Amusing algorithms and data-structures that power Lucene and Elasticsearch

Adrien Grand

Agenda

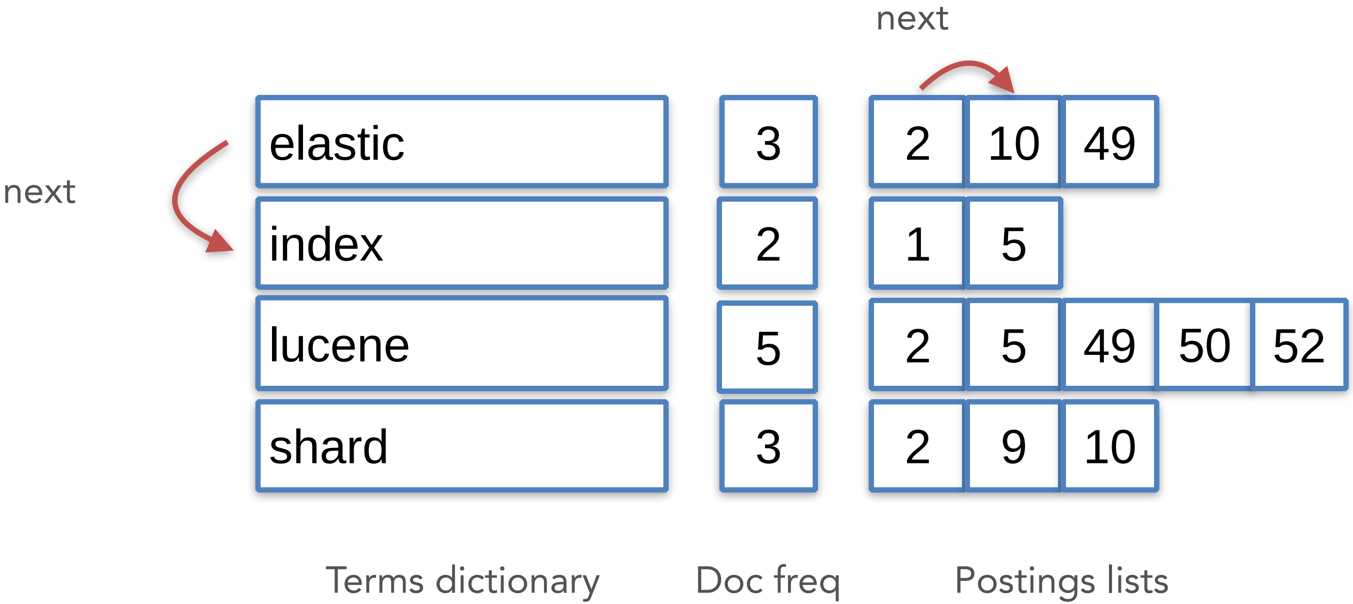
- conjunctions
- regexp queries
- numeric doc values compression
- cardinality aggregation

How are conjunctions implemented?

Inverted index

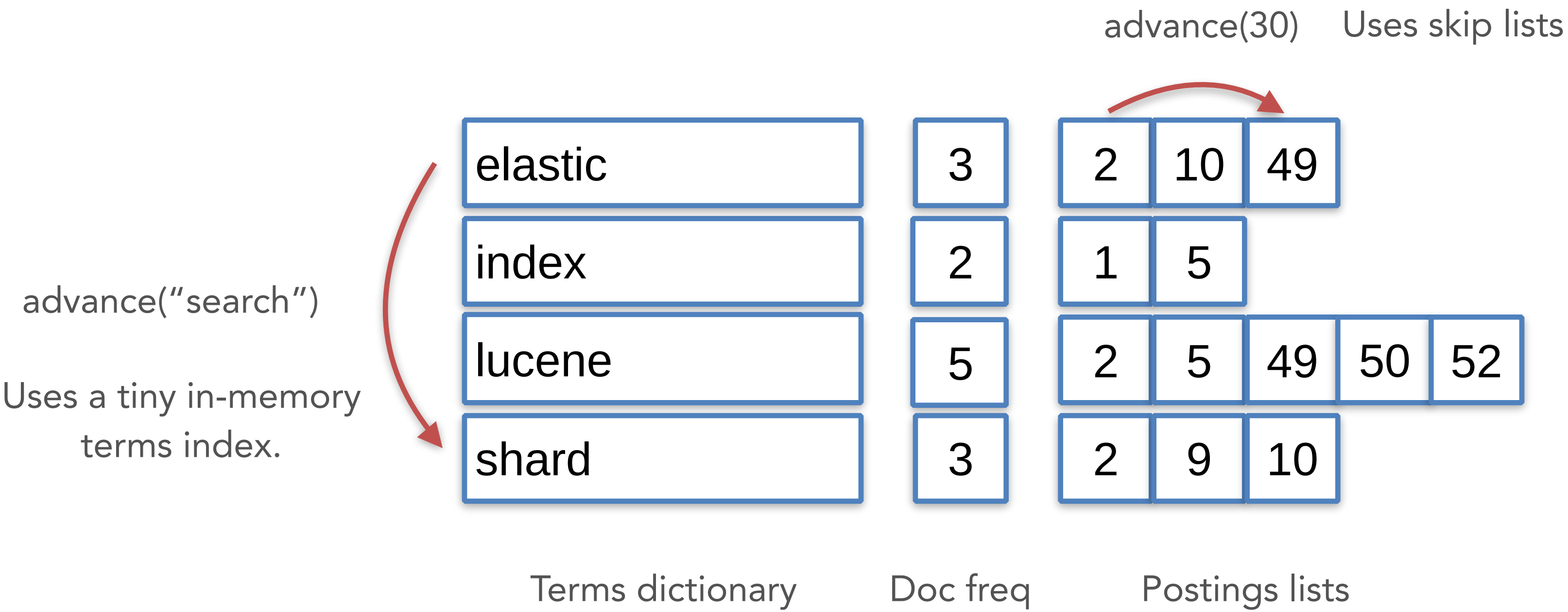
Terms dictionary	Doc freq	Postings lists				
elastic	3	2	10	49		
index	2	1	5			
lucene	5	2	5	49	50	52
shard	3	2	9	10		

Inverted index



Terms dictionary	Doc freq	Postings lists				
elastic	3	2	10	49		
index	2	1	5			
lucene	5	2	5	49	50	52
shard	3	2	9	10		

Inverted index



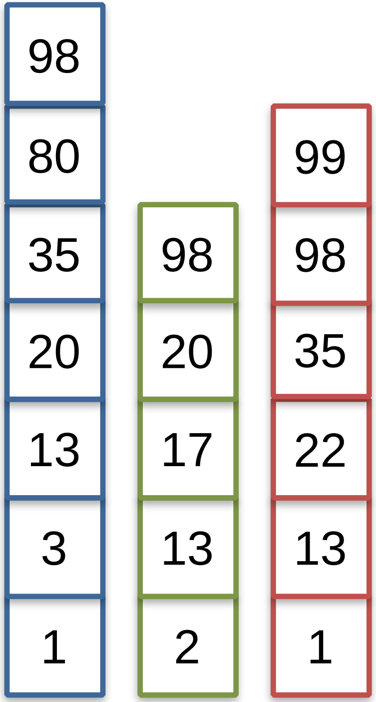
advance(30) Uses skip lists

advance("search")

Uses a tiny in-memory terms index.

Terms dictionary	Doc freq	Postings lists
elastic	3	2 10 49
index	2	1 5
lucene	5	2 5 49 50 52
shard	3	2 9 10

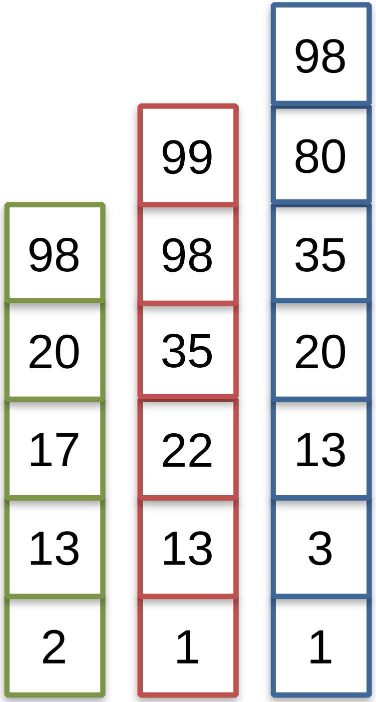
Conjunctions



98		
80		99
35	98	98
20	20	35
13	17	22
3	13	13
1	2	1

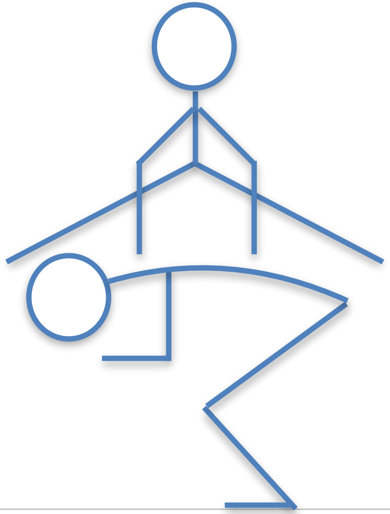
1. Sort by cost

Conjunctions

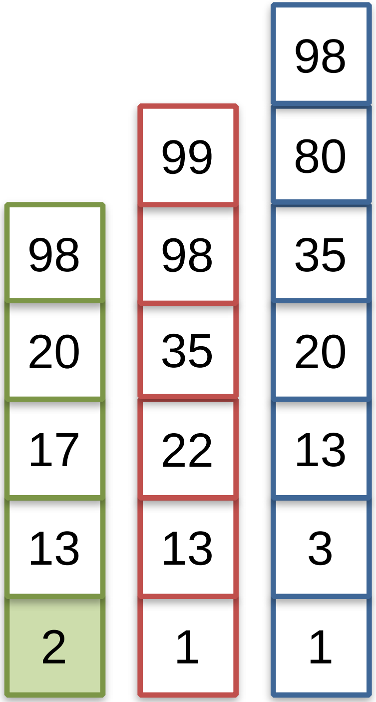


		98
	99	80
98	98	35
20	35	20
17	22	13
13	13	3
2	1	1

1. Sort by cost
2. Leap frog!



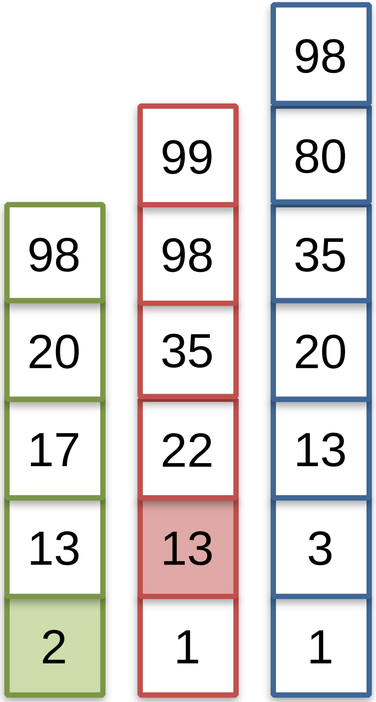
Conjunctions



		98
	99	80
98	98	35
20	35	20
17	22	13
13	13	3
2	1	1

next → 2

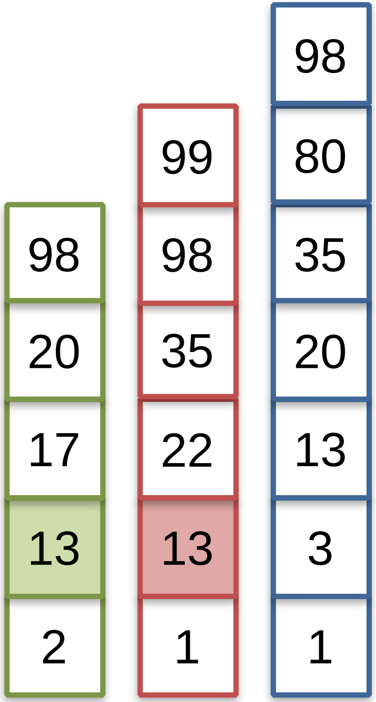
Conjunctions



next → 2

advance(2) → 13 TOO FAR

Conjunctions



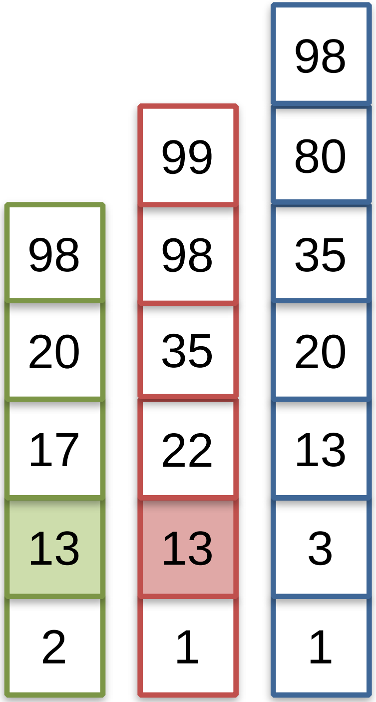
		98
	99	80
98	98	35
20	35	20
17	22	13
13	13	3
2	1	1

next → 2

advance(2) → 13 TOO FAR

advance(13) → 13

Conjunctions



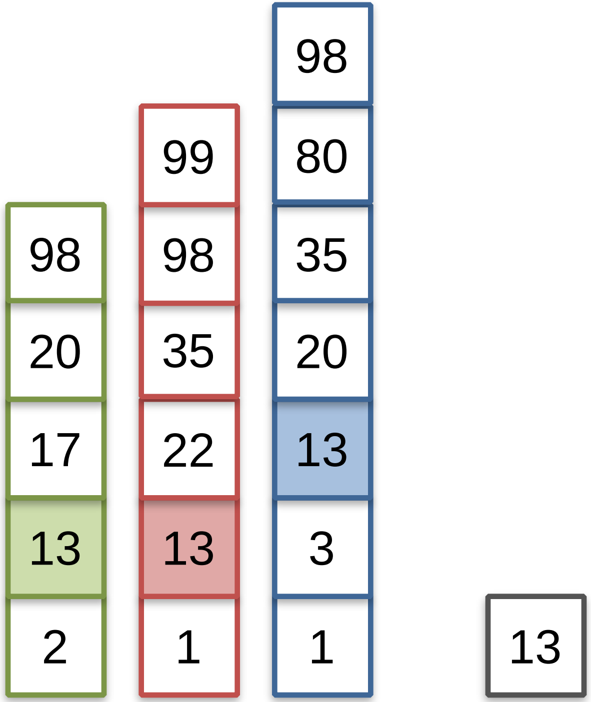
next → 2

advance(2) → 13 TOO FAR

advance(13) → 13

already on 13

Conjunctions



next → 2

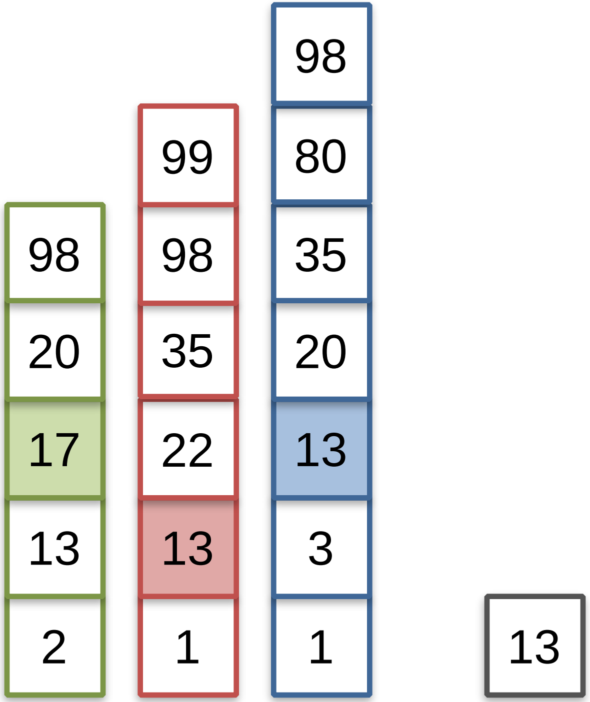
advance(2) → 13 TOO FAR

advance(13) → 13

already on 13

advance(13) → 13 MATCH

Conjunctions



next → 2

advance(2) → 13 TOO FAR

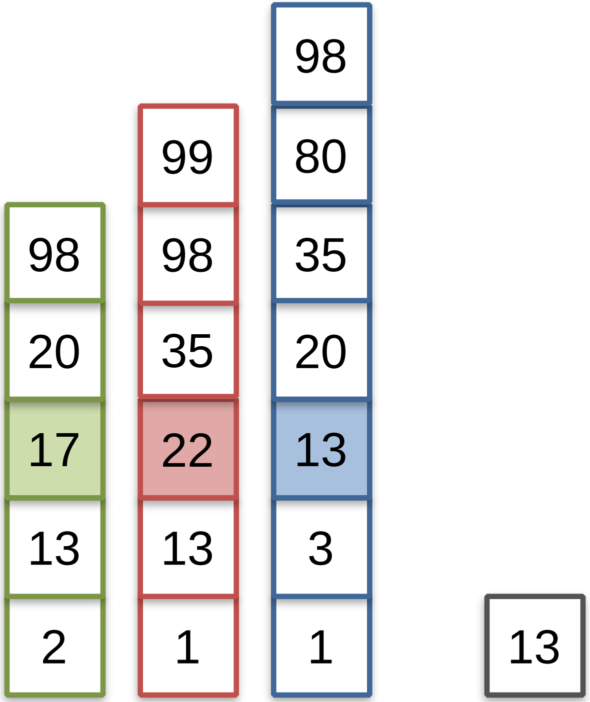
advance(13) → 13

already on 13

advance(13) → 13 MATCH

next → 17

Conjunctions



next → 2

advance(2) → 13 TOO FAR

advance(13) → 13

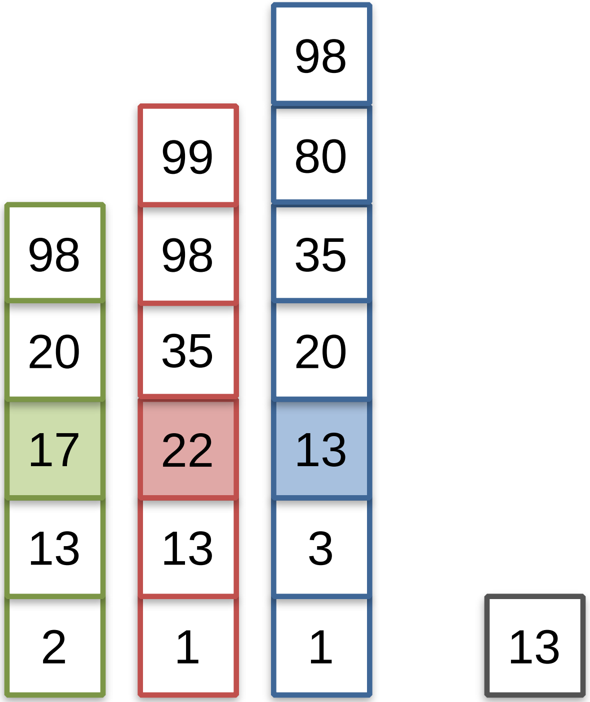
already on 13

advance(13) → 13 MATCH

next → 17

advance(17) → 22 TOO FAR

Conjunctions



		98	
	99	80	
98	98	35	
20	35	20	
17	22	13	
13	13	3	
2	1	1	13

next → 2

advance(2) → 13 TOO FAR

advance(13) → 13

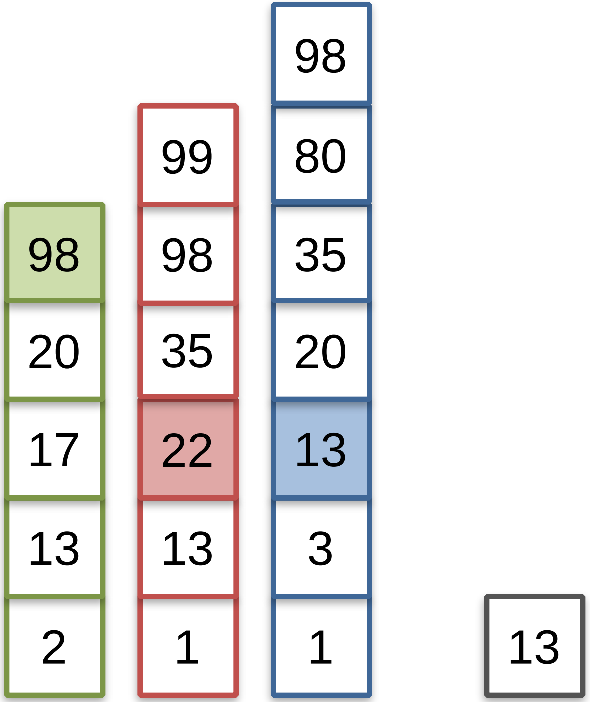
already on 13

advance(13) → 13 MATCH

next → 17

advance(17) → 22 TOO FAR

Conjunctions



next → 2

advance(2) → 13 TOO FAR

advance(13) → 13

already on 13

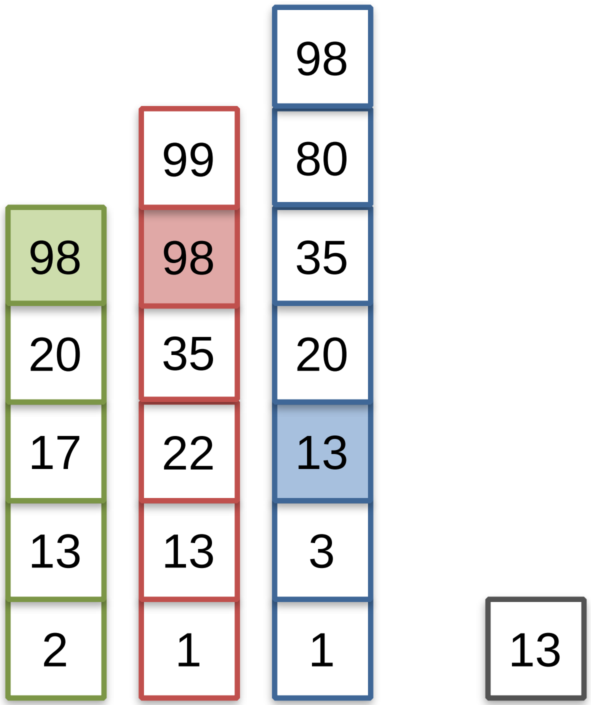
advance(13) → 13 MATCH

next → 17

advance(17) → 22 TOO FAR

advance(22) → 98

Conjunctions



next → 2

advance(2) → 13 TOO FAR

advance(13) → 13

already on 13

advance(13) → 13 MATCH

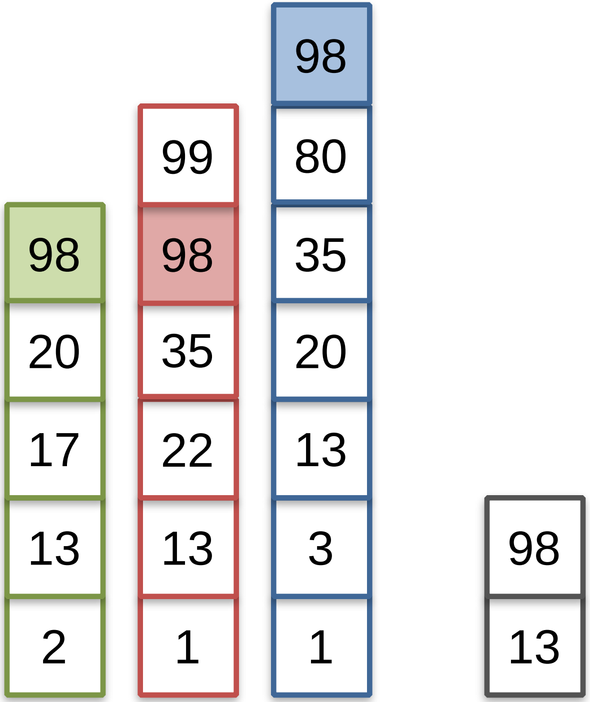
next → 17

advance(17) → 22 TOO FAR

advance(22) → 98

advance(98) → 98

Conjunctions



next → 2

advance(2) → 13 TOO FAR

advance(13) → 13

already on 13

advance(13) → 13 MATCH

next → 17

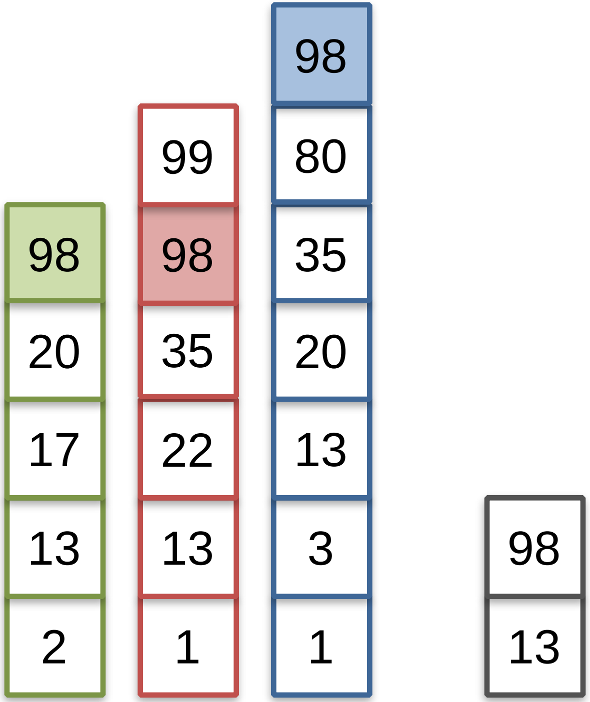
advance(17) → 22 TOO FAR

advance(22) → 98

advance(98) → 98

advance(98) → 98 MATCH

Conjunctions



next → 2	
advance(2) → 13	TOO FAR
advance(13) → 13	
already on 13	
advance(13) → 13	MATCH
next → 17	
advance(17) → 22	TOO FAR
advance(22) → 98	
advance(98) → 98	
advance(98) → 98	MATCH
next → ∞	END

How do regexp queries work?

Regexp queries

elastic	2	10	49	
index	1	5		
lucene	2	5	49	50
search	5	10		
shard	2	9	10	

Challenge: find matching terms and merge postings lists

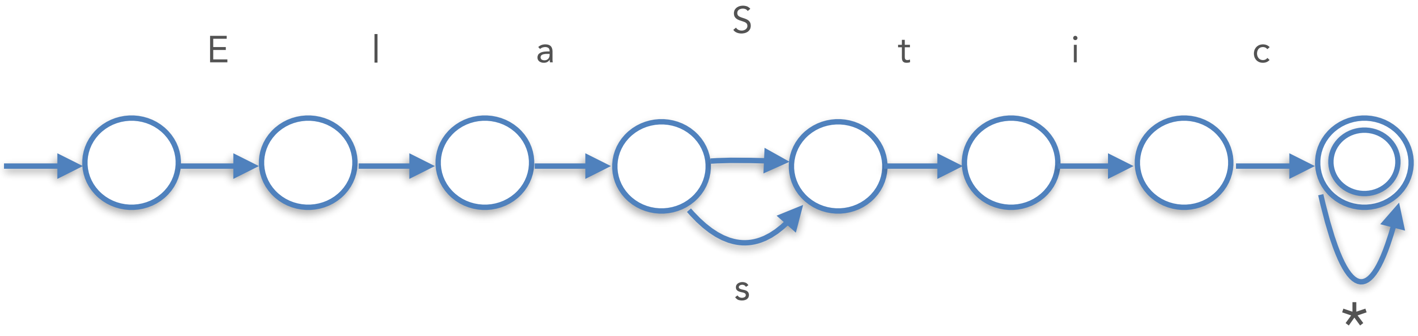
Naive way:

- iterate over terms
- evaluate regexp against every term

SLOWWWWWW

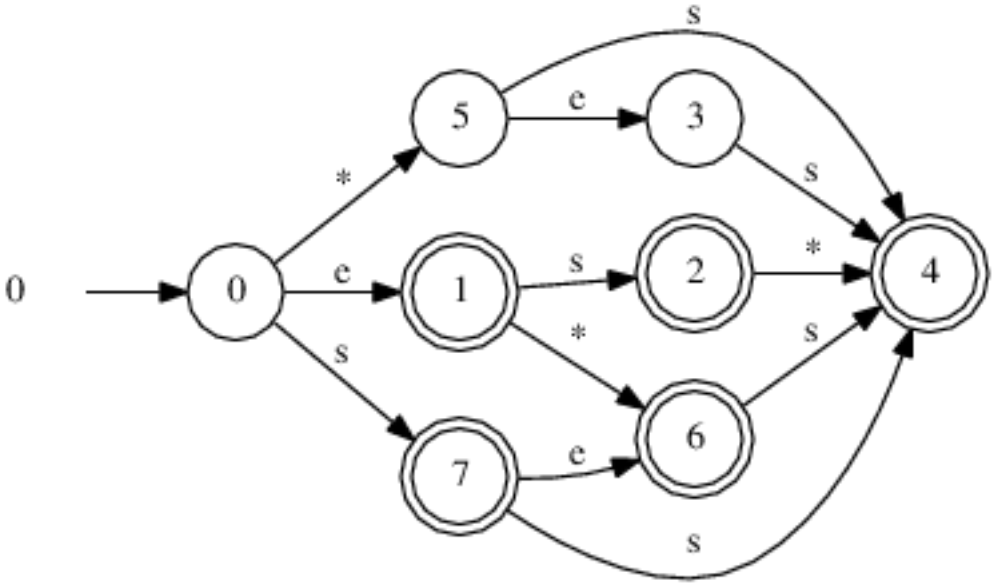
Regexp queries

Ela[Ss]tic.*



Regexp queries

- Not limited to regexps
- Fuzzy queries too!
  - example: es~1



How are numeric doc values compressed?

Aggregation Execution

What is average price of green docs?

"color"	Doc IDs
blue	0
green	1, 4, 5
red	3

(inverted index)

Aggregation Execution

What is average price of green docs?

"color"	Doc IDs
blue	0
green	1, 4, 5
red	3

Doc ID	"price"
0	10
1	20
2	20
3	60
4	60
5	20

(field data)

$$\frac{(20 + 60 + 20)}{3} = 33.33$$

Field Data and Doc Values

Field Data

- In-memory, lives on JVM Heap
- All-or-nothing
- Lazily constructed at query-time

Doc Values

- Disk-based, leverages OS FS cache
- Pages in/out of FS cache
- Precomputed at index-time
    - Allows better compression

“Allows better compression”

Lots of cool tricks, let’s dive in!

Default strategy

Delta Encoding + bit packing

Doc ID	"price"
0	5
1	2
2	2
3	5
4	9
5	4

Encoding

- Compute min and max values:
  - min=2
  - max=9
- max - min = 7
  - so deltas require 3 bits per value
- Encode the min value
- Encode deltas on 3 bits per value:
  - [3,0,0,3,7,2] requires 18 bits

Default strategy

Delta Encoding + bit packing

Doc ID	“price”
0	5
1	2
2	2
3	5
4	9
5	4

Decoding

- For doc ID *d*
  - Read 3 bits at offset *d*3 bits*
  - Add the min value

Less than 256 unique values

Table encoding

Doc ID	"price"
0	5
1	2
2	2
3	5
4	9
5	4

Encoding

- Dedup and sort values:
  - [2, 4, 5, 9]
- Encode the table index for every doc:
  - [2, 0, 0, 2, 3, 1]
  - 2 bits per value

Less than 256 unique values

Table encoding

Doc ID	"price"
0	5
1	2
2	2
3	5
4	9
5	4

Decoding

- For doc ID *d*:
    - Read 2 bits at offset *d*2 bits*
    - Gives table index *t*
    - Read value in table at index *t*

Timestamps without full precision

GCD compression

Doc ID	“price”
0	11
1	31
2	21
3	1
4	71
5	51

Encoding

- Compute minimum value:
  - 1
- Compute deltas:
  - [10, 30, 20, 0, 70, 50]
- Compute GCD of the deltas:
  - 10
- Encode min value and GCD
- Then encode quotients using bit packing
  - [1,3,2,0,7,5]: 3 bits per value

Timestamps without full precision

GCD compression

Doc ID	"price"
0	11
1	31
2	21
3	1
4	71
5	51

Decoding

- For doc ID *d*:
  - Read 3 bits at offset *d*3 bits*
  - Multiply by the GCD
  - Add the min value

How does the Cardinality agg work?

bit-pattern observable magic

Distinct Counts: Naive Solution

Cardinality agg == SELECT Count(Distinct foo)

Distinct Counts: Naive Solution

Maintain a set of all values

blue
green
red
…

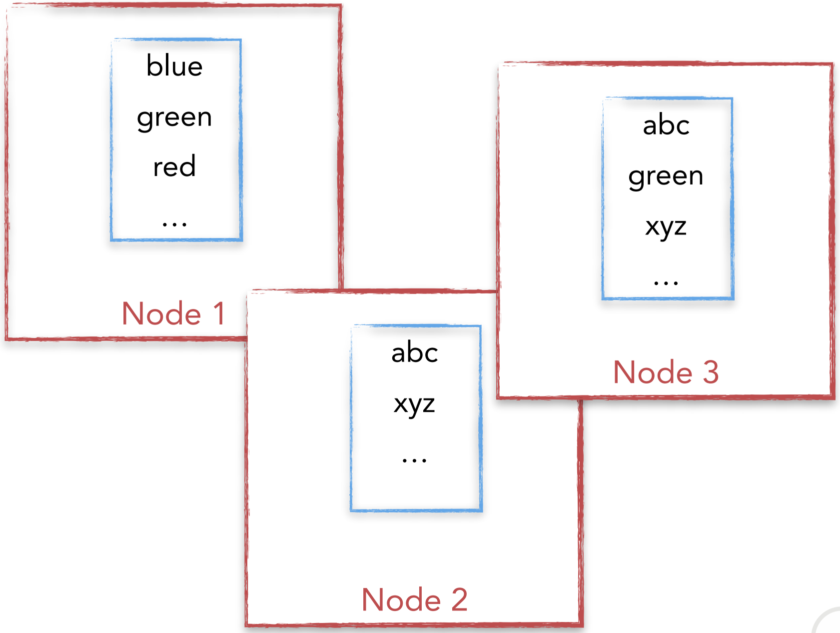
- Cardinality == set.size()

- set.size() == n
- Memory usage == n * size of each term

(Ignoring set overhead)

Distinct Counts: Naive Solution

Gets worse in distributed environment



blue
green
red
...

Node 1

abc
xyz
...

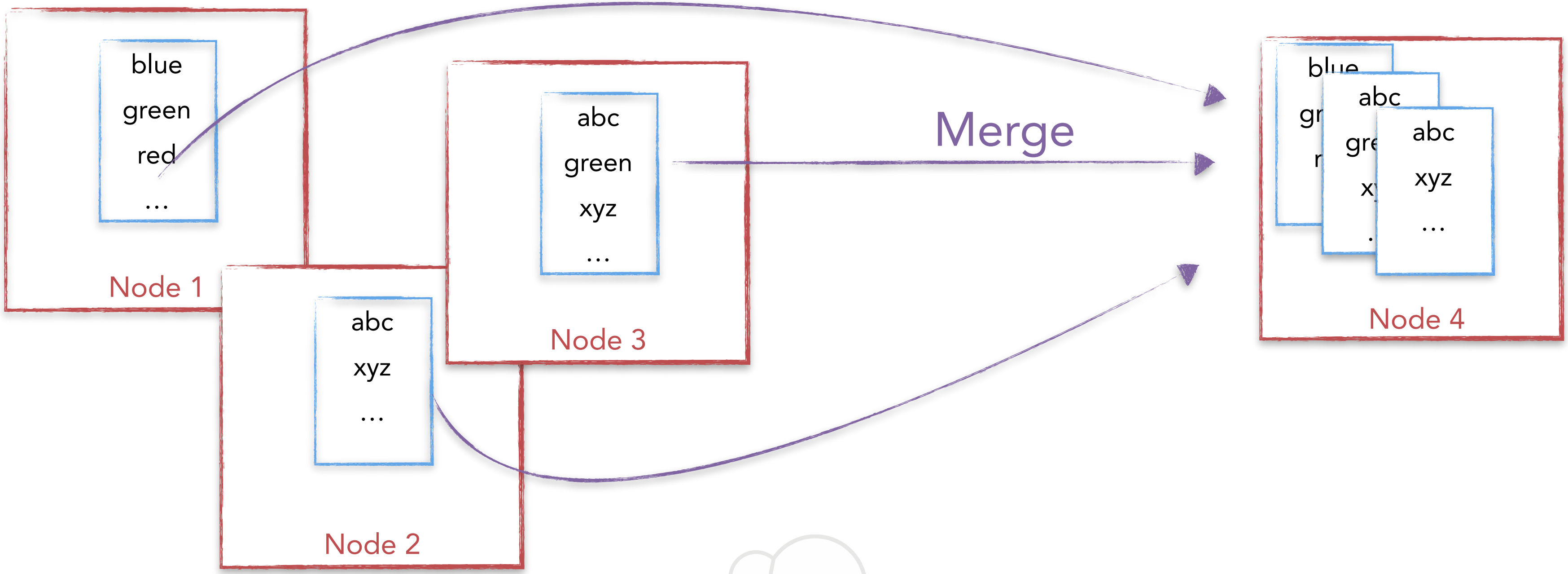
Node 2

abc
green
xyz
...

Node 3

Distinct Counts: Naive Solution

Gets worse in distributed environment



Distinct Counts: HyperLogLog++

Cardinality agg uses HyperLogLog++ instead

- Approximates cardinality
- Uses only a few Kb of memory for billions of distinct values
- Fast!
- Lossless unions

Bit-Observable Patterns

Let's flip some coins…

Probability of a "run"

$$\frac{1}{2^n}$$

Bit-Observable Patterns

Let’s flip some coins…

Probability of a “run”

$$\frac{1}{2^n}$$

5 heads in a row

$$\frac{1}{32}$$

Bit-Observable Patterns

Let’s flip some coins…

Probability of a “run”

$$\frac{1}{2^n}$$

5 heads in a row

$$\frac{1}{32}$$

20 heads in a row

$$\frac{1}{1048576}$$

Bit-Observable Patterns

Let’s flip some coins…

Probability of a “run”

$$\frac{1}{2^n}$$

5 heads in a row

$$\frac{1}{32}$$

20 heads in a row

$$\frac{1}{1048576}$$

Could do this in one sitting

Might take several days

Key Insight:

2^(Length of the run) ~= duration of coin flipping

Bit-Observable Patterns

Let’s hash values, instead of flipping coins…

$$v \quad = \ 12345$$

$$h(v) = \ 11001011111101011010$$

Run of 1 zero

Bit-Observable Patterns

Let’s hash values, instead of flipping coins…

Set “register” to 1

1

$$v = 12345$$

$$h(v) = 11001011111101011010$$

Bit-Observable Patterns

Let's hash values, instead of flipping coins…

$\boxed{1}$

$$v = 3456$$

$$h(v) = 10001101001100111000$$

Run of 3 zeros

Bit-Observable Patterns

Let’s hash values, instead of flipping coins…

$\boxed{3}$

Set “register” to 3

$$v = 3456$$

$$h(v) = 10001101001100111000$$

Bit-Observable Patterns

Let’s hash values, instead of flipping coins…

$$v = 948$$

$$h(v) = 01000111110100110100$$

$\boxed{3}$

Don’t update register

Run of 2 zeros

Key Insight:

cardinality

2^(Length of the run) ~= ~~duration of coin flipping~~

Probability of a “run”

$$\frac{1}{2^n}$$

5 zeros in a row

$$\frac{1}{32}$$

~32 distinct values

20 zeros in a row

$$\frac{1}{1048576}$$

~1048576 distinct values

What if you get unlucky on first value?

$$v = 938$$

$$h(v) = 0000010000000000$$

Run of 10 zeros

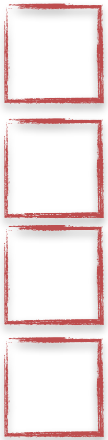
oops :(

Stochastic Averaging

Solution: keep multiple counters

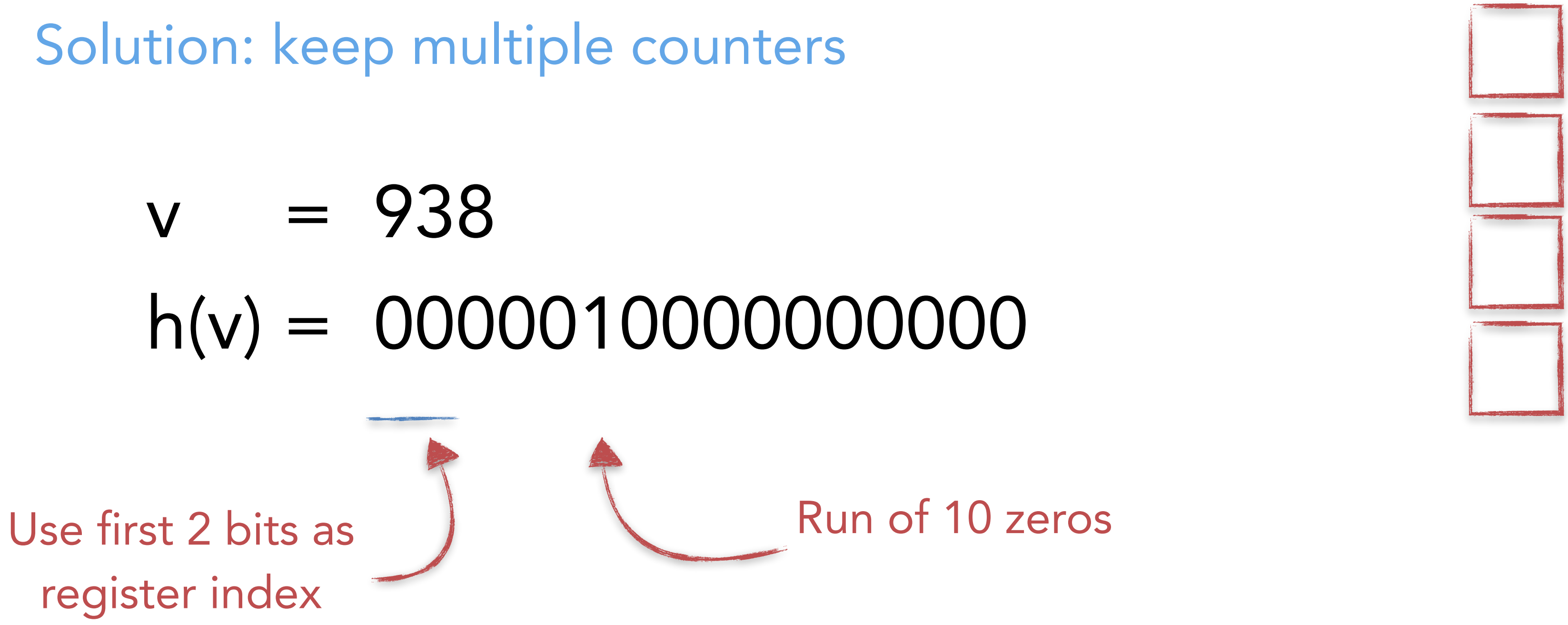
$v = 938$

$h(v) = 0000010000000000$



Stochastic Averaging

Solution: keep multiple counters



$$v = 938$$

$$h(v) = \underline{00}00010000000000$$

Use first 2 bits as register index

Run of 10 zeros

Stochastic Averaging

Solution: keep multiple counters

$$v = 938$$

$$h(v) = \underline{00}00010000000000$$

Set register[0] to 10

Stochastic Averaging

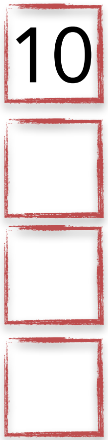
Solution: keep multiple counters

v = 7482

h(v) = 1001110100111010

Use first 2 bits as register index

Run of 1 zero

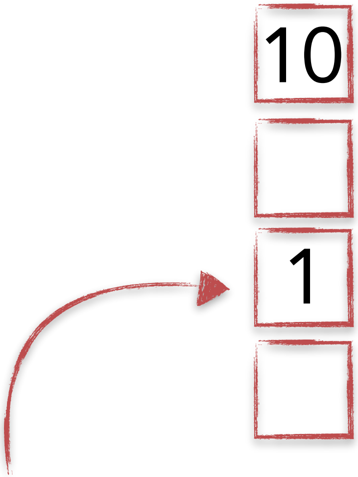


Stochastic Averaging

Solution: keep multiple counters

$$v = 7482$$

$$h(v) = \underline{10}01110100111010$$



Set register[2] to 1

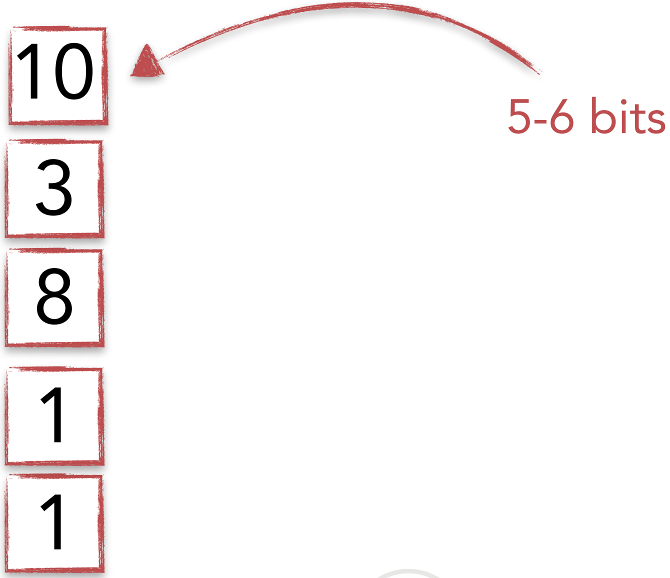
Stochastic Averaging

10	2^10=1024
2	2^2=4
3	2^3=8
1	2^1=2

- Naive approach: sum
  - 1024 + 4 + 8 + 2
  - = 1038
- In practice harmonic mean times number of registers works better:
  - 4 * 4 / (1/1024 + 1/4 + 1/8 + 1/2)
  - =18.3
  - Gives less weight to outliers

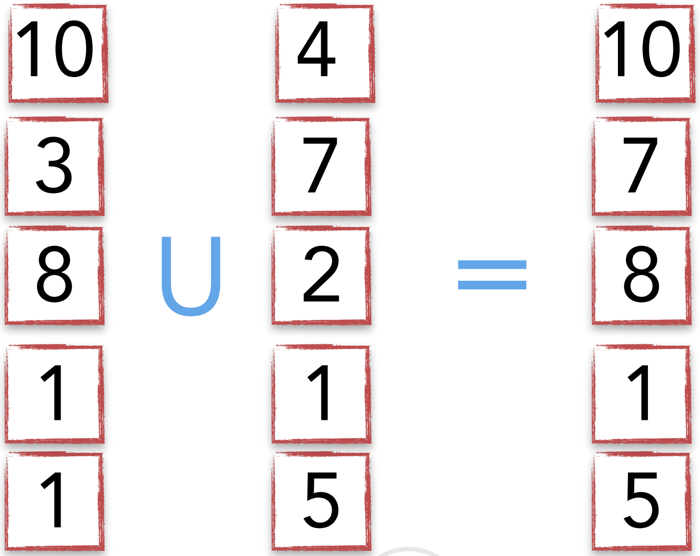
Other neat attributes

Registers are small!



Other neat attributes

Unions are lossless!

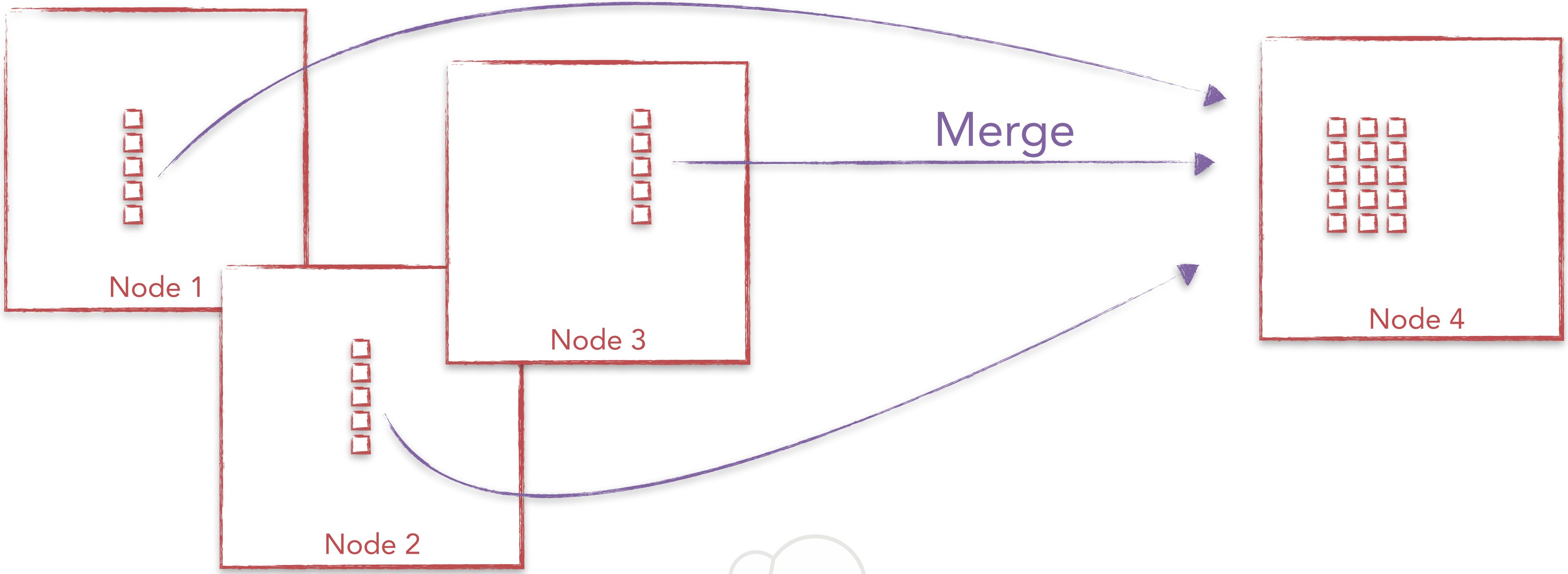


10		4		10
3		7		7
8	U	2	=	8
1		1		1
1		5		5

Take max of each register

Other neat attributes

Which is perfect for distributed environments



In closing…

Stop worrying and learn to love approximate algorithms

Thank you!

@jpountz