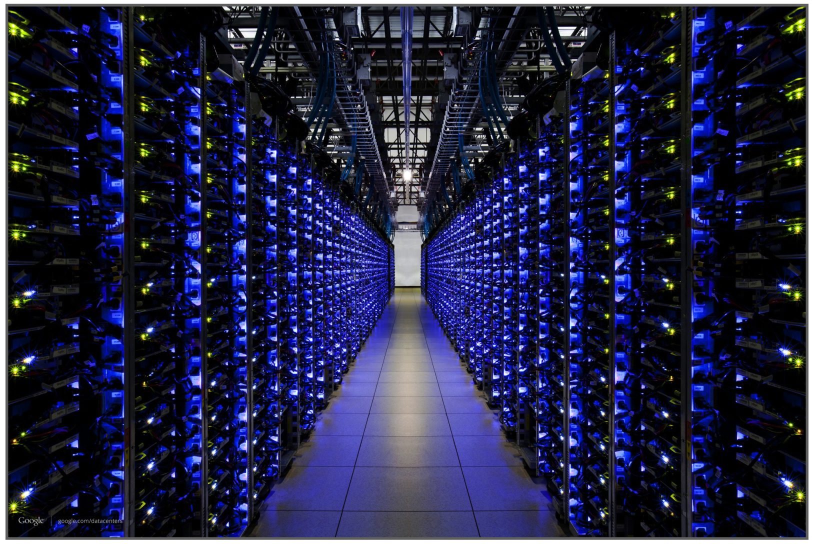
V.4 MapReduce

1. **System Architecture**
2. **Programming Model**
3. **Hadoop**

Based on **MRS Chapter 4** and **RU Chapter 2**

Why MapReduce?



- Large clusters of **commodity computers** (as opposed to few supercomputers)
- Challenges:
  - **load balancing**
  - **fault tolerance**
  - **ease of programming**

Jeff Dean

- **MapReduce**
  - system for distributed data processing
  - programming model

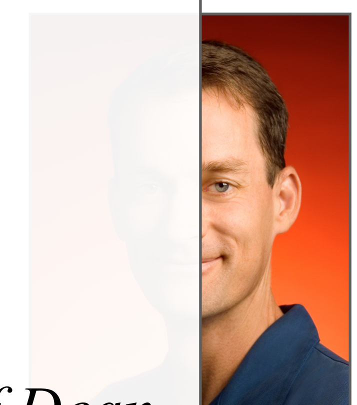
Sanjay Ghemawat

- Full details: [Ghemawat et al. '03][Dean and Ghemawat '04]

Why MapReduce?

- Large clusters of **commodity computers** (as opposed to few supercomputers)
- Challenges:
  - load balancing
  - fault tolerance
  - ease of programming

> **Jeff Dean Facts:**
>
> *When Jeff Dean designs software, he first codes the binary and then writes the source as documentation.*
>
> *Compilers don't warn Jeff Dean. Jeff Dean warns compilers.*
>
> *Jeff Dean's keyboard has two keys: 1 and 0.*
>
> *When Graham Bell invented the telephone, he saw a missed call from Jeff Dean.*
>
> Source: http://www.quora.com/Jeff-Dean/What-are-all-the-Jeff-Dean-facts



Jeff Dean

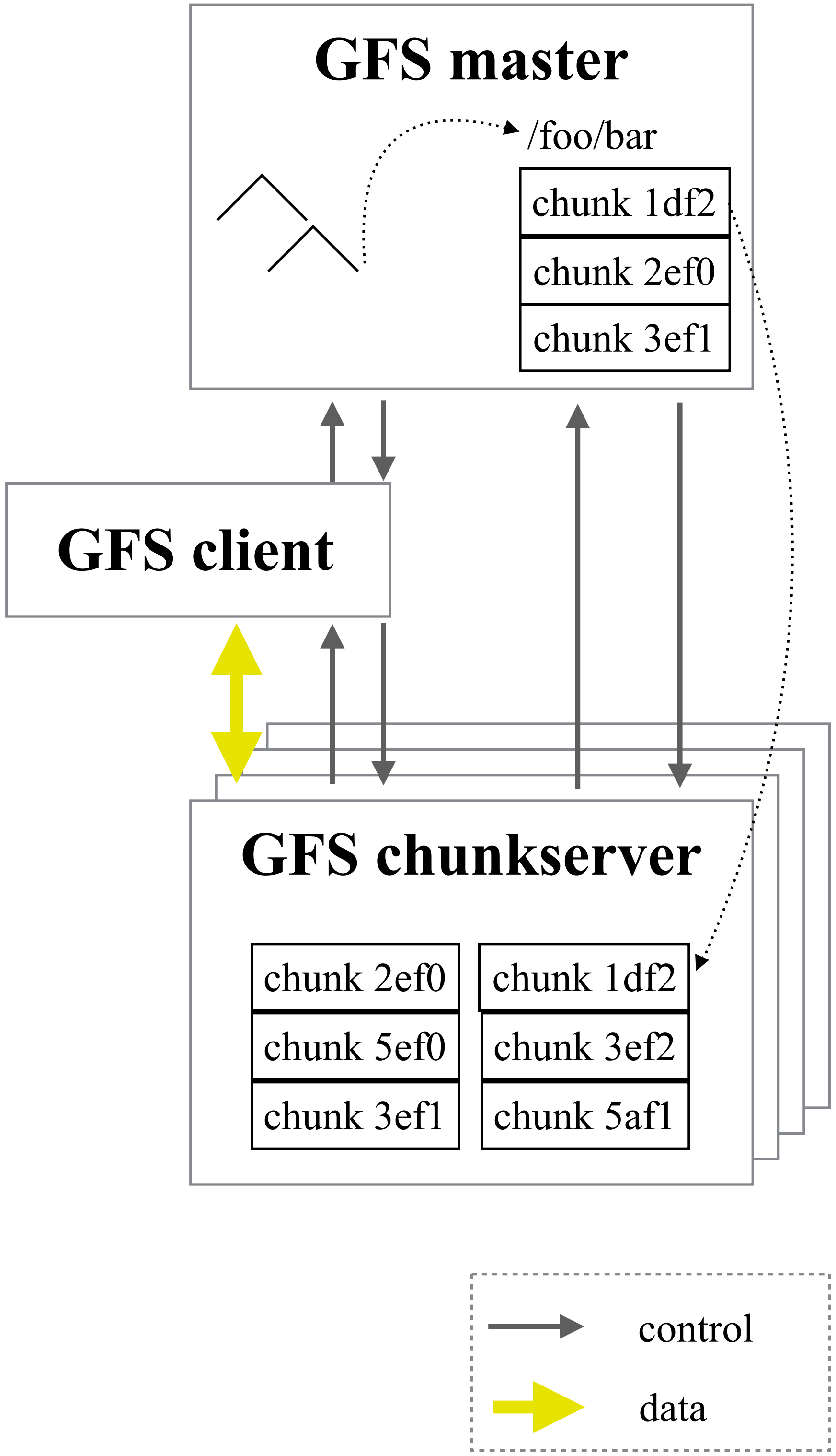
- MapReduce
  - system for distributed data processing
  - programming model

Sanjay Ghemawat

- Full details: [Ghemawat et al. '03][Dean and Ghemawat '04]

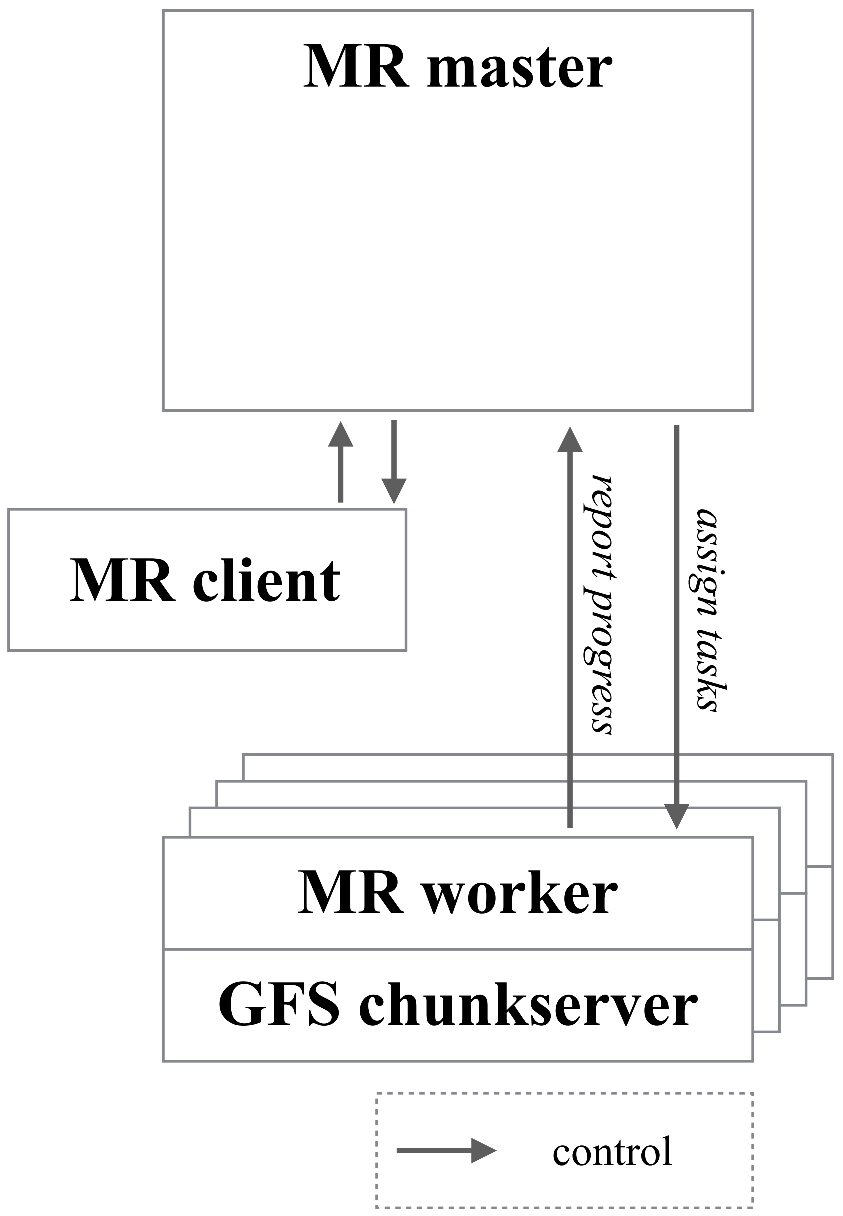
1. System Architecture

- **Google File System** (GFS)
  - distributed file system for large clusters
  - tunable replication factor
  - **single master**
    - manages namespace (/home/user/data)
    - coordinates replication of data chunks
    - first point of contact for clients
  - **many chunkservers**
    - keep data chunks (typically 64 MB)
    - send/receive data chunks to/from clients
  - Full details: [Ghemawat et al. ’03]



System Architecture (cont'd)

- **MapReduce** (MR)
  - system for distributed data processing
  - moves computation to the data for locality
  - copes with failure of workers
  - **single master**
    - coordinates execution of job
    - (re-)assigns map/reduce tasks to workers
  - **many workers**
    - execute assigned map/reduce tasks
- Full details: [Dean and Ghemawat '04]



2. Programming Model

- Inspired by **functional programming** (i.e., no side effects)
- Input/output are **key-value pairs** (k, v) (e.g., string and int)
- Users implement two functions
  - ***map***: ($k1$, $v1$) => *list*($k2$, $v2$)
  - ***reduce***: ($k2$, list($v2$)) => *list*($k3$, $v3$) with input sorted by key $k2$
- Anatomy of a MapReduce job
  - Workers execute ***map***() on their portion of the input data in GFS
  - Intermediate data from ***map***() is **partitioned and sorted**
  - Workers execute ***reduce***() on their partition and write output data to GFS
- Users may implement ***combine***() for local aggregation of intermediate data and ***compare***() to control how data is sorted

WordCount

- Problem: Count how often every word *w* occurs in the document collection (i.e., determine *cf*(*w*))

```
map(long did, string content) {
    for(string word : content.split()) {
        emit(word, 1)
    }
}
```

```
reduce(string word, list<int> counts) {
    int total = 0
    for(int count : counts) {
        total += count
    }
    emit(word, total)
}
```

Execution of WordCount

```
map(long did, string content) {
    for(string word : content.split()) {
        emit(word, 1)
    }
}
```

```
reduce(string word, list<int> counts) {
    int total = 0
    for(int count : counts) {
        total += count
    }
    emit(word, total)
}
```

Execution of WordCount

d123

a x b
b a y

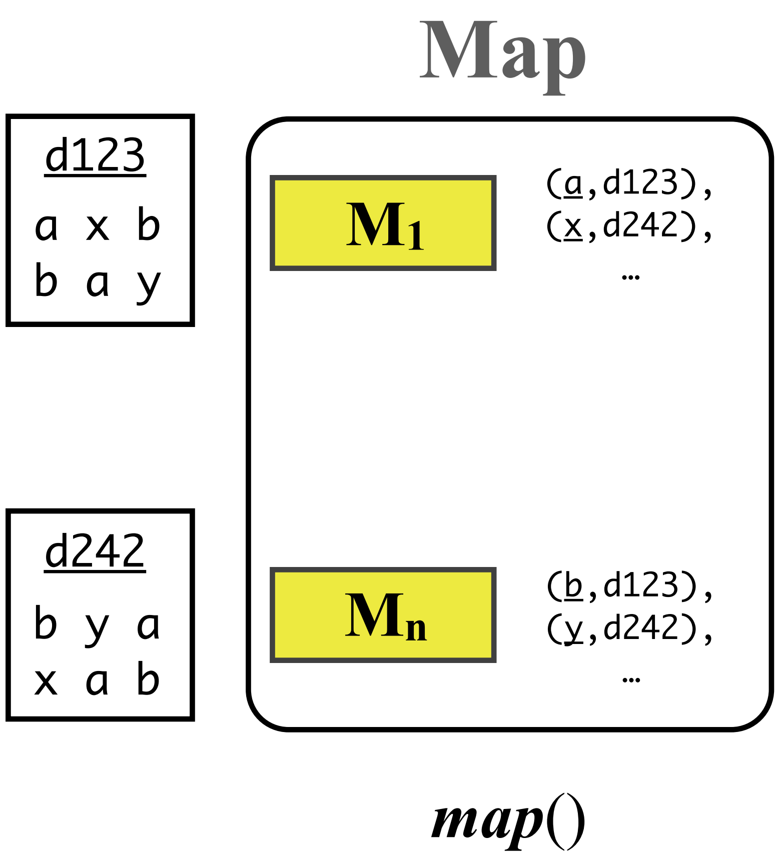
d242

b y a
x a b

```
map(long did, string content) {
    for(string word : content.split()) {
        emit(word, 1)
    }
}
```

```
reduce(string word, list<int> counts) {
    int total = 0
    for(int count : counts) {
        total += count
    }
    emit(word, total)
}
```

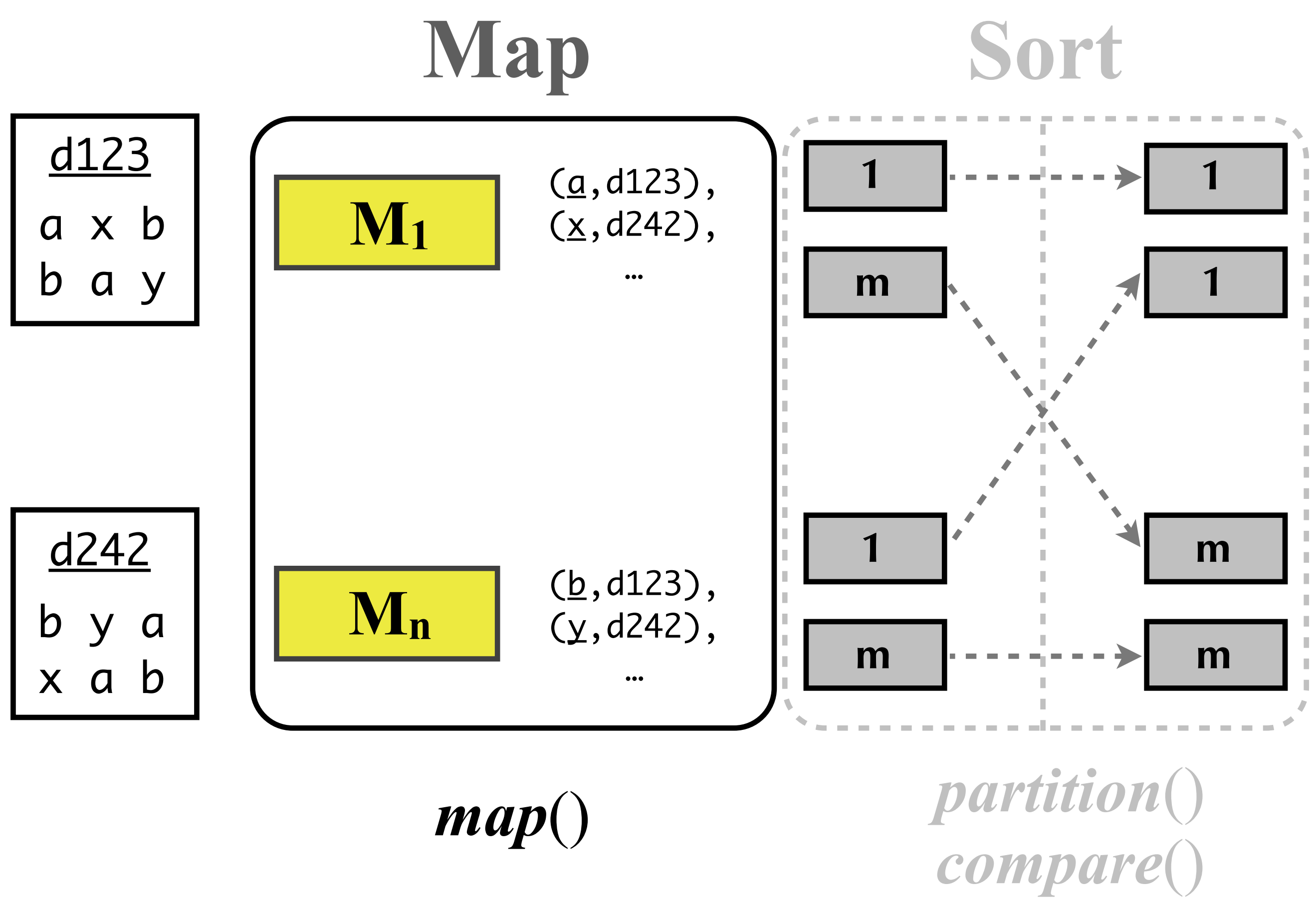
Execution of WordCount




```
map(long did, string content) {
    for(string word : content.split()) {
        emit(word, 1)
    }
}
```

```
reduce(string word, list<int> counts) {
    int total = 0
    for(int count : counts) {
        total += count
    }
    emit(word, total)
}
```

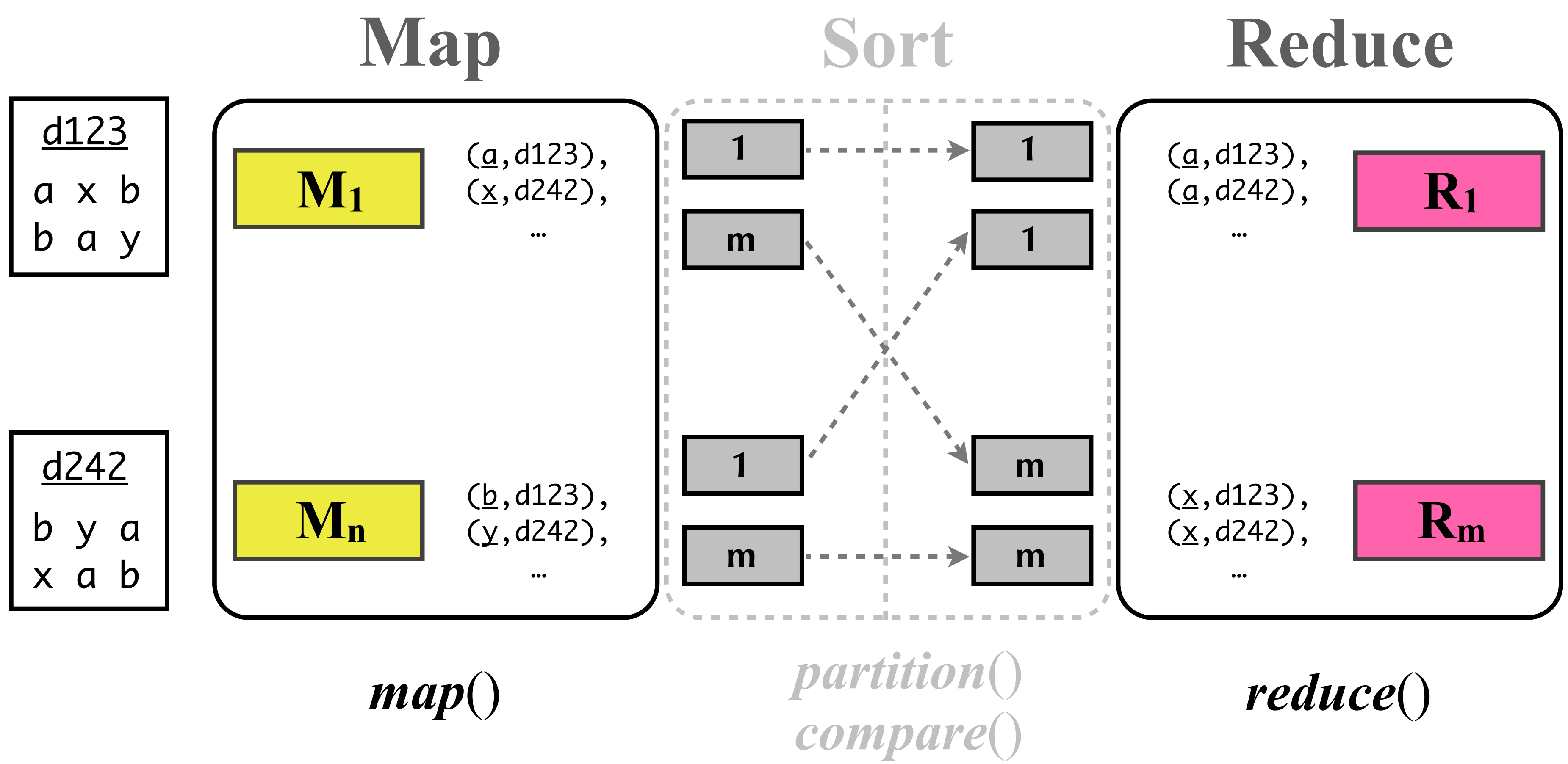
Execution of WordCount




```
map(long did, string content) {
    for(string word : content.split()) {
        emit(word, 1)
    }
}
```

```
reduce(string word, list<int> counts) {
    int total = 0
    for(int count : counts) {
        total += count
    }
    emit(word, total)
}
```

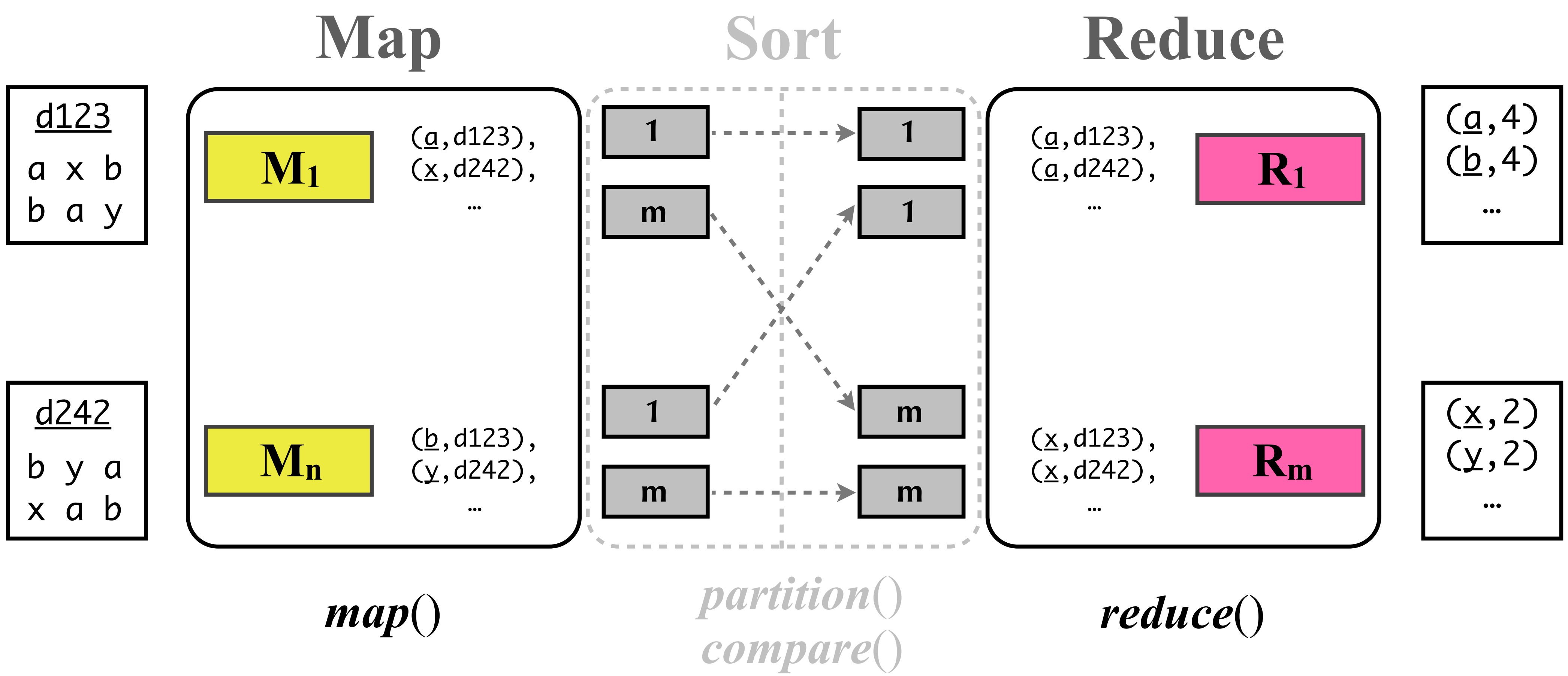
Execution of WordCount




```
map(long did, string content) {
    for(string word : content.split()) {
        emit(word, 1)
    }
}
```

```
reduce(string word, list<int> counts) {
    int total = 0
    for(int count : counts) {
        total += count
    }
    emit(word, total)
}
```

Execution of WordCount




```
map(long did, string content) {
    for(string word : content.split()) {
        emit(word, 1)
    }
}
```

```
reduce(string word, list<int> counts) {
    int total = 0
    for(int count : counts) {
        total += count
    }
    emit(word, total)
}
```

Inverted Index Construction

- Problem: Construct a positional inverted index with postings containing positions (e.g., {d_{123}, 3, [1, 9, 20] })

```
map(long did, string content) {
    int pos = 0
    map<string, list<int>> positions = new map<string, list<int>>()
    for(string word : content.split()) {                      // tokenize document content
        positions.get(word).add(pos++)                        // aggregate word positions
    }
    for(string word : map.keys()) {
        emit(word, new posting(did, positions.get(word)))     // emit posting
    }
}
```

```
reduce(string word, list<posting> postings) {
    postings.sort()                                           // sort postings (e.g., by did)
    emit(word, postings)                                      // emit posting list
}
```

3. Hadoop

- **Open source implementation** of GFS and MapReduce

- **Hadoop File System** (HDFS)
  - name node (master)
  - data node (chunkserver)

- **Hadoop MapReduce**
  - job tracker (master)
  - task tracker (worker)

Doug Cutting

- Has been successfully deployed on clusters of **10,000s machines**
- **Productive use** at Yahoo!, Facebook, and many more

Jim Gray Benchmark

- **Jim Gray Benchmark**:
  - sort large amount of 100 byte records (10 first bytes are keys)
  - **minute sort**: sort as many records as possible in under a minute
  - **gray sort**: must sort at least 100 TB, must run at least 1 hours

- **November 2008**: Google sorts 1 TB in 68 s and 1 PB in 6:02 h on MapReduce using a cluster of 4,000 computers and 48,000 hard disks
  http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html

- **May 2011**: Yahoo! sorts 1 TB in 62 s and 1 PB in 16:15 h on Hadoop using a cluster of approximately 3,800 computers 15,200 hard disks
  http://developer.yahoo.com/blogs/hadoop/posts/2009/05/hadoop_sorts_a_petabyte_in_162/

Summary of V.4

- **MapReduce**
  a system of distributed data processing
  a programming model

- **Hadoop**
  a widely-used open-source implementation of MapReduce

Additional Literature for V.4

- **Apache Hadoop (http://hadoop.apache.org)**
- **J. Dean and S. Ghemawat**: *MapReduce: Simplified Data Processing on Large Clusters*, OSDI 2004
- **J. Dean and S. Ghemawat**: *MapReduce: Simplified Data Processing on Large Clusters*, CACM 51(1):107-113, 2008
- **S. Ghemawat, H. Gobioff, and S.-T. Leung**: *The Google File System*, SOPS 2003
- **J. Lin and C. Dyer**: *Data-Intensive Text Processing with MapReduce*, Morgan & Claypool Publishers, 2010 (http://lintool.github.io/MapReduceAlgorithms)

V.5 Near-Duplicate Detection

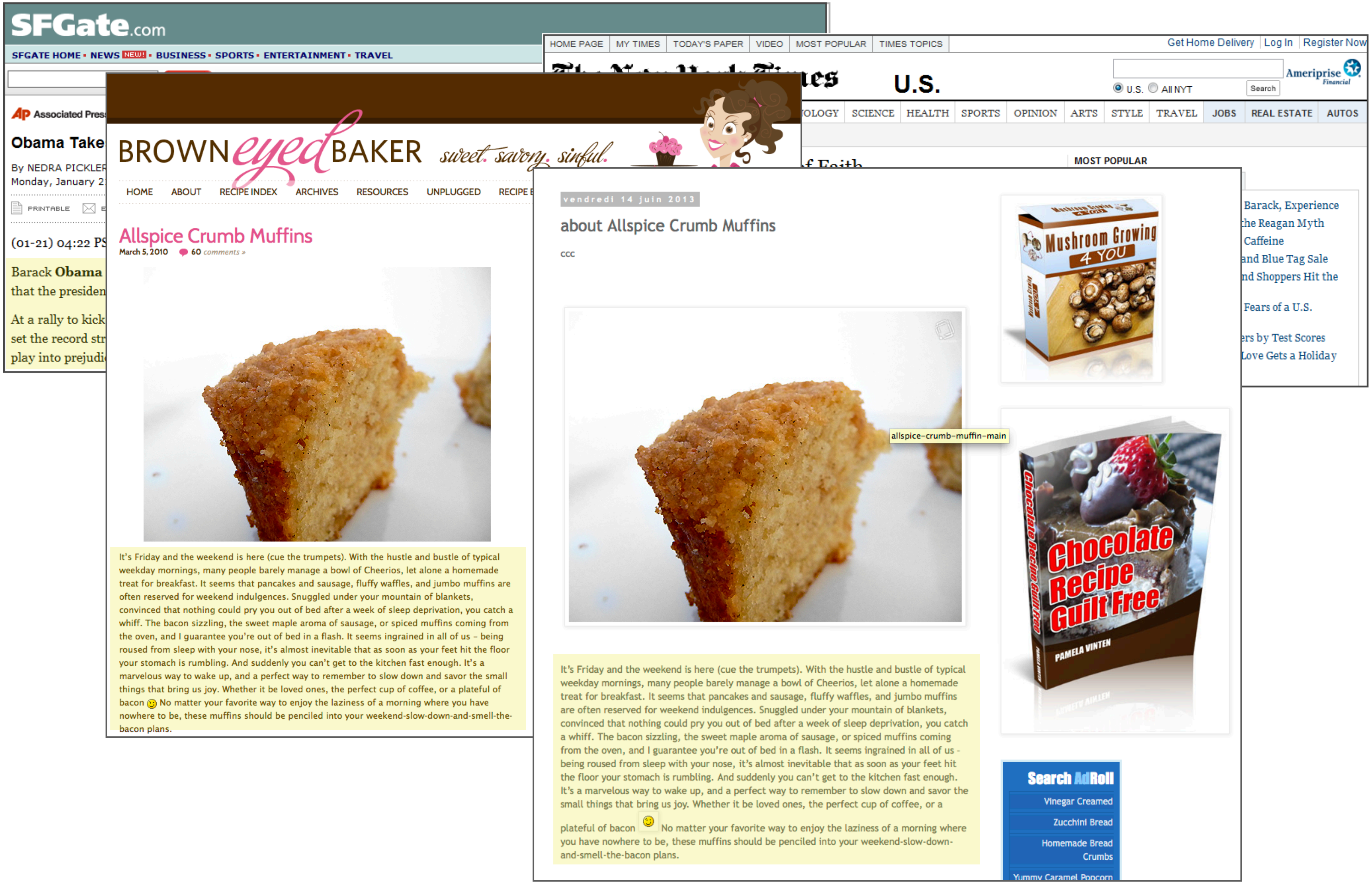
1. **Shingling**
2. **SpotSigs**
3. **Min-Wise Independent Permutations**
4. **Locality-Sensitive Hashing**

Based on **MRS Chapter 19** and **RU Chapter 3**

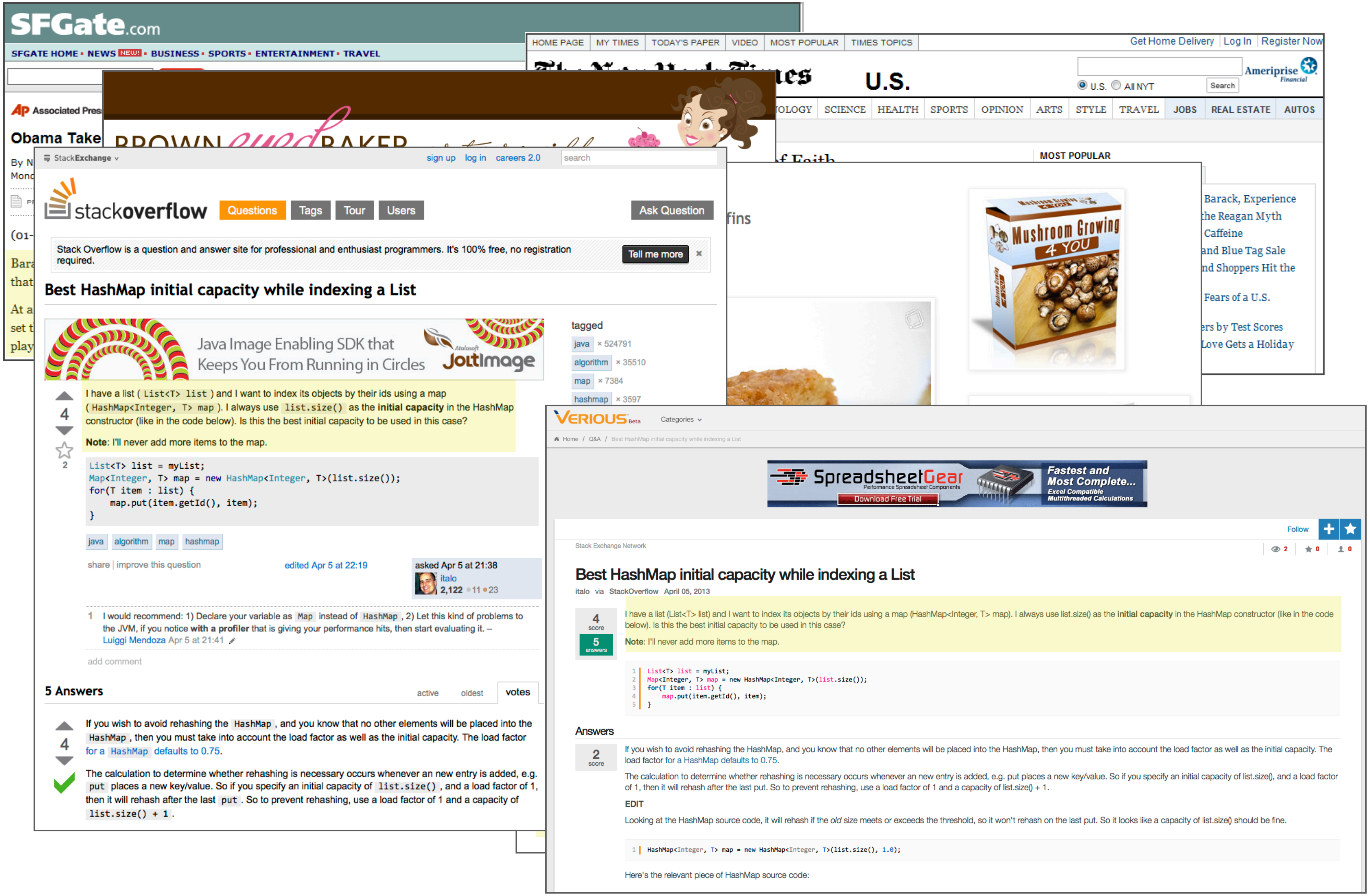
Near-Duplicate Detection

Near-Duplicate Detection

Near-Duplicate Detection



Near-Duplicate Detection



Near-Duplicate Detection

- Why near-duplicate detection?
  - **smaller indexes** and thus **faster response times**
  - improved **result quality**

- **Building blocks** of a near-duplicate detection method
  - **document representation** (e.g., bag of words, bag of *n*-grams, set of links, anchor text of inlinks, set of relevant queries, feature vector)
  - **similarity measure** (e.g., Jaccard coefficient, cosine similarity)
  - **near-duplication detection algorithm**
    - sorting- and indexing-based approaches
    - similarity hashing (e.g., MIPS, LSH)

1. Shingling

- Observation: Duplicates on the Web are often **slightly perturbed** (e.g., due to different boilerplate, minor rewordings, etc.)

- **Document fingerprinting** (e.g., SHA-1 or MD5) is not effective, since we need to allow for minor differences between documents

- **Shingling** represents document d as set $S(d)$ of **word-level *n*-grams** (*shingles*) and compares documents based on these sets

$\boldsymbol{n} = 3$

the little brown fox jumps over the green frog $\rightarrow$ { *the little brown*, *little brown fox*, *brown fox jumps*, *fox jumps over*, … }

Shingling

- Encode shingles by **hash fingerprints** (e.g., using SHA-1), yielding a set of numbers $S(d) \subseteq [1, \ldots, n]$ (e.g., for $n = 2^{64}$)

n = 3

{ *the little brown*, *little brown fox*, *brown fox jumps*, *fox jumps over*, … } $\rightarrow$ { 141,944, 13,031,980, 21,111,978, *6,012,014*, … }

- Compare suspected near-duplicate documents *d* and *d'* by
  - **Resemblance** $\frac{|S(d) \cap S(d')|}{|S(d) \cup S(d')|}$ (Jaccard coefficient)
  - **Containment** $\frac{|S(d) \cap S(d')|}{|S(d)|}$ (Relative overlap)

Shingle-Based Clustering

- Remove near-duplicate document *d'* if resemblance or containment is **above a user-specified threshold τ**

- How to **avoid** comparing all pairs of documents?

1. Compute **shingle set** *S*(*d*) for each document *d*
2. Build **inverted index**: shingle => list of document identifiers
3. Compute (*d*, *d'*, *c*) table with common-shingle count *c* by considering **all pairs of documents** (*d*, *d'*) **per shingle**
4. Keep all pairs of documents (*d*, *d'*) with **similarity above threshold** and add (*d*, *d'*) as edge to a graph
5. Compute **connected components** of graph (using union-find algorithm) as clusters of near-duplicate documents

Super Shingles and Complexity

- **Super shingles** (shingles over shingles) can be used to speed up steps 2 and 3 of the algorithm, since documents with many common shingles are likely to have common super shingle

- Algorithm considers **only pairs** of documents that have **at least one shingle in common**, but worst case remains at $O(n^2)$

- Problem: Shingle sets can become quite large, making the similarity computation expensive

- Full details: [Broder et al. ’97]

2. SpotSigs

- Problem: Near-duplicate detection on the Web fails for web pages with **same core content** but **different navigation, header, etc.**
- Observation: **Stopwords** tend to occur mostly in core content
- **SpotSigs** considers only those shingles that begin with a stopword

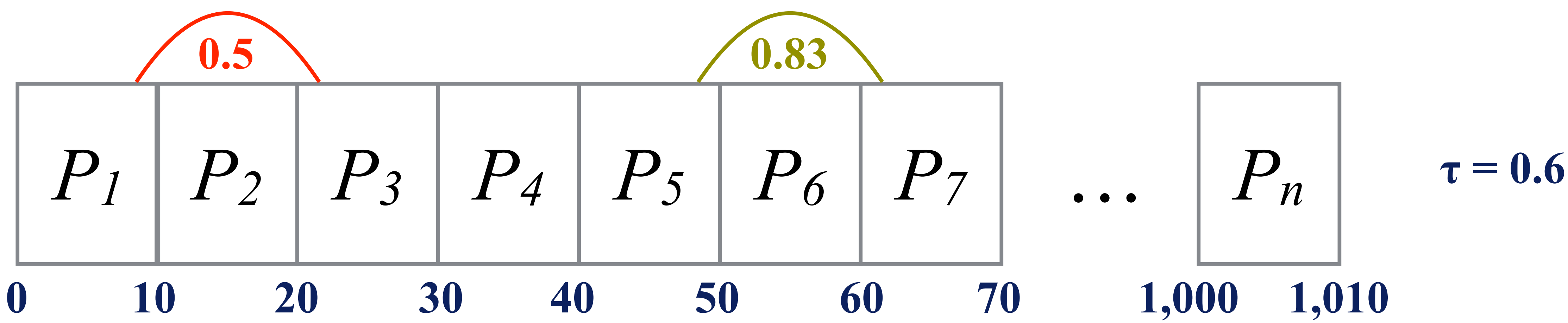
- Problem: How can we perform fewer similarity computations?
- **Upper bound** for Jaccard coefficient

$$r(A,B) = \frac{|A \cap B|}{|A \cup B|} \leq \frac{min(|A|,|B|)}{max(|A|,|B|)}$$

$$\leq \frac{|A|}{|B|} \quad \text{(assuming } |A| \leq |B| \text{ w.l.o.g.)}$$

SpotSigs

- Do not compare any sets $|A|$ and $|B|$ with $|A| / |B| \leq \tau$
- Given similarity threshold τ, **partition the documents** (based on their signature set cardinality) into partitions $P_1, \ldots, P_n$



- Consider document pairs in $P_i \times P_j$ ($i \leq j$) only if

$$\frac{|max\{|S(d)| \mid d \in P_i\}|}{|min\{|S(d)| \mid d \in P_j\}|} > \tau$$

- **Clever partitioning** to compare at most neighboring partitions
- Full details: [Theobald et al. ’08]

3. Min-Wise Independent Permutations

- **Statistical sketch** to estimate the resemblance of $S(d)$ and $S(d')$
  - consider m **independent random permutations** of the two sets, implemented by applying m **independent hash functions**
  - keep the **minimum value** observed for each of the m hash functions, yielding a m-dimensional MIPs vector for each document
  - **estimate resemblance** of $S(d)$ and $S(d')$ based on MIPs(d) and MIPs(d')

$$\hat{r}(d, d') = \frac{|\{1 \leq i \leq m \mid MIPs(d)[i] = MIPs(d')[i]\}|}{m}$$

- Full details: [Broder et al. ’00]

Min-Wise Independent Permutations

Set of shingle fingerprints

$S(d) = \{ 3, 8, 12, 17, 21, 24\}$

$\boldsymbol{h_1(x) = 7x + 3 \bmod 51}$
$\{ 24, 8, 36, 20, 48, 18\}$

$\boldsymbol{h_2(x) = 5x + 6 \bmod 51}$
$\{ 21, 46, 15, 40, 9, 24\}$

.
.
.

$\boldsymbol{h_m(x) = 3x + 9 \bmod 51}$
$\{ 18, 33, 45, 9, 21, 30\}$

8
9
.
.
.
9

MIPs(*d*)
MIPs vector

MIPs(*d*)		*MIPs*(*d'*)
8	↔	8
9	↔	1
5	↔	2
9	↔	9

Estimated resemblance: 2 / 4

- MIPs are an **unbiased estimator of resemblance**

$$P[min\{h(x)|x \in A\} = min\{h(y)|y \in B\}] = |A \cap B|/|A \cup B|$$

- MIPs can be seen as **repeated random sampling** of x,y from A,B

4. Locality Sensitive Hashing (for MIPs)

- General idea behind **locality sensitive hashing** (LSH)
  - hash each item l times so that **similar items map to same bucket**
  - consider pairs of items similar that mapped **at least once to same bucket**

- Locality sensitive hashing with MIPs vectors
  - compute ***l* independent MIPs vectors of length *m*** for each document
  - consider document pairs with **at least one common MIPs vector**

$S(d) = \{3, 8, 12, 17, 21, 24\}$ $\rightarrow$ $\begin{bmatrix}8\\2\end{bmatrix}$ $\begin{bmatrix}9\\1\end{bmatrix}$ $\cdots$ $\begin{bmatrix}5\\7\end{bmatrix}$

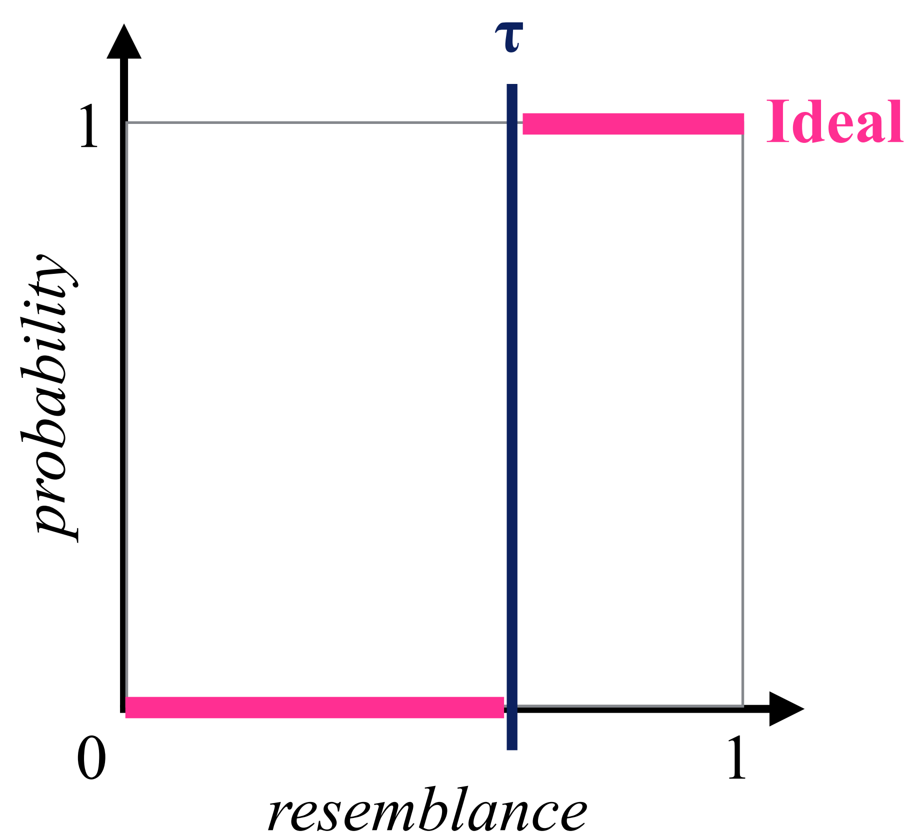
$MIPs_1(d)$ $MIPs_2(d)$ $MIPs_l(d)$

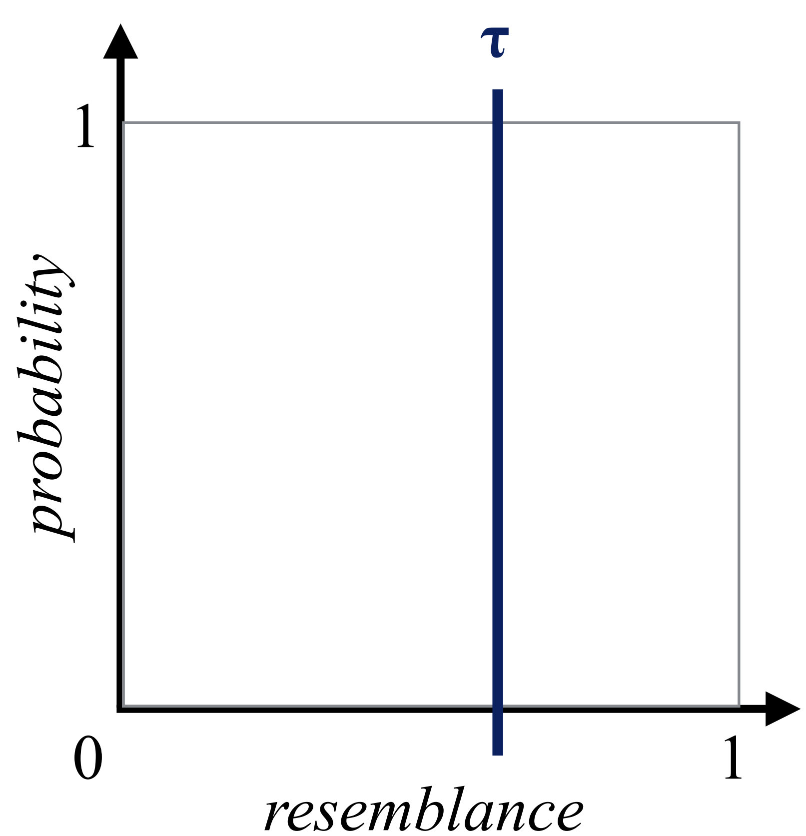
$S(d') = \{3, 5, 12, 17, 22, 24\}$ $\rightarrow$ $\begin{bmatrix}3\\8\end{bmatrix}$ $\begin{bmatrix}9\\1\end{bmatrix}$ $\cdots$ $\begin{bmatrix}2\\5\end{bmatrix}$

$MIPs_1(d')$ $MIPs_2(d')$ $MIPs_l(d')$

LSH Analysis

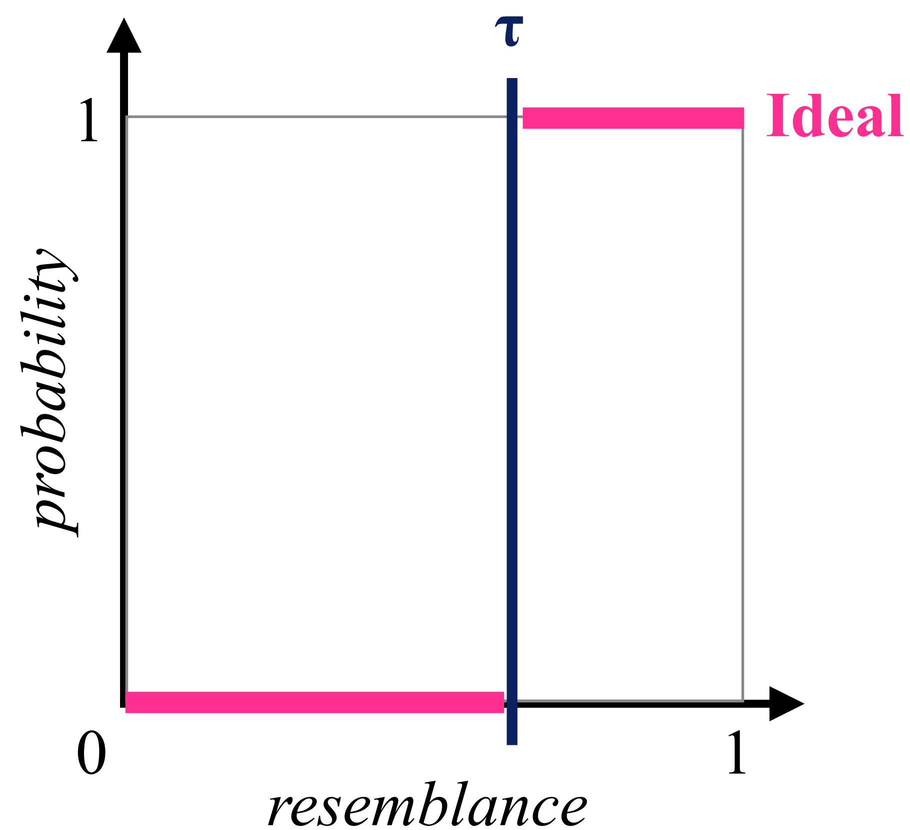
- Let $r = r(d, d')$ denote the resemblance between d and d'
  - $P[MIPs_i(\mathrm{d}) = MIPs_i(\mathrm{d}')] = r^{\mathrm{m}}$: same i-th MIPs vector
  - $1 - r^{m}$: different i-th MIPs vector
  - $(1 - r^{m})^{l}$: all MIPs vectors different
  - $1 - (1 - r^{m})^{l}$: at least one MIPs vector in common

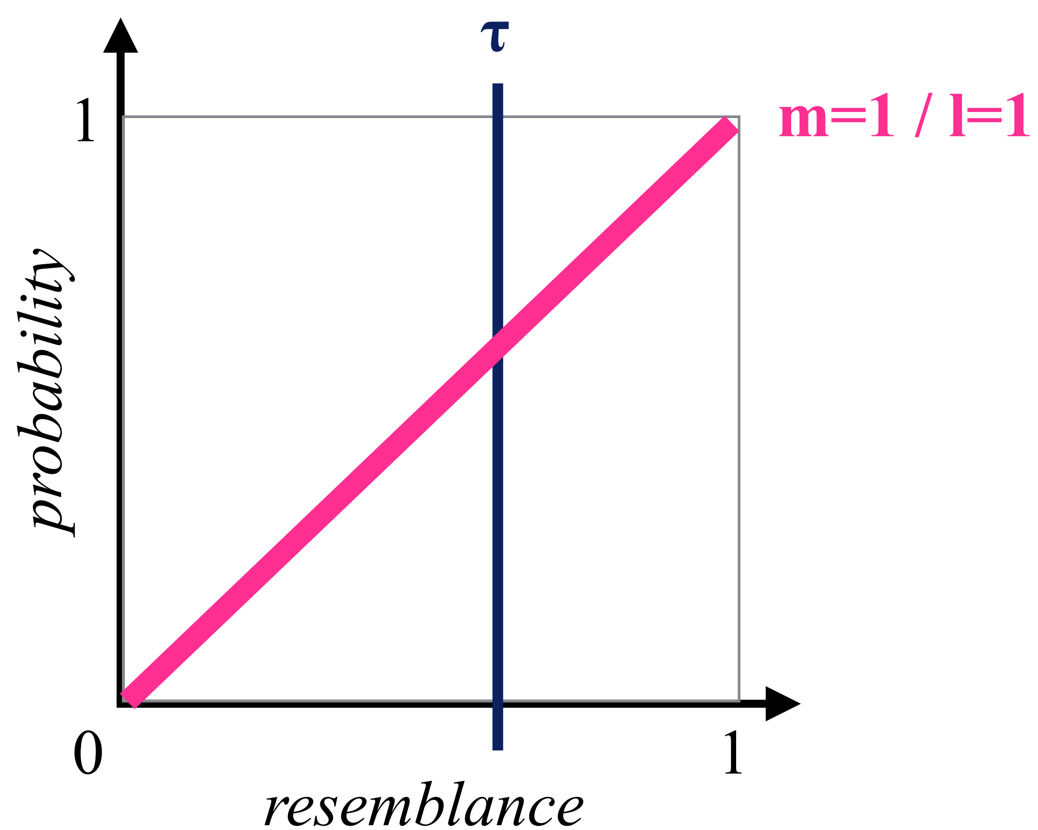




LSH Analysis

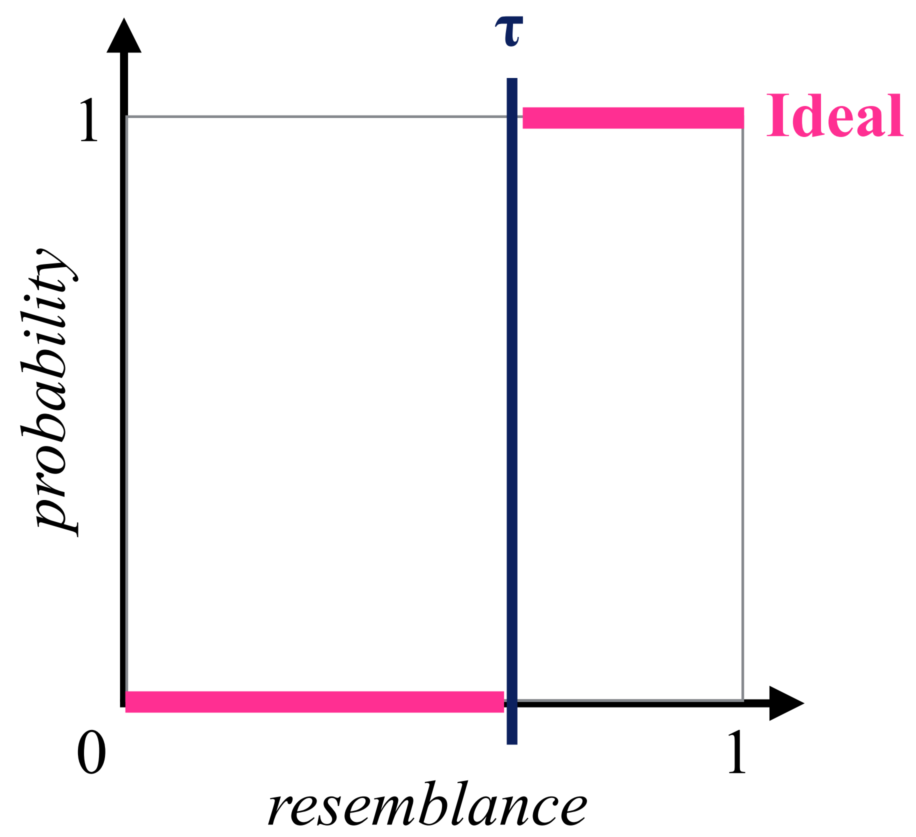
- Let $r = r(d, d')$ denote the resemblance between d and d'
  - $P[MIPs_i(\mathrm{d}) = MIPs_i(\mathrm{d}')] = r^{\mathrm{m}}$: same i-th MIPs vector
  - $1 - r^{m}$: different i-th MIPs vector
  - $(1 - r^{m})^{l}$: all MIPs vectors different
  - $1 - (1 - r^{m})^{l}$: at least one MIPs vector in common

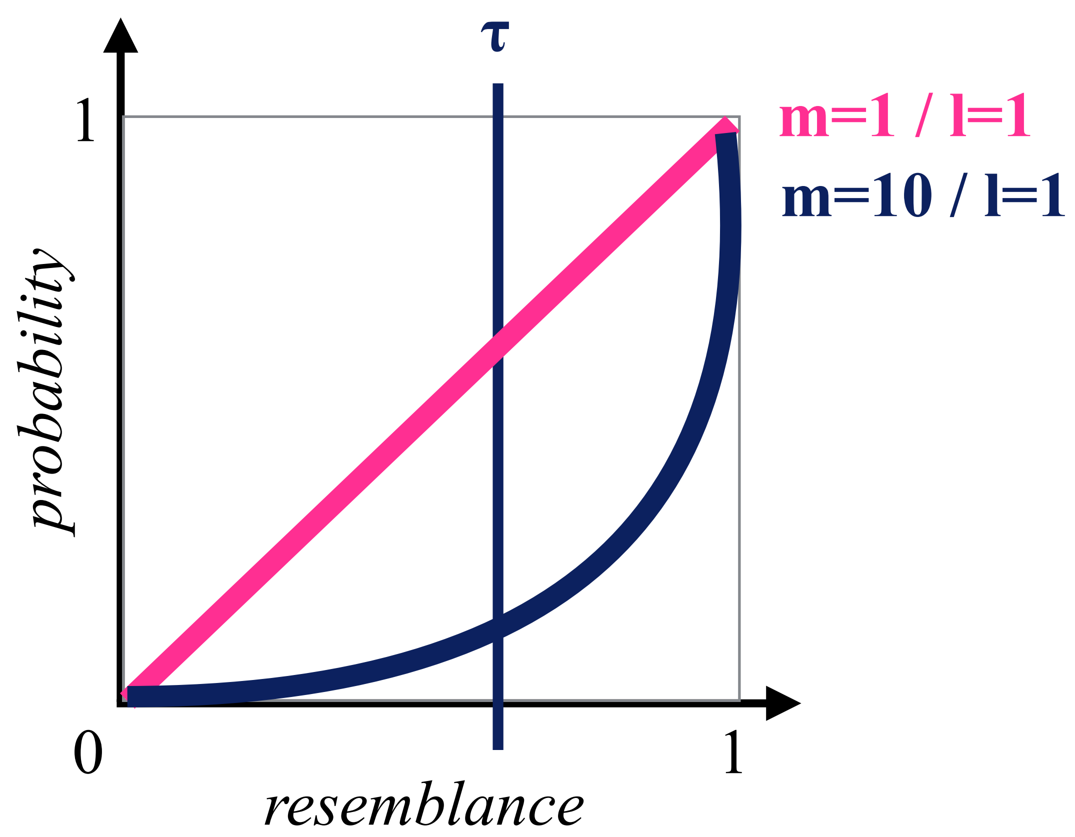




LSH Analysis

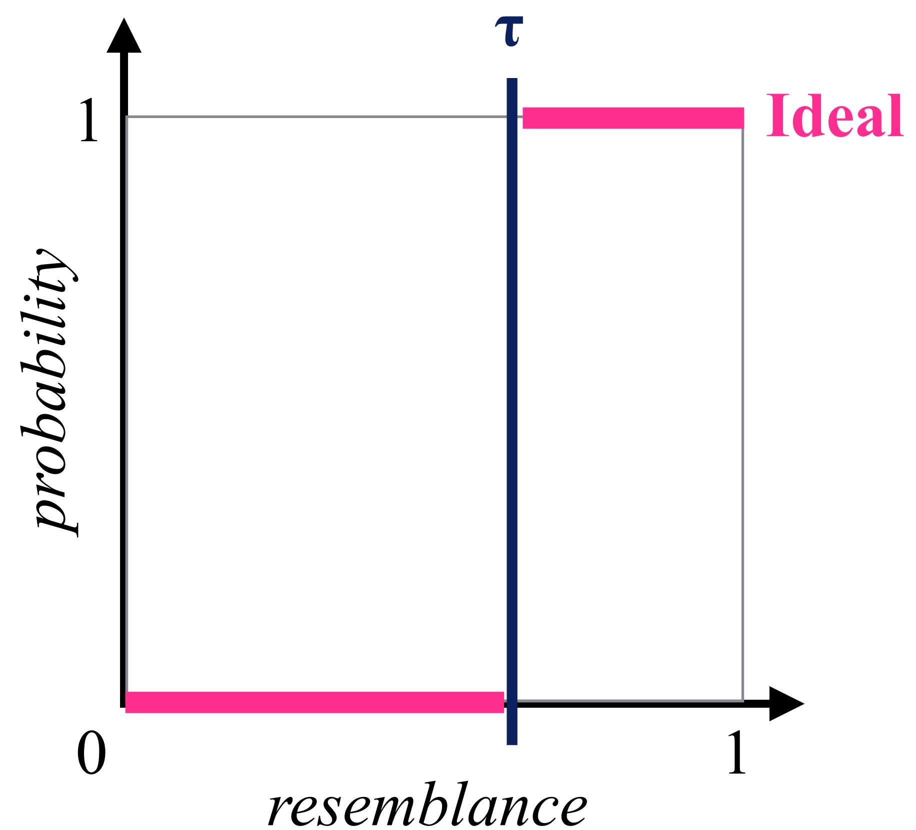
- Let $r = r(d, d')$ denote the resemblance between d and d'
  - $P[MIPs_i(d) = MIPs_i(d')] = r^{m}$: same i-th MIPs vector
  - $1 - r^{m}$: different i-th MIPs vector
  - $(1 - r^{m})^{l}$: all MIPs vectors different
  - $1 - (1 - r^{m})^{l}$: at least one MIPs vector in common

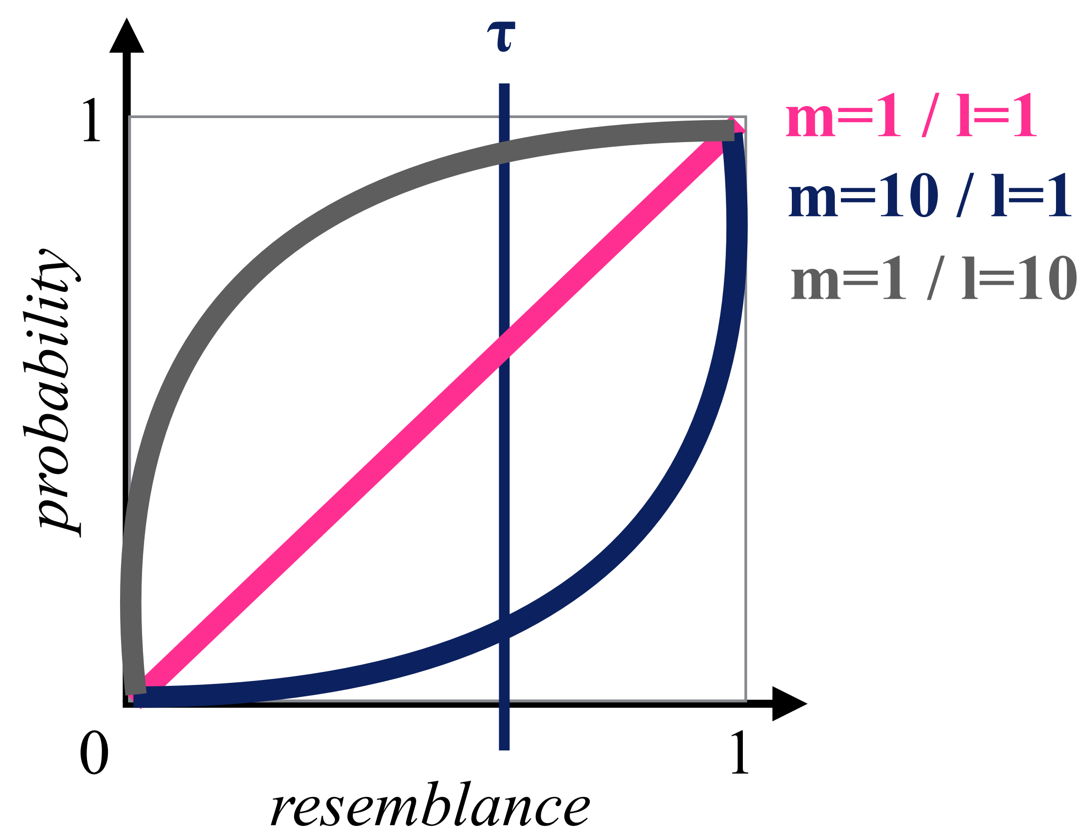




LSH Analysis

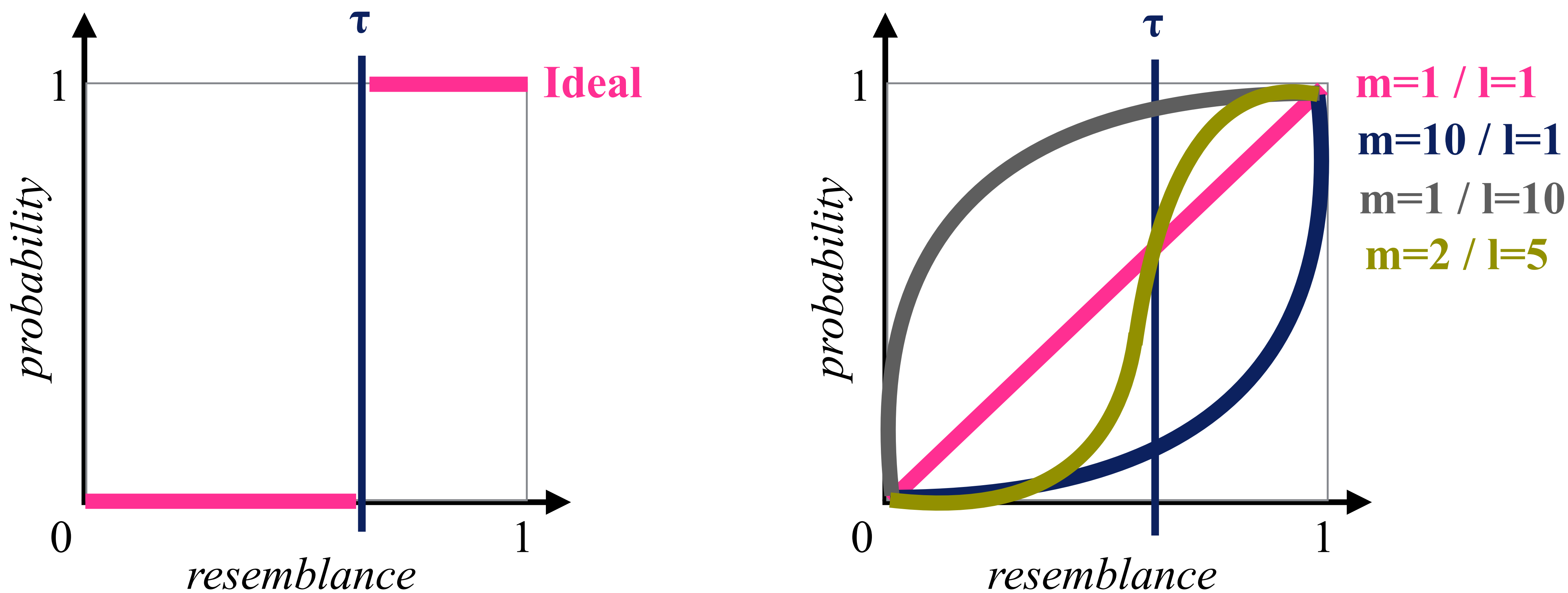
- Let $r = r(d, d')$ denote the resemblance between d and d'
  - $P[MIPs_i(d) = MIPs_i(d')] = r^m$: same i-th MIPs vector
  - $1 - r^m$: different i-th MIPs vector
  - $(1 - r^m)^l$: all MIPs vectors different
  - $1 - (1 - r^m)^l$: at least one MIPs vector in common





LSH Analysis

- Let $r = r(d, d')$ denote the resemblance between d and d'
  - $P[MIPs_i(d) = MIPs_i(d')] = r^m$: same i-th MIPs vector
  - $1 - r^m$: different i-th MIPs vector
  - $(1 - r^m)^l$: all MIPs vectors different
  - $1 - (1 - r^m)^l$: at least one MIPs vector in common



LSH Analysis

- Example: For a pair of documents d and d' with $r(d, d') = 0.8$, $m = 5$, and $l = 20$, the probability of missing the pair is $(1 - 0.8^{5})^{20} = 3.56 \times 10^{-4}$

- Full details: [Gionis et al. '99]

Summary of V.5

- **Near-Duplicate Detection**
  essential for smaller indexes and better result quality
- **Shingling**
  to deal with small perturbations in otherwise duplicate documents
- **SpotSigs**
  focuses on shingles beginning with a stopword
  uses smart blocking to compare fewer document pairs
- **Min-Wise Independent Permutations**
  as a statistical sketch to approximate resemblance
- **Locality-Sensitive Hashing**
  as a method to reduce the number of document comparisons

Additional Literature for V.5

- **A. Broder, S. Glassman, M. Manasse, and G. Zweig**: *Syntactic Clustering of the Web*, WWW 1997
- **A. Broder, M. Charikar, A. Frieze, M. Mitzenmacher**: *Min-Wise Independent Permutations,* JCSS 60(3):630-659, 2000
- **A. Gionis, P. Indyk, and R. Motwani**: *Similarity Search in High Dimensions via Hashing,* VLDB 1999
- **M. Henzinger**: *Finding Near-Duplicate Web Pages: a Large-Scale Evaluation of Algorithms*, SIGIR 2006
- **M. Theobald, J. Siddharth, and A. Paepcke**: *SpotSigs: Robust and efficient near duplicate detection in large web collections,* SIGIR 2008