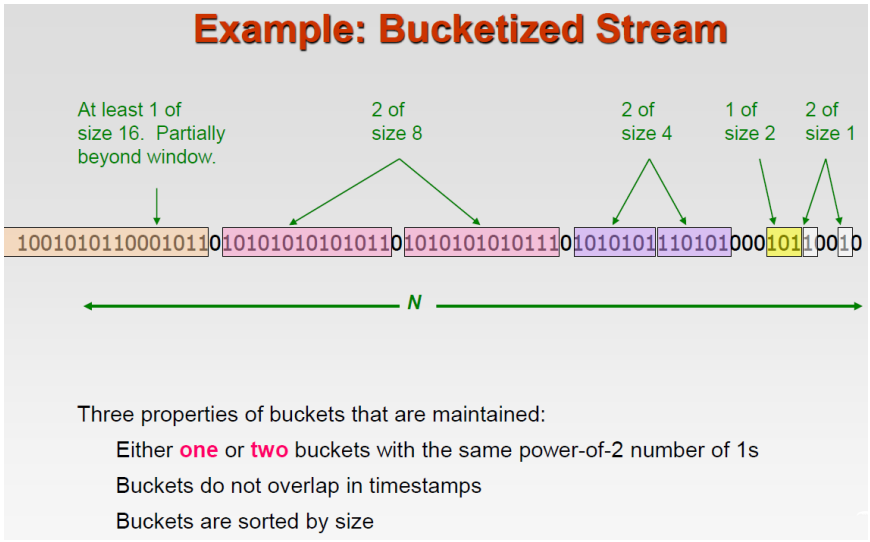
That explains the $O(\text{loglog}\, N)$ in (B) above

每个 bucket 之存储两样东西：

1.结束的那个 1 的 timestamp O(logN)

2.bucket 里面有多少个 1 O(loglogN)

其他的 timestamp 哪去了？被舍弃了…



每个 bucket：

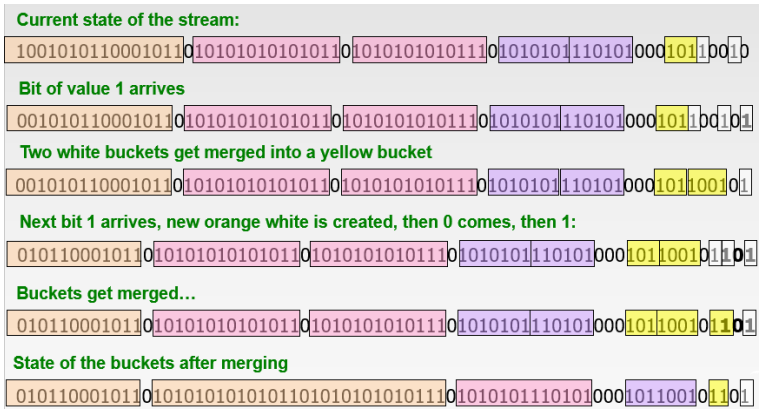
The right end of a bucket is always a position with a 1 （最右边是 1）

Every position with a 1 is in some bucket（至少有一个 1）

Either **one** or **two** buckets with the same **power-of-2 number** of **1s （2 的若干次方的 1）**

Buckets do not overlap in timestamps （不重叠）

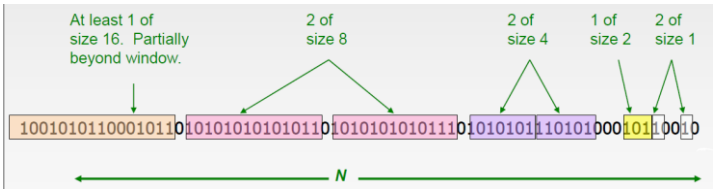
Update bucket:



2 cases: 0 →什么都不用动，1→递归更新（遇到 3 个长度相同的 bucket 就合并前两个）

如果某个 bucket 的 timestamp 大于 N，整个 bucket 被舍弃

How to query



把前面 bucket 的 size 加起来，最后的那个 bucket 加一半

比方说这个就是 1+1+2+4+4+8+8+16/2

Error bound: 50%

Error 是 误差/实际值 当误差最大，并且实际值最小的时候 error 最大。即每个 size 的 bucket 只有一个，并且最后一个 bucket 只有一点点留在 window 里，此时 error 最大。

有 r 个 bucket，此时 error 为 2^{r-1}，实际值为 $1 + 2 + 4 + .. + 2^{r-1} = 2^r - 1$, error bound 为 50%

Further Reducing the Error

Instead of maintaining **1** or **2** of each size bucket, we allow either ***r*-1** or ***r*** buckets (***r* > 2**)

Except for the largest size buckets; we can have any number between **1** and ***r*** of those

Error is at most *O(1/r)*

By picking ***r*** appropriately, we can tradeoff between number of bits we store and the error

7.47- 第三个问题：Filtering Data Streams

例如：有 1000 个好的 email address(filter)，现在有 1000000000 个 email，把这些 email 中属于好的 email address 筛选出来。

First Cut Solution (一刀切)

Given a set of keys ***S*** that we want to filter

Create a **bit array *B*** of ***n*** bits, initially all ***0*s**

Choose a **hash function *h*** with range **[*0,n*)**

Hash each member of $s \in S$ to one of ***n*** buckets, and set that bit to **1**, i.e., ***B[h(s)]=1***

Hash each element ***a*** of the stream and output only those that hash to bit that was set to **1**

Output *a* if B[h(a)] == 1

步骤：1.所有 B 中值初始化为 0

2.求所有 filter 中元素的 hash 值

3. 把 B 中这些 hash 值对应的值变为 1

4. 求目标的 hash 值

B 中该 hash 值对应的值不为 1 的话，就肯定不是我们要的。（实际上等于 1 的话也不一定是我们想要的）

False negative：满足条件的，被丢弃了 （这个算法里不会产生 false negative）

False positive：不满足条件的，给我们了 （这个算法里会产生 false positive）

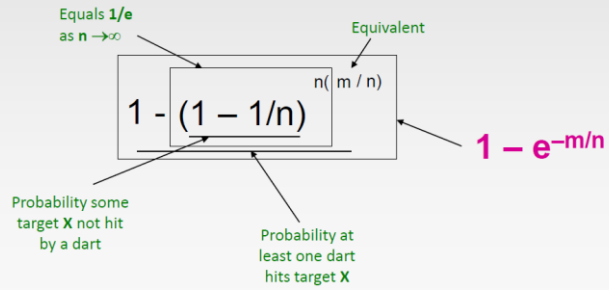
以下假设：

|S| = 1 billion email addresses
|B|= 1GB = 8 billion bits

实际上在这个 hash table 里 filter 所占的比例不到 1/8，因为有的会重复，那究竟是多少呢？

We have ***m*** darts, ***n*** targets

What is the probability that a target gets at least one dart?



Fraction of 1s in the array B
= probability of false positive = 1 – $e^{-m/n}$

Example: 10^9 darts, $8 \cdot 10^9$ targets
Fraction of **1s** in **B** = 1 – $e^{-1/8}$ = **0.1175**
Compare with our earlier estimate: **1/8 = 0.125**

这个概率有两层涵义：

有多少 target 被 dart 占据了

有多少比率的 false positive

Bloom filter （多刀切）

Consider: **|S| = *m*, |B| = *n***

Use ***k*** independent hash functions $h_1, \ldots, h_k$

Initialization:

Set **B** to all **0s**

Hash each element $s \in S$ using each hash function h_i, set $B[h_i(s)] = 1$ (for each $i = 1, .., k$)

Run-time:

When a stream element with key ***x*** arrives

- If $B[h_i(x)] = 1$ for all $i = 1, \ldots, k$ then declare that ***x*** is in ***S***
  - That is, ***x*** hashes to a bucket set to **1** for every hash function $h_i(x)$
- Otherwise discard the element ***x***

多个 hash function

Consider a Bloom filter of size m=10 and number of hash functions k=3. Let H(x) denote the result of the three hash functions.

The 10-bit array is initialized as below

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Insert x_0 with $H(x_0) = \{1, 4, 9\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	1

Insert x_1 with $H(x_1) = \{4, 5, 8\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

Query y_0 with $H(y_0) = \{0, 4, 8\}$ => ???

Query y_1 with $H(y_1) = \{1, 5, 8\}$ => ??? **False positive!**

这个例子中 filter 有 2 个元素，3 个 hash function，即每个元素理论上占有 10 个位置中的三个位置

What fraction of the bit vector B are 1s?

Throwing $k \cdot m$ darts at ***n*** targets

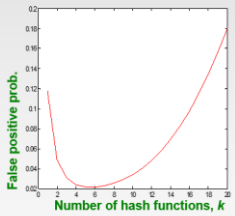
So fraction of **1**s is $(1 - e^{-km/n})$

But we have ***k*** independent hash functions and we only let the element ***x*** through **if all *k*** hash element ***x*** to a bucket of value **1**

So, false **positive probability** = $(1 - e^{-km/n})^k$

这些地方被占据

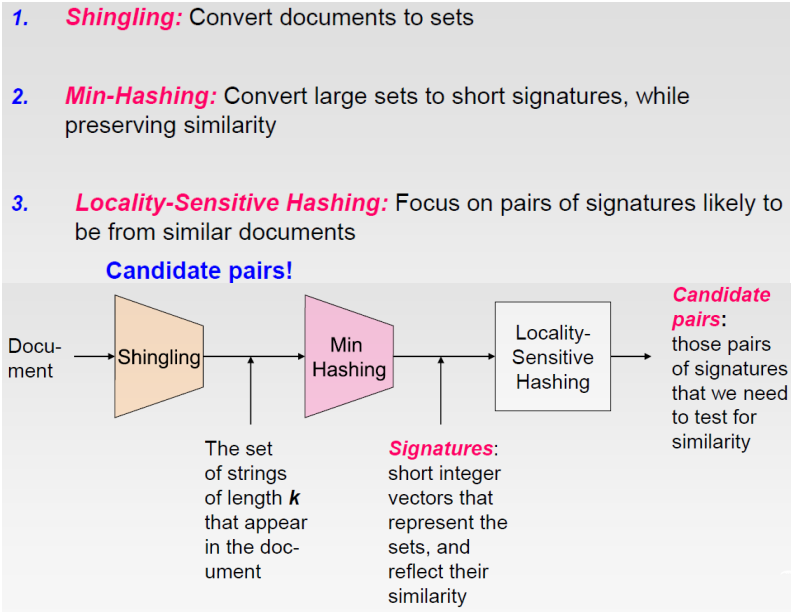
- $m = 1$ billion, $n = 8$ billion
  - $k = 1$: $(1 - e^{-1/8}) = 0.1175$
  - $k = 2$: $(1 - e^{-1/4})^2 = 0.0493$
- What happens as we keep increasing *k*?



- "Optimal" value of *k*: $n/m \ln(2)$
  - In our case: Optimal $k = 8 \ln(2) = 5.54 \approx 6$
    - Error at $k = 6$: $(1 - e^{-1/6})^2 = 0.0235$

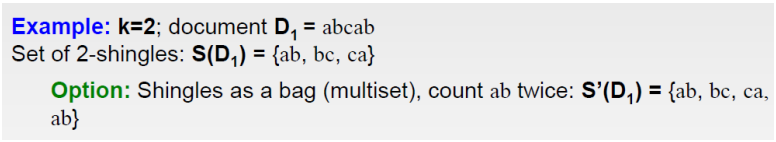
9.1-9.7 背景介绍，背景就是我们要准备一套算法，这个算法就是在给定 corpus 里，找到相似度大于某个阈值的 pair document，相似度用 Jacarrd similarity

9.8-9.9 三个步骤



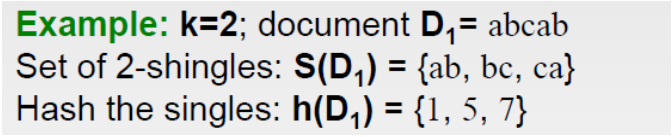
9.9-9.17 shingling

k-shingle: 逐次提取 k 个元素，放到 set 里。这个元素可以是 characters，words，具体是什么看要求！！！



不提额外要求的话，用上面那种，set 里面不能有重复元素。

相似的文档会有很多相同的 shingle



在计算相似度的时候，我们不会用原始的字符串比较，而是把他们 hash 成整数值，来替代 shingle 字符串，提高效率

于是每个 shingle 对应一个数值，所以每个 document 可以表示成每个维度都是 0/1 的 vector，1 代表 document 包含该维度所表示的 shingle。这个 vector 是相当稀疏的…

9.18-9.40

根据上面的转换，每个 document 可以表示成：

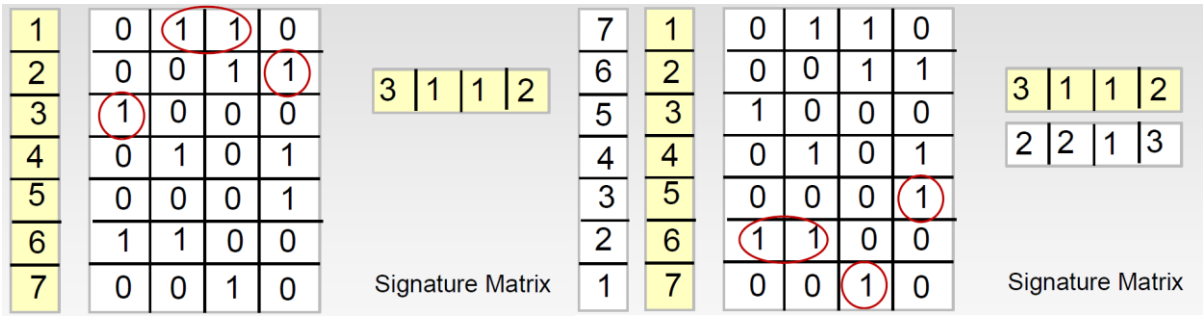
Example: S_1 = {a, d}, S_2 = {c}, S_3 = {b, d, e}, and S_4 = {a, c, d}

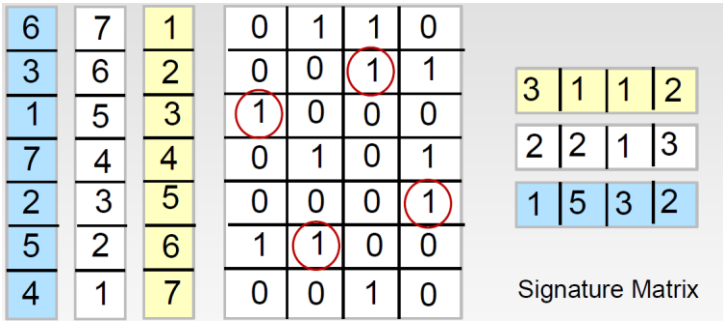
Element	S_1	S_2	S_3	S_4
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0

这里面 a, b, c, d, e…是 shingle，s1, s2, s3, …是 document

有一种方法，能够使这个很长的 vector 转化成一个很短的 vector (signature)，并且保持相似性。-- min-hashing

做法：取最小的 1





为啥这种方式就对呢？证明：

Given cols C_1 and C_2, rows may be classified as:

	C_1	C_2
A	1	1
B	1	0
C	0	1
D	0	0

a = # rows of type A, etc.

Note: sim(C_1, C_2) = a/(a +b +c)

Then: Pr[$h(C_1) = h(C_2)$] = *Sim*(C_1, C_2)

Look down the cols C_1 and C_2 until we see a 1

If it's a type-*A* row, then $h(C_1) = h(C_2)$

If a type-*B* or type-*C* row, then not

针对 C1 和 C2 来讲

a 代表形如 signature matrix 中(1,1)的个数 …

根据 Jaccard similarity sim(C1,C2) = a / (a+b+c)

而在随机排列中，(1, 1)这种形式的 pair 战胜(0,1)和(1,0)，排在他们前面的概率是 a/ (a+b+c) (0,0) 对结果是没有影响的

即 Pr[h(C1) = h(C2)] 也恰好是 a/ (a+b+c)

所以 sim(C1,C2) = Pr[h(C1) = h(C2)]

用 permutation（洗牌）的方式会消耗大量计算资源，可以使用 hash 的方法替代

Row	S_1	S_2	S_3	S_4	$x+1 \mod 5$	$3x+1 \mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

0. Initialize all ***sig(C)[i]*** = ∞

	S_1	S_2	S_3	S_4
h_1	∞	∞	∞	∞
h_2	∞	∞	∞	∞

Row 0: we see that the values of $h_1(0)$ and $h_2(0)$ are both 1, thus $sig(S_1)[0] = 1$, $sig(S_1)[1] = 1$, $sig(S_4)[0] = 1$, $sig(S_4)[1] = 1$,

	S_1	S_2	S_3	S_4
h_1	1	∞	∞	1
h_2	1	∞	∞	1

Row 1, we see $h_1(1) = 2$ and $h_2(1) = 4$, thus $sig(S_3)[0] = 2$, $sig(S_3)[1] = 4$

	S_1	S_2	S_3	S_4
h_1	1	∞	2	1
h_2	1	∞	4	1

Row 2: $h_1(2) = 3$ and $h_2(2) = 2$, thus $sig(S_2)[0] = 3$, $sig(S_2)[1] = 2$, no update for S_4

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	1	2	4	1

Row 3: $h_1(2) = 4$ and $h_2(2) = 0$, update $sig(S_1)[1] = 0$, $sig(S_3)[1] = 0$, $sig(S_4)[1] = 0$,

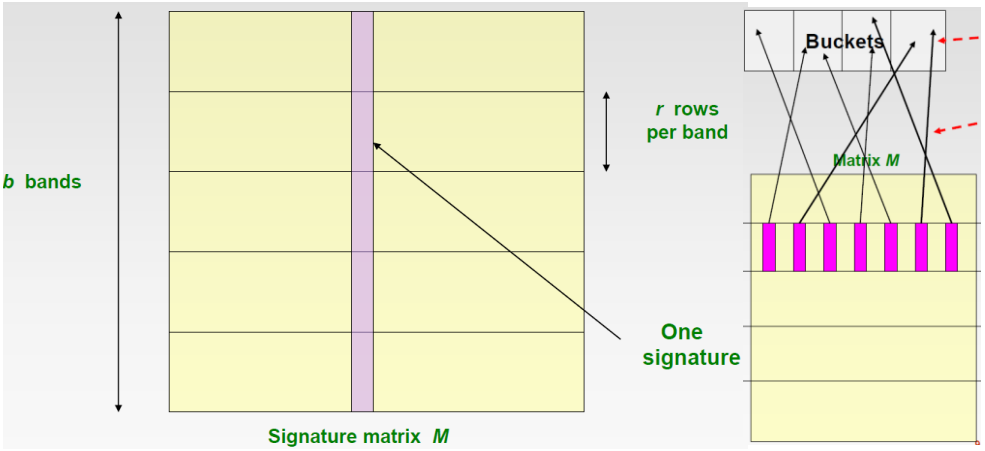
	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	0	2	0	0

Row 4: $h_1(2) = 0$ and $h_2(2) = 3$, update $sig(S_3)[0] = 0$,

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

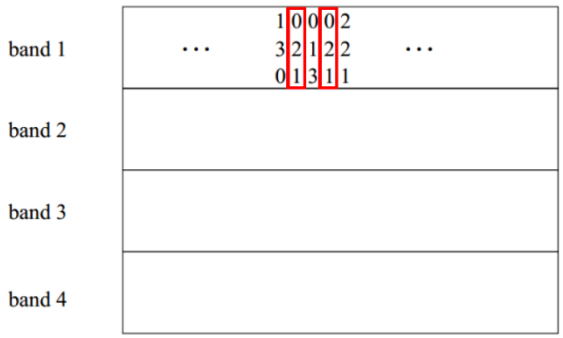
9.41 - LSH： 目标是通过 signature matrix 找到 candidate pair

做法：把 signature matrix M 分成 b 个 bands，每个 bands 有 r 个 rows，然后找出至少有一个 band 完全相同 pair，即为 candidate pair



如何找到完全相同的 pair 呢？ Hash

这里面有个简化模型的假设：完全相同的才能被 hash 到同一个 bucket 里



比如这种情况，在 band1 中，这两个已经完全相同了，则被认为是 candidate pair。如果在 band1 里不完全相同，还有其他三次机会…（在 band2 band3 band4 里有可能相同）

接下来证明为什么这种方法是合理的：

Assume the following case:

Suppose 100,000 columns of ***M*** (100k docs)

Signatures of 100 integers (rows)

Therefore, signatures take 40Mb

Choose ***b*** = 20 bands of ***r*** = 5 integers/band

Goal: Find pairs of documents that are at least ***s*** = *0.8* similar

这些是前提

Find pairs of ≥ *s*=0.8 similarity, set b=20, r=5

Assume: $sim(C_1, C_2) = 0.8$

Since $sim(C_1, C_2) \geq$ **s**, we want C_1, C_2 to be a **candidate pair**: We want them to hash to at **least 1 common bucket** (at least one band is identical)

Probability C_1, C_2 identical in one particular band: $(0.8)^5 = 0.328$

Probability C_1, C_2 are ***not*** similar in all of the 20 bands: $(1-0.328)^{20} = 0.00035$

i.e., about 1/3000th of the 80%-similar column pairs are **false negatives** (we miss them)

We would find 99.965% pairs of truly similar documents

当 C1，C2 的 similarity 本身是 0.8 的时候，是我们想要的 pair，它被丢弃的概率是 0.00035

Find pairs of ≥ *s*=0.8 similarity, set b=20, r=5

Assume: $sim(C_1, C_2) = 0.3$

Since $sim(C_1, C_2) <$ **s** we want C_1, C_2 to hash to **NO common buckets** (all bands should be different)

Probability C_1, C_2 identical in one particular band: $(0.3)^5 = 0.00243$

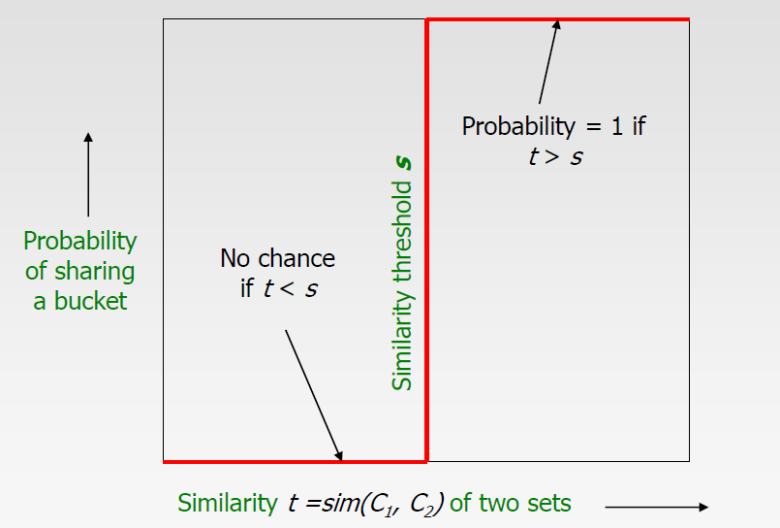
Probability C_1, C_2 identical in at least 1 of 20 bands: $1 - (1 - 0.00243)^{20} = 0.0474$

In other words, approximately 4.74% pairs of docs with similarity 0.3% end up becoming **candidate pairs**

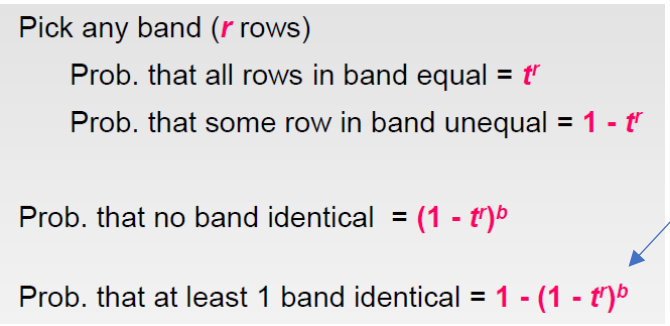
- They are **false positives** since we will have to examine them (they are candidate pairs) but then it will turn out their similarity is below threshold **s**

当 C1，C2 的 similarity 本身是 0.3 的时候，不是我们想要的 pair，它被收录为 candidate pair 的概率是 0.0474

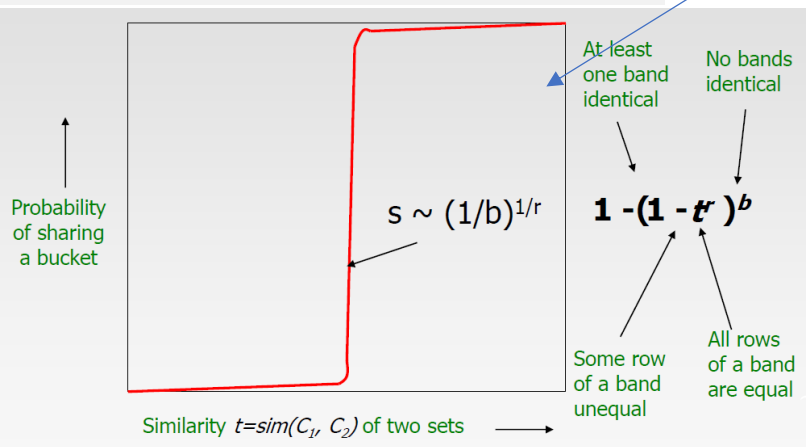
Analysis of LSH



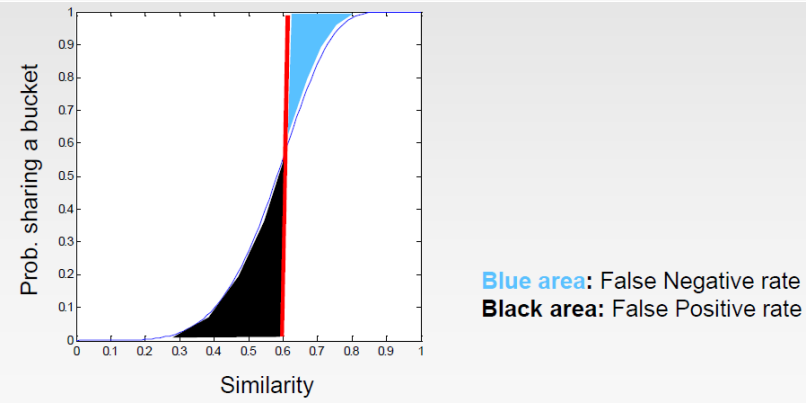
这个是理想状态我们想要的结果，小于 threshold 都 share bucket，大于 threshold 都 share bucket



然而现实情况是：share bucket 的概率是这样的　　图像是这样的



经过数学计算，当 s 等于这个数值的时候，最接近我们想要的结果。所以当选取 b 和 r 的时候，要遵循这个公式。



Suppose we wish to find similar sets, and we do so by minhashing the sets 10 times and then applying locality-sensitive hashing using 5 bands of 2 rows (minhash values) each. If two sets had Jaccard similarity 0.6, what is the probability that they will be identified in the locality-sensitive hashing as candidates (i.e. they hash at least once to the same bucket)? You may assume that there are no coincidences, where two unequal values hash to the same bucket. A correct expression is sufficient: you need not give the actual number.

Answer: $1-(1-0.6^2)^5=0.893$

11.1-11.8 介绍 recommender system 没啥用…

11.9 基本模型

X = set of Customers

S = set of Items

Utility function u: ***X*** × ***S*** → ***R***

R = set of ratings

R is a totally ordered set

e.g., 0-5 stars, real number in [0,1]

Utility Matrix

	Avatar	LOTR	Matrix	Pirates
Alice	1		0.2	
Bob		0.5		0.3
Carol	0.2		1	
David				0.4

这是一个基本模型，建立一个 matrix，行和列分别是 item 和 user，matrix 的内容是 rate

11.10 problem

11.11 如何收集数据

Explicit

Ask people to rate items

Doesn't work well in practice – people can't be bothered

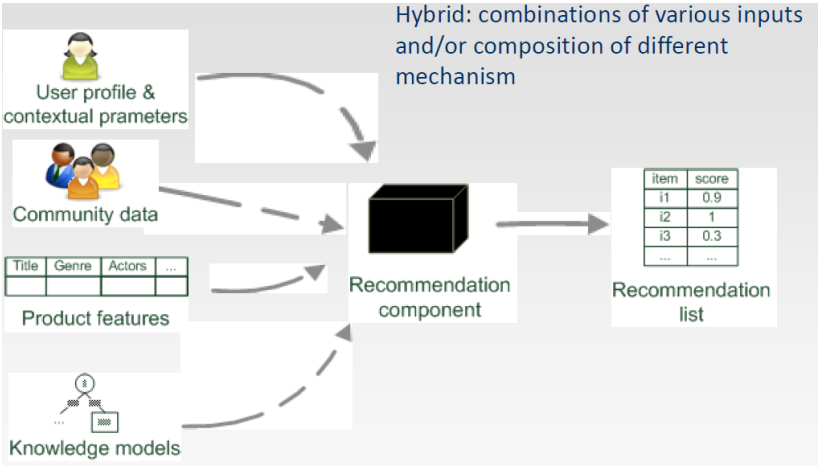
Implicit

Learn ratings from user actions

- E.g., purchase implies high rating

直接向用户要，或者根据用户的行为去推断

11.12-11.17 推荐系统的原理和分类

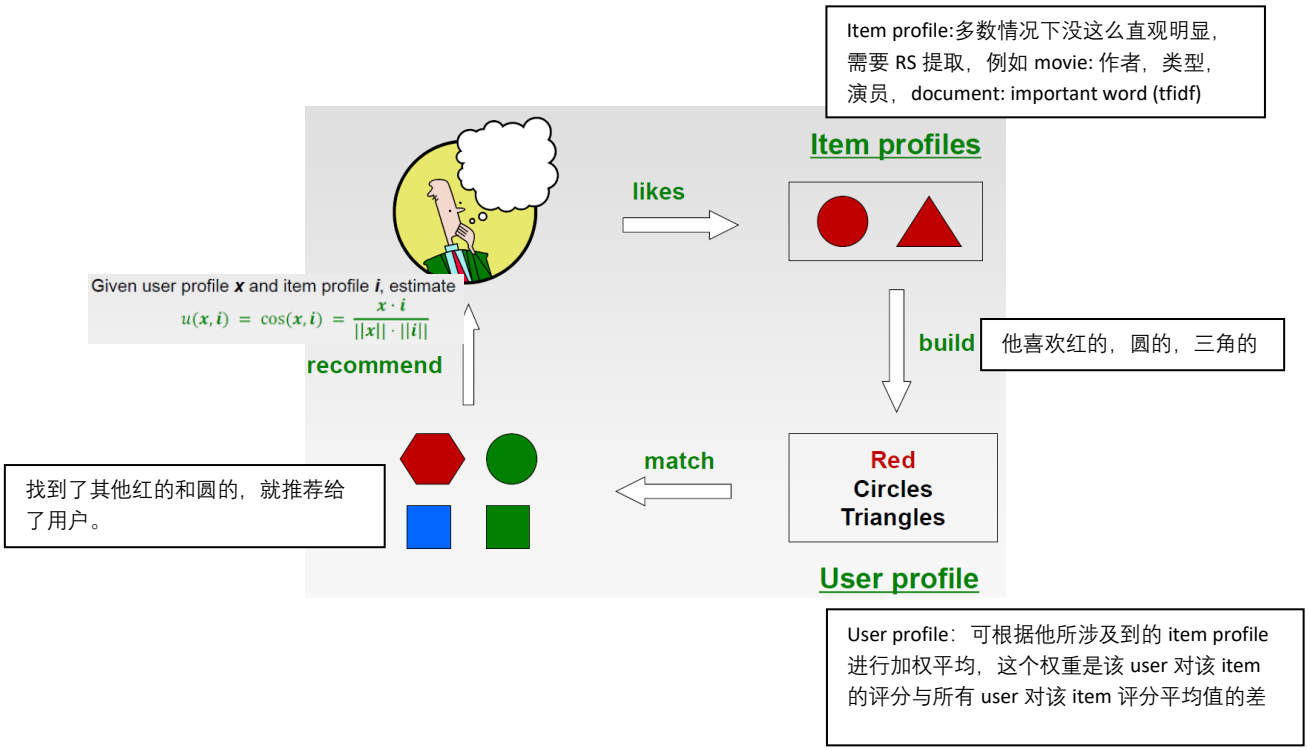


Collaborative：根据我"朋友"的 rate 的情况，来给我推荐，比方说我朋友都喜欢看《色戒》，那我也很可能喜欢看… 这里的朋友指的是志趣相同的人。推荐系统可以根据相似度计算找到这些人。

Content-based：只根据我以往看电影的情况给我推荐，比方说历史记录里都是岛国动作片，推荐系统很可能推荐苍老师的作品。

Knowledge model 不考

11.18-11.26



+: No need for data on other users

+: Able to recommend to users with unique tastes

+: Able to recommend new & unpopular items

No first-rater problem

+: Able to provide explanations

Can provide explanations of recommended items by listing content-features that caused an item to be recommended

优点：不需要其他 user 的信息

用户口味不变的话，可以持续推荐

只要相似度高就会被推荐，不管是什么样的 item

提供解释

–: Finding the appropriate features is hard

E.g., images, movies, music

–: Recommendations for new users

How to build a user profile?

–: Overspecialization

Never recommends items outside user's content profile

People might have multiple interests

Unable to exploit quality judgments of other users

缺点：feature 不好找

新的 user 什么还什么都没做，无法知道他喜欢什么

永远都在同一个领域找 item

人是会变的…

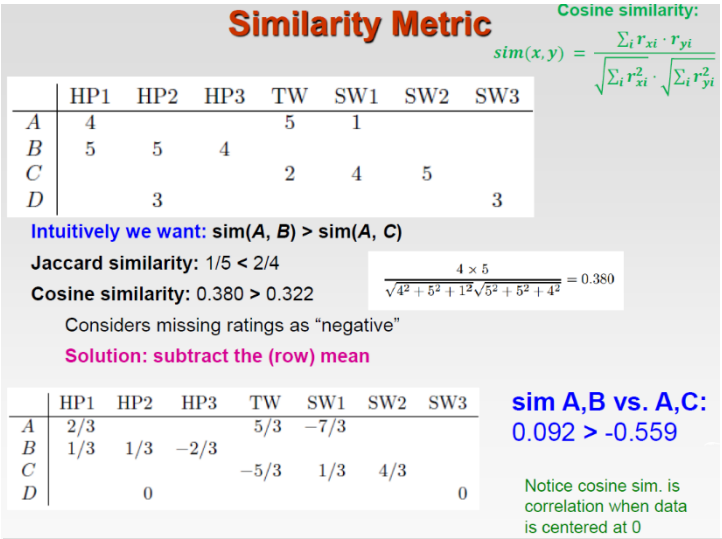
无法利用“朋友”们的力量

11.27-11. collaborative filtering 推测空缺位置的值

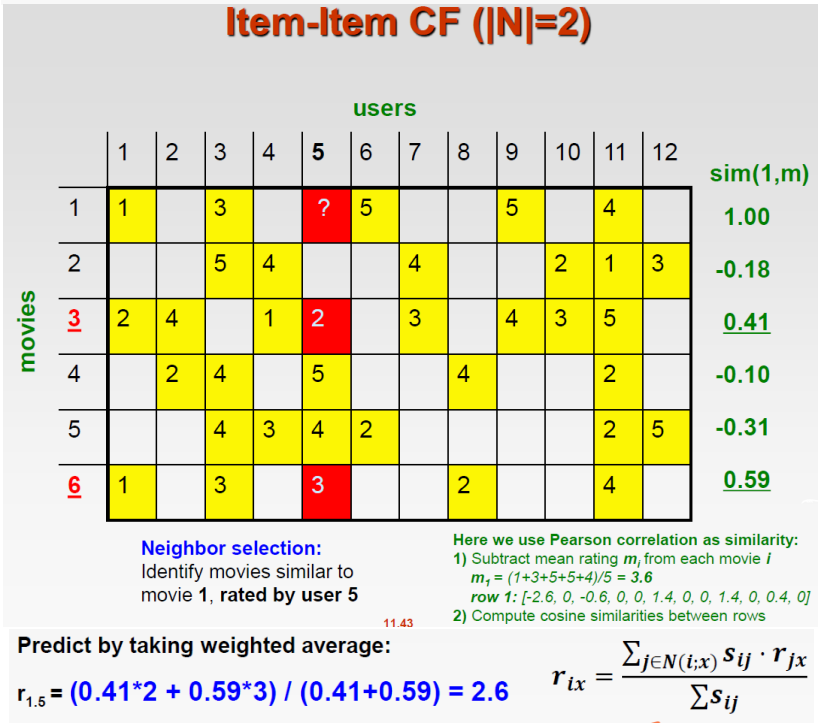
可分成 user-user 和 item-item 两种，他们的流程一样，都分成两步：

Step1：找“朋友”，即找相似度最大的若干个“朋友”，user 找相似度最大的若干个 user，item 找相似度最大的若干个 item

Step2：根据朋友推测某空缺位置的值



Step1 找相似，有很多种相似度的方法，比如 Jaccard Similarity，cosine similarity，Pearson correlation，ppt 中的结论是 Pearson correlation 的方法最好，但实际上考试里和练习题里用的是 cosine similarity



注意：一定别忘了除以总的 similarity

In practice, it has been observed that item-item often works better than user-user

Consider three users u_1, u_2, and u_3, and four movies m_1, m_2, m_3, and m_4. The users rated the movies using a 4-point scale: -1: bad, 1: fair, 2: good, and 3: great. A rating of 0 means that the user did not rate the movie. The three users' ratings for the four movies are: u_1 = (3, 0, 0, -1), u_2 = (2, -1, 0, 3), u_3 = (3, 0, 3, 1)

Which user has more similar taste to u_1 based on cosine similarity, u_2 or u_3? Show detailed calculation process.

Answer: sim(u1, u2) = (3*2 -1*3)/(sqrt(10)*sqrt(14)) ≈ 0.2535, sim(u1, u3) = (3*3-1*1)/(sqrt(10)*sqrt(19)) ≈ 0.5804. Thus u3 is more similar to u1.

User u_1 has not yet watched movies m_2 and m_3. Which movie(s) are you going to recommend to user u_1, based on the user-based collaborative filtering approach? Justify your answer.

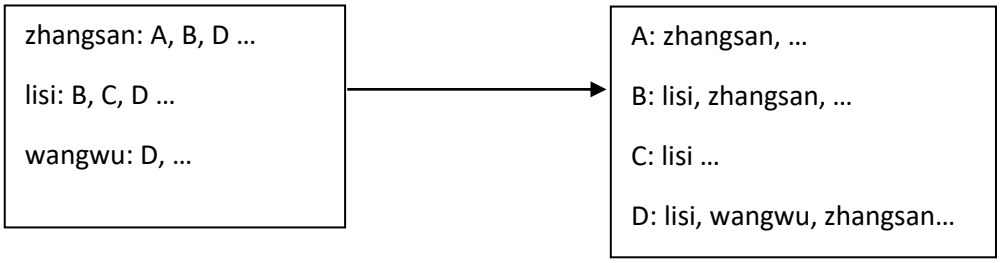
Answer: You can use either cosine similarity or Pearson correlation coefficient to compute the similarities between users. However, the conclusion should be that only m3 is recommended to u1.

疑似去年的理论题...

Memory-based and Model-based Approaches

- User-based CF is said to be "memory-based"
  - the rating matrix is directly used to find neighbors / make predictions
  - does not scale for most real-world scenarios
  - large e-commerce sites have tens of millions of customers and millions of items
- Model-based approaches
  - based on an offline pre-processing or "model-learning" phase
  - at run-time, only the learned model is used to make predictions
  - models are updated / re-trained periodically
  - large variety of techniques used
  - model-building and updating can be computationally expensive

Problem1.1



Mapper → <A, zhangsan>, <B, zhangsan>, <D, zhangsan>

<B, lisi>, <C, lisi>, <D, lisi>

<D, wangwu>

Reducer → <A, (zhangsan)>

<B, (zhangsan, lisi)>

<C, (lisi)>

<D, (zhangsan, lisi, wangwu)>　　这不是我们想要的，value 并没有排序

解决方法，用 secondary sort：

Mapper → <<A, zhangsan>, zhangsan>, <<B, zhangsan>, zhangsan>, <<D, zhangsan>, zhangsan>

<<B, lisi>, lisi>, <<C, lisi>, lisi>, <<D, lisi>, lisi>

<<D, wangwu>, wangwu> →　这地方用到 WritableComparator

Shuffle and sort →<<A, zhangsan>, zhangsan>　→ <<A, zhangsan>, (zhangsan)>

<<B, lisi>, lisi>　　<<B, lisi>, (lisi, zhangsan)>

<<B, zhangsan>, zhangsan>　　<<C, lisi>, (lisi)>

<<C, lisi>, lisi>　　<<D, lisi>, (lisi, wangwu, zhangsan)>

<<D, lisi>, lisi>

<<D, wangwu>, wangwu>

<<D, zhangsan>, zhangsan>

这地方能按要求排好序是因为我们写了 pair 里的 comparTo 方法

所有复合 Key 中，第一个元素相同的被分到同一个 partitioner 进行计算，是因为写了 partitioner

→ Reducer → <A, (zhangsan)>

<B, (lisi, zhangsan)>

<C, (lisi)>

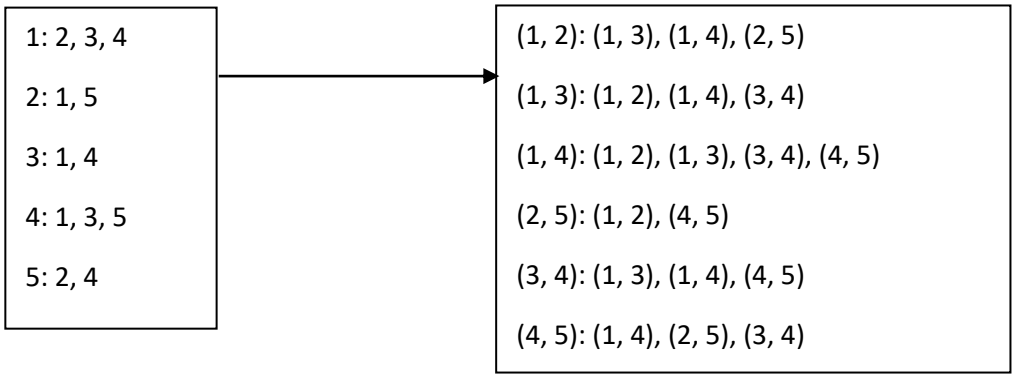
<D, (lisi, wangwu, zhangsan)>

Problem2.2

Given a large text file, your task is to find out the top-k most frequent co-occurring term pairs. The co-occurrence of (w, u) is defined as: u and w appear in the same line (this also means that (w, u) and (u, w) are treated equally). Your Spark program should generate a list of *k* key-value pairs ranked in descending order according to the frequencies, where the keys are the pair of terms and the values are the co-occurring frequencies (Hint: you need to define a function which takes an array of terms as input and generate all possible pairs).

```
def pairGen(wordArray: Array[String]) : ArrayBuffer[(String, Int)] = {
    val abuf = new ArrayBuffer[(String, Int)]
    for(i <- 0 to wordArray.length -1){
      val term1 = wordArray(i) //从Array中获得一个term1
      if(term1.length()>0){
        for(j <- i+1 to wordArray.length - 1){
          val term2 = wordArray(j) // 从term1的后面再获得一个term2
          if(term2.length()>0){
            if(term1 < term2){abuf.+=:(term1 + "," + term2, 1)}
            else {abuf.+=:(term2 + "," + term1, 1)}
          }              //把term1和term2合并到一个字符串里
        }                //题目中说(term1,term2)与(term2,term1)是相同的
      }                  //所以要把他们调整成相同的
    }
    return abuf
  }
val textFile = sc.textFile(inputFile)    //读取文件，形成RDD，每一行的形式为String
val words = textFile.map(_.split(" ").toLowerCase)  //每一行的形式为Array("term1", "term2", "term3", ...)
val pairs = words.flatMap(x => pairGen(x)).reduceByKey(_+_)  //把这个Array输入到pairGen函数中
val topk = pairs.map(_.swap).sortByKey(false).take(k).map(_.swap) //要按照第二项排序，先把第二项与第一项对换
                                                                  //降序排完之后，再换回来
topk.foreach(x => println(x._1, x._2))
```

Assignment problem1



Mapper→ (1, 2), (1, 3), (1, 4) → (<(1, 2), (1, 3)>, (1, 3)), (<(1, 2), (1, 4)>, (1, 4)), (<(1, 3), (1, 2)>, (1, 2))
(<(1, 3), (1, 4)>, (1, 4)), (<(1, 4), (1, 2)>, (1, 2)), (<(1, 4), (1, 3)>, (1, 3))
(1, 2), (2, 5) (<(1, 2), (2, 5)>, (2, 5)), (<(2, 5), (1, 2)>, (1, 2))
(1, 3), (3, 4) (<(1, 3), (3, 4)>, (3, 4)), (<(3, 4), (1, 3)>, (1, 3))
(1, 4), (3, 4), (4, 5) (<(1, 4), (3, 4)>, (3, 4)), (<(1, 4), (4, 5)>, (4, 5)), (<(3, 4), (1, 4)>, (1, 4))
(<(3, 4), (4, 5)>, (4, 5)), (<(4, 5), (1, 4)>, (1, 4)), (<(4, 5), (3, 4)>, (3, 4))
(2, 5), (4, 5) (<(2, 5), (4, 5)>, (4, 5)), (<(4, 5), (2, 5)>, (2, 5))

→ shuffle and sort

(<(1, 2), (1, 3)>, (1, 3)), (<(1, 2), (1, 4)>, (1, 4)), (<(1, 2), (2, 5)>, (2, 5)),
(<(1, 3), (1, 2)>, (1, 2)), (<(1, 3), (1, 4)>, (1, 4)), (<(1, 3), (3, 4)>, (3, 4)),
(<(1, 4), (1, 2)>, (1, 2)), (<(1, 4), (1, 3)>, (1, 3)), (<(1, 4), (3, 4)>, (3, 4)), (<(1, 4), (4, 5)>, (4, 5)),
(<(2, 5), (1, 2)>, (1, 2)), (<(2, 5), (4, 5)>, (4, 5)),
(<(3, 4), (1, 3)>, (1, 3)), (<(3, 4), (1, 4)>, (1, 4)), (<(3, 4), (4, 5)>, (4, 5)),
(<(4, 5), (1, 4)>, (1, 4)), (<(4, 5), (2, 5)>, (2, 5)), (<(4, 5), (3, 4)>, (3, 4))

→ (<(1, 2), (1, 3)>, List((1, 3), (1, 4), (2, 5)))
(<(1, 3), (1, 2)>, List((1, 2), (1, 4), (3, 4)))
(<(1, 4), (1, 2)>, List((1, 2), (1, 3), (3, 4), (4, 5)))
(<(2, 5), (1, 2)>, List((1, 2), (4, 5)))
(<(3, 4), (1, 3)>, List((1, 3), (1, 4), (4, 5)))
(<(3, 4), (1, 3)>, List((1, 4), (2, 5), (3, 4)))

→reducer (1, 2): (1, 3), (1, 4), (2, 5)
(1, 3): (1, 2), (1, 4), (3, 4)
(1, 4): (1, 2), (1, 3), (3, 4), (4, 5)
(2, 5): (1, 2), (4, 5)
(3, 4): (1, 3), (1, 4), (4, 5)
(4, 5): (1, 4), (2, 5), (3, 4)