Søketek uke 5

Gruppe 1

Recap

Lecture 09.09

- N-gram overlap
- Jaccard coefficient
- BSBI
- SPIMI
- Distributed indexing

Lecture 20.09

- Heaps' law
- Zipf's law
- Dictionary compression
- Front coding
- Posting compression
- Encoding: VB, Gamma, Rice, Golomb, Simple9

Lecture 09.09

N-gram overlap

- Create n-grams of query and compare with other n-grams
- Example: tri-gram of «november»:

nov, ove, vem, emb, mbe, ber

- Can compare with tri-gram of «december»:

dec, ece, cem, emb, mbe, ber

- Is there an overlap?

N-gram overlap

- Create n-grams of query and compare with other n-grams
- Example: tri-gram of «november»:

nov, ove, vem, emb, mbe, ber

- Can compare with tri-gram of «december»:

dec, ece, cem, emb, mbe, ber

- Is there an overlap?
- **Yes!**
- **emb**, **mbe**, and **ber** occur in both

Jaccard coefficient

- A way of measuring the overlap between two n-grams
- Will be a number between 0 and 1
    - 1 means full overlap, 0 means no overlap

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Jaccard coefficient

1. Put the n-grams in separate **sets**
   - A = {nov, ove, vem, emb, mbe, ber}
   - B = {dec, ece, cem, emb, mbe, ber}

Jaccard coefficient

1. Put the n-grams in separate **sets**
   - A = {nov, ove, vem, emb, mbe, ber}
   - B = {dec, ece, cem, emb, mbe, ber}

2. Take the intersection of A and B, and the union of A and B
   - **AB_intersection** = {emb, mbe, ber}
   - **AB_union** = {nov, ove, vem, emb, mbe, ber, dec, ece, cem}

Jaccard coefficient

1. Put the n-grams in separate **sets**
   - A = {nov, ove, vem, emb, mbe, ber}
   - B = {dec, ece, cem, emb, mbe, ber}

2. Take the intersection of A and B, and the union of A and B
   - **AB_intersection** = {emb, mbe, ber}
   - **AB_union** = {nov, ove, vem, emb, mbe, ber, dec, ece, cem}

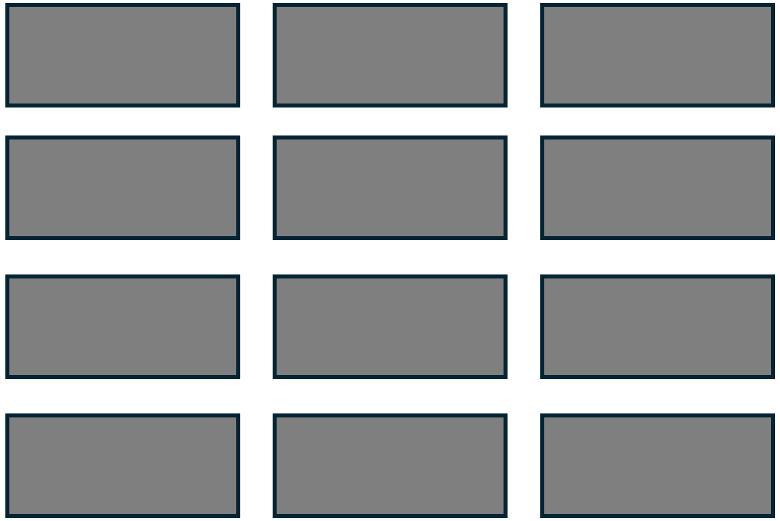
3. Divide the **length** of AB_intersection on the **length** of AB_union
   - jc = len(AB_intersection) / len(AB_union)
   - jc = 3 / 9
   - jc = 0.333...

Index construction

- Problem 1: An inverted index is huge, so we can’t fit the entire thing in main memory
- Problem 2: Disk seeks are expensive

Blocked sort-based indexing

- Break corpus into **blocks** which can approximately fit in main memory

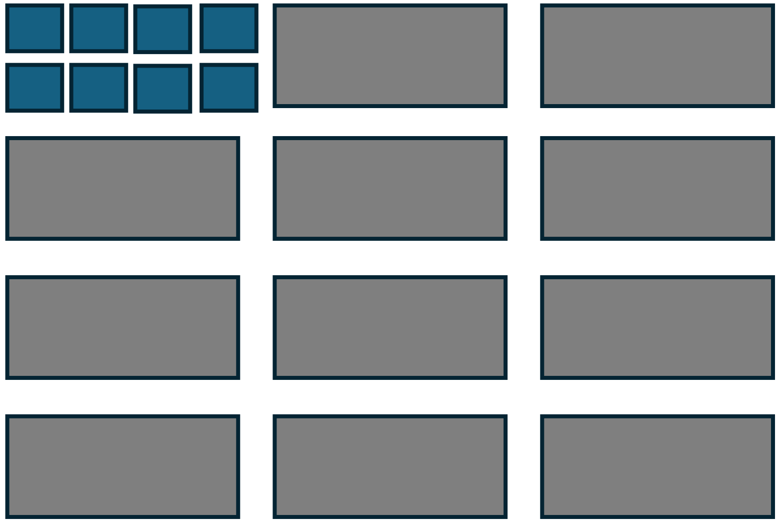


Blocked sort-based indexing

- Read **one block at a time**

Blocked sort-based indexing

- **Tokenise** the documents



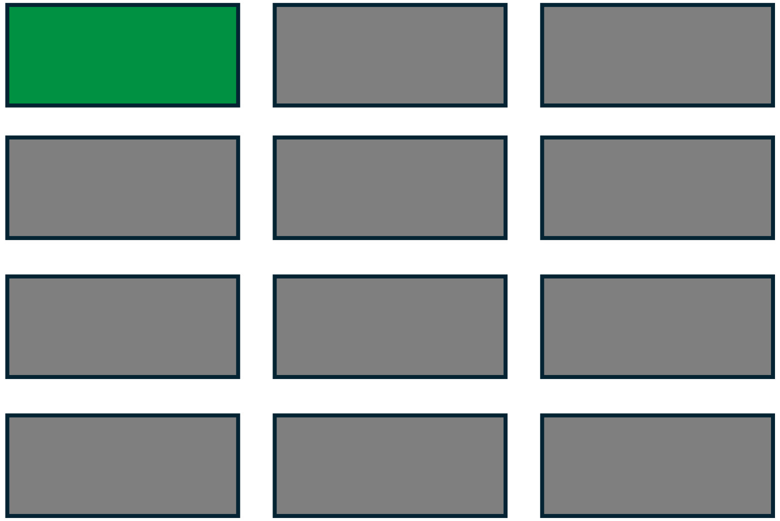
Blocked sort-based indexing

- Create **postings** from all terms, **sort** alphabetically

Term1, doc: 1	Term2, doc: 1	Term3, doc: 1	Term4, doc: 1
Term5, doc: 1	Term6, doc: 2	Term7, doc: 2	Term8, doc: 3

Blocked sort-based indexing

- Write it **back** to disk

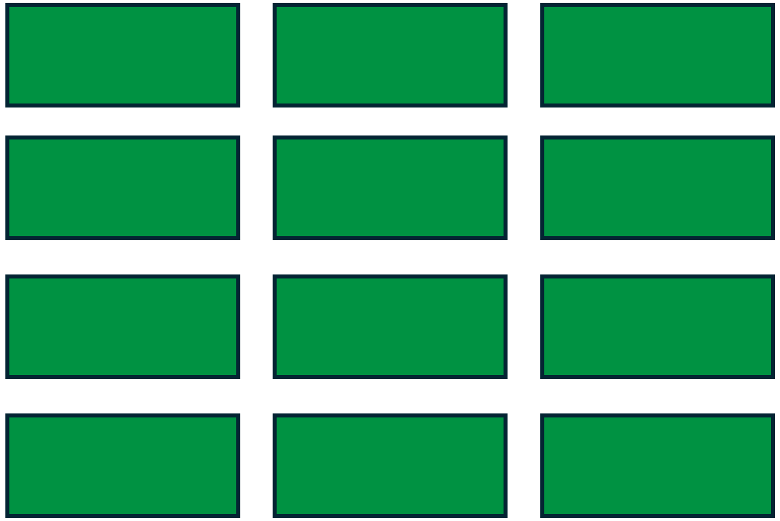


Blocked sort-based indexing

- Start over with a **new** block

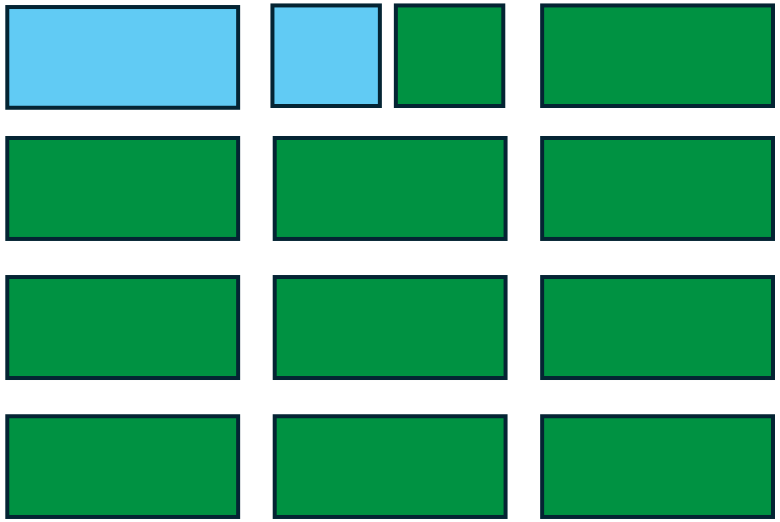
Blocked sort-based indexing

- Continue until **all** blocks are converted



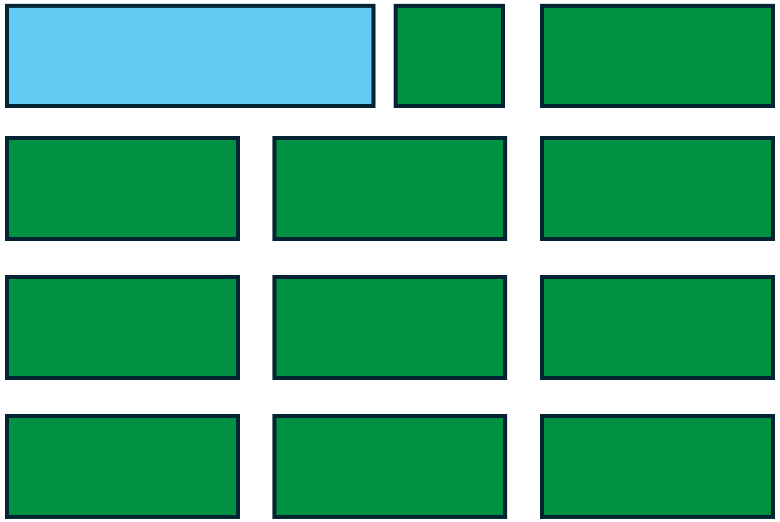
Blocked sort-based indexing

- Read **parts** of blocks from disk



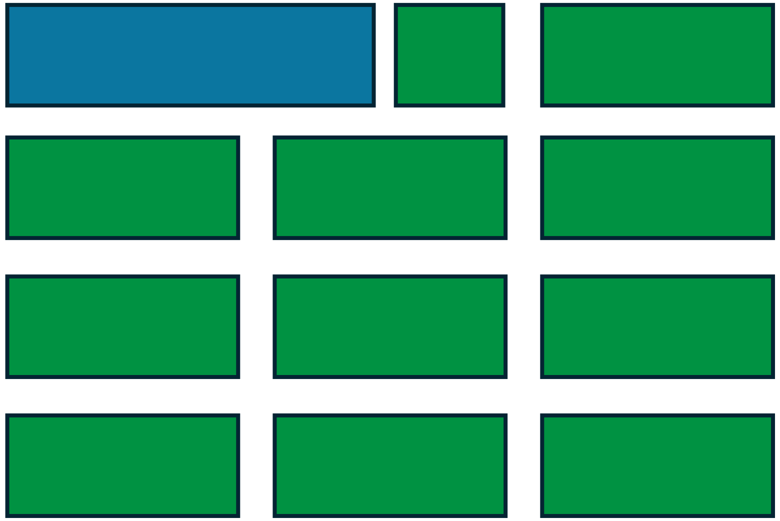
Blocked sort-based indexing

- **Merge** them



Blocked sort-based indexing

- Write them **back** to disk



Blocked sort-based indexing

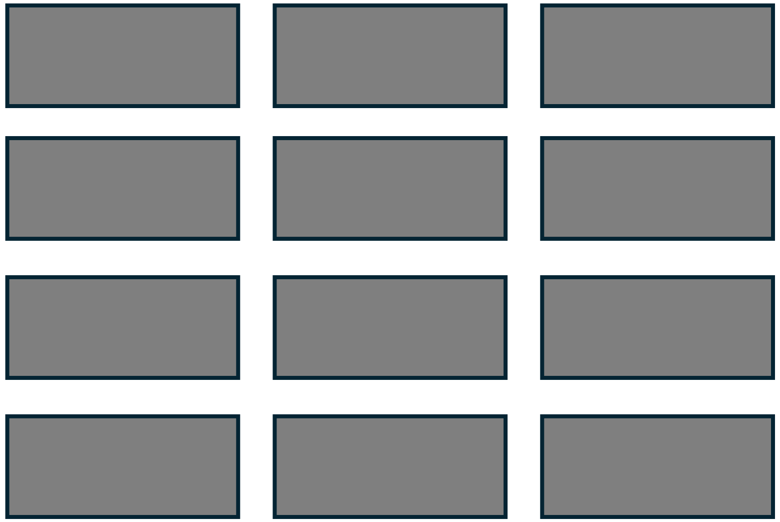
- Continue until **all** the postings are one big inverted index

Single-pass in-memory indexing

- Given a huge corpus

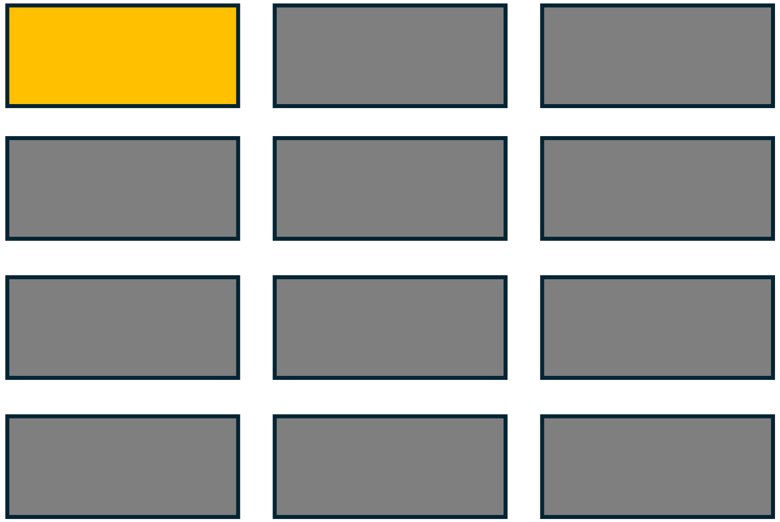
Single-pass in-memory indexing

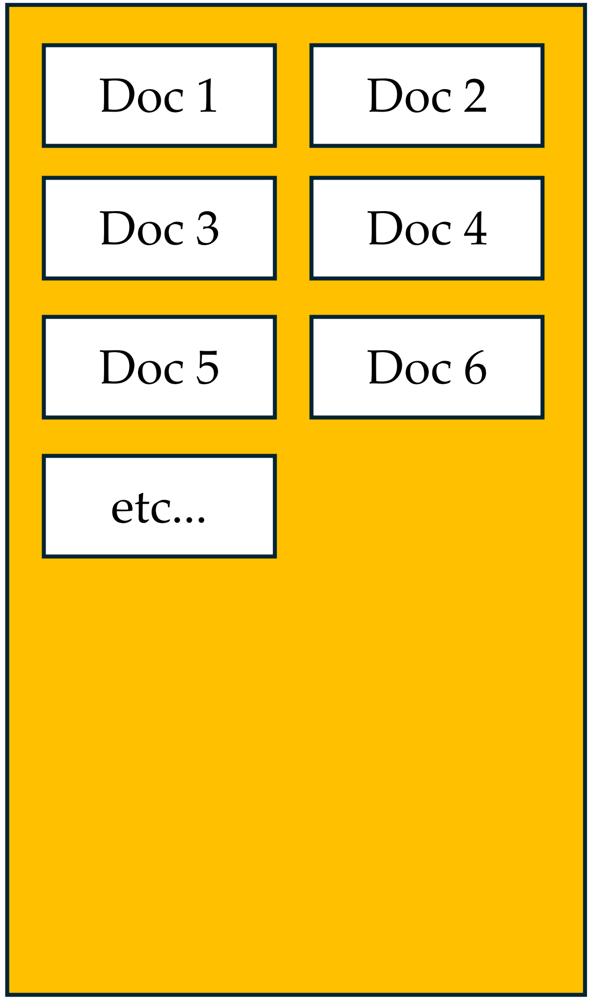
- Break corpus into blocks which can approximately fit in main memory



Single-pass in-memory indexing

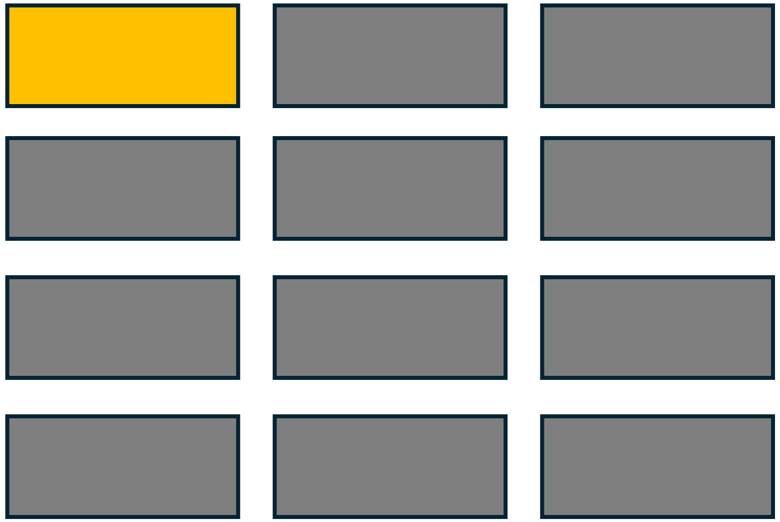
- Read one block at a time

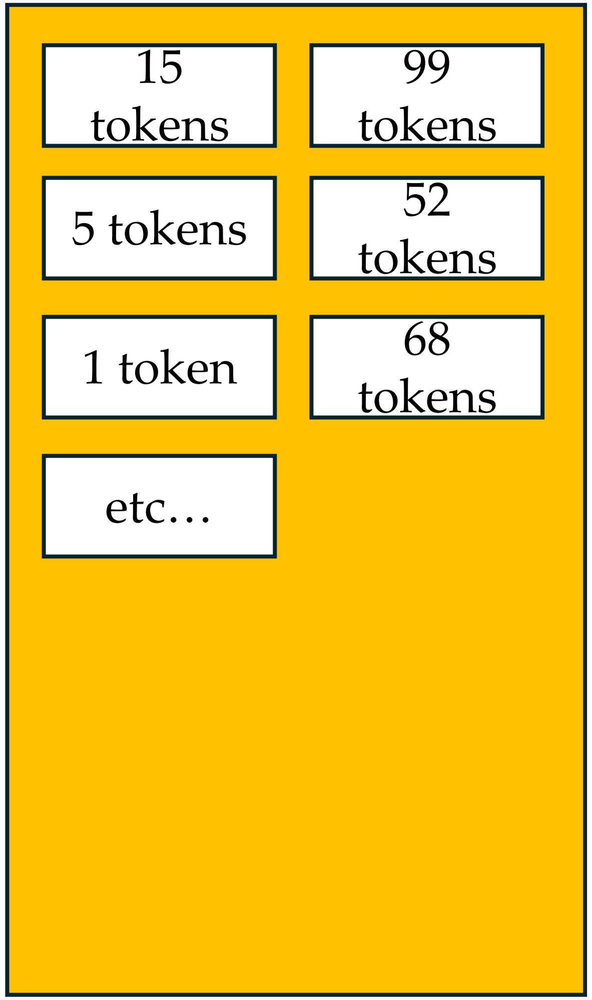




Single-pass in-memory indexing

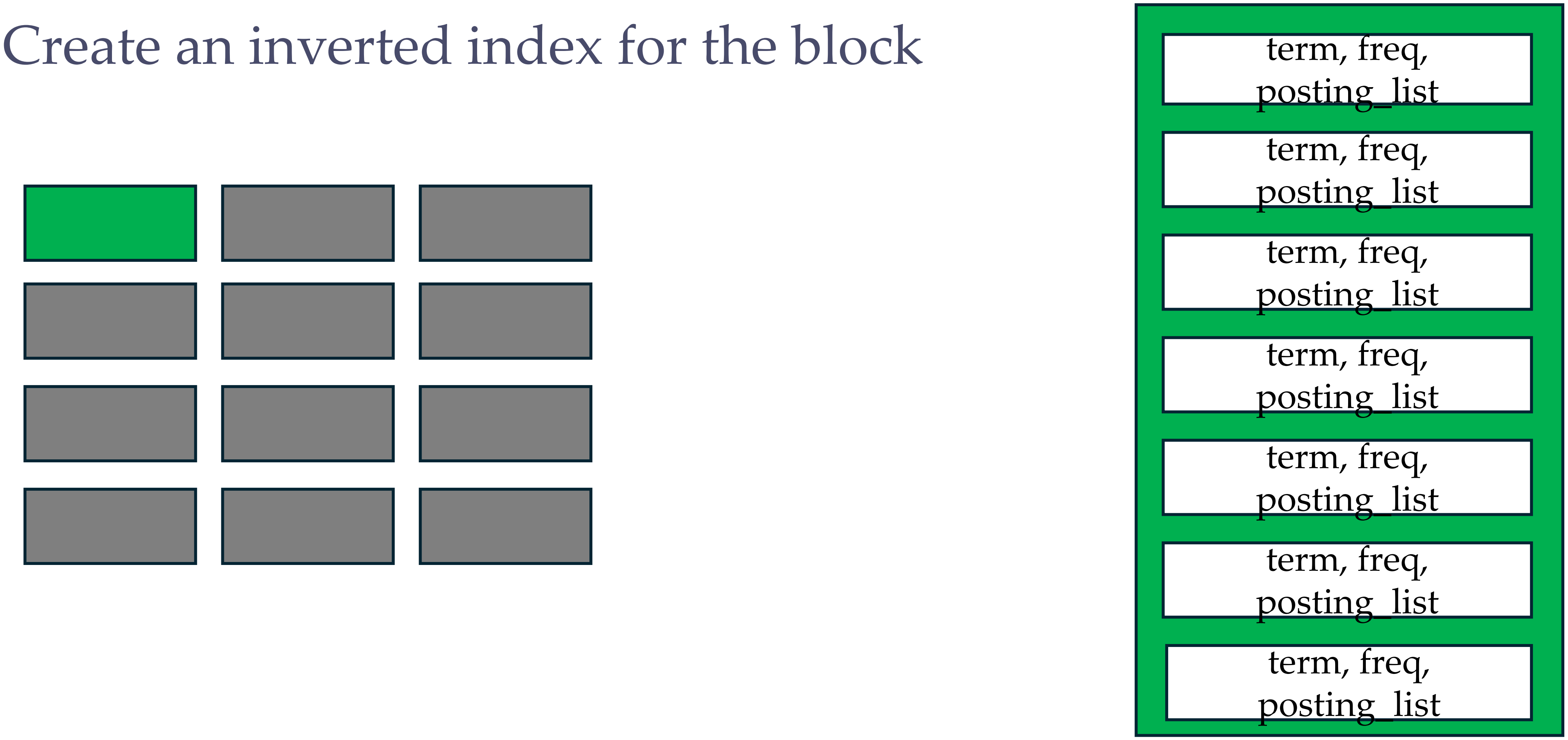
- Tokenise the documents





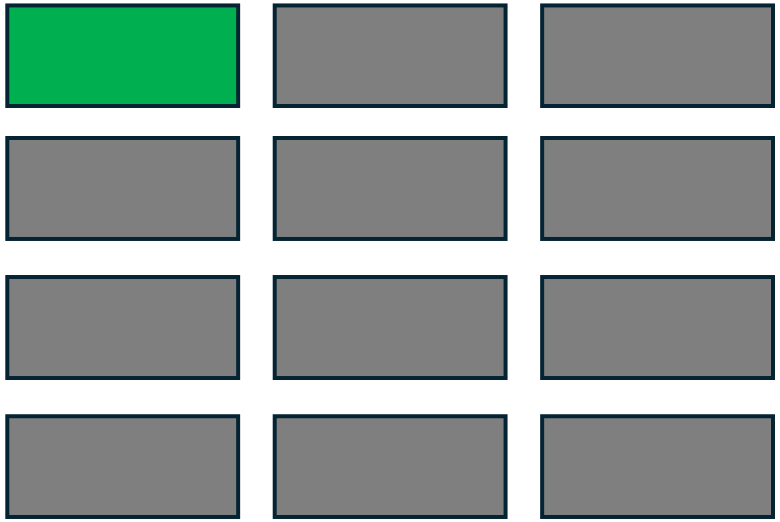
Single-pass in-memory indexing

- Create an inverted index for the block



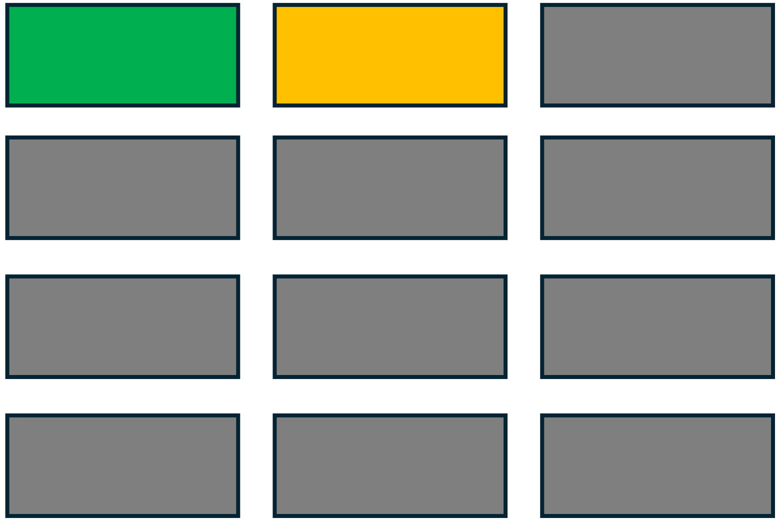
Single-pass in-memory indexing

- Write the block/mini inverted index to disk



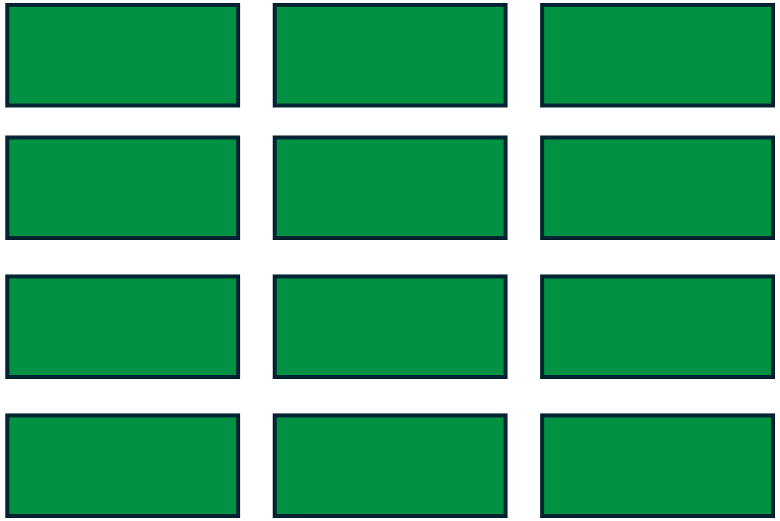
Single-pass in-memory indexing

- Start over with a new block



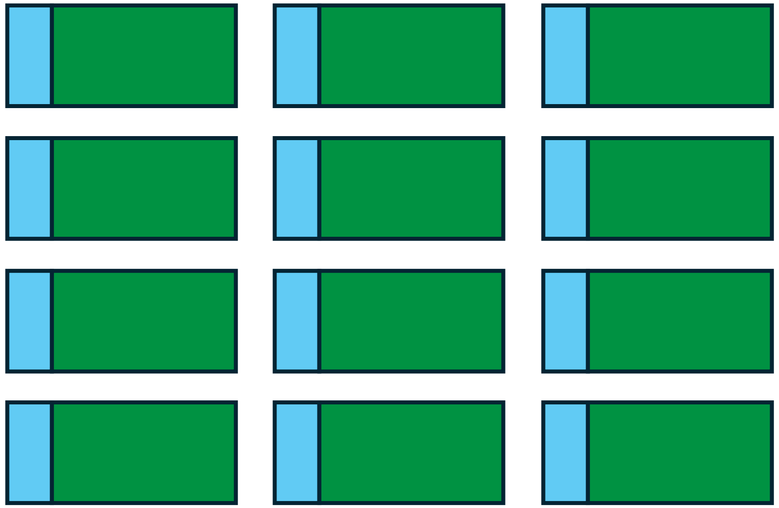
Single-pass in-memory indexing

- Continue until all blocks are converted



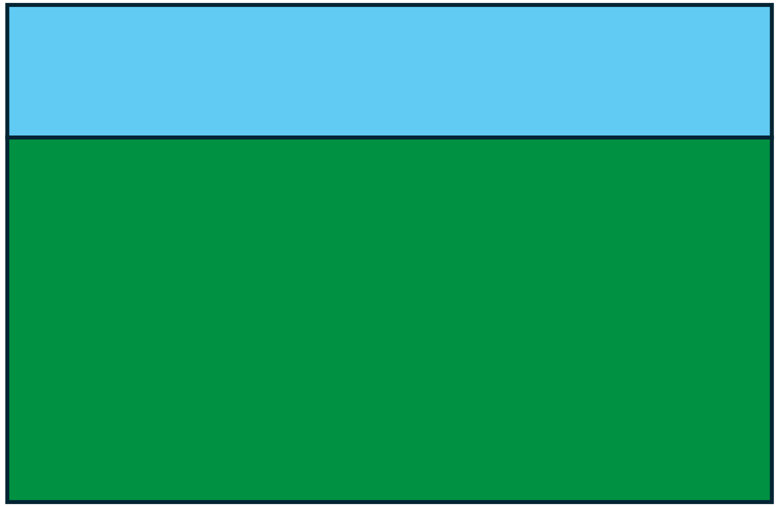
Single-pass in-memory indexing

- Using a k-way merge, merge the blocks to a single inverted index



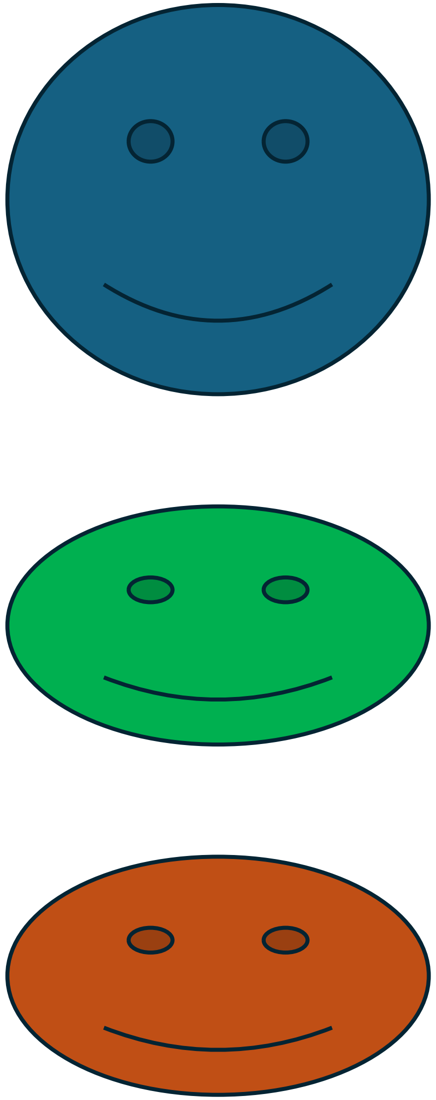
Single-pass in-memory indexing

- We keep some of it in memory and store the rest on disk



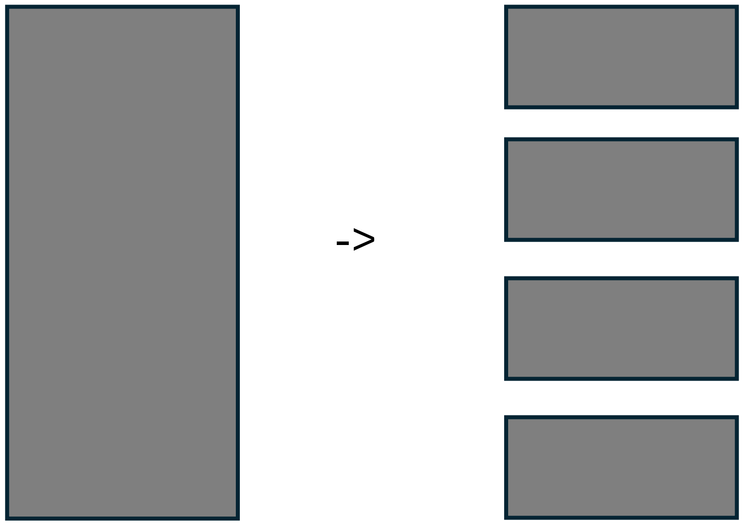
Distributed indexing

- Cannot rely on a single machine
- **Distribute** the tasks
- One **master** machine distributes work
- Multiple **parsers** read from corpus and creates (term, document-ID) pairs
- Multiple **inverters** sort pairs and read them to posting lists



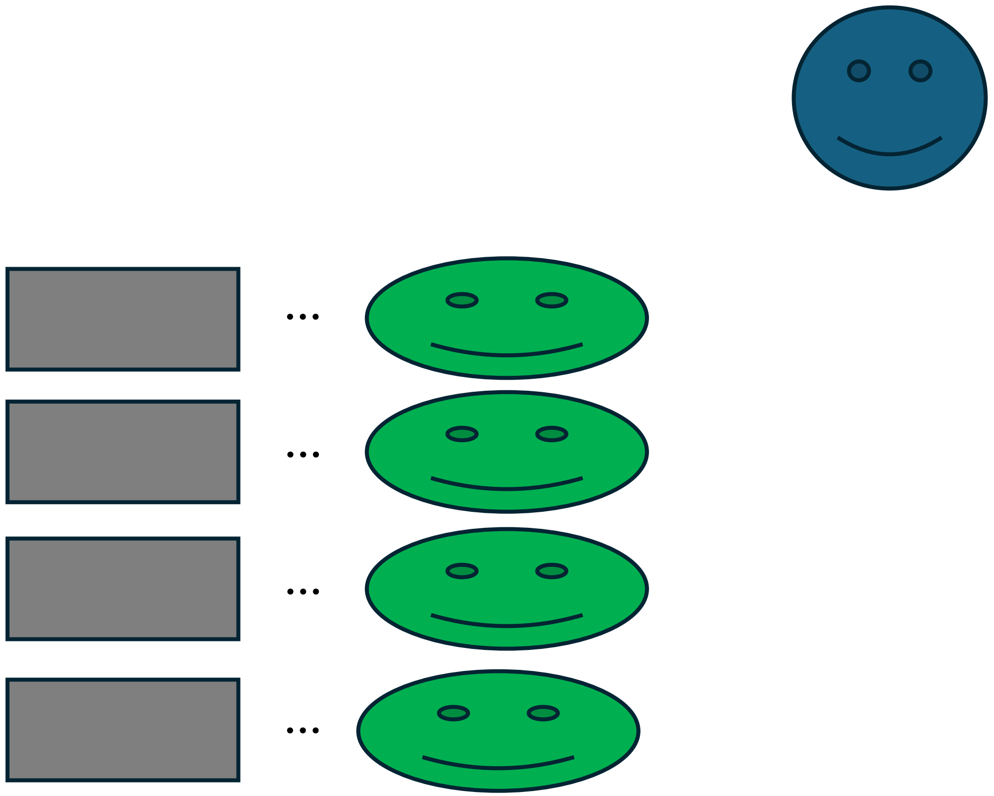
Distributed indexing

- Split corpus into blocks again



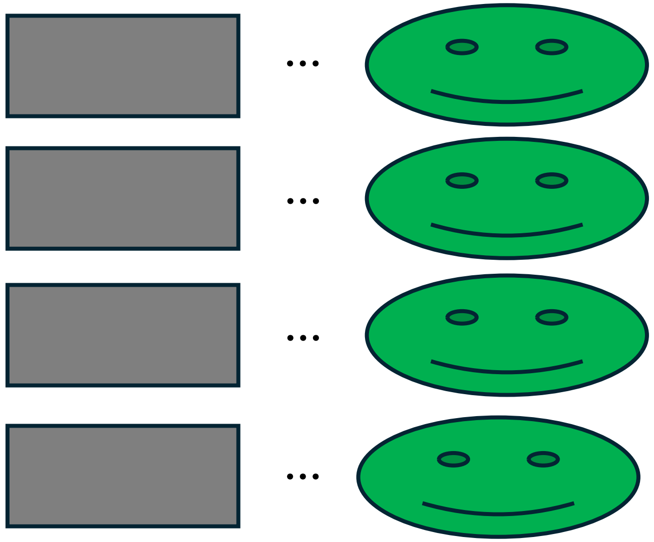
Distributed indexing

- Master machine assigns random splits to random parsers



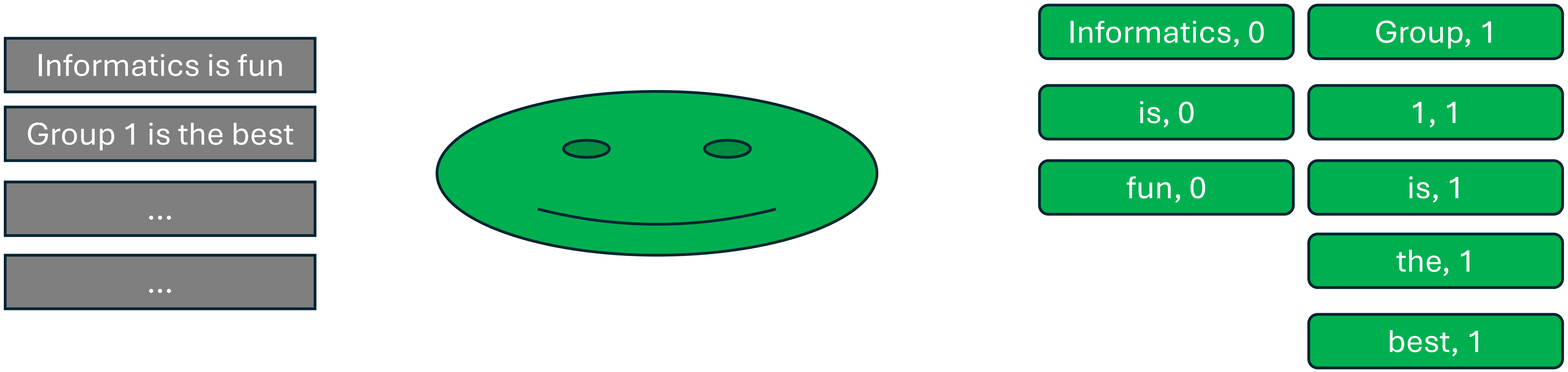
Distributed indexing

- Parsers read one document at a time, and create (term, document-ID) pairs



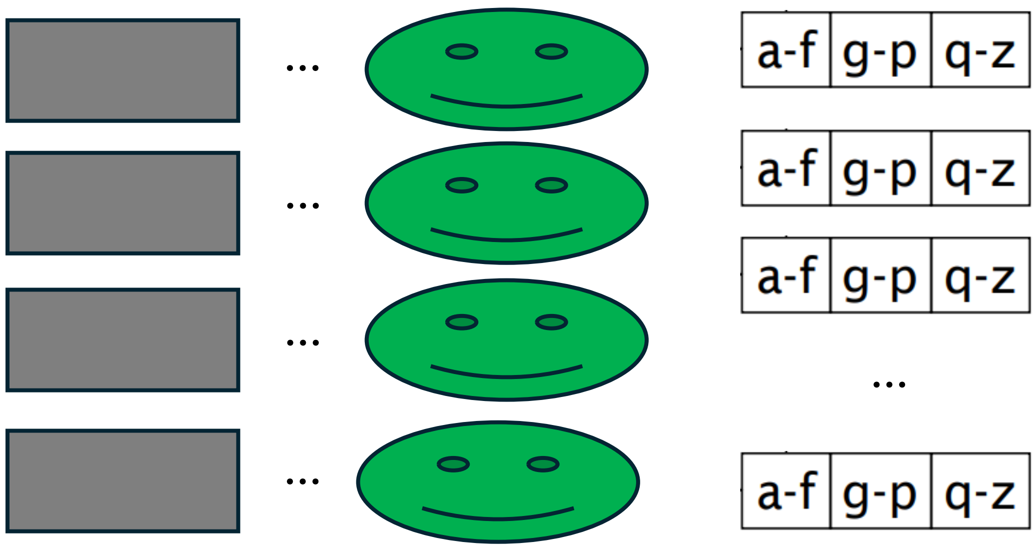
Distributed indexing

- Parsers read one document at a time, and create (term, document-ID) pairs



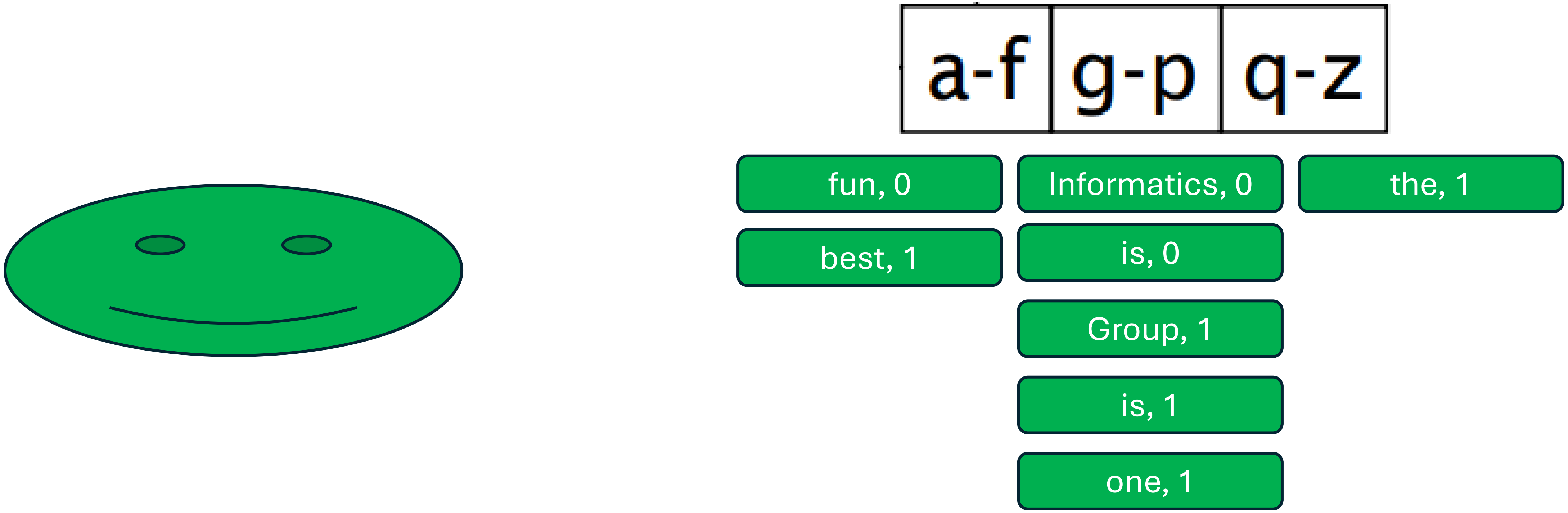
Distributed indexing

- Parsers read one document at a time, and create (term, document-ID) pairs
- Parsers write them to partitions



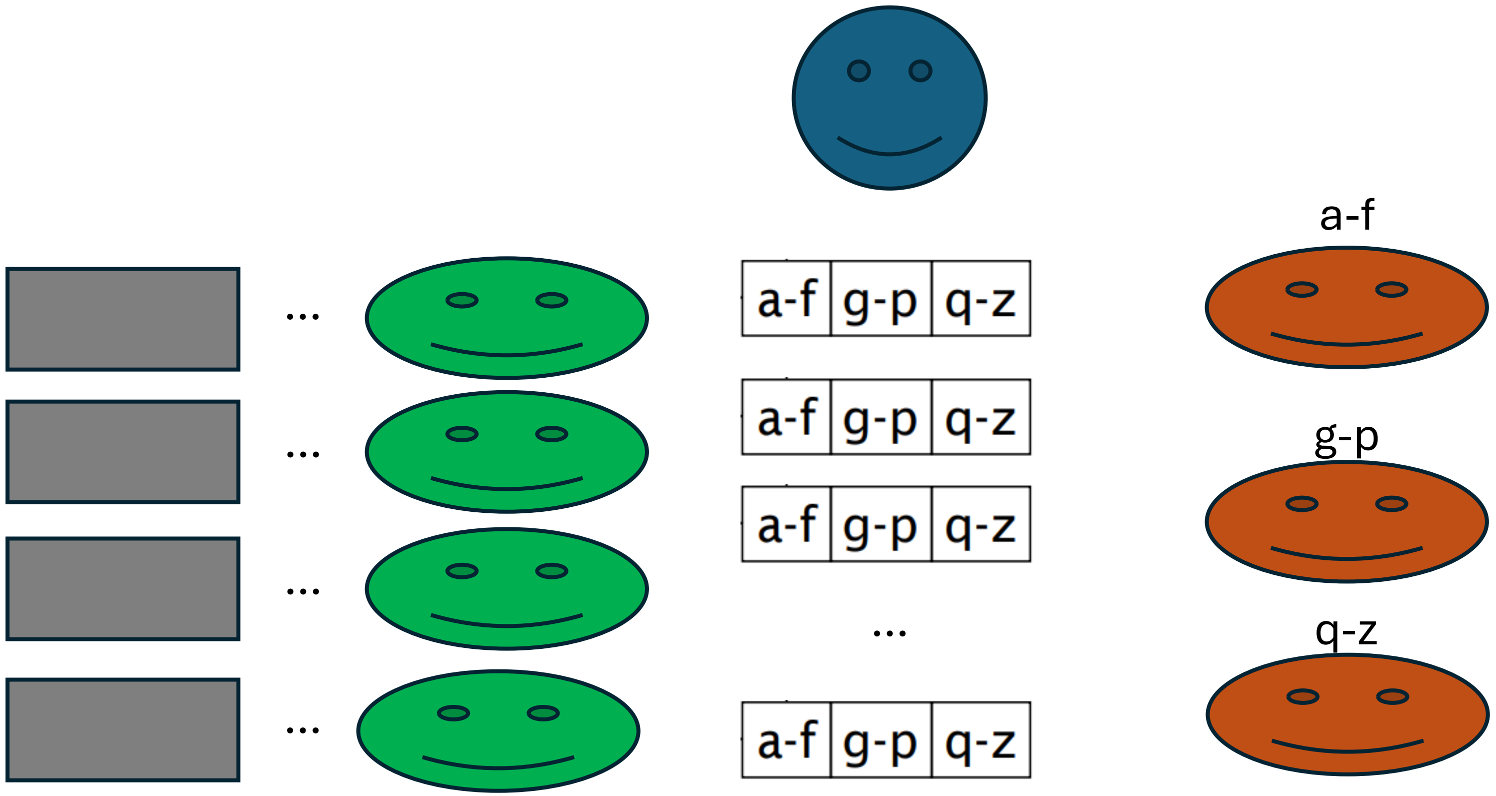
Distributed indexing

- Parsers read one document at a time, and create (term, document-ID) pairs
- Parsers write them to partitions



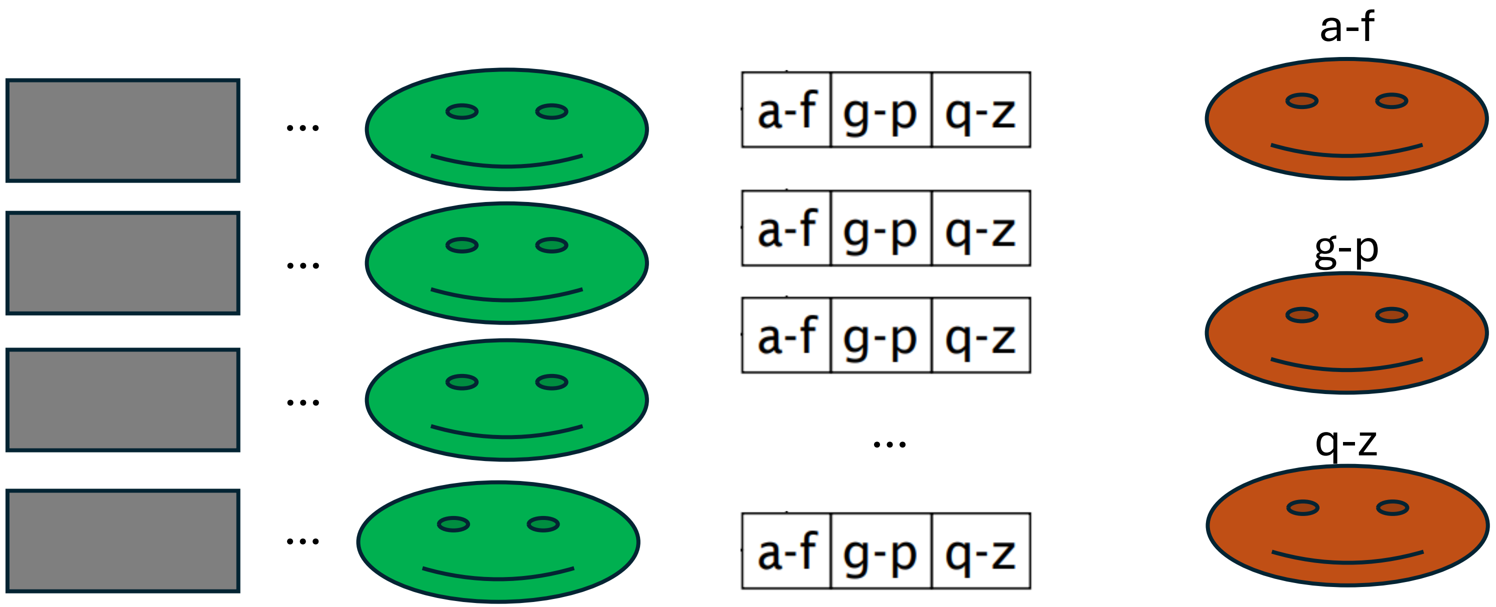
Distributed indexing

- Master machine assigns one partition to each inverter



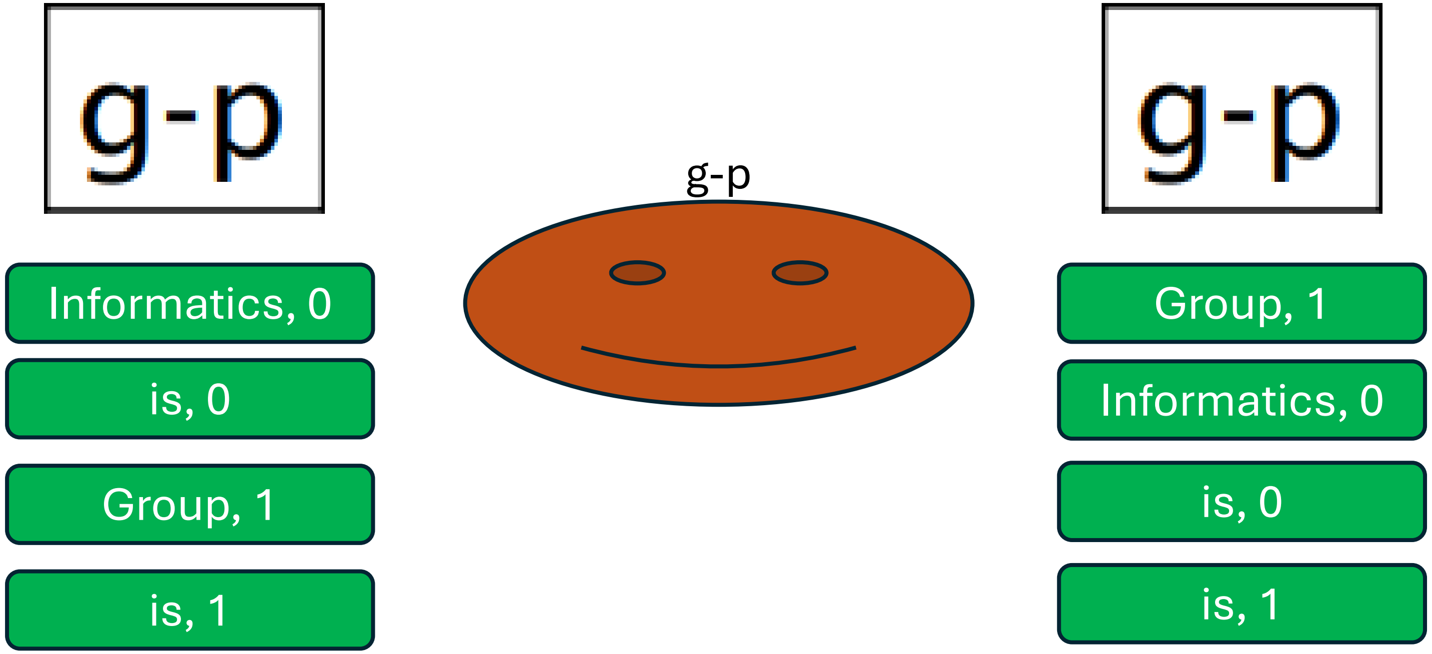
Distributed indexing

- Each inverter sorts their partition alphabetically



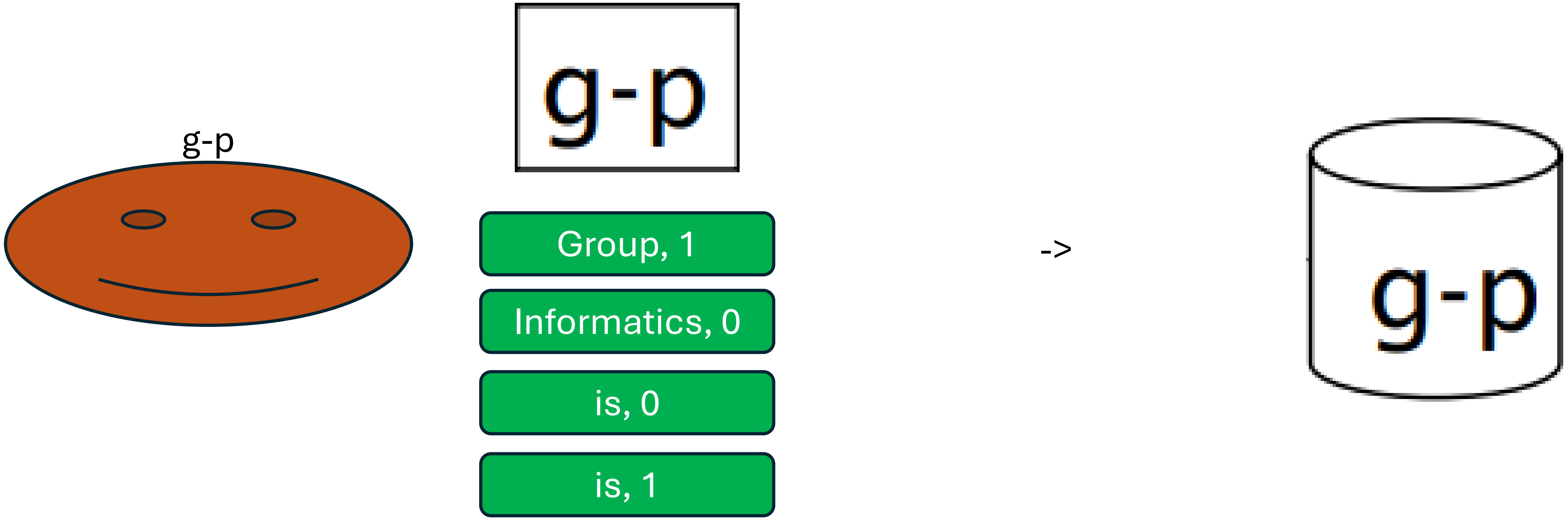
Distributed indexing

- Each inverter sorts their partition alphabetically



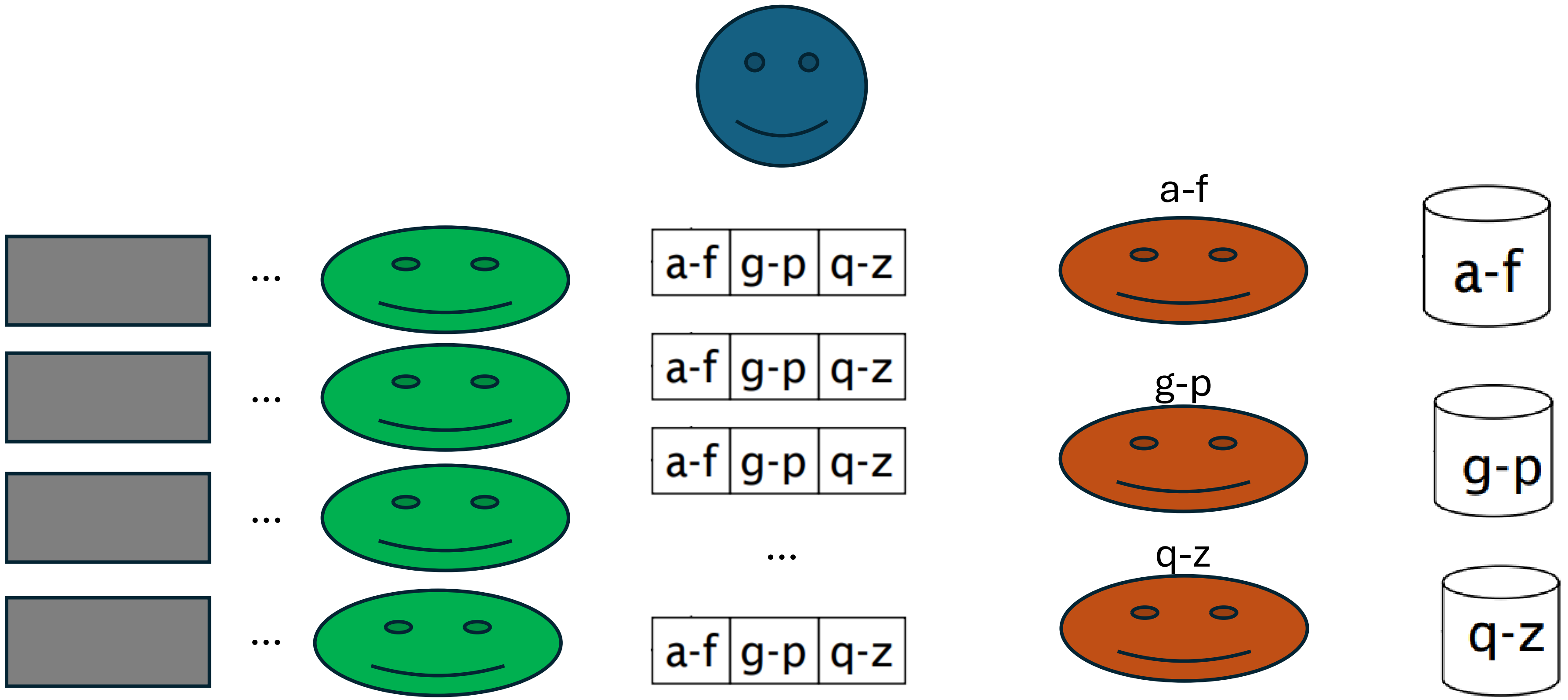
Distributed indexing

- … and merges them with an existing inverted index from their partition

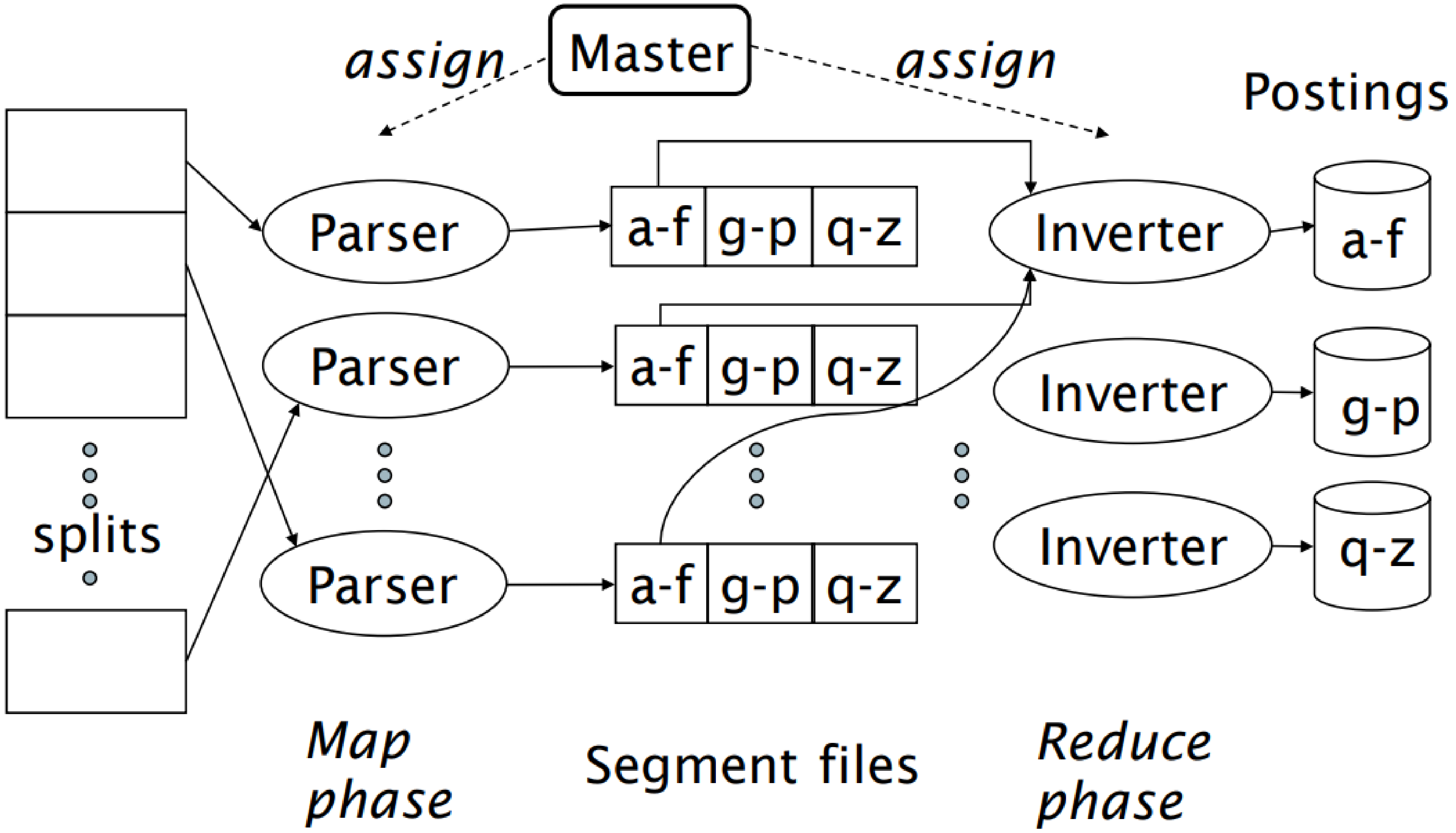


Distributed indexing

- And everyone is happy



Distributed indexing



Lecture 20.09

Heap's law

- How many distinct letters are there in a corpus?
- Apparently:

$$\mathrm{M} = kT^{b}$$

- **M** = Vocabulary, number of distinct terms in corpus
- **T** = Total number of tokens in corpus, including duplicates
- 30 <= **k** <= 100
- 0.4 <= **b** <= 0.6

Heap's law example

- k = 30
- b = 0.6
- T = 1 000 000

- $M = k * T^b$
- $M = 30 * 1\,000\,000^{0.6}$
- $M = 30 * 3981$
- $M = 119\,432$

The point of Heap's law

- Heap's law suggests that the vocabulary growth slows as the size of the corpus increases

- T = 1 000 000 M = 119 432
- T = 2 000 000 M increases by 61 593
- T = 3 000 000 M increases by 49 858
- T = 4 000 000 M increases by 43 498
- T = 5 000 000 M increases by 39 308

Zipf's law

- Shows distribution of terms in a corpus
- Crazy scary formula on the slides and in the book

Basically

- The second most frequent term occurs half as often as the most frequent term

Example

The 3 most frequent terms in "corpus X" are **the**, **yes** and **hello**.

"the" occurs 100 times, "yes" occurs 50 times, and "hello" 33 times

Dictionary compression

- Search starts in the dictionary!
- Want to keep as much as possible in main memory

```
dict = [
    { term: "I", freq: 105, postinglist_pointer },
    { term: "cant", freq: 82, postinglist_pointer },
    ...
]
```

- Sooo tempting to compress!!

Dictionary-as-a-string

```
dictstring = "Icantwaittohearwhatthisweeksbookis"

dict = [
    { freq, postinglist_pointer, term_pointer },
    { freq, postinglist_pointer, term_pointer },
    { freq, postinglist_pointer, term_pointer },
    ...
]
```

Dictionary-as-a-string *with blocking*

```
dictstring = "1I4cant4wait2to4hear4what4this5weeks4book2is"

dict = [
    { freq, postinglist_pointer, term_pointer },
    { freq, postinglist_pointer },
    { freq, postinglist_pointer },
    { freq, postinglist_pointer, term_pointer },
    ...
]
```

Front coding

```
dictstring = "8automata8automate9automatic10automation"
```

Front coding

```
dictstring = "8automat*a1◊e2◊ic3◊ion"
```

Front coding breakdown

"8automat*a1◊e2◊ic3◊ion"

1. 8automat*: 8 chars, automat is a prefix
2. "a" is the first suffix
3. "1◊e": suffix of 1 char incoming! (e)
4. "2◊ic": suffix of 2 chars incoming! (ic)
5. "3◊ion": suffix of 2 chars incoming! (ion)

Front coding breakdown

```
"8automat* a 1◊e 2◊ic 3◊ion"
```

1. `8automat*`: 8 chars, automat is a prefix
2. `"a"` is the first suffix
3. `"1◊e"`: suffix of 1 char incoming! (e)
4. `"2◊ic"`: suffix of 2 chars incoming! (ic)
5. `"3◊ion"`: suffix of 2 chars incoming! (ion)

Front coding breakdown

8automat*a1◊e2◊ic3◊ion

«why no 1◊ in front of the first suffix?»

Front coding breakdown

8automat*a1◊e2◊ic3◊ion

«why no 1◊ in front of the first suffix?»

7automat*1◊a1◊e2◊ic3◊ion

Just spent 2 extra chars.. For nothing

Front coding with 2 prefixes

"8automat*a1◊e2◊ic3◊ion"

"6inter*n3◊nal3◊net5◊ested"

"8automat*a1◊e2◊ic3◊ion5inter*n3◊nal3◊net5◊ested"

"8automata8automate9automatic10automation6intern8internal8internet10interested"

Posting compression

- Postings are larger than the dictionary!
- More compression potential 😝

- Obs! For simplicity, a postings in this context is just a document ID

Gaps

```
computer: 33, 47, 154, 159, 202

=>

computer: 33, 47-33, 154–47, 159-154, 202-159

=>

computer: 33, 14, 107, 5, 43
```

Gaps (stop word example)

```
the: ..., 283042, 283043, 283044, 283045, ...

=>

the: ..., 1, 1, 1, 1, ...
```

VB codes encoding algorithm

1. Convert the document ID from decimal to binary
2. Divide the binary number into splits of at most 7 bits, from right to left
3. If the left-most number has length < 7, pad it with 0s
4. Prefix the right-most number with a 1, and the rest with a 0

VB codes encoding example

Example number = **300**

1. Convert: 300 = **100101100**
2. Divide: **10**, **0101100**
3. Pad: **0000010**, 0101100
4. Add prefix: **0**0000010, **1**0101100

Final bit sequences: 00000010 10101100

VB codes decoding algorithm

1. Read bit sequences from left to right
    1. If the first bit is 0, read the next bit sequence too
    2. If the first bit is 1, stop when we've read the current byte
2. Remove the first bit of each sequence
3. Remove trailing 0s

VB codes decoding example

1. Read bit sequences 00000010 10101100
    1. **0**0000010 starts with a **0**, continue
    2. **1**0101100 starts with a **1**, finish reading and stop
2. Remove prefixes: **0000010**, **0101100**
3. Remove trailing 0s: **10 0101100**

Converting back to decimal we get 300

Gamma encoding algorithm

1. Convert the document ID to binary
2. Remove the prefix bit
3. Take care of the length of the binary number *in unary*
4. Add a 0 at the end of the result of step 3
5. Concatenate the results of steps 3-4 and steps 1-2

Gamma encoding example

Example number = **13**

1. Convert: 13 = **1101**
2. Chop: **101**
3. Length in unary: **111**
4. Pad with 0: **1110**
5. Concatenate: **1110** + **101** = **1110101**

Final bit sequence: 1110101

Gamma decoding algorithm

1. Read N bits until we hit 0
2. Read the next N bits
3. Pad 1 to the result of step 2

Gamma decoding example

1. Read until we hit 0: 111**0**101
2. Read the next N bits: **101**
3. Pad 1: **1101**

Converting back to decimal we get 13

Delta encoding

- The same as Gamma encoding, but we also Gamma encode the length!

Example

- 13 with Gamma encoding: 1110101
- 13 with Delta encoding: 110101

- Difference: **110** prefix instead of **1110**

Rice encoding

- Dealing with the entire posting list

Rice encoding algorithm

```
computer: 34, 178, 291, 453
```

1. Convert IDs to gaps
2. Find the average
3. Round down to the avg's closest power of two, call it **b**
4. For each **gap_int** in the posting list, get
    1. (**gap_int** – 1) / **b** in unary
    2. (**gap_int** – 1) % **b** in binary

Rice encoding example

`computer: 34, 178, 291, 453`

1. Convert to gaps: `34, 144, 113, 162`
2. Find average: (34 + 144 + 113 + 162) / 4 = 113
3. Find **b** = 64 (because 2^7 = 128, 2^6 = 64)

4. Iterating our list:
    1. (34 – 1) / **64** = 0
    2. (34 – 1) % **64** = 100001
    3. etc.

Rice encoding example

```
computer: 34, 178, 291, 453
```

In the end we end up with

```
computer: [0 100001,
           110 001111,
           10 110000,
           110 100001]
```

Golomb encoding

- Just like Rice coding, with 1 exception:
- **b** is average * 0.69

Simple9 encoding

- Uses a 32 bit **word**.
- First 4 bits is a "selector"
- Remaining 28 bits are integers

- Selector tells us how the following ints are distributed
- 0000 = 1 int of 28 bits
- 0001 = 2 ints of 14 bits
- 0010 = 3 ints of 9 bits
- ...
- 1000 = 28 ints of 1 bit

Simple9 encoding example 1

- Selector: **0000**
- The remaining 28 bits contain 1 integer, encoded in binary

- Example decimal: **134 217 726**
- Encoded to binary: **11111111111111111111111110**

- Simple9 encoding: 0000 11111111111111111111111110

Simple9 encoding example 2

- Selector: **0100**
- The remaining 28 bits contain 14 integers, encoded in binary

- Example decimals: **1**, **2**, **2**, **2**, **3**, **0**, **1**, **2**, **2**, **3**, **0**, **1**, **2**, **2**, and **3**
- Encoded to binary: **01**, **10**, **10**, **10**, **11**, **00**, **01**, **10**, **10**, **11**, **00**, **10**, **10**, **11**

- Simple9 encoding: 0100 0110101011000110101100101011