Large dictionaries

Paolo Boldi
DSI
LAW (Laboratory for Web Algorithmics)
Università degli Studi di Milan

Term map

Term map

- While building the inverted index we build a map from terms to numbers.

Term map

- While building the inverted index we build a map from terms to numbers.
- The same kind of map is needed while building the graph: in that case it is a map from URLs to numbers:

`http://pippo.pluto/x/y`	0
`http://topolino.minnie/w.htm`	1
`http://topolino.minnie/z.htm`	2
...	...

Term map

- While building the inverted index we build a map from terms to numbers.
- The same kind of map is needed while building the graph: in that case it is a map from URLs to numbers:

`http://pippo.pluto/x/y`	0
`http://topolino.minnie/w.htm`	1
`http://topolino.minnie/z.htm`	2
...	...

- The numbering used is arbitrary, but lexicographic numbering turns out to be convenient for many reasons. . .

Building the graph

Building the graph

- From the map, one can build a graph by scanning the pages (one at a time), parsing them and determining outgoing links (anchors).

Building the graph

- From the map, one can build a graph by scanning the pages (one at a time), parsing them and determining outgoing links (anchors).
- Given a link to, say, `http://foo/bar/index.html`, we just have to determine the *number* to which this URL corresponds.

Building the graph

- From the map, one can build a graph by scanning the pages (one at a time), parsing them and determining outgoing links (anchors).
- Given a link to, say, `http://foo/bar/index.html`, we just have to determine the *number* to which this URL corresponds.
- Keeping URLs in an array and doing a binary search is out of questions for reasons of time (1G nodes=30 comparisons) and space (1G nodes=300GB of data!).

The problem

The problem

- In other words, we want to represent a function like:

x	$f(x)$
`http://pippo.pluto/x/y`	0
`http://topolino.minnie/w.htm`	1
`http://topolino.minnie/z.htm`	2
...	...

The problem

- In other words, we want to represent a function like:

x	$f(x)$
`http://pippo.pluto/x/y`	0
`http://topolino.minnie/w.htm`	1
`http://topolino.minnie/z.htm`	2
...	...

- with a data structure that can compute $f(x)$ *quickly* given x.

The problem

- In other words, we want to represent a function like:

x	$f(x)$
`http://pippo.pluto/x/y`	0
`http://topolino.minnie/w.htm`	1
`http://topolino.minnie/z.htm`	2
...	...

- with a data structure that can compute $f(x)$ *quickly* given x.
- Construction time is not a problem (within reasonable limits)!

The problem

- In other words, we want to represent a function like:

x	$f(x)$
`http://pippo.pluto/x/y`	0
`http://topolino.minnie/w.htm`	1
`http://topolino.minnie/z.htm`	2
...	...

- with a data structure that can compute $f(x)$ *quickly* given x.
- Construction time is not a problem (within reasonable limits)!
- We *don't need* the inverse function.

Short interlude: hash functions

Short interlude: hash functions

- Given a universe Ω and an integer m, a *an m-bucket hash function for* Ω is a function $h : \Omega \to [m] = \{0, 1, \ldots, m-1\}$

Short interlude: hash functions

- Given a universe Ω and an integer m, a *an m-bucket hash function for* Ω is a function $h : \Omega \to [m] = \{0, 1, \ldots, m-1\}$
- It must be easy to compute and as "injective" as possible.

Short interlude: hash functions

- Given a universe Ω and an integer m, a *an m-bucket hash function for* Ω is a function $h : \Omega \to [m] = \{0, 1, \ldots, m-1\}$
- It must be easy to compute and as "injective" as possible.
- In particular, we are interested in its behaviour on a specific set $S \subseteq \Omega$.

Short interlude: hash functions

- Given a universe Ω and an integer m, a *an m-bucket hash function for* Ω is a function $h : \Omega \to [m] = \{0, 1, \ldots, m-1\}$
- It must be easy to compute and as "injective" as possible.
- In particular, we are interested in its behaviour on a specific set $S \subseteq \Omega$.
- Ideally, if $|S| \leq m$, we would like h to be injective on S. In such a case we say that *h is perfect for S*.

Short interlude: hash functions

- Given a universe Ω and an integer m, a *an m-bucket hash function for* Ω is a function $h : \Omega \to [m] = \{0, 1, \ldots, m-1\}$
- It must be easy to compute and as "injective" as possible.
- In particular, we are interested in its behaviour on a specific set $S \subseteq \Omega$.
- Ideally, if $|S| \leq m$, we would like h to be injective on S. In such a case we say that *h is perfect for S*.
- If moreover $|S| = m$, we say that h is *minimal perfect*.

Short interlude: hash functions

- Given a universe Ω and an integer m, a *an m-bucket hash function for* Ω is a function $h : \Omega \to [m] = \{0, 1, \ldots, m-1\}$
- It must be easy to compute and as "injective" as possible.
- In particular, we are interested in its behaviour on a specific set $S \subseteq \Omega$.
- Ideally, if $|S| \leq m$, we would like h to be injective on S. In such a case we say that *h is perfect for S*.
- If moreover $|S| = m$, we say that h is *minimal perfect*.
- Usually, obtaining a minimal perfect hash function is impossible because S is *unknown*. Not in our case, though. . .

Short interlude: hash functions

- Given a universe Ω and an integer m, a *an m-bucket hash function for* Ω is a function $h : \Omega \to [m] = \{0, 1, \ldots, m-1\}$
- It must be easy to compute and as "injective" as possible.
- In particular, we are interested in its behaviour on a specific set $S \subseteq \Omega$.
- Ideally, if $|S| \leq m$, we would like h to be injective on S. In such a case we say that *h is perfect for S*.
- If moreover $|S| = m$, we say that h is *minimal perfect*.
- Usually, obtaining a minimal perfect hash function is impossible because S is *unknown*. Not in our case, though. . .
- We will present a technique introduced by Majewski, Wormald, Havas and Czech.

Short interlude: string hashing

Short interlude: string hashing

- As a concrete example: let $\Omega = \Sigma^{\leq w}$ (the set of all strings of length $\leq w$ on an alphabet Σ);

Short interlude: string hashing

- As a concrete example: let $\Omega = \Sigma^{\leq w}$ (the set of all strings of length $\leq w$ on an alphabet Σ);
- Let m be an integer

Short interlude: string hashing

- As a concrete example: let $\Omega = \Sigma^{\leq w}$ (the set of all strings of length $\leq w$ on an alphabet Σ);
- Let m be an integer
- Let us draw w weights (at random) between 0 and $m-1$:

3	12	7	41	33	...	5

Short interlude: string hashing

- As a concrete example: let $\Omega = \Sigma^{\leq w}$ (the set of all strings of length $\leq w$ on an alphabet Σ);
- Let m be an integer
- Let us draw w weights (at random) between 0 and $m - 1$:

3	12	7	41	33	...	5

- Given a string $x \in \Omega$

n	i	n	o

we look at it as a sequence of w numbers (padding it with zeroes at the end):

110	105	110	111	0	...	0

Short interlude: string hashing

- As a concrete example: let $\Omega = \Sigma^{\leq w}$ (the set of all strings of length $\leq w$ on an alphabet Σ);
- Let m be an integer
- Let us draw w weights (at random) between 0 and $m - 1$:

3	12	7	41	33	...	5

- Given a string $x \in \Omega$

n	i	n	o

we look at it as a sequence of w numbers (padding it with zeroes at the end):

110	105	110	111	0	...	0

- $h(x)$ is defined multiplying each character to the corresponding weight, summing up and taking the result modulo m: $(3 \times 110 + 12 \times 105 + \dots) \bmod m$.

The MWHC Algorithm: 1

The MWHC Algorithm: 1

- Take some $m \geq n$, and choose *uniformly at random* two hash functions h_1 and h_2 from strings to $\{0, 1, \ldots, m-1\}$

The MWHC Algorithm: 1

- Take some $m \geq n$, and choose *uniformly at random* two hash functions h_1 and h_2 from strings to $\{0, 1, \ldots, m-1\}$
- For example

x	$f(x)$	$h_1(x)$	$h_2(x)$
`http://pippo.pluto/x/y`	0	231	3443
`http://topolino.minnie/w.htm`	1	32	5534
`http://topolino.minnie/z.htm`	2	231	32
...	...	...	...

The MWHC Algorithm: 1

- Take some $m \geq n$, and choose *uniformly at random* two hash functions h_1 and h_2 from strings to $\{0, 1, \ldots, m-1\}$
- For example

x	$f(x)$	$h_1(x)$	$h_2(x)$
`http://pippo.pluto/x/y`	0	231	3443
`http://topolino.minnie/w.htm`	1	32	5534
`http://topolino.minnie/z.htm`	2	231	32
...	...	...	...

- Build a graph whose vertices are $\{0, 1, \ldots, m-1\}$ and with an edge for every string: the edge for string x connects $h_1(x)$ and $h_2(x)$.

The MWHC Algorithm: 1

- Take some $m \geq n$, and choose *uniformly at random* two hash functions h_1 and h_2 from strings to $\{0, 1, \ldots, m-1\}$
- For example

x	$f(x)$	$h_1(x)$	$h_2(x)$
`http://pippo.pluto/x/y`	0	231	3443
`http://topolino.minnie/w.htm`	1	32	5534
`http://topolino.minnie/z.htm`	2	231	32
...	...	...	...

- Build a graph whose vertices are $\{0, 1, \ldots, m-1\}$ and with an edge for every string: the edge for string x connects $h_1(x)$ and $h_2(x)$.
- Special cases: degenerate arcs? coincident arcs? cyclic graph? We throw h_1 and h_2 away and generate another pair.

The MWHC Algorithm: 1

- Take some $m \geq n$, and choose *uniformly at random* two hash functions h_1 and h_2 from strings to $\{0, 1, \ldots, m-1\}$
- For example

x	$f(x)$	$h_1(x)$	$h_2(x)$
`http://pippo.pluto/x/y`	0	231	3443
`http://topolino.minnie/w.htm`	1	32	5534
`http://topolino.minnie/z.htm`	2	231	32
...	...	...	...

- Build a graph whose vertices are $\{0, 1, \ldots, m-1\}$ and with an edge for every string: the edge for string x connects $h_1(x)$ and $h_2(x)$.
- Special cases: degenerate arcs? coincident arcs? cyclic graph? We throw h_1 and h_2 away and generate another pair.
- **Theorem**: if m is large enough ($m \geq 1.75n$) with high probability (with an expected number of $e^{4/5} \approx 2$ attempts) we will get a graph satisfying the constraints.

The MWHC Algorithm: 2

The MWHC Algorithm: 2

- Let's go back to our example:

x	$f(x)$	$h_1(x)$	$h_2(x)$
`http://pippo.pluto/x/y`	0	231	3443
`http://topolino.minnie/w.htm`	1	32	5534
`http://topolino.minnie/z.htm`	2	231	32
...	...	...	...

The MWHC Algorithm: 2

- Let's go back to our example:

x	$f(x)$	$h_1(x)$	$h_2(x)$
`http://pippo.pluto/x/y`	0	231	3443
`http://topolino.minnie/w.htm`	1	32	5534
`http://topolino.minnie/z.htm`	2	231	32
...	...	...	...

- We associate to every vertex a variable (the variable associated to 231 is x_{231}) and look at the graph as a system of modular equations:

$$\begin{aligned} (x_{231} + x_{3443}) \bmod m &= 0 \\ (x_{32} + x_{5534}) \bmod m &= 1 \\ (x_{231} + x_{32}) \bmod m &= 2 \end{aligned}$$

The MWHC Algorithm: 2

- Let's go back to our example:

x	$f(x)$	$h_1(x)$	$h_2(x)$
`http://pippo.pluto/x/y`	0	231	3443
`http://topolino.minnie/w.htm`	1	32	5534
`http://topolino.minnie/z.htm`	2	231	32
...	...	...	...

- We associate to every vertex a variable (the variable associated to 231 is x_{231}) and look at the graph as a system of modular equations:

$$\begin{aligned}(x_{231}+x_{3443}) \bmod m &= 0\\(x_{32}+x_{5534}) \bmod m &= 1\\(x_{231}+x_{32}) \bmod m &= 2\end{aligned}$$

- **Theorem**: If the graph is acyclic, this system admits a solution (it can be found with a DFS).

The MWHC Algorithm: 2 (bis)

The MWHC Algorithm: 2 (bis)

- More precisely, graph acyclicity implies that you can re-order the equations in such a way that every equation contains a variable that *never appear before*:

$$\begin{aligned}(x_{231} + x_{3443}) \bmod m &= 0\\(x_{32} + x_{5534}) \bmod m &= 1\\(x_{231} + x_{32}) \bmod m &= 2\end{aligned}$$

The MWHC Algorithm: 2 (bis)

- More precisely, graph acyclicity implies that you can re-order the equations in such a way that every equation contains a variable that *never appear before*:

$$\begin{aligned}(x_{231} + x_{3443}) \bmod m &= 0\\ (x_{32} + x_{5534}) \bmod m &= 1\\ (x_{231} + x_{32}) \bmod m &= 2\end{aligned}$$

... becomes ...

$$\begin{aligned}(x_{231} + x_{3443}) \bmod m &= 0\\ (x_{231} + x_{32}) \bmod m &= 2\\ (x_{32} + x_{5534}) \bmod m &= 1\end{aligned}$$

The MWHC Algorithm: 2 (bis)

- More precisely, graph acyclicity implies that you can re-order the equations in such a way that every equation contains a variable that *never appear before*:

$$\begin{aligned}(x_{231}+x_{3443}) \bmod m &= 0\\(x_{32}+x_{5534}) \bmod m &= 1\\(x_{231}+x_{32}) \bmod m &= 2\end{aligned}$$

. . . becomes . . .

$$\begin{aligned}(x_{231}+x_{3443}) \bmod m &= 0\\(x_{231}+x_{32}) \bmod m &= 2\\(x_{32}+x_{5534}) \bmod m &= 1\end{aligned}$$

- The "new" variable that appears in every equation is called the "hinge" of that equation.

The MWHC Algorithm: 2 (bis)

- More precisely, graph acyclicity implies that you can re-order the equations in such a way that every equation contains a variable that *never appear before*:

$$\begin{aligned}(x_{231}+x_{3443}) \bmod m &= 0\\ (x_{32}+x_{5534}) \bmod m &= 1\\ (x_{231}+x_{32}) \bmod m &= 2\end{aligned}$$

. . . becomes . . .

$$\begin{aligned}(x_{231}+x_{3443}) \bmod m &= 0\\ (x_{231}+x_{32}) \bmod m &= 2\\ (x_{32}+x_{5534}) \bmod m &= 1\end{aligned}$$

- The "new" variable that appears in every equation is called the "hinge" of that equation.
- Hinges are free! (So you find a solution to the system by assigning to every hinge the value you need to make the equation right)

The MWHC Algorithm: 3

The MWHC Algorithm: 3

- We store an array $x[]$ with the solution found: this array has the property that, for every string x:

$$(x[h_1(x)] + x[h_2(x)]) \bmod m = f(x).$$

The MWHC Algorithm: 3

- We store an array $x[]$ with the solution found: this array has the property that, for every string x:

$$(x[h_1(x)] + x[h_2(x)]) \bmod m = f(x).$$

- So to compute $f(x)$ we just need $x[]$ ($m \approx 2n$ integers, $2n \log n$ bits) and the two weight vectors for computing the two hash functions (the size of this is *independent* from n, it just depends on the string length)

The MWHC Algorithm: 3

- We store an array $x[]$ with the solution found: this array has the property that, for every string x:

$$(x[h_1(x)] + x[h_2(x)]) \bmod m = f(x).$$

- So to compute $f(x)$ we just need $x[]$ ($m \approx 2n$ integers, $2n \log n$ bits) and the two weight vectors for computing the two hash functions (the size of this is *independent* from n, it just depends on the string length)
- Remarks:
  - we are not storing the strings, so it will be IMPOSSIBLE to compute f^{-1}

The MWHC Algorithm: 3

- We store an array $x[]$ with the solution found: this array has the property that, for every string x:

$$(x[h_1(x)] + x[h_2(x)]) \bmod m = f(x).$$

- So to compute $f(x)$ we just need $x[]$ ($m \approx 2n$ integers, $2n \log n$ bits) and the two weight vectors for computing the two hash functions (the size of this is *independent* from n, it just depends on the string length)
- Remarks:
  - we are not storing the strings, so it will be IMPOSSIBLE to compute f^{-1}
  - Observe that the MWHC construction gives much more than a simple minimal perfect hash: it is an *order preserving* one (OPMPH)!

The real MWHC algorithm

The real MWHC algorithm

- The same idea can be applied to more than two hash functions, working with hypergraphs instead of graphs.

The real MWHC algorithm

- The same idea can be applied to more than two hash functions, working with hypergraphs instead of graphs.
- The advantage is that we can get acyclicity with *less* vertices (in the case of graphs, we need $m \geq 1.75n$).

The real MWHC algorithm

- The same idea can be applied to more than two hash functions, working with hypergraphs instead of graphs.
- The advantage is that we can get acyclicity with *less* vertices (in the case of graphs, we need $m \geq 1.75n$).
- It can be shown that the optimum is obtained with *three* hash functions, in which case we just need $m \geq 1.21n$.

The real MWHC algorithm

- The same idea can be applied to more than two hash functions, working with hypergraphs instead of graphs.
- The advantage is that we can get acyclicity with *less* vertices (in the case of graphs, we need $m \geq 1.75n$).
- It can be shown that the optimum is obtained with *three* hash functions, in which case we just need $m \geq 1.21n$.
- With the latter technique, you need $1.21n \log n$ bits to store a minimal order-preserving hash.

The real MWHC algorithm

- The same idea can be applied to more than two hash functions, working with hypergraphs instead of graphs.
- The advantage is that we can get acyclicity with *less* vertices (in the case of graphs, we need $m \geq 1.75n$).
- It can be shown that the optimum is obtained with *three* hash functions, in which case we just need $m \geq 1.21n$.
- With the latter technique, you need $1.21n \log n$ bits to store a minimal order-preserving hash.
- The same idea is actually more general: every function $h : S \subseteq \Omega \to \Psi$ can be represented using only $1.21n \log |\Psi|$ bits, with constant-time evaluation.

A perfect hash

- If you just need a *perfect* hash...

A perfect hash

- If you just need a *perfect* hash...
- You proceed exactly like explained, and you get a system of equations, one per edge:

$$
\begin{aligned}
(x_{231} + x_{3443}) \bmod ??? &= ???\\
(x_{32} + x_{5534}) \bmod ??? &= ???\\
(x_{231} + x_{32111}) \bmod ??? &= ???
\end{aligned}
$$

A perfect hash

- If you just need a *perfect* hash...
- You proceed exactly like explained, and you get a system of equations, one per edge:

$$\begin{aligned}(x_{231} + x_{3443}) \bmod ??? &= ???\\ (x_{32} + x_{5534}) \bmod ??? &= ???\\ (x_{231} + x_{32111}) \bmod ??? &= ???\end{aligned}$$

- Each equation is of the form

$$(x_{h_1(w)} + x_{h_2(w)}) \bmod ??? = ???$$

for some $w \in S$. Acyclicity guarantees that it is possible to *reorder* these equations in some order so that every equation contains a variable (either $h_1(w)$ or $h_2(w)$) that never appeared before.

A perfect hash (cont'd)

- In other words, it is possible to write the system so that

$$(x_{h_1(w)} + x_{h_2(w)}) \bmod 2 = 0 \text{ or } 1$$

depending on whether the hinge is $h_1(w)$ or $h_2(w)$.

A perfect hash (cont'd)

- In other words, it is possible to write the system so that

$$(x_{h_1(w)} + x_{h_2(w)}) \bmod 2 = 0 \text{ or } 1$$

  depending on whether the hinge is $h_1(w)$ or $h_2(w)$.
- The system has a solution (because of acyclicity). In other words, you can determine a vector $x[]$ (of bits) such that, for every $w \in S$,

$$1 + (x[h_1(w)] + x[h_2(w)]) \bmod 2$$

  gives an index $j(w)$ such that th $h_{j(w)}(w)$ are all distinct.

A perfect hash (cont'd)

- In other words, it is possible to write the system so that

$$(x_{h_1(w)} + x_{h_2(w)}) \bmod 2 = 0 \text{ or } 1$$

  depending on whether the hinge is $h_1(w)$ or $h_2(w)$.
- The system has a solution (because of acyclicity). In other words, you can determine a vector $x[]$ (of bits) such that, for every $w \in S$,

$$1 + (x[h_1(w)] + x[h_2(w)]) \bmod 2$$

  gives an index $j(w)$ such that th $h_{j(w)}(w)$ are all distinct.
- In other words, the function $g(w) := h_{j(w)}(w)$ is a perfect hash (not a minimal one).

A perfect hash (cont'd)

- In other words, it is possible to write the system so that

$$(x_{h_1(w)} + x_{h_2(w)}) \bmod 2 = 0 \text{ or } 1$$

depending on whether the hinge is $h_1(w)$ or $h_2(w)$.

- The system has a solution (because of acyclicity). In other words, you can determine a vector $x[]$ (of bits) such that, for every $w \in S$,

$$1 + (x[h_1(w)] + x[h_2(w)]) \bmod 2$$

gives an index $j(w)$ such that th $h_{j(w)}(w)$ are all distinct.

- In other words, the function $g(w) := h_{j(w)}(w)$ is a perfect hash (not a minimal one).
- Just needs m bits (besides the weights for the two hashes!). Actually, $2m$ for hypergraphs (because the possible hinges are three, so 1 bit is not enough).

A minimal perfect hash

- Combining the construction explained (using $2m$ bits) with a bitvector of m bits for ranking, we obtain a minimal perfect hash.

A minimal perfect hash

- Combining the construction explained (using $2m$ bits) with a bitvector of m bits for ranking, we obtain a minimal perfect hash.
- This structure uses $2m + m + o(m) = 3m + o(m) = 3.63n$ bits, plus the (constant) bits needed to store the hash functions h_1 and h_2.

A minimal perfect hash

- Combining the construction explained (using $2m$ bits) with a bitvector of m bits for ranking, we obtain a minimal perfect hash.
- This structure uses $2m + m + o(m) = 3m + o(m) = 3.63n$ bits, plus the (constant) bits needed to store the hash functions h_1 and h_2.
- Note that it is minimal and perfect, but *not* order preserving.

All in all...

- We can store a *perfect hash* (ph) in $1.21n$ bits

All in all...

- We can store a *perfect hash* (ph) in $1.21n$ bits
- A *minimal perfect hash* (mph) in $3.63n$ bits

All in all...

- We can store a *perfect hash* (ph) in $1.21n$ bits
- A *minimal perfect hash* (mph) in $3.63n$ bits
- An *order-preserving minimal perfect hash* (opmph) in $1.21n \log n$ bits

All in all. . .

- We can store a *perfect hash* (ph) in $1.21n$ bits
- A *minimal perfect hash* (mph) in $3.63n$ bits
- An *order-preserving minimal perfect hash* (opmph) in $1.21n \log n$ bits
- An arbitrary r-bit-valued function in $1.21nr$ bits

MapReduce primer

MapReduce is Google programming model to process large datasets. Although originally invented at Google, it has since got a number of free implementations, most notably Apache's Hadoop.

MapReduce primer

MapReduce is Google programming model to process large datasets. Although originally invented at Google, it has since got a number of free implementations, most notably Apache's Hadoop.

It is based on two basic steps, called *map* and *reduce*. Both work on bags (i.e., multisets) of key/value pairs:

MapReduce primer

MapReduce is Google programming model to process large datasets. Although originally invented at Google, it has since got a number of free implementations, most notably Apache's Hadoop.

It is based on two basic steps, called *map* and *reduce*. Both work on bags (i.e., multisets) of key/value pairs:

$$\mathrm{map}: \qquad (k, v) \mapsto \{(k_1, v_1), \ldots, (k_n, v_n)\}$$

MapReduce primer

MapReduce is Google programming model to process large datasets. Although originally invented at Google, it has since got a number of free implementations, most notably Apache's Hadoop.

It is based on two basic steps, called *map* and *reduce*. Both work on bags (i.e., multisets) of key/value pairs:

$$
\begin{aligned}
\text{map}: &\quad (k, v) \mapsto \{(k_1, v_1), \ldots, (k_n, v_n)\} \\
\text{reduce}: &\quad (k, \{v_1, \ldots, v_p\}) \mapsto \{(k'_1, v'_1), \ldots, (k'_m, v'_m)\}
\end{aligned}
$$

MapReduce: map phase

Every MapReduce iteration starts with a bag B_0 of key/values; the first step is applying

$$\mathrm{map}: \qquad (k, v) \mapsto \{(k_1, v_1), \ldots, (k_n, v_n)\}$$

to each key/value pair in the input set. The collection of all outputs so obtained is the bag B_1 on which the next phase will be applied.

MapReduce: map phase

Every MapReduce iteration starts with a bag B_0 of key/values; the first step is applying

$$\mathrm{map}: \qquad (k, v) \mapsto \{(k_1, v_1), \ldots, (k_n, v_n)\}$$

to each key/value pair in the input set. The collection of all outputs so obtained is the bag B_1 on which the next phase will be applied.

Note that all maps are independent, so they can be executed by many machines (potentially, one per pair!).

MapReduce: intermediate phase

The key/value pairs obtained in the map phase B_1 are then shuffled: all pairs *with the same key* are put together, and mapped to a single pair whose value component collects all the values

$$(k, v_1), \dots, (k, v_n) \mapsto (k, \{v_1, \dots, v_n\}).$$

The key/value pairs obtained in the map phase B_1 are then shuffled: all pairs *with the same key* are put together, and mapped to a single pair whose value component collects all the values

$$(k, v_1), \ldots, (k, v_n) \mapsto (k, \{v_1, \ldots, v_n\}).$$

The new set of key/value pairs will be called B_1'.

MapReduce: reduce phase

The key/value pairs in the set B'_1 (of the form $(k, \{v_1, \ldots, v_n\})$ because of the intermediate phase) are then passed to the reduce function

MapReduce: reduce phase

The key/value pairs in the set B'_1 (of the form $(k, \{v_1, \dots, v_n\})$ because of the intermediate phase) are then passed to the reduce function

$$\text{reduce}: \qquad (k, \{v_1, \dots, v_p\}) \mapsto \{(k'_1, v'_1), \dots, (k'_m, v'_m)\}$$

The collection of all outputs so obtained is the bag B_2 which is the final output of the MapReduce iteration.

MapReduce: reduce phase

The key/value pairs in the set B_1' (of the form $(k, \{v_1, \dots, v_n\})$ because of the intermediate phase) are then passed to the reduce function

$$\text{reduce}: \qquad (k, \{v_1, \dots, v_p\}) \mapsto \{(k_1', v_1'), \dots, (k_m', v_m')\}$$

The collection of all outputs so obtained is the bag B_2 which is the final output of the MapReduce iteration.

Note that all reduces are independent, so they can be executed by many machines (potentially, one per key!).

MapReduce example

A classical example of usage is the construction of inverted indices. Here we show how to count word occurrences in a document collection.

MapReduce example

A classical example of usage is the construction of inverted indices. Here we show how to count word occurrences in a document collection.

Input. Pairs $(\mathrm{docId}, \mathrm{doc})$ where docId is a document id (a number) and doc is a document.

MapReduce example

A classical example of usage is the construction of inverted indices. Here we show how to count word occurrences in a document collection.

Input. Pairs $(\mathrm{docId}, \mathrm{doc})$ where docId is a document id (a number) and doc is a document.

Map. Map maps the pair (x, d) to the set of pairs (w, x), w being a word that occurs in d.

MapReduce example

A classical example of usage is the construction of inverted indices. Here we show how to count word occurrences in a document collection.

Input. Pairs $(\mathrm{docId}, \mathrm{doc})$ where docId is a document id (a number) and doc is a document.

Map. Map maps the pair (x, d) to the set of pairs (w, x), w being a word that occurs in d.

Reduce. Reduce maps the pair $(w, \{d_1, \ldots, d_n\})$ to the single pair (w, n).

Using MapReduce to peel a graph

Peeling a (hyper)graph with MapReduce is possible. You start with the set of edges and count the degree:

Using MapReduce to peel a graph

Peeling a (hyper)graph with MapReduce is possible. You start with the set of edges and count the degree:

Using MapReduce to peel a graph

Peeling a (hyper)graph with MapReduce is possible. You start with the set of edges and count the degree:

$$\mathrm{map}: \qquad (x, y) \mapsto \{(x, y), (y, x)\}$$

Using MapReduce to peel a graph

Peeling a (hyper)graph with MapReduce is possible. You start with the set of edges and count the degree:

$$\begin{aligned}\text{map}: &\quad (x, y) \mapsto \{(x, y), (y, x)\}\\ \text{reduce}: &\quad (x, \{y_1, \dots, y_d\}) \mapsto \{(x, d)\}\end{aligned}$$

Using MapReduce to peel a graph

Peeling a (hyper)graph with MapReduce is possible. You start with the set of edges and count the degree:

$$\begin{aligned} \text{map}: &\quad (x, y) \mapsto \{(x, y), (y, x)\} \\ \text{reduce}: &\quad (x, \{y_1, \ldots, y_d\}) \mapsto \{(x, d)\} \end{aligned}$$

Then you can remove all arcs that insist on a node of degree 1. And proceed. This is obtained as a join (a primitive that can be easily implemented in MapReduce).

Using MapReduce to peel a graph

Peeling a (hyper)graph with MapReduce is possible. You start with the set of edges and count the degree:

$$\begin{aligned} \text{map}: &\quad (x, y) \mapsto \{(x, y), (y, x)\} \\ \text{reduce}: &\quad (x, \{y_1, \ldots, y_d\}) \mapsto \{(x, d)\} \end{aligned}$$

Then you can remove all arcs that insist on a node of degree 1. And proceed. This is obtained as a join (a primitive that can be easily implemented in MapReduce).

Arcs should be peeled in reverse order w.r.t. removal.