map (k_1, v_1) → list [<k_2, v_2>]
Transforms input into key-value pairs to process
reduce (k_2, [v_2]) → [<k_3, v_3>]
Aggregates the list of values for each key
Values with same key are sent to same reducer
combine (k_2, [v_2]) → [<k_3, v_3>]
Reducers that run in memory after map phase
Used as an optimisation to reduce network traffic
Same output key-value types as Mapper
Works if reduce() is commutative and associative
Generally, not interchangeable with reducer
partition (k_2, num of partitions) → partition for k_2
Often a hash of the key, e.g., hash(k_2) mod n
Divides up key space for parallel reduce ops
Num partitions == num reduce tasks
Grouping comparator: controls which keys are grouped together for a single call to `reduce()`

In-mapper combining (e.g word count)
Scalability issue (not suitable for huge data)
Pairs ({a, b} -> 1, {a, d} -> 5) Easy to implement
Creates lots of pairs, combiner aren't effective
Stripes (a -> {b:1, d:5}) Less sorting
May suffer from memory problem
Order inversion(using pairs)No memory problem
Special key value pairs to capture the marginal
Control sort order so the special pair comes first
(dog, *) [6327,8514…] 42908
(dog, aardvark) [1,2] f(aardvark|dog)=3/42908
Value-to-key conversion 把要sort的放在key里

Resilient: fault-tolerant, can recompute missing or damaged partitions due to node failures.
Distributed: data reside on N nodes in a cluster
Dataset: a collection of partitioned elements

```
Transformations
map(f: T->U): RDD[T]->RDD[U]
flatMap(f: T->Seq[U]): RDD[T]->RDD[U]
mapValues(f: V->W): RDD[(K,V)]->RDD[(K,W)]
filter(f: T->Bool): RDD[T]->RDD[T]
reduceByKey(f: (V,V)->V): RDD[(K,V)]-> RDD[(K,V)]
groupByKey(): RDD[(K,V)]->RDD[(K,Seq[V])]
countByKey(): RDD[(K,V)]->Map[T,Long]
sort(c: Comparator[K]): RDD[(K,V)]->RDD[(K,V)]
sortBy(f: T->K, asc: Bool): RDD[T]->RDD[T]
sample(frac: Float): RDD[T]->RDD[T]
union(): (RDD[T],RDD[T])->RDD[T]
join(): (RDD[(K,V)],RDD[(K,W)])->RDD[(K,(V,W)]
crossProduct(): (RDD[T],RDD[U])->RDD[(T,U)]
partitionBy(p: Parter[K]): RDD[(K,V)]->RDD[(K,V)]
Actions
count(): RDD[T]->Long
collect(): RDD[T]->Seq[T]
reduce(f: (T,T)->T): RDD[T]->T
lookup(k: K): RDD[(K,V)]->Seq[V]
save(path: String)
```

```
val output = doc.map(line => {
  val cols = line.split(",")
  (cols(2).toInt, cols(1).toInt)
})
  .distinct
  .countByKey
  .toSeq
  .sortWith(_._2 >= _._2)
  .take(5)
```

A *k*-**shingle** (or *k*-gram) for a document is a sequence of *k* tokens that appears in the doc
Tokens can be characters, words or something else
e.g k=2 D=abcab set of 2-shingles S(D)={ab, bc, ca}
Documents that are intuitively similar will have many shingles in common.
Minhashing: Convert large sets to short signatures, while preserving similarity
LSH:
1. Divide matrix **M** into **b** bands of **r** rows.
2. For each band, hash its portion of col to **k** buckets
3. **Candidate** col pairs are those who hash to the **same bucket(i.e col portions are identical)** for **>= 1** band
4. Tune **b** and **r** to catch most similar pairs, but few non-similar pairs

```
// Cols are documents, items in cols are shingles
Row S1 S2 S3 S4 (x+1)%5 (3x+1)%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
R0  S1 S2 S3 S4  R2  S1 S2 S3 S4  R4  S1 S2 S3 S4
h1   1  #  #  1  h1   1  3  2  1  h1   1  3  0  1
h2   1  #  #  1  h2   1  2  4  1  h2   0  2  0  0
R1  S1 S2 S3 S4  R3  S1 S2 S3 S4  SigSim(S1, S4)=1
h1   1  #  2  1  h1   1  3  2  1  Jaccard
h2   1  #  4  1  h2   0  2  0  0  Sim(S1, S4)=2/3
```

Top-k
```
val words = textFile.flatMap(_.toLowerCase().split("[\\s*$&#/\"'\\,.:;?!\\[\\](){}<>~\\-_]+").distinct)
val pairs = words.filter(_.length>0).filter(x => x.charAt(0) <='z' && x.charAt(0)>='a').map(x=>(x,1)).reduceByKey(_+_)
val topk = pairs.sortBy(_._1).sortBy(_._2, false).take(k).map(x => x._1 + '\t' + x._2)
```

Reverse graph edge direction
```
val pairs = sc.textFile(graphFile).map(x=>(x.split('\t')(0),x.split('\t')(1)))
val res = pairs.map(_.swap).groupByKey().sortBy(_._1.toInt).mapValues(x=>x.toSeq.sortWith(_.toInt<_.toInt))
val fmtres = res.map{case(x, y) => s"""$x\t${y.mkString(",")}"""}
fmtres.saveAsTextFile(outputFolder)
```

b=20 r=5 want sim>=0.8
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)
Assume: sim(C_1, C_2) = 0.3Since 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.
公式$1-(1-t^r)^b$

Sample a **fixed proportion**. As the stream grows the sample also gets bigger.
Stream of tuples: (user, query, time). How often did a user run the same query in a single days?
Naive sol: gen random int [0,9] for each query
What fraction of queries by an average user are duplicates? Suppose each user issues **x** queries once and **d** queries twice. Ans: **d/(x+d)**
Sample contains x/10 of the singleton queried and 2d/10 of the duplicate queries at least once
Only 1/10*1/10*d=d/100 pairs of duplicates
Of d "duplicates" 18d/100 appear exactly once
18d/100 = ((1/10*9/10)+(9/10*1/10))*d
Solution: sample users instead of queries

Reservoir sampling:

1. Store first **s** elements of stream
2. **n^th^** element arrives(**n > s**)
   - With probability **s/n**, keep the **n^th^** element, else discard it
   - If we picked **n^th^** element, it replaces one of the **s** elements, picked uniformly at random

Proof: induction
For elements already in S in step n, probability it is still in S in step n+1 is

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

$$\frac{s}{n}\frac{n}{n+1}=\frac{s}{n+1}$$

DGIM bucket properties:

1. Right end of a bucket is always 1
2. Either **one** or **two** buckets with the **same power-of-2 number** of **1s**
3. Buckets do not overlap in timestamps
4. Earlier buckets are larger or equal to later one
5. Buckets disappear when their end-time is > **N**(window size) time units in the past

Algorithm:

1. New bit arrives, drop the oldest bucket if its end-time is prior to **N** time units before current time
2. If current bit is 0, no other changes needed
3. If current bit is 1
   1) Create a new bucket of size 1 for this bit (end timestamp = current time)
   2) If there are now 3 buckets of size 1, combine the oldest two into a bucket of size 2
   3) Do step 2 recursively

Consider: **|S| = *m*, |B| = *n***
Use ***k*** independent hash functions $h_1, \dots, h_k$
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 $i = 1,\dots, k$ then declare that ***x*** is in ***S***
- That is, ***x*** hashes to a bucket set to **1** for every hash function $h_i(x)$

prob(0)=e^-(km/n)
prob(1)=1-prob(0)
prob(false positive)=p(1)^k

```
insert h1 h2 h3 0 1 2 3 4 5 6 7 8 9
    x0  1  4  9 0 1 0 0 1 0 0 0 0 1
    x1  4  5  8 0 1 0 0 1 1 0 0 1 1
 query
    y0  0  4  8 false
    y1  1  5  8 maybe true
```

Content-based Pros No community required, comparison between items possible
Cons Content descriptions necessary, cold start for new users, no surprises
Collaborative Pros No knowledge-engineering effort, serendipity of results, learns market segments
Cons Requires some form of rating feedback, cold start for new users and new items
Knowledge-based Pros Deterministic recommendations, assured quality, no cold-start, can resemble sales dialogue
Cons Knowledge engineering effort to bootstrap, basically static, does not react to short-term trends
Content-based Main idea: Recommend items to customer *x* similar to previous items rated highly by *x*
e.g. Recommend movies with same actor, genre, …
For each item, create an **item profile**. Profile is a set (vector) of features. (e.g actor, genre…)
TF-IDF(i,j)=TF(i,j)*IDF(i)
Let freq(i,j) number of occurrences of keyword i in doc j
Let maxOthers(i,j) denote the highest number of occurrences of another keyword of j
TF(i,j)=freq(i,j)/maxOthers(i,j)
N: number of all recommendable documents
n(i): number of documents in which keyword i appears
IDF(i)=log(N/n(i))
Collaborative filtering
Jaccard sim 交集个数除以并集个数
Cosine sim 点乘除以各自的模
Pearson correlation coefficient就是再cos的基础上减去row mean

$$\boldsymbol{sim(x,y)}=\frac{\sum_{s\in S_{xy}}(r_{xs}-\overline{r_x})(r_{ys}-\overline{r_y})}{\sqrt{\sum_{s\in S_{xy}}(r_{xs}-\overline{r_x})^2}\sqrt{\sum_{s\in S_{xy}}(r_{ys}-\overline{r_y})^2}}$$

```
    m1 m2 m3 m4 m5 m6 m7         (u1, u2) (u1, u3)
 u1  4        5  1        Jaccard    1/5      2/4
 u2  5  5  4               Cosine  0.380    0.322
 u3           2  4  5     Pearson  0.092   -0.559
 u4     3              3
Pearson (subtract row mean)
      m1    m2    m3    m4    m5 m6 m7
 u1  2/3             5/3 -7/3
 u2  1/3  1/3 -2/3
...
算item based就是把矩阵transpose一下
    m1 m2 m3 m4 m5 m6 m7 m8 m9 10 11 12 sim(1,m)
 u1  1     3        5        5     4        1.00
 u2        5  4        4        2  1  3    -0.18
 u3  2  4     1  2     3     4  3  5        0.41
 u4     2  4     5        4        2       -0.10
 u5        4  3  4  2              2  5    -0.31
 u6  1     3     3        2        4        0.59
predict(u1,m5)=weighted_average((u3,m5),(u6,m5))
=(0.41*2 + 0.59*3) / (0.41+0.59) = 2.6
Pearson predict只选>0, cos全都选
```

```
val puPair = textFile.filter(_(VoteTypeId) == "5").map(x => (x(PostId), x(UserId))).distinct
val res = puPair.groupByKey().filter(_._2.size>10).mapValues(x => x.toList.sortBy(_.toInt)).sortBy(_._1.toInt)
```