# Low-cost Management of Inverted Files for Online Full-Text Search

**Giorgos Margaritis** , Stergios V. Anastasiadis
Systems Research Group, University of Ioannina, Greece

CIKM '09, Hong Kong, November 2 - 6, 2009

# Introduction

## Prices of hard disks drop

- Amount of stored data increases
- Need for automated search

## Full-text search

- User: defines terms and operands (e.g. logical)
- System:  finds documents that contain terms and satisfy operands
  - ↳ usually sorts returned documents by "relevance"

## Online full-text search

- Index updates concurrent with document searches
- Need to constantly maintain index up-to-date

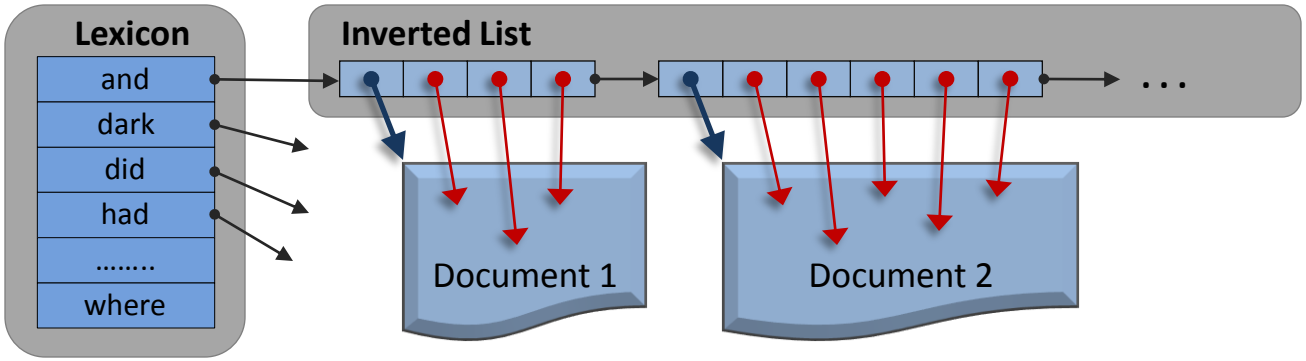
# Inverted Index

## For each term store an **inverted (or posting) list**

- list of pointers to documents containing term
- pointers specify exact position in document (e.g. <*doc*, *offset*>)
- each pointer is called **posting**

## Lexicon

- associates terms to their inverted lists



# Build time vs. Search time Trade-off

## Contiguous disk storage of inverted lists:

- improves access efficiency
- needs complex dynamic storage management
- requires frequent or bulky relocations

## List fragmentation

- improves build time
- degrades search time

# Inverted List Contiguity

## Recent methods

- Maintain a (relatively small) number of partial indexes
- Retrieve an inverted list: retrieve its fragments from all partial indexes

## Majority of inverted lists

- Disk size of hundreds of kilobytes

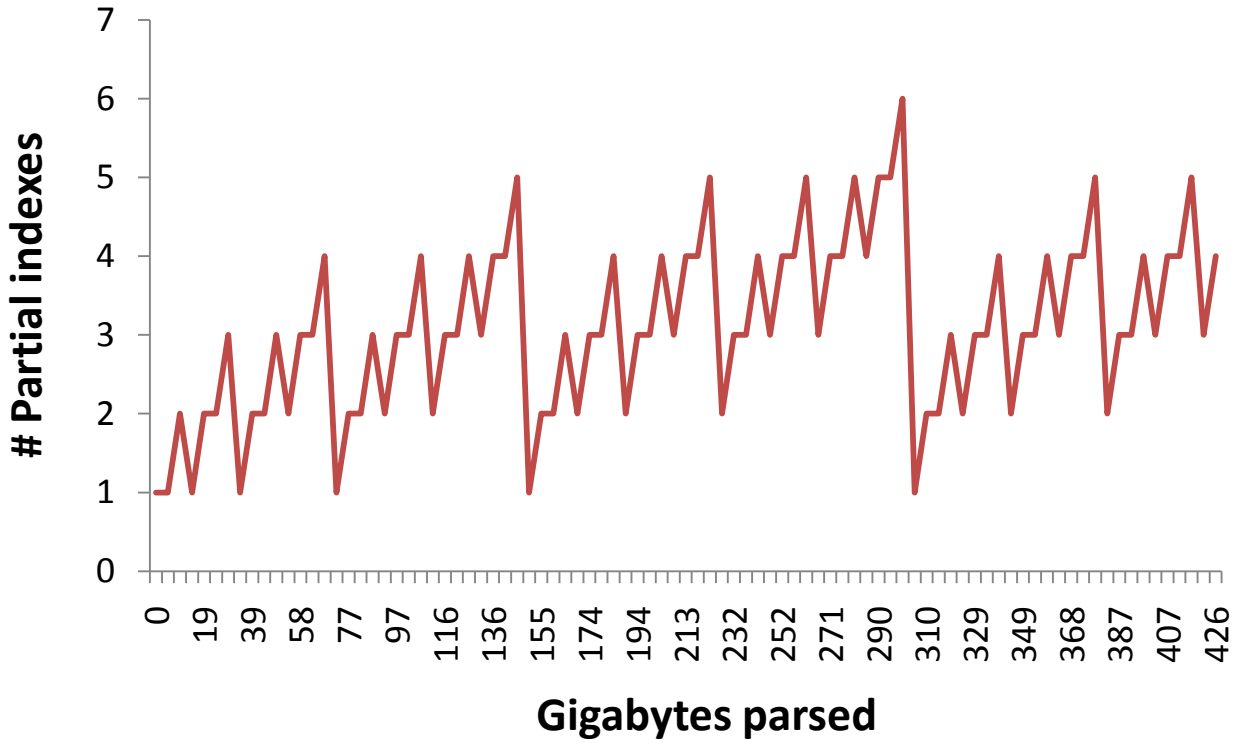
## Fragmentation can really **harm retrieval times of small lists**

- More than 99% of terms have list sizes < 10KB (426GB text collection)
- Typical disk: 12ms positional time + 0.1ms to sequentially read 10KBb
- N fragments → N positional times ≈ N x (contiguous list retrieval time)

# Inverted List Contiguity

## Hybrid Logarithmic Merge [Büttcher et al., 2006]

- Create inverted index for 426GB



# Design Objectives

1. Keep small lists contiguous

2. Keep large lists contiguous, or in few large fragments

3. Keep build time low

4. Retrieve lists fast

# Outline

Motivation

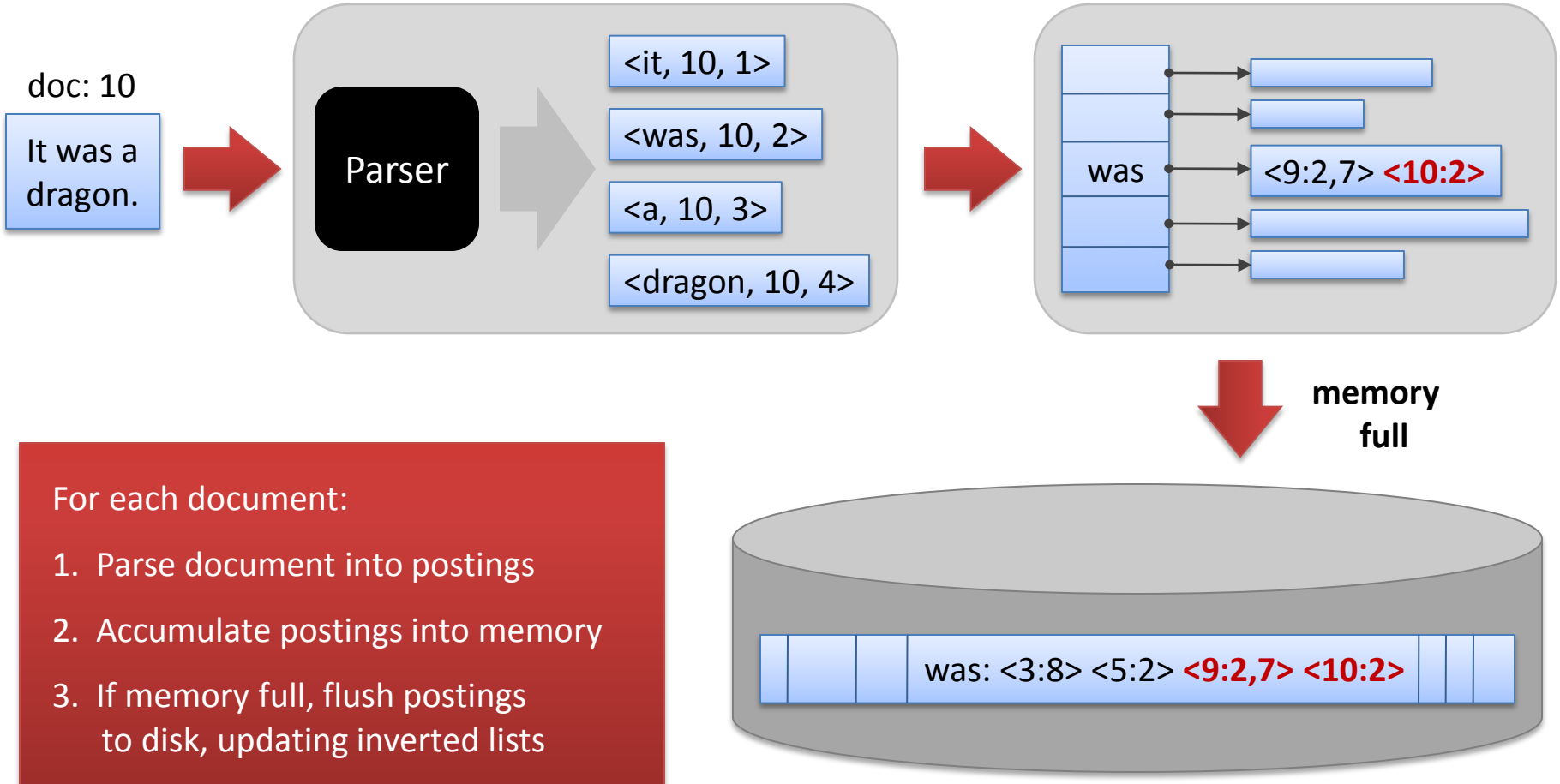
➡ Background

Proposed Algorithm

Experiments

Conclusions

# Creating an Inverted Index




For each document:

1. Parse document into postings
2. Accumulate postings into memory
3. If memory full, flush postings to disk, updating inverted lists

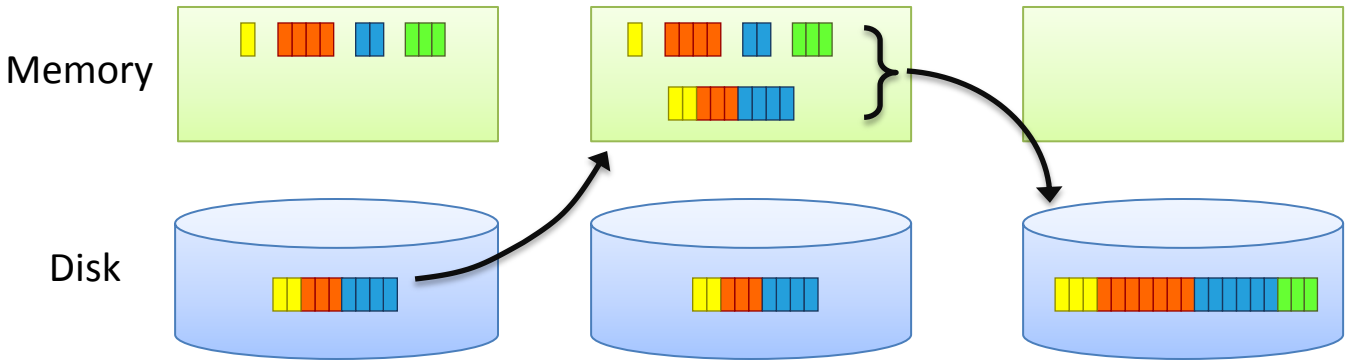
# Postings Flushing: Re-Merge

## Re-Merge [Lester et al., 2004]

- Inverted lists stored in lexicographic order
- For each list: 1. Read in memory, 2. Add new postings, 3. Write to disk

## Cost

- Efficient for lists with few postings (sequential read)
- Retrieve ALL lists from disk and write them ALL back to disk



# Postings Flushing: In-Place Update

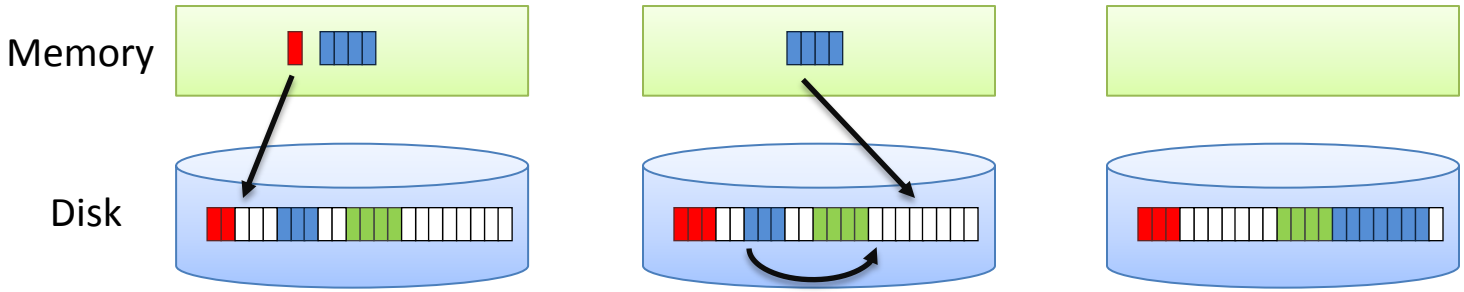
## In-Place Update [Tomasic et al., 1994]

- Create list: pre-allocate some empty space after each it
- For each list: append new postings on disk.

  If not enough space, relocate list in position with sufficient space

## Cost

- Efficient for lists with lots of postings (append instead of read + write)
- Lists can't be kept in lexicographical order → random seeks for each list



# Postings Flushing: Hybrid Merge

## Hybrid Merge [Büttcher et al., 2008]

- Categorizes terms into **short** (rare) and **long** (frequent), based on their inverted lists length
- Use Re-Merge method for short terms and In-Place Update for long

## Combines the benefits of the two methods

- Re-Merge: proper for updating a big number of short lists
- In-Place Update: proper for updating a small number of long lists

## Two problems

- Cost of reading and writing back all short lists for Re-Merge
- Cost of relocations for In-Place Update

# Outline

Motivation

Background

➡ Proposed Algorithm

Experiments

Conclusions

# System Architecture

Organize disk index in fixed-size **blocks**

Categorize terms in **short** or **long**, based on their list size

Group short terms into disjoint lexicographical **ranges**

- Lists of a range must fit into one block

A block may contain:

- a group of short lists, updated using "read lists, add postings, write lists"
- postings of a long list, updated using appends

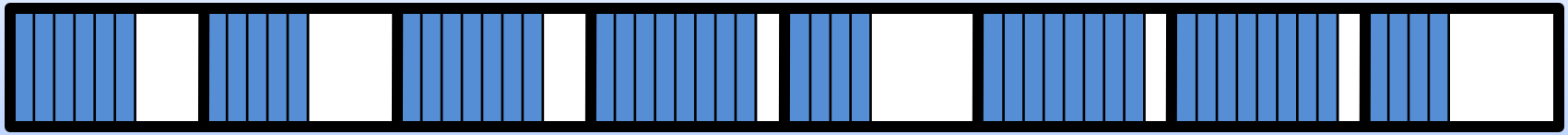
When memory is full:

- selectively flush *some* postings from memory to disk
- update only those blocks that can be efficiently updated

# Proteus

MEMORY

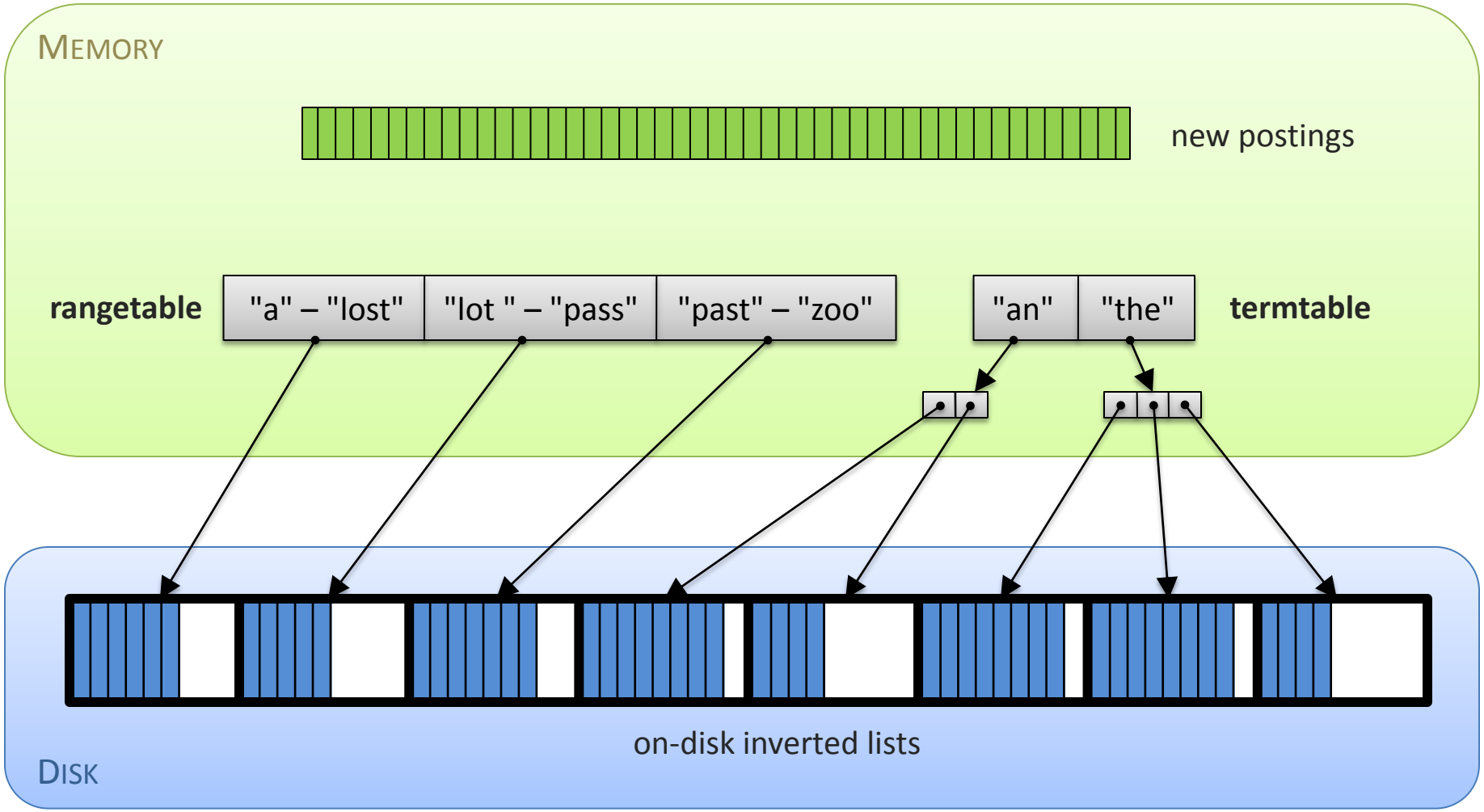
new postings



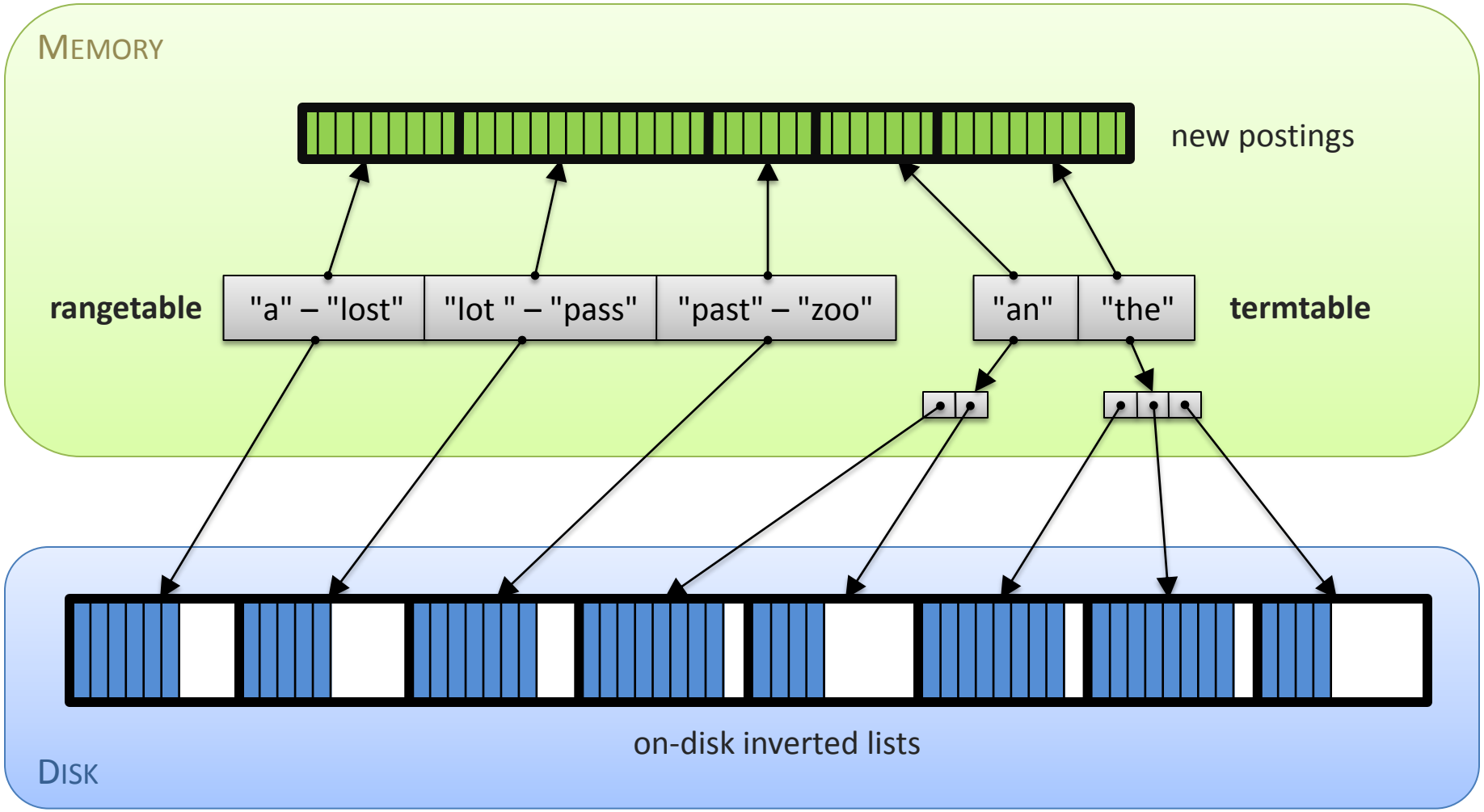
on-disk inverted lists

DISK

# Proteus


MEMORY
new postings
rangetable
"a" – "lost"
"lot " – "pass"
"past" – "zoo"
"an"
"the"
termtable
on-disk inverted lists
DISK

# Proteus


MEMORY
new postings
rangetable
"a" – "lost"
"lot " – "pass"
"past" – "zoo"
"an"
"the"
termtable
on-disk inverted lists
DISK

# Parameters

## Preference Factor (***F***)

- Prefer to flush long term *T* instead of range *R*:
  - flush long term: cheap – single disk append
  - flush range: costly – read all lists, add postings, write all lists
- Caveat: if *T* has too few memory postings does not worth flushing
  - if *T* has ***F*** times less postings than *R*, flush *R*

## Flush Memory (***M***)

- Flush only a small portion of memory (e.g. ***M*** = 20MB)
  - Update only blocks that can be efficiently updated
  - Free some memory space

# "Selective Range Flush" Algorithm

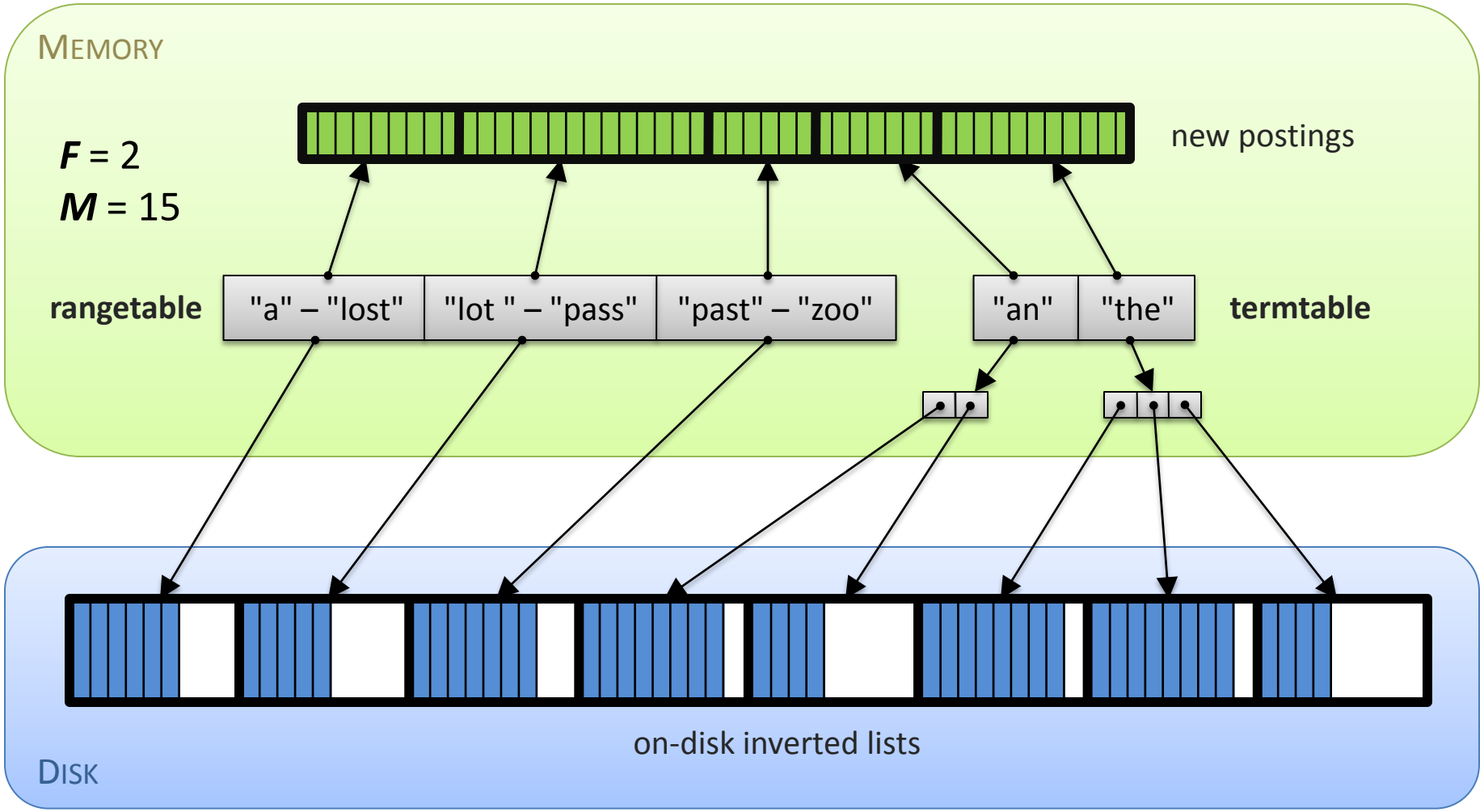
1. Parse document into postings

2. When memory is full:

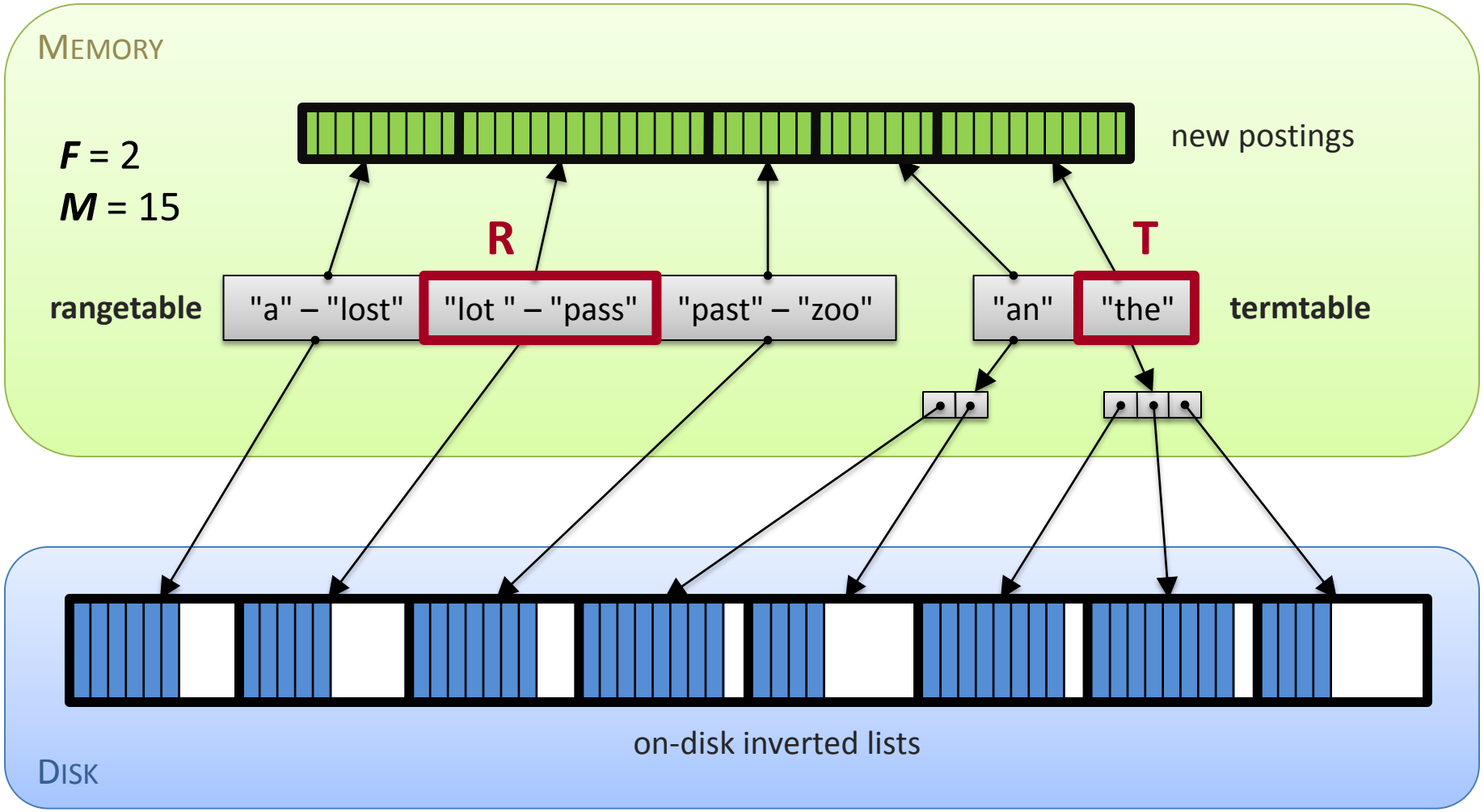
```
While ( size of flushed postings < M )
    R := range with max postings in memory ;
    T := long term with max postings in memory ;
    If ( R.mem_postings < F × T.mem_postings ) then
        flush T ;     /* append new postings on disk */
    Else
        flush R ;     /* read lists from block, add new postings, write lists back to block */
```

3. Return to parsing phase

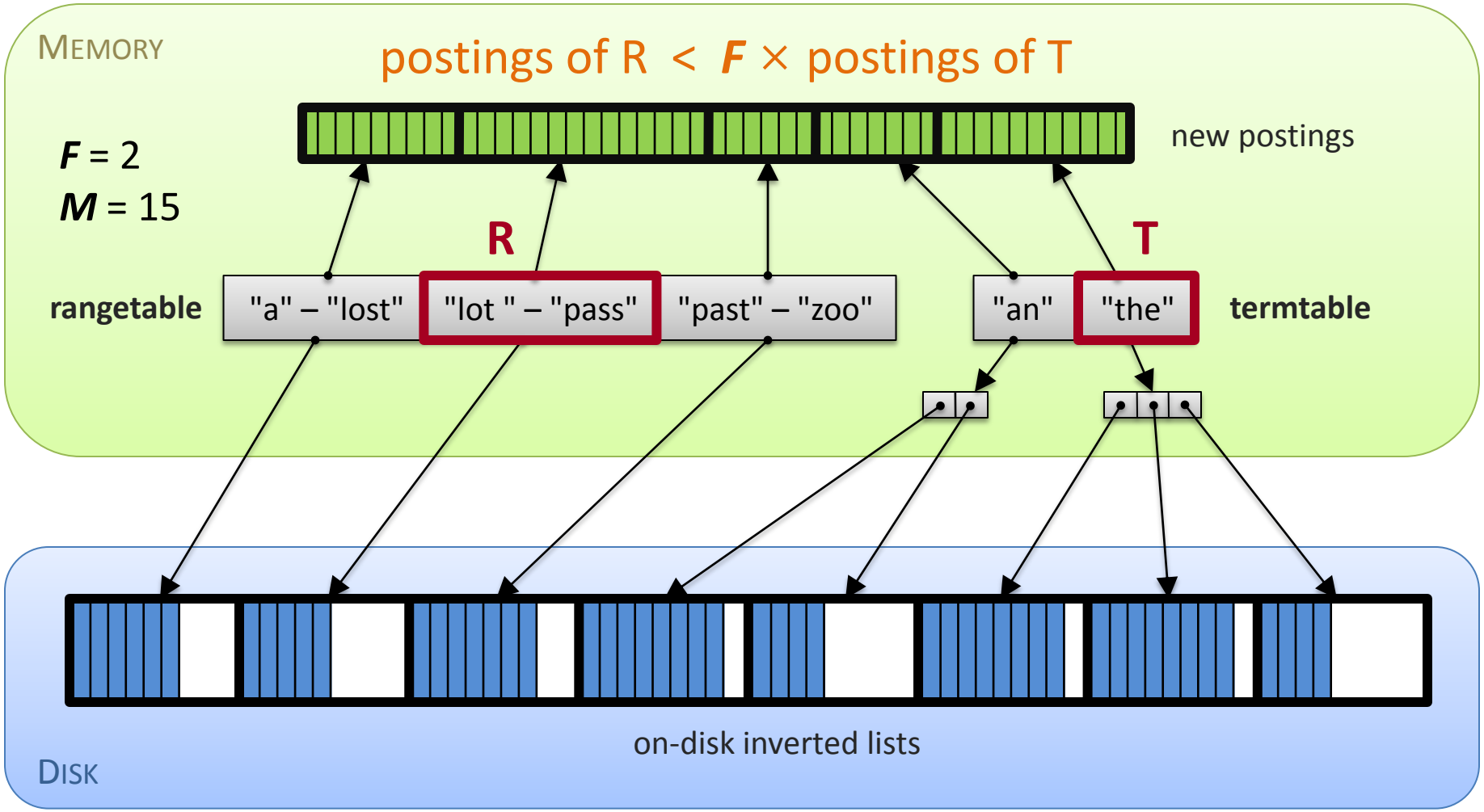
# Proteus


MEMORY
F = 2
M = 15
new postings
rangetable
"a" – "lost"
"lot " – "pass"
"past" – "zoo"
"an"
"the"
termtable
on-disk inverted lists
DISK

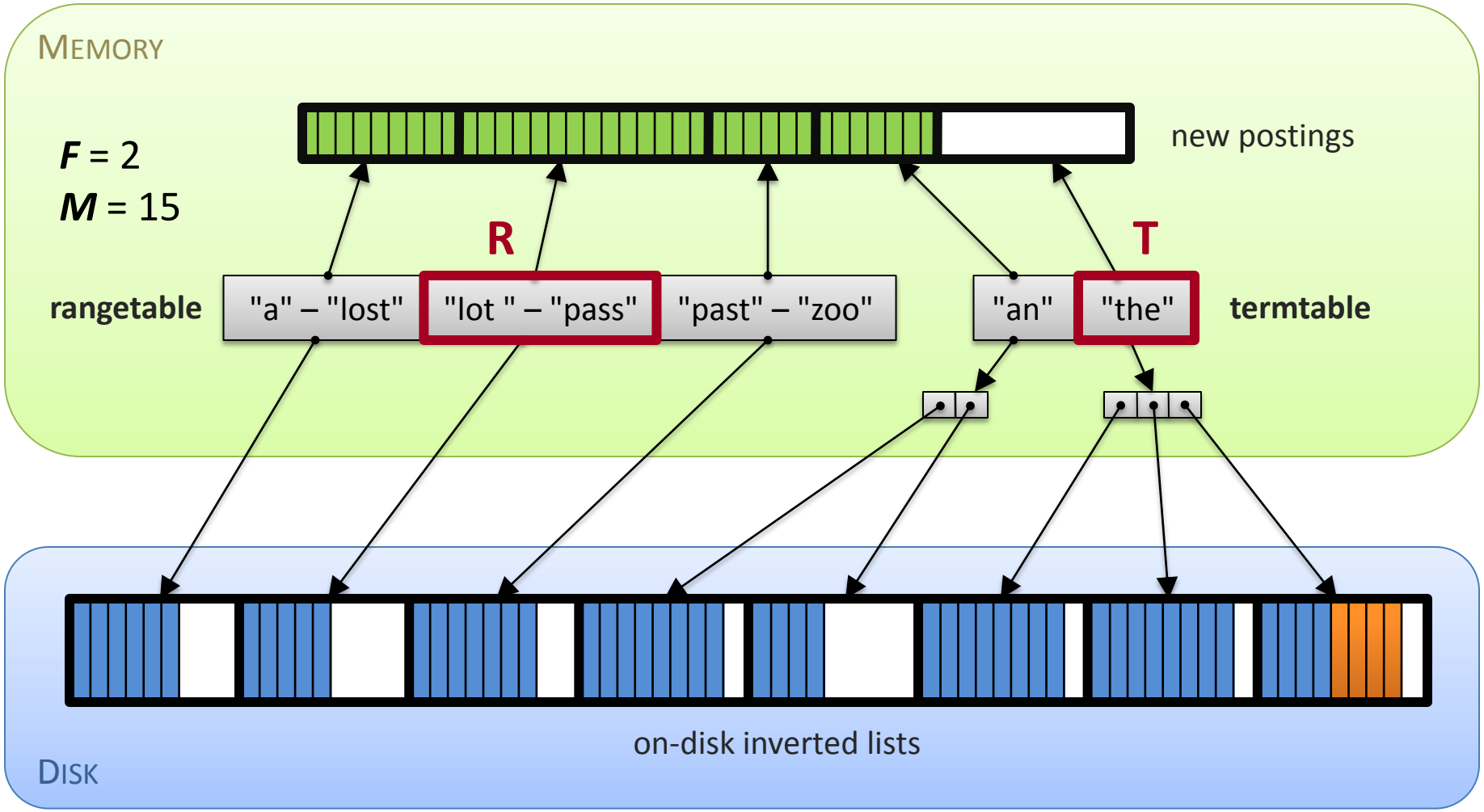
# Proteus


MEMORY
F = 2
M = 15
new postings
R
T
rangetable
"a" – "lost"
"lot " – "pass"
"past" – "zoo"
"an"
"the"
termtable
on-disk inverted lists
DISK

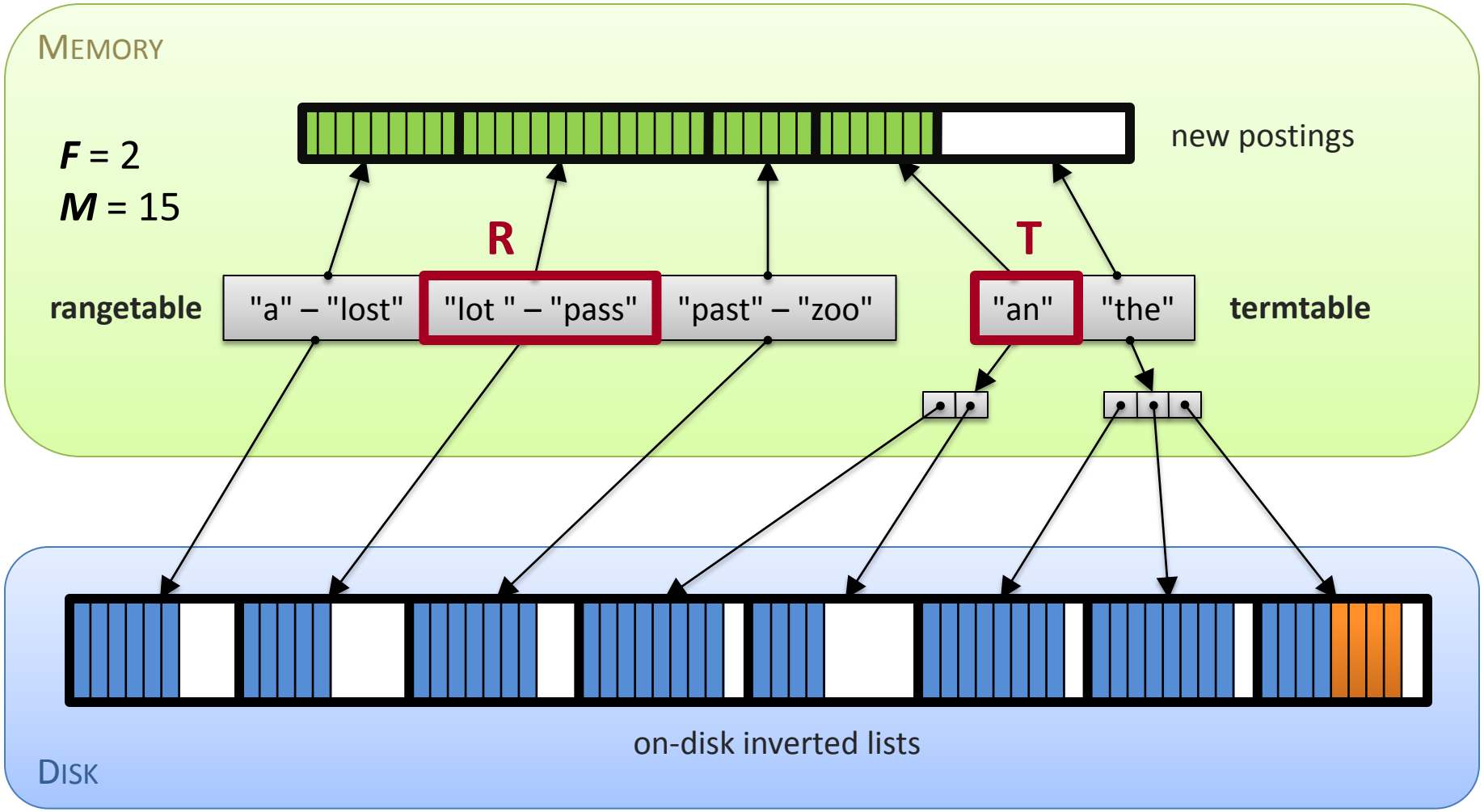
# Proteus


MEMORY
postings of R < F × postings of T
F = 2
M = 15
new postings
R
T
rangetable
"a" – "lost"
"lot " – "pass"
"past" – "zoo"
"an"
"the"
termtable
on-disk inverted lists
DISK

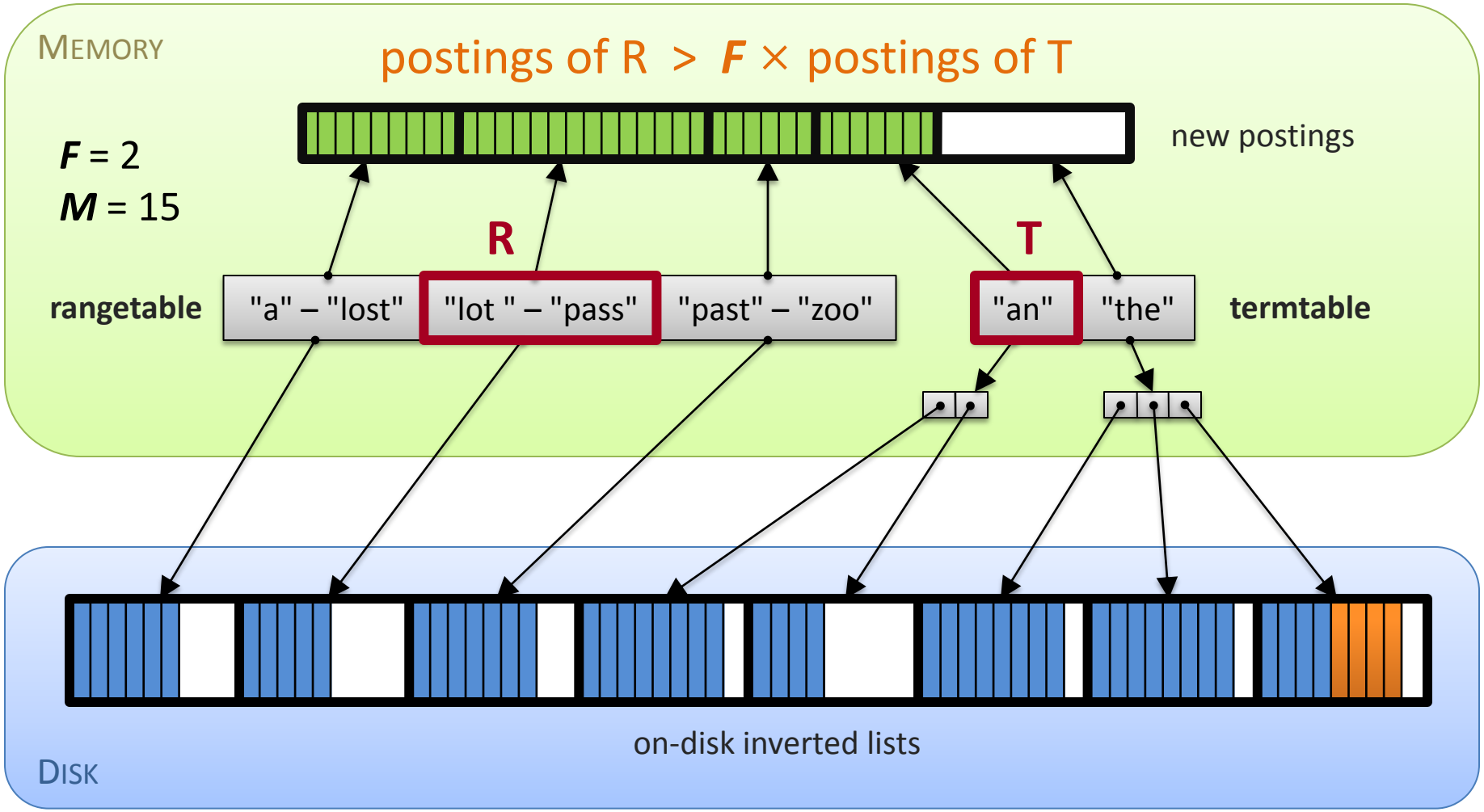
# Proteus


MEMORY
F = 2
M = 15
new postings
R
T
rangetable
"a" – "lost"
"lot " – "pass"
"past" – "zoo"
"an"
"the"
termtable
on-disk inverted lists
DISK

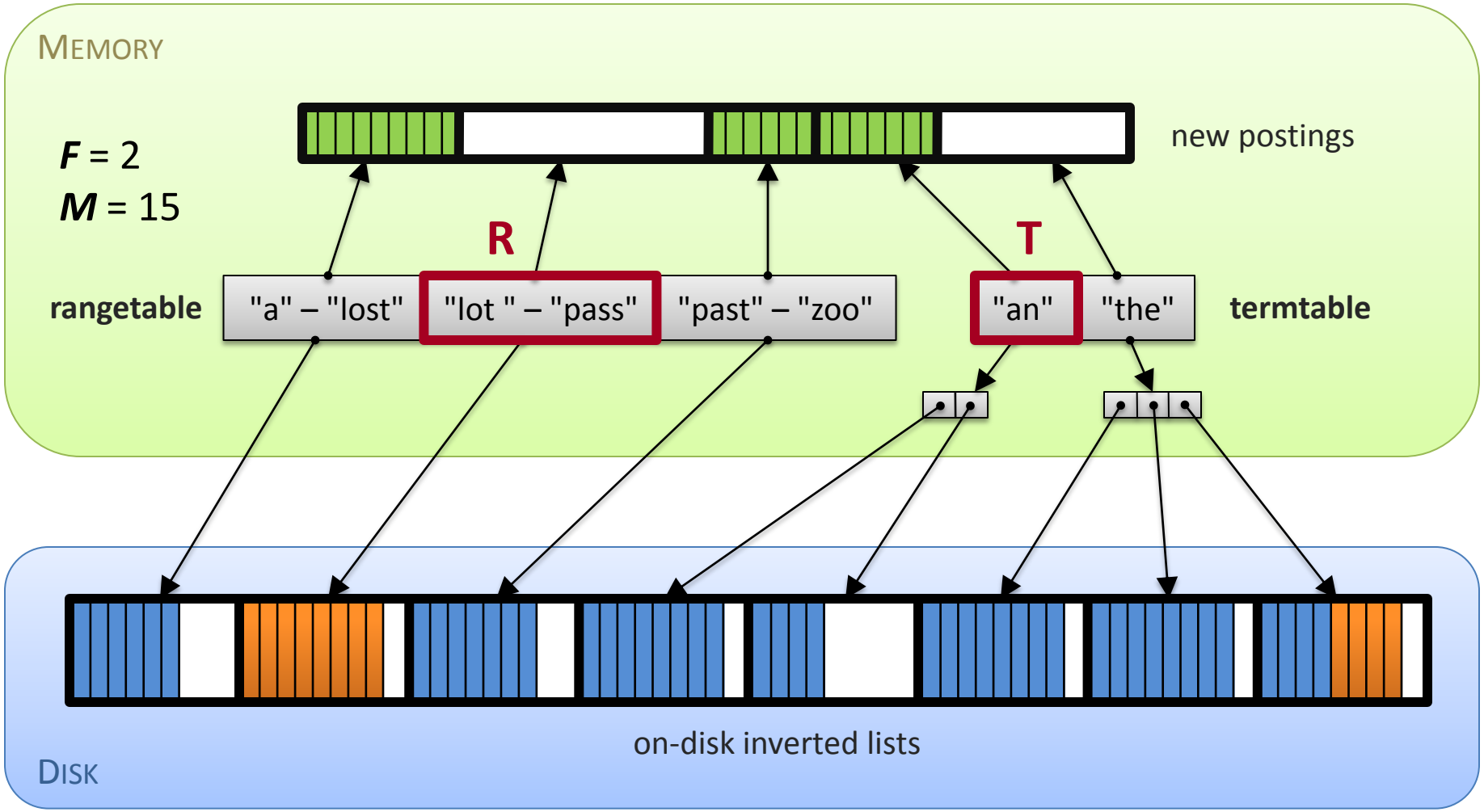
# Proteus


MEMORY
F = 2
M = 15
new postings
R
T
rangetable
"a" – "lost"
"lot " – "pass"
"past" – "zoo"
"an"
"the"
termtable
on-disk inverted lists
DISK

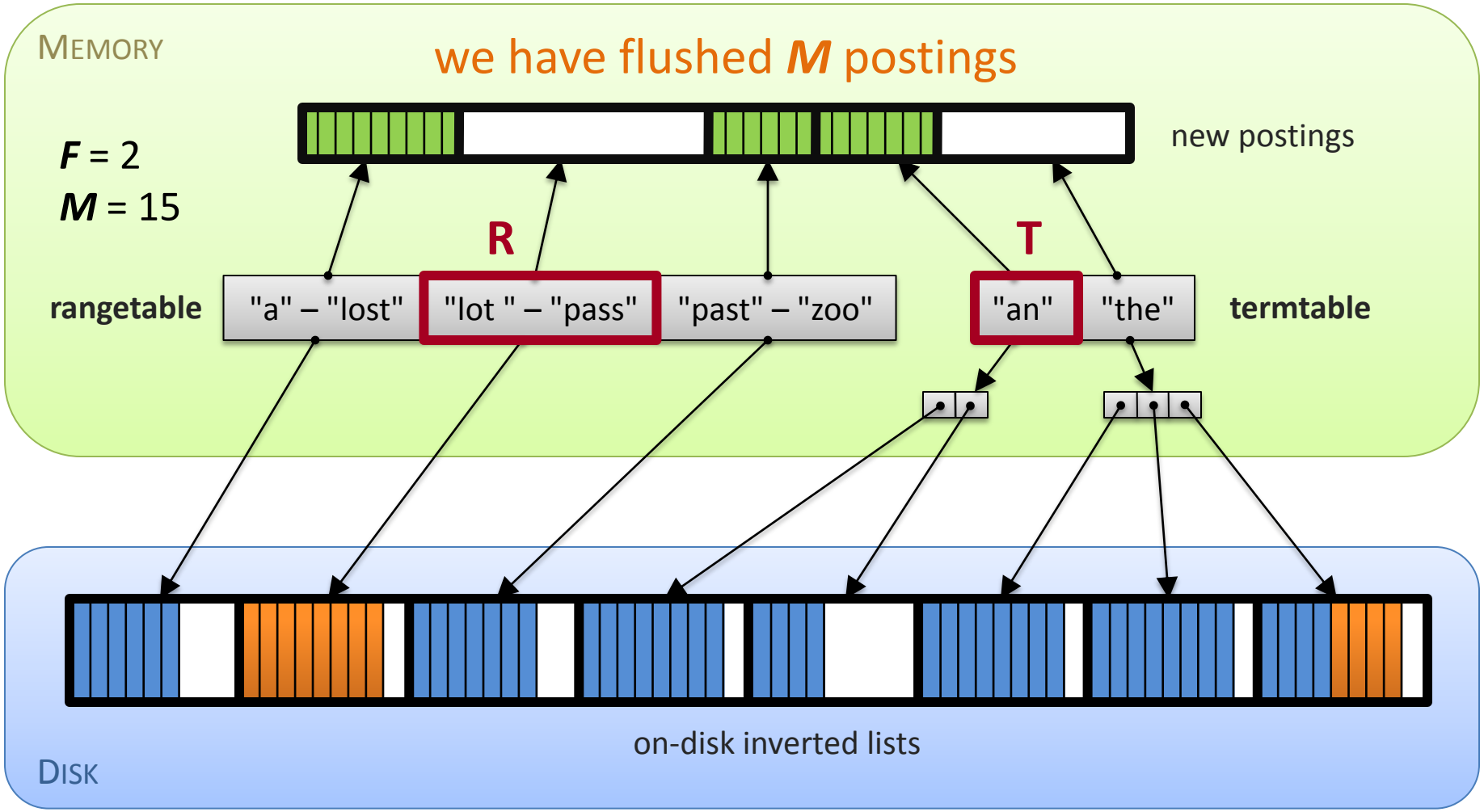
# Proteus


MEMORY
postings of R > F × postings of T
F = 2
M = 15
new postings
R
T
rangetable
"a" – "lost"
"lot " – "pass"
"past" – "zoo"
"an"
"the"
termtable
on-disk inverted lists
DISK

# Proteus


MEMORY
F = 2
M = 15
new postings
R
T
rangetable
"a" – "lost"
"lot " – "pass"
"past" – "zoo"
"an"
"the"
termtable
on-disk inverted lists
DISK

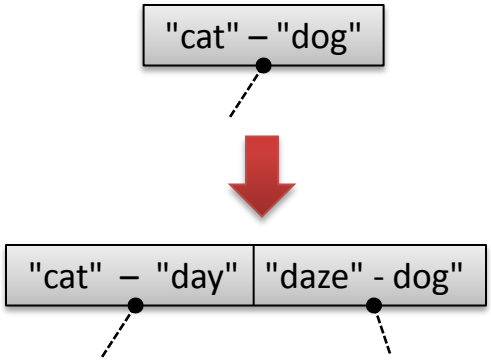
# Proteus


MEMORY
we have flushed M postings
F = 2
M = 15
new postings
R
T
rangetable
"a" – "lost"
"lot " – "pass"
"past" – "zoo"
"an"
"the"
termtable
on-disk inverted lists
DISK

# Block Overflow

## Block of range *R* overflows:

- allocate block
- move half of postings to new block
- split *R* according to the lists each block contains




## Last block of list of long term *T* overflows:

- allocate block
- move overflown postings to new block

# Innovative Work

Simplify disk management by splitting postings into disk blocks

- Relax requirement of full contiguity for long terms
- Maintain requirement of full contiguity for each short term

Treat short terms in ranges for efficiency

Partially flush both short and long terms when memory full

Dynamically decide whether to flush long term or range

# Outline

Motivation

Background

Proposed Algorithm

➡ Experiments

Conclusions

# Experiments

## Experimental environment

- Nodes: quad-core at 2.33Ghz, 3GB memory, 2 SATA disks (500GB each)
- Text collection: 426GB GOV2 TREC Terabyte track
- Search queries: Efficiency Topics query set of TREC 2005 Terabyte Track

## Proteus: Selective Range Flush prototype implementation

- Parser from Zettair search engine (RMIT University, Australia)

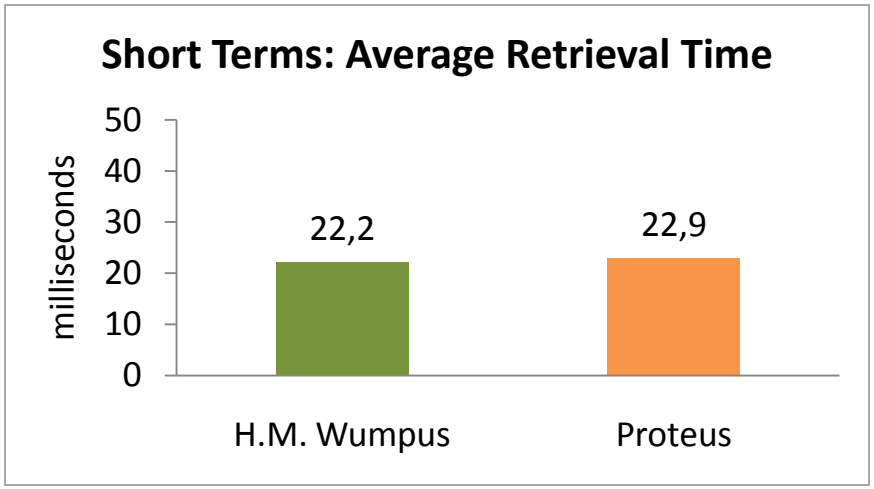
## Compare with Hybrid Merge

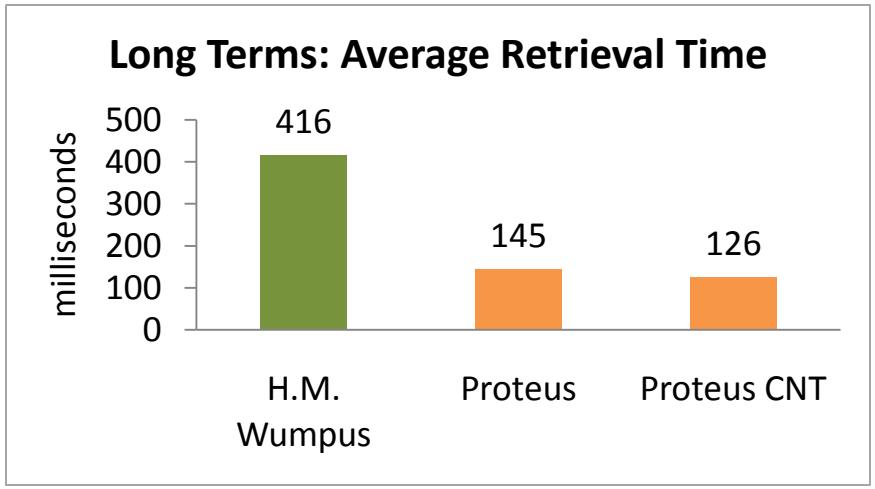
- Wumpus search engine (U. Waterloo, Canada)

# Parameters

| Parameter | Default value |
|---|---|
| Block size | 8MB |
| Postings memory | 1GB |
| Flush memory | 20MB |
| Preference factor | 3 |
| Long list threshold | 1MB |

# List Retrieval




## Proteus short list retrieval times similar to Hybrid Merge

- both systems keep short lists contiguous

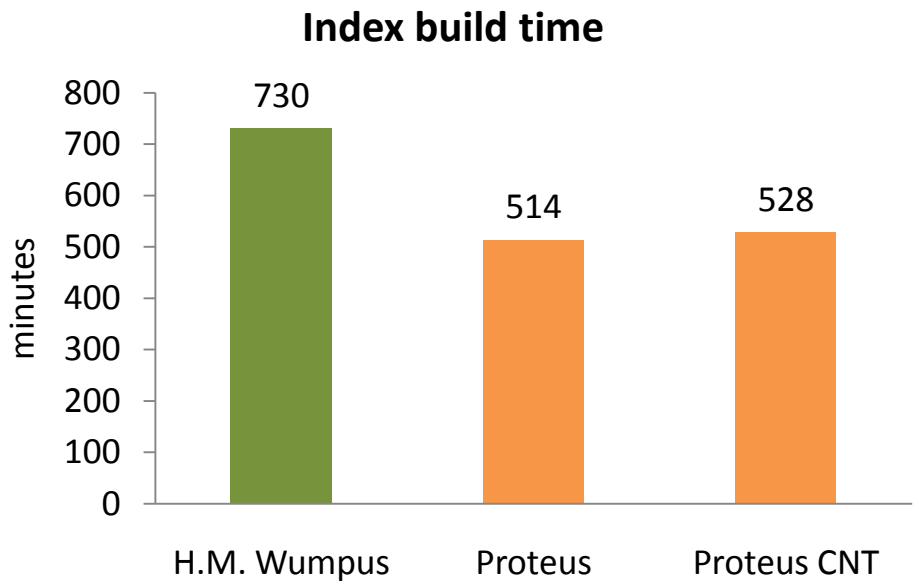
## Proteus long list retrieval times 2-3 times lower than Hybrid Merge

- due to Wumpus implementation, not H.M. algorithm

## Proteus "CNT" variant:

- keeps long lists contiguous using relocations

# Inverted Index Construction



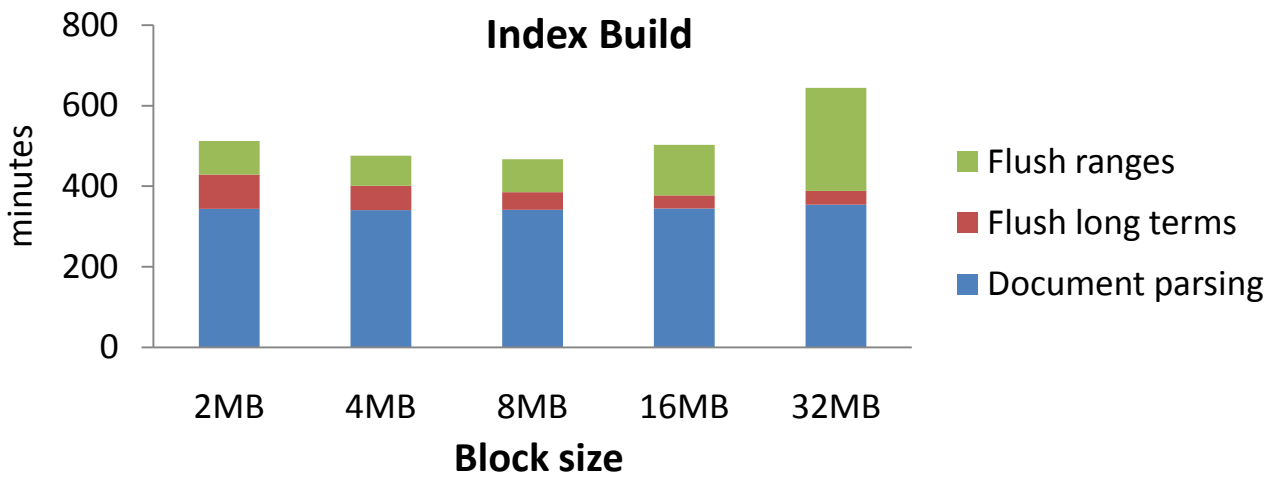

## Proteus:

- 30% reduction in build time comparing to H.M.

## Proteus CNT:

- 3% more time (14 min) to keep *all* lists contiguous

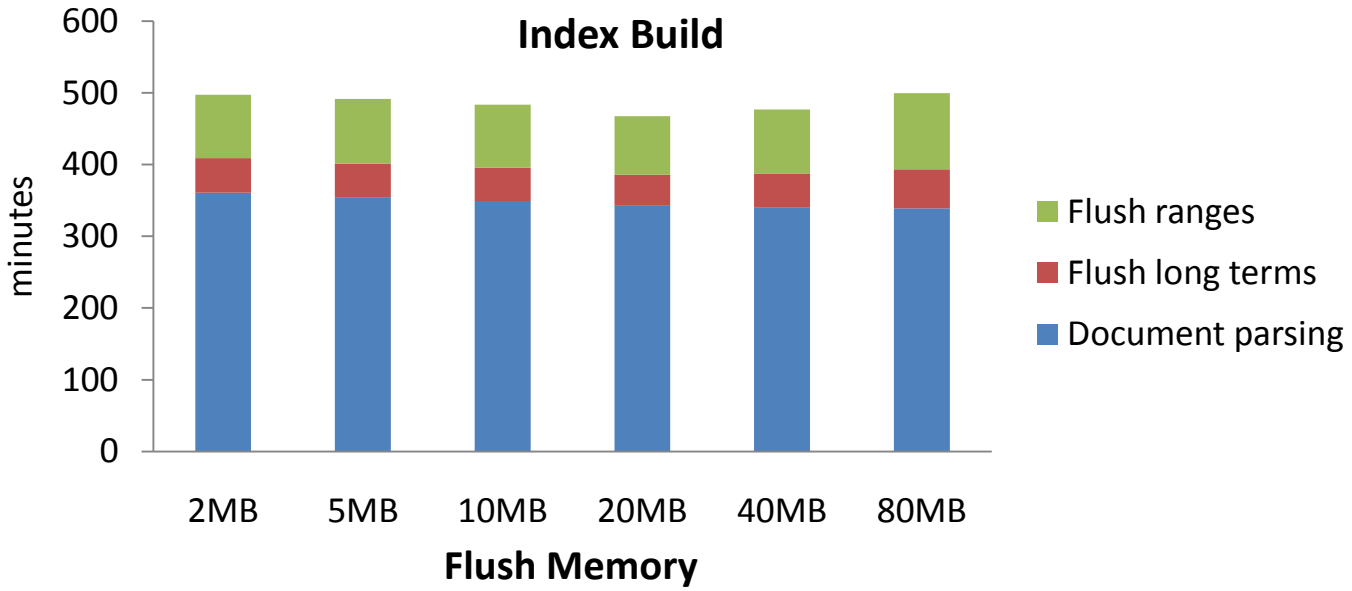
# Sensitivity Analysis: Block size




## Increase block size

- increase range flushing time:
  - more postings per range block
  - more bytes transferred between memory and disk per flush
- decrease long term flushing time:
  - same amount of bytes flushed to disk, fewer appends

# Flush memory



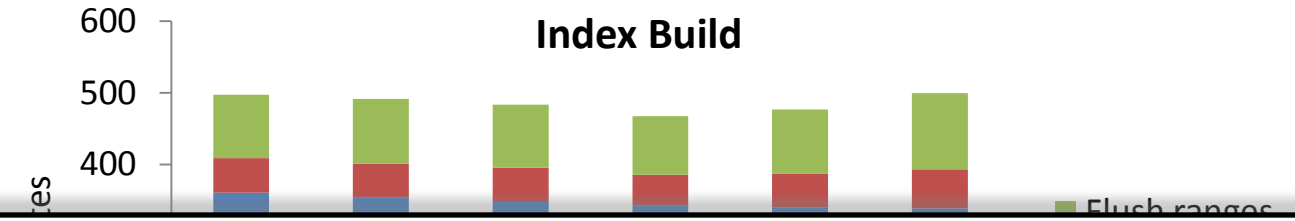

## Small values:

- don't create sufficient space to accumulate new postings

## Large values:

- flush ranges and long terms with very few postings →
- disk head movement overhead

# Flush memory




*"Flush only a small percentage (2% - 3%) of total memory ..."*

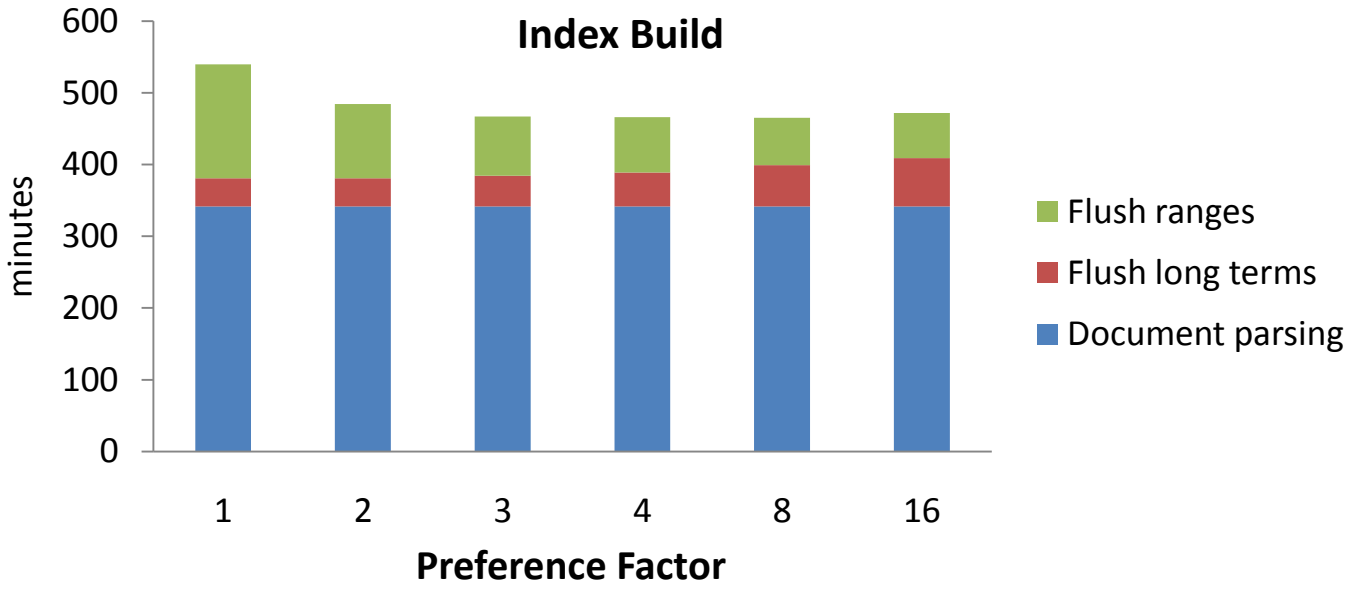
Sr

- 

Large values:

- flush ranges and long terms with very few postings →
- disk head movement overhead

# Preference Factor



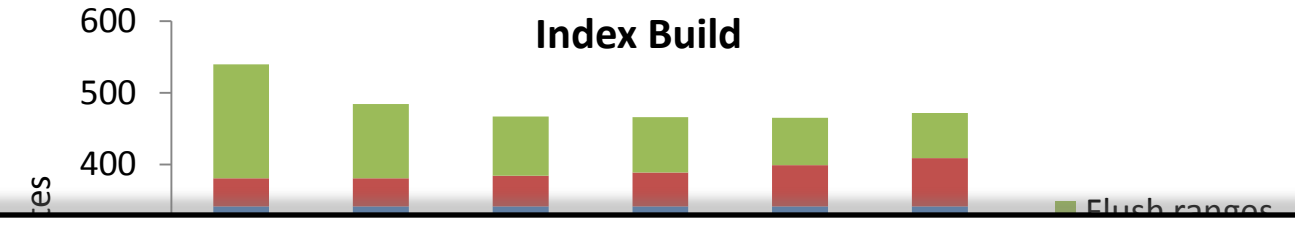

Small values:

- underestimate cost of flushing ranges

Large values:

- underestimate cost of flushing long terms

# Preference Factor




*"Postpone flushing of ranges, but not too much …"*

Sr

- underestimate cost of flushing ranges

Large values:

- underestimate cost of flushing long terms

# Conclusions & Future work

## Fragmentation can harm retrieval times for short lists

## Proteus:

- manage inverted index in blocks
- keep short lists contiguous, long lists in large fragments
- memory full: selectively flush some postings
- update only blocks that can be efficiently updated
  - **30% reduction in build time**
  - **constant 15% increase in retrieval times of lists > 8MB**
  - **3% increase in build time to keep *all* lists contiguous**
  - **relatively insensitive to parameter values**

## Future work:

- different block sizes for short and long terms, auto adjust parameter values based on dataset / hardware, different cost models for flushing