DedupSearch: Two-Phase Deduplication Aware Keyword Search

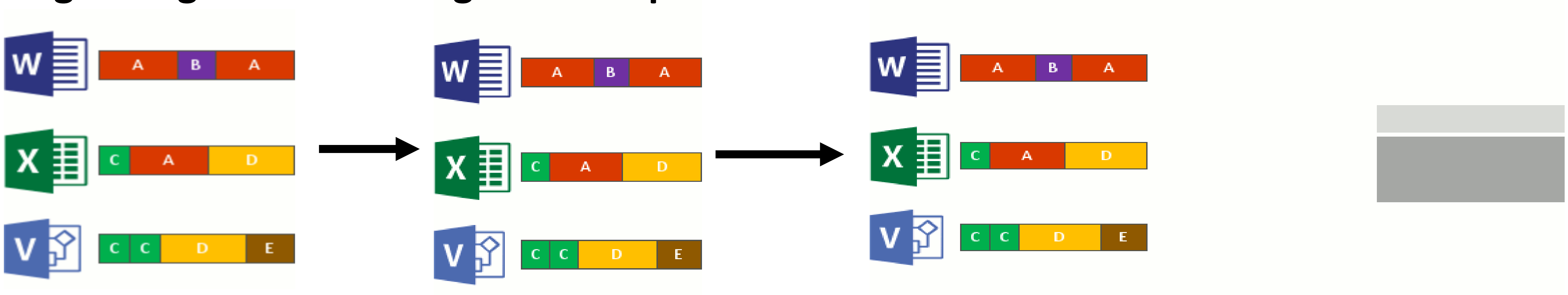
Nadav Elias, Technion-Israel Institute of Technology; Philip Shilane, Dell Technologies; Sarai Sheinvald, ORT Braude College of Engineering; Gala Yadgar, Technion - Israel Institute of Technology

Speaker wrl

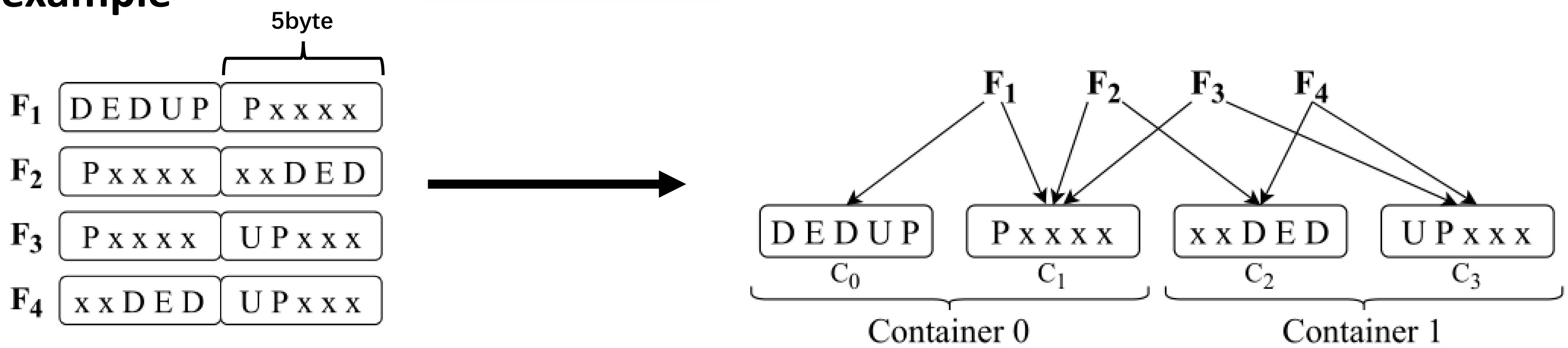
FAST 2022

Background

➢ **Deduplication:** **the purpose is to reduce the physical capacity required to store the growing amounts of logical backup data.**



- **example**



(a) Traditional system

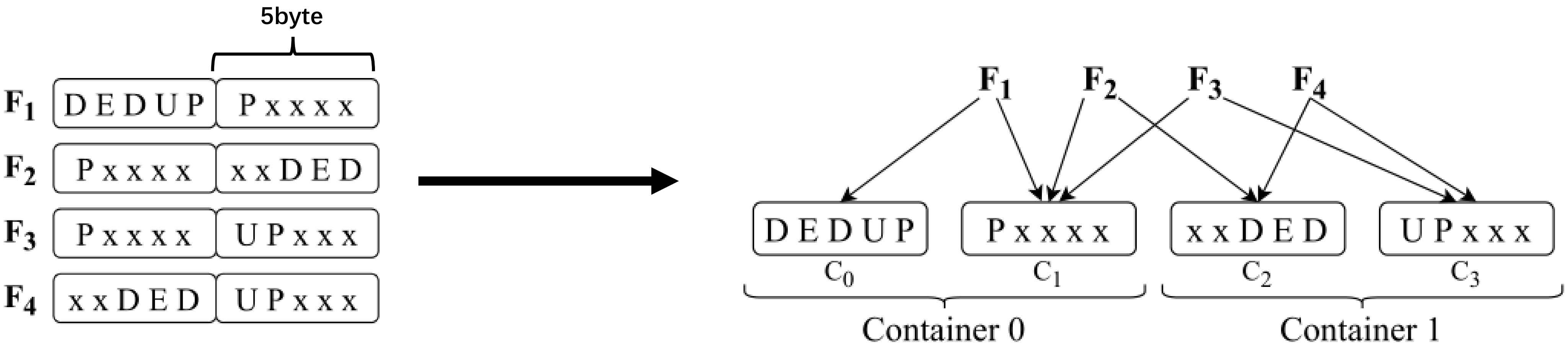
Background

- Keyword search: identify relevant documents with a string search.

- Logging and data analytics systems: construct an index of strings during data ingestion
- The problem is 1. Such Indexes carry non-negligible overheads
  2. Not useful for binary strings or more complex keyword patterns.

- naïve search method: process a file system by progressing through the files, opening each file, and scanning its content for the specified keywords.

Problem



(a) Traditional system

The same naïve search algorithm can also be applied to the deduplicated storage.

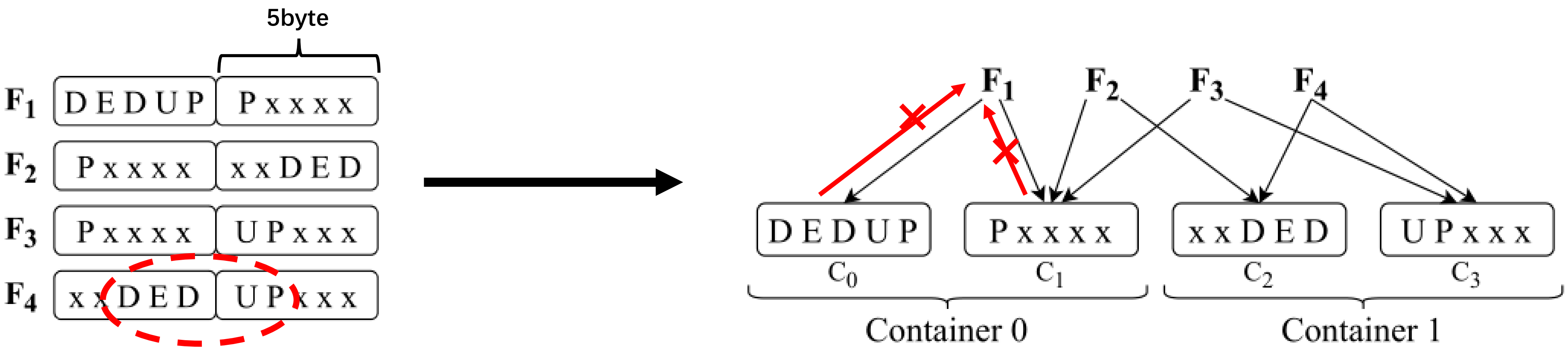
Following the file recipes (F1,F2,F3,F4) it would scan the chunks in the following order: C0,C1,C1,C2,C1,C3,C2,C3—a total of eight chunk reads.

1. If this access pattern spans a large number of containers (larger than the cache size), entire containers might be fetched from the disk several times.

2. Moreover, the data in each chunk will be processed by the underlying keyword-search algorithm multiple times—once for each occurrence in a file.

Idea

➢ **Main idea:** **Read and process each chunk in the system only once.**



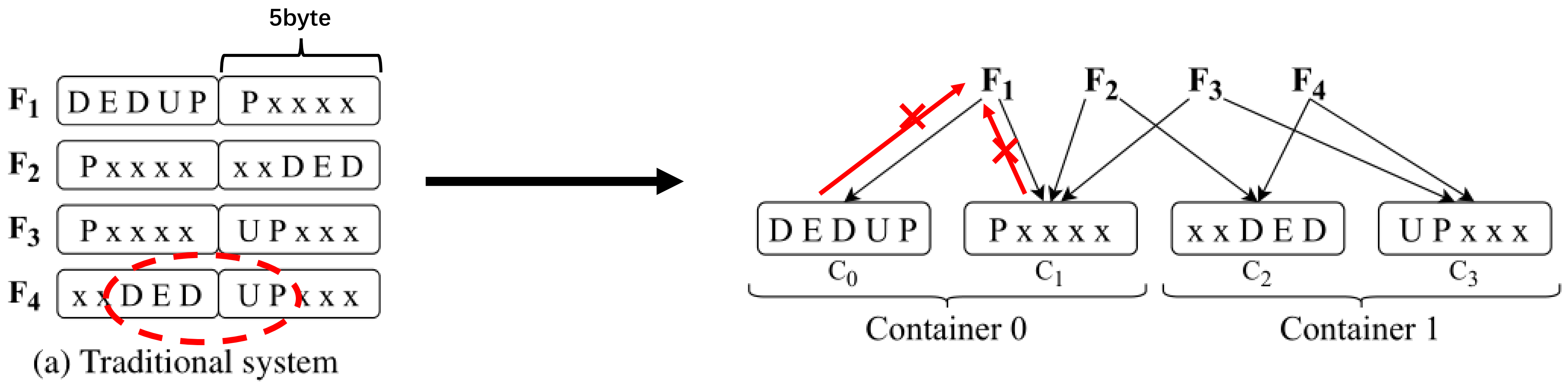
(a) Traditional system

- **challenge1:** **we cannot directly associate keyword matches in a chunk with the corresponding file or files**

- **challenge2:** **keywords might be split between adjacent chunks in a file**

Idea

- **Main idea:** **Read and process each chunk in the system only once.**



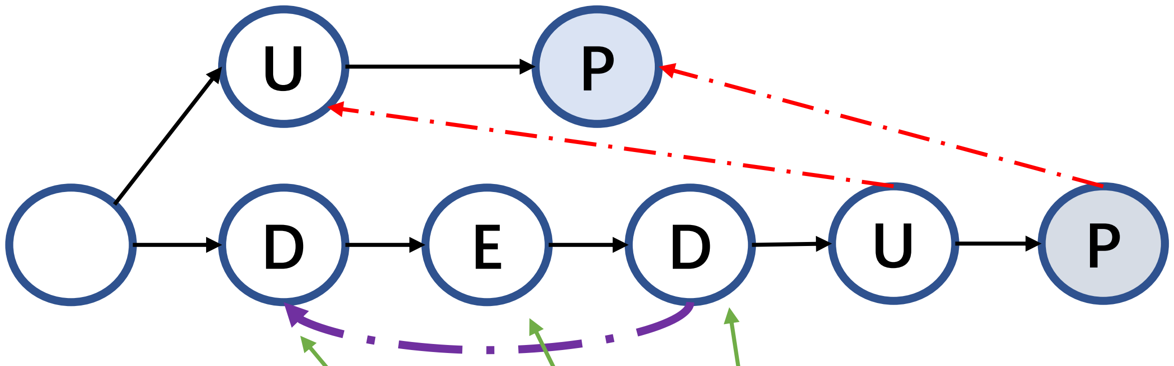
- **This paper's solution is**: **Propose an alternative algorithm, DedupSearch, that progresses in two main phases.**

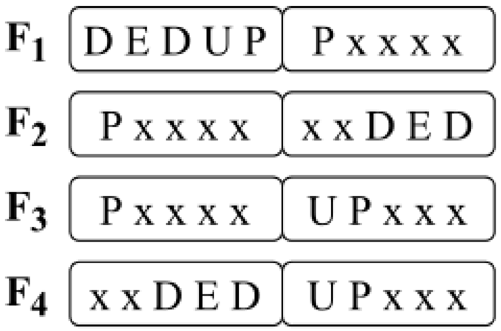
The **physical phase** performs a physical scan of the storage system and scans each chunk of data for the keywords.

The **logical phase** performs a logical scan of the file system by traversing the chunk pointers that make up the files.

Design —String-matching algorithm

The dictionary is {dedup, up}





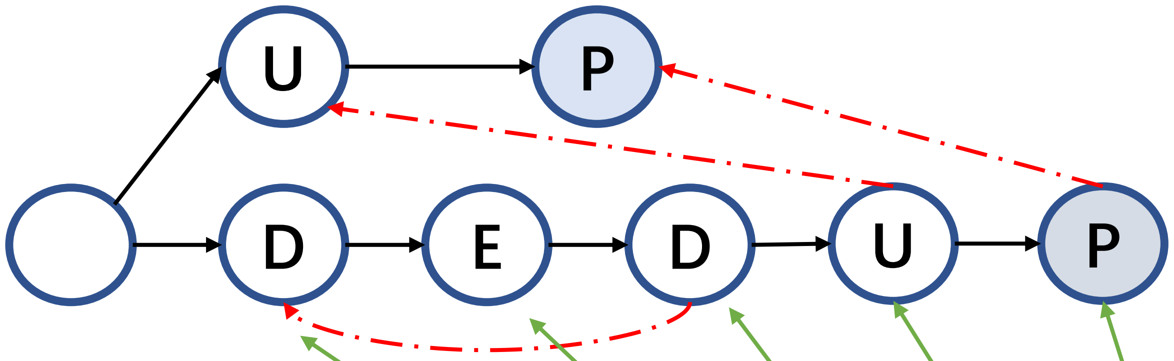
(a) Traditional system

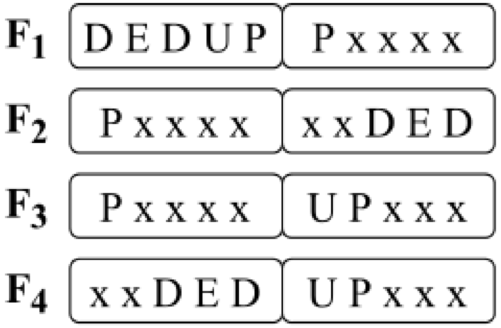
If our input is SECDEDEDUPSUPDEDU

S E C D E D E D U P S U P D E D U

Design —String-matching algorithm

The dictionary is {dedup, up}





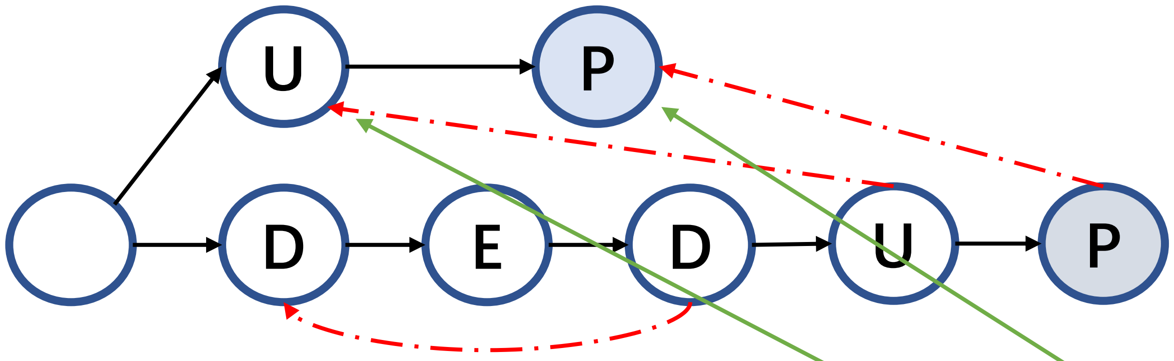
(a) Traditional system

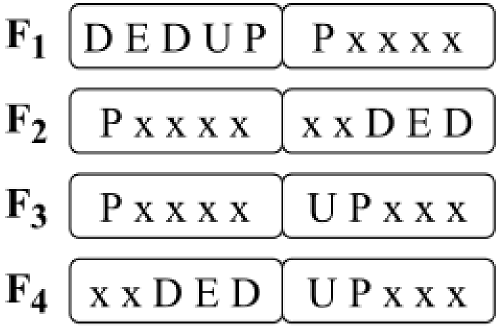
If our input is SECDEDEDUPSUPDEDU

S E C D E D E D U P S U P D E D U

Design —String-matching algorithm

The dictionary is {dedup, up}





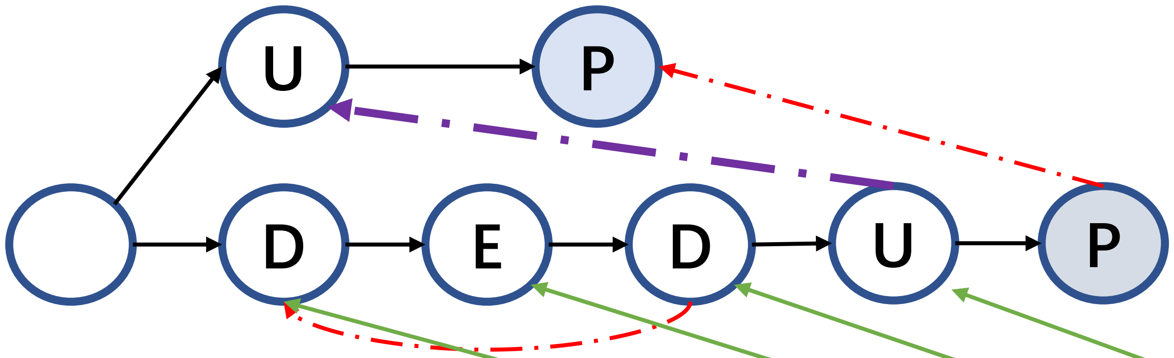
(a) Traditional system

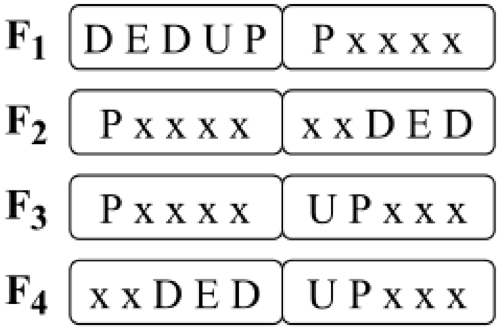
If our input is SECDEDEDUPSUPDEDU

S E C D E D E D U P S U P D E D U

Design —String-matching algorithm

The dictionary is {dedup, up}





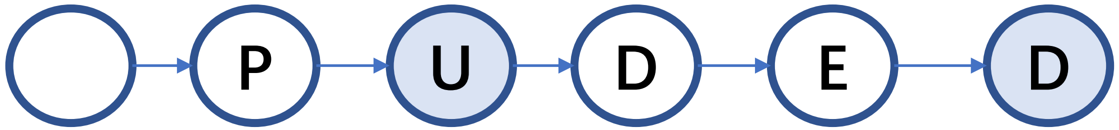
(a) Traditional system

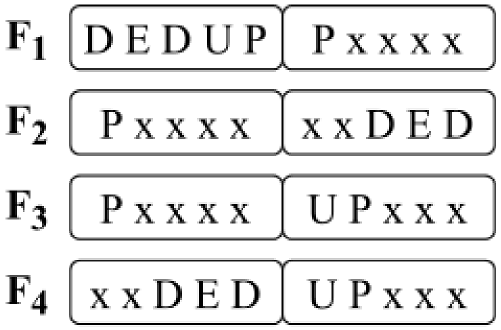
If our input is SECDEDEDUPSUPDEDU

S E C D E D E D U P S U P D E D U

Design —String-matching algorithm

The dictionary is {dedup, up}





(a) Traditional system

The first n bytes of the chunk, where n is the length of the longest string in the dictionary.

PXXXX → XXXXP

Design —Partial matches.

Let N denote the length of the keyword

Prefix as P_i Suffix as S_j

If $P_i + S_j$ = N, then they are match

$\mathbf{F_1}$	D E D U P	P x x x x
$\mathbf{F_2}$	P x x x x	x x D E D
$\mathbf{F_3}$	P x x x x	U P x x x
$\mathbf{F_4}$	x x D E D	U P x x x

The new problem is: a chunk may contain several prefix or suffix matches

P_1 = D or P_3 = DED record only the longest prefix and longest suffix in each chunk

---

D+EDUP, DE+DUP, DED+UP, DEDU+P, DED+EDUP

{(1,4),(2,3),(3,2),(4,1),(3,4)}.

	$j = 1$	2	3	4
$i = 1$				0 [D+EDUP]
2			0 [DE+DUP]	
3		0 [DED+UP]		2 [DED+EDUP]
4	0 [DEDU+P]			

Table 1: Partial-match table for DEDUP

Design —Match result database

In practice, the vast majority of the chunks contain at most one exact match

Chunk-result record

20byte	1byte	1byte	2byte	2byte
FP	|Prefix|	|Suffix|	# Exact	Offset
FP_0	0	0	1	0
FP_1	0	1	0	0
FP_2	3	0	0	0
FP_3	0	2	0	0

Memory-hash

$\mathbf{F_1}$ $\mathbf{F_2}$ $\mathbf{F_3}$ $\mathbf{F_4}$

FP	ID	|Pre|	|Suf|	#Exact	Off	ID	|Pre|	|Suf|	#Exact	Off
FP_3	0	0	2	0	0	1	0	0	1	0

C_0 C_1 C_2 C_3

Container 0 Container 1

Location-list record

20byte	1byte			
FP	Offset1	Offset2	Offset3	Offset4
FP_x	xx	xx	xx	xx

Memory-hash

Long location-list record:

20byte	1byte	1byte		
FP	Exact	Offset2	Offset3	Offset4
FP_x	xx	xx	xx	xx

Memory-hash

Design —Match result database

The new problem is: Keywords that begin or end with frequent letters in the alphabet might result in the allocation of numerous chunk-result records whose partial matches never generate a full match.

The tiny-result records are allocated only if this is the only match in the chunk

Chunk-result record

FP (20byte)	|Prefix| (1byte)	|Suffix| (1byte)	# Exact (2byte)	Offset (2byte)
FP_0	0	0	1	0
FP_1	0	1	0	0
FP_2	3	0	0	0
FP_3	0	2	0	0

Memory-hash

FP (20byte)	|Prefix| (1byte)	|Suffix| (1byte)
FP_1	0	1

Disk-hash

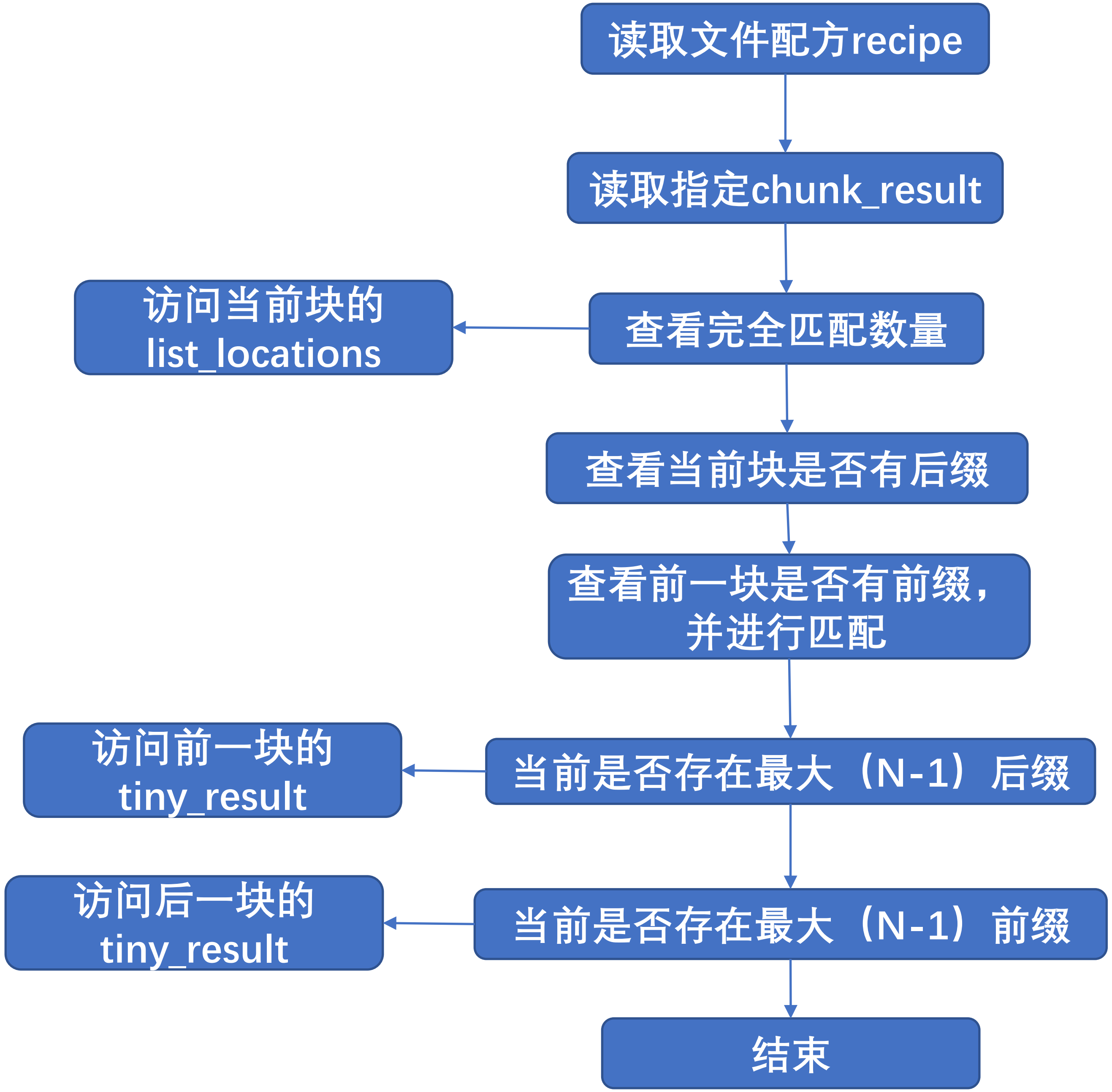
Tiny-result records are accessed during the logical phase only if the adjacent chunk contains a prefix or suffix of length n−1.

Design —logical phase

Algorithm 1 DedupSearch Logical Phase: handling FP_i in File F

Input: FP_i, FP_{i-1}, FP_{i+1}, res_{i-1}

$res_i \leftarrow chunk\_result[FP_i]$
if $res_i =$ NULL **then**
  `return`
if $res_i.exact\_matches > 0$ **then**
  add $file\ name$, $match\ offset$ to output
  if $res_i.exact\_matches > 1$ **then**
    $locations \leftarrow list\_locations[FP_i]$
    for all offsets in $locations$ **do**
      add $file\ name$, $offset$ to output
if $res_i.longest\_suffix > 0$ **then**
  if $res_{i-1} \neq$ NULL **then**
    if $res_{i-1}.longest\_prefix > 0$ **then**
      for all matches in $partial\text{-}match\_table[res_{i-1}.longest\_prefix, res_i.longest\_suffix]$ **do**
        add $file\ name$, $match\ offset$ to output
  else if $res_i.longest\_suffix = n - 1$ **then**
    $tiny \leftarrow tiny\_result[FP_{i-1}]$
    if $tiny \neq$ NULL & $tiny =$ `prefix` **then**
      add $file\ name$, $match\ offset$ to output
if $res_i.longest\_prefix = n - 1$ **then**
  $tiny \leftarrow tiny\_result[FP_{i+1}]$
  if $tiny \neq$ NULL & $tiny =$ `suffix` **then**
    add $file\ name$, $match\ offset$ to output

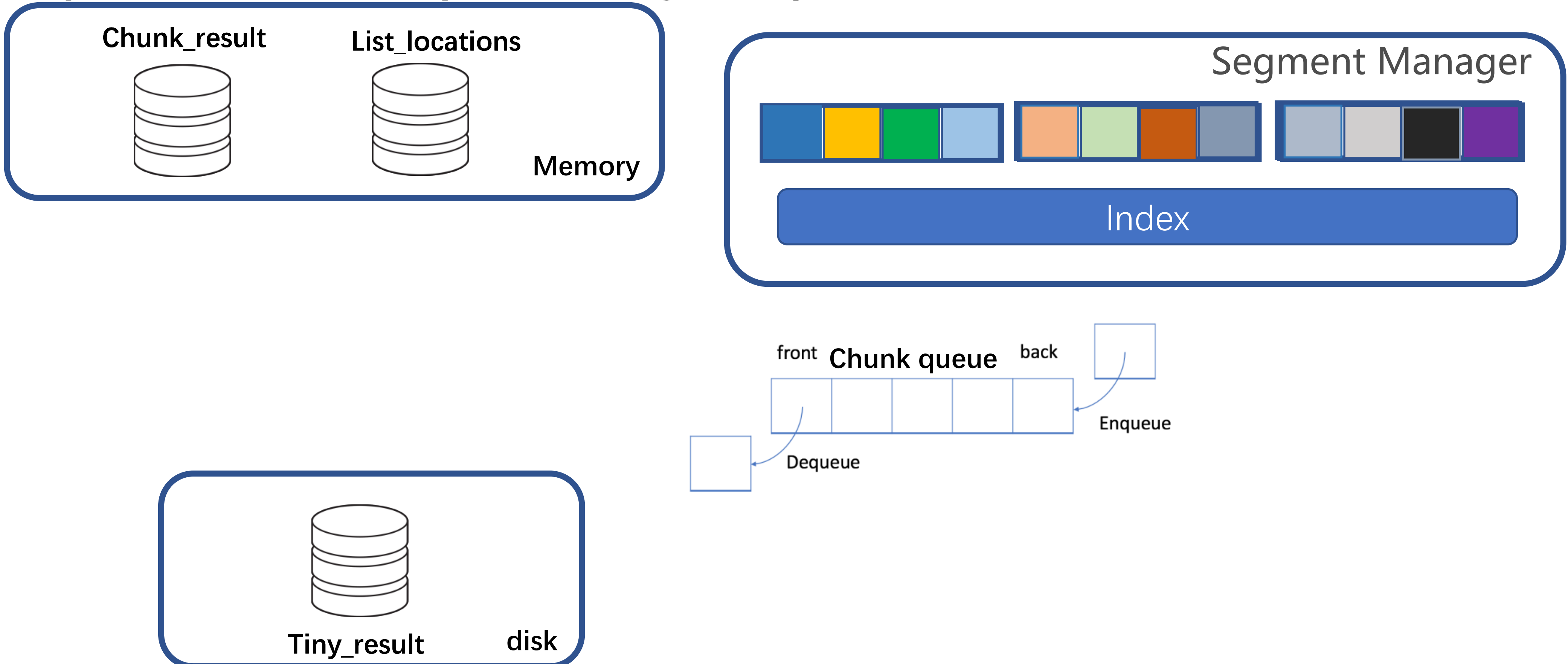


Implementation

- **Platform**

  Hardware：Xeon 4210 + 128GB DDR4 RAM + Dell R8DN1Y 1TB 2.5" SA TA HDD

  Software：(Open-source deduplication system) **Destor + Ubuntu 16.04.7**

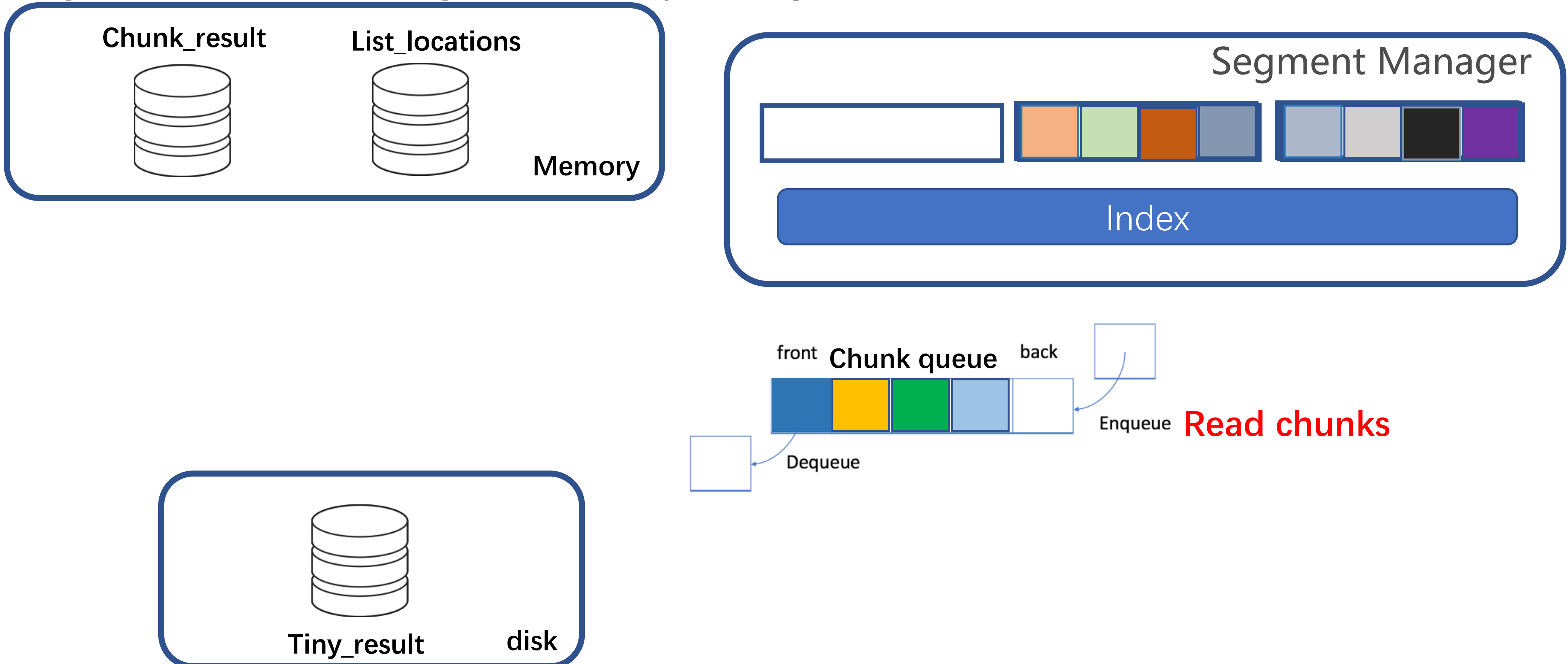


Implementation

➢ **Platform**

Hardware： Xeon 4210 + 128GB DDR4 RAM + Dell R8DN1Y 1TB 2.5" SA TA HDD

Software： (Open-source deduplication system) Destor + Ubuntu 16.04.7

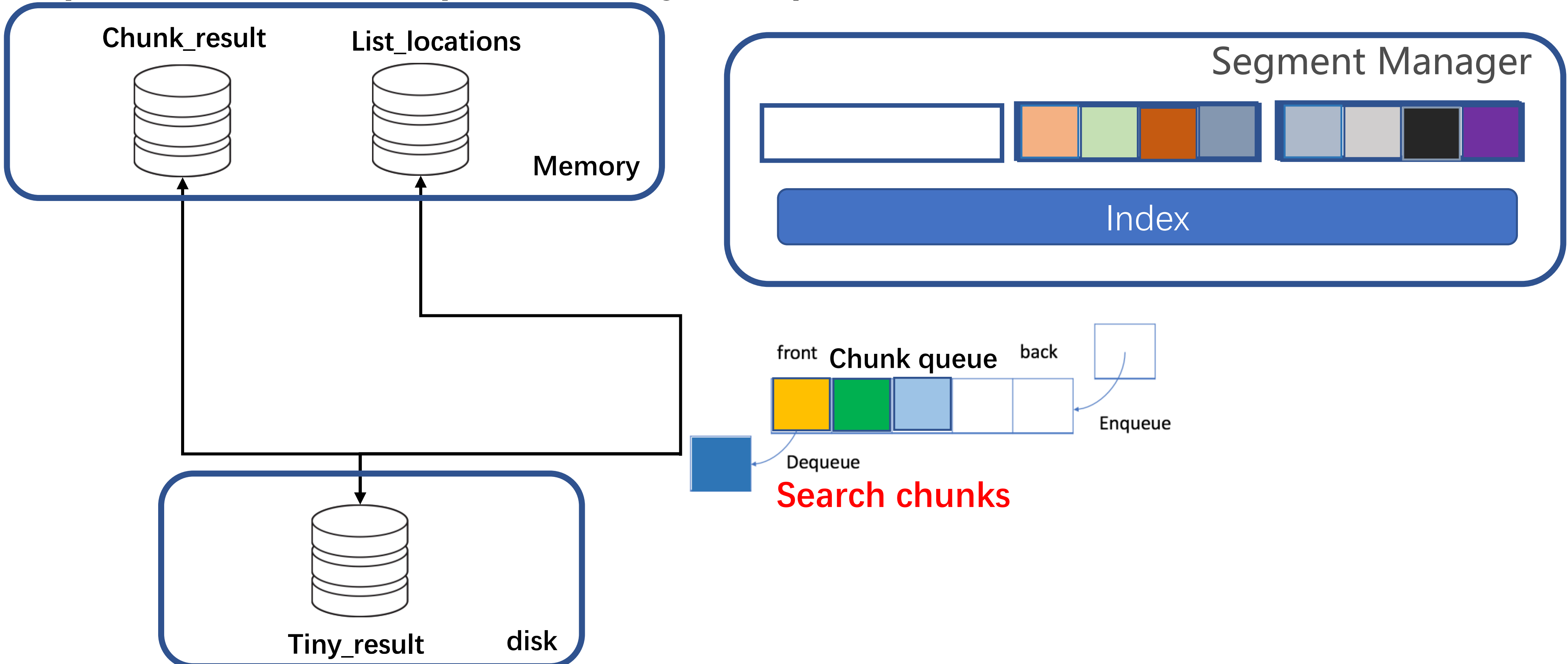


Implementation

➢ **Platform**

Hardware：Xeon 4210 + 128GB DDR4 RAM + Dell R8DN1Y 1TB 2.5" SA TA HDD

Software：(Open-source deduplication system) Destor + Ubuntu 16.04.7

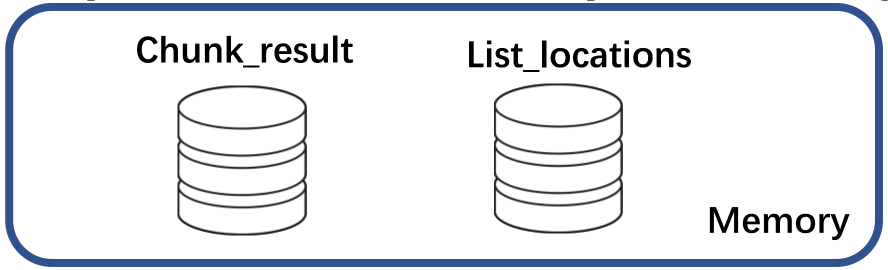


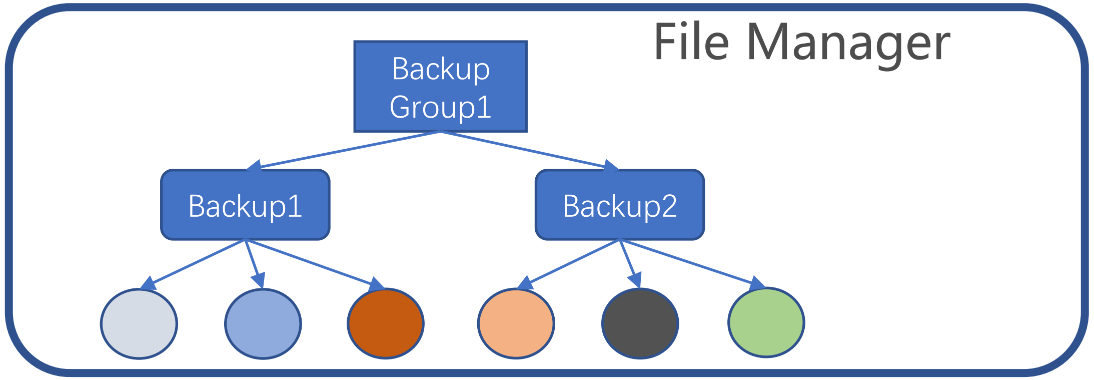
Implementation

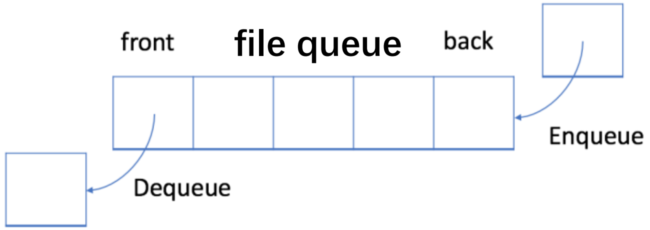
- **Platform**

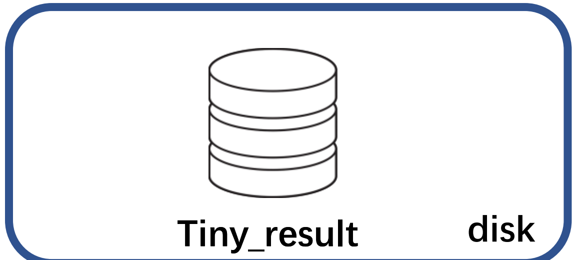
Hardware： Xeon 4210 + 128GB DDR4 RAM + Dell R8DN1Y 1TB 2.5" SA TA HDD

Software： (Open-source deduplication system) **Destor + Ubuntu 16.04.7**







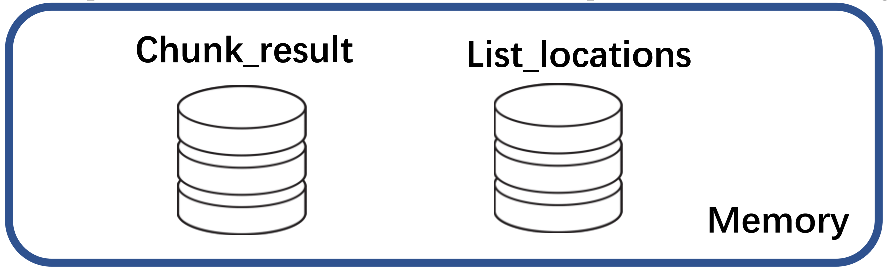


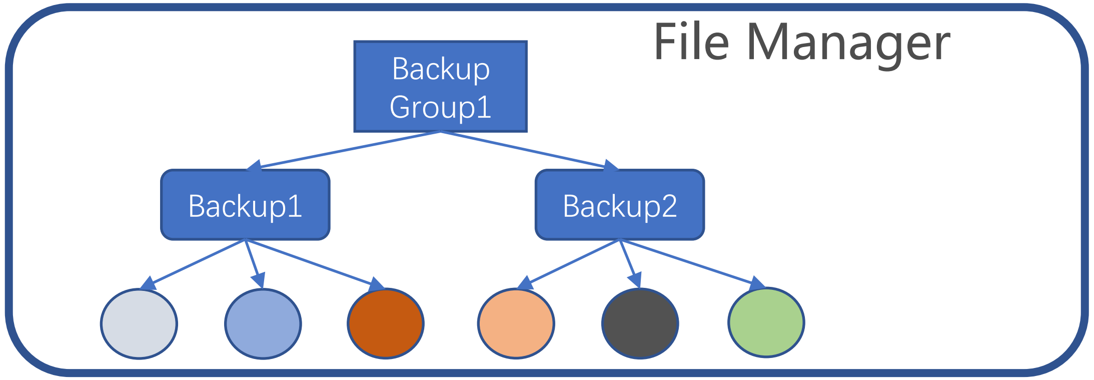
Implementation

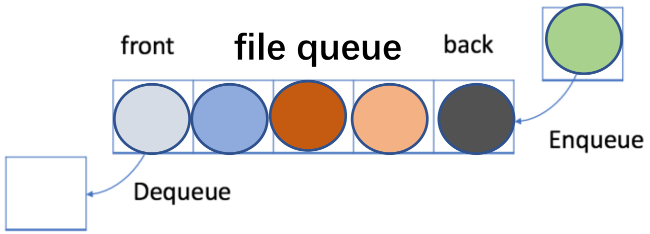
- **Platform**

Hardware： Xeon 4210 + 128GB DDR4 RAM + Dell R8DN1Y 1TB 2.5" SA TA HDD

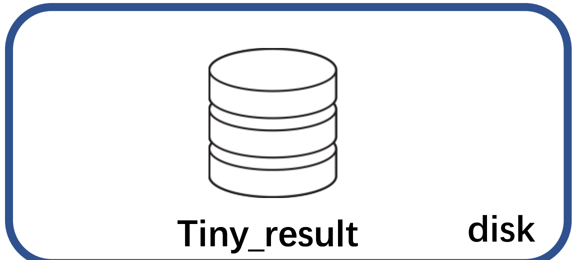
Software： (Open-source deduplication system) Destor + Ubuntu 16.04.7







Read file-recipe

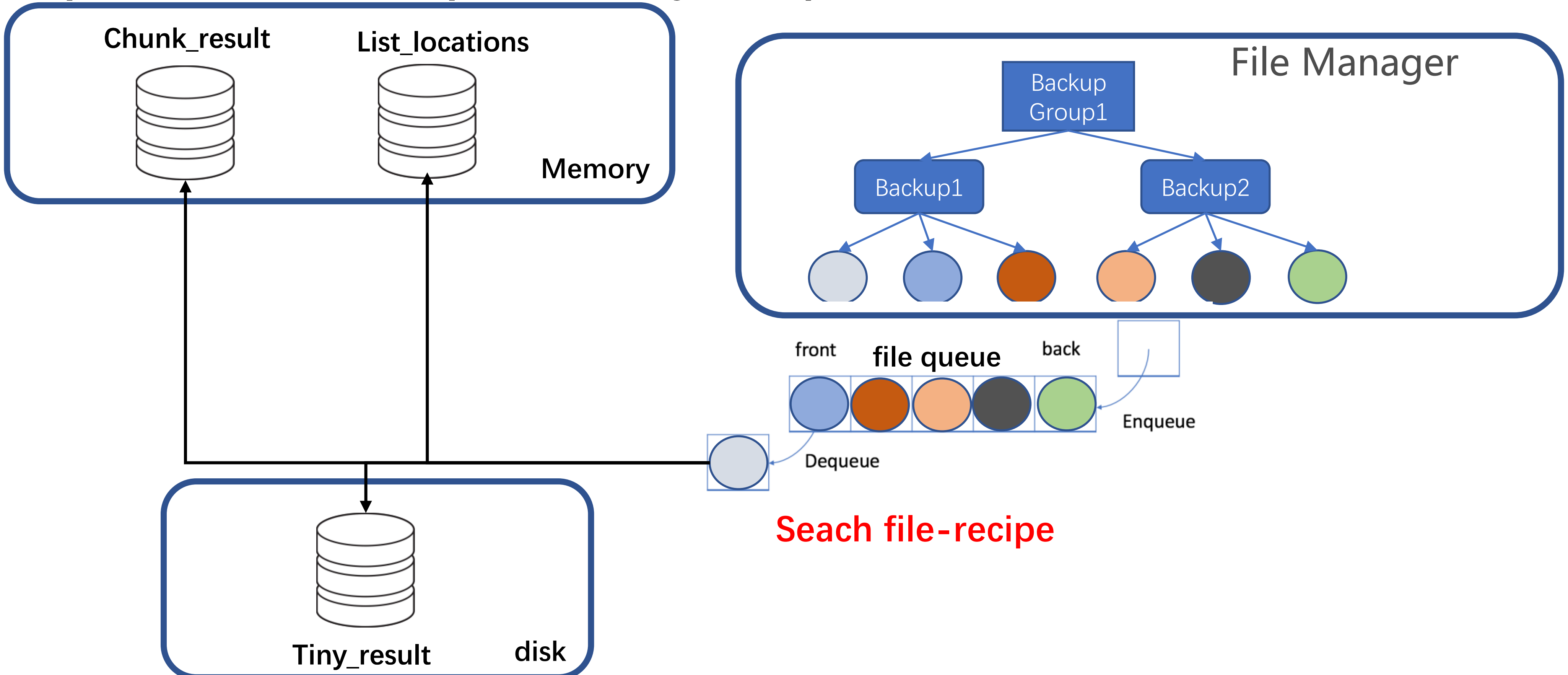


Implementation

- **Platform**

  Hardware： Xeon 4210 + 128GB DDR4 RAM + Dell R8DN1Y 1TB 2.5" SA TA HDD

  Software： (Open-source deduplication system) Destor + Ubuntu 16.04.7



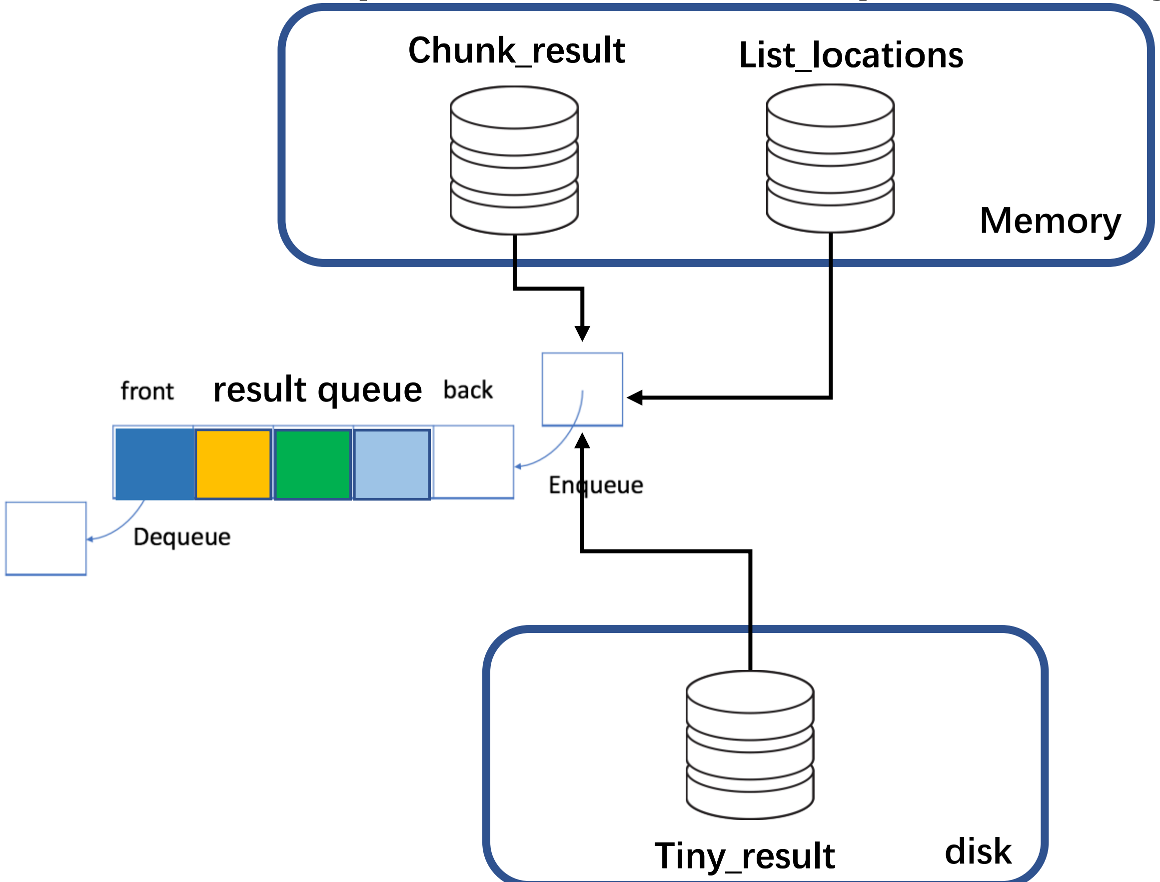
Seach file-recipe

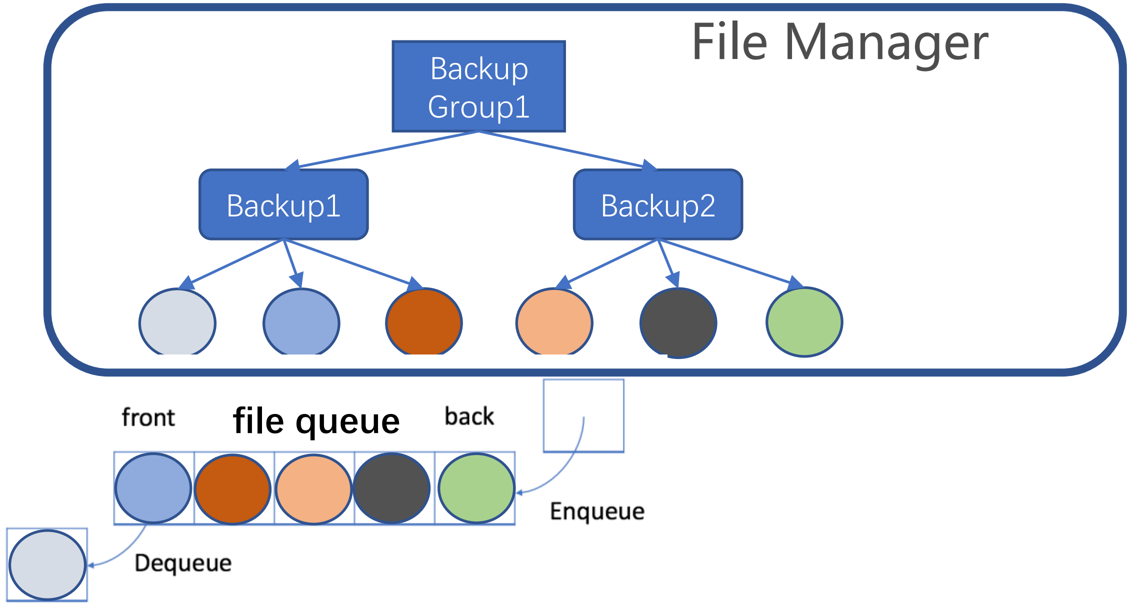
Implementation

- **Platform**

Hardware： Xeon 4210 + 128GB DDR4 RAM + Dell R8DN1Y 1TB 2.5" SA TA HDD

Software： (Open-source deduplication system) Destor + Ubuntu 16.04.7





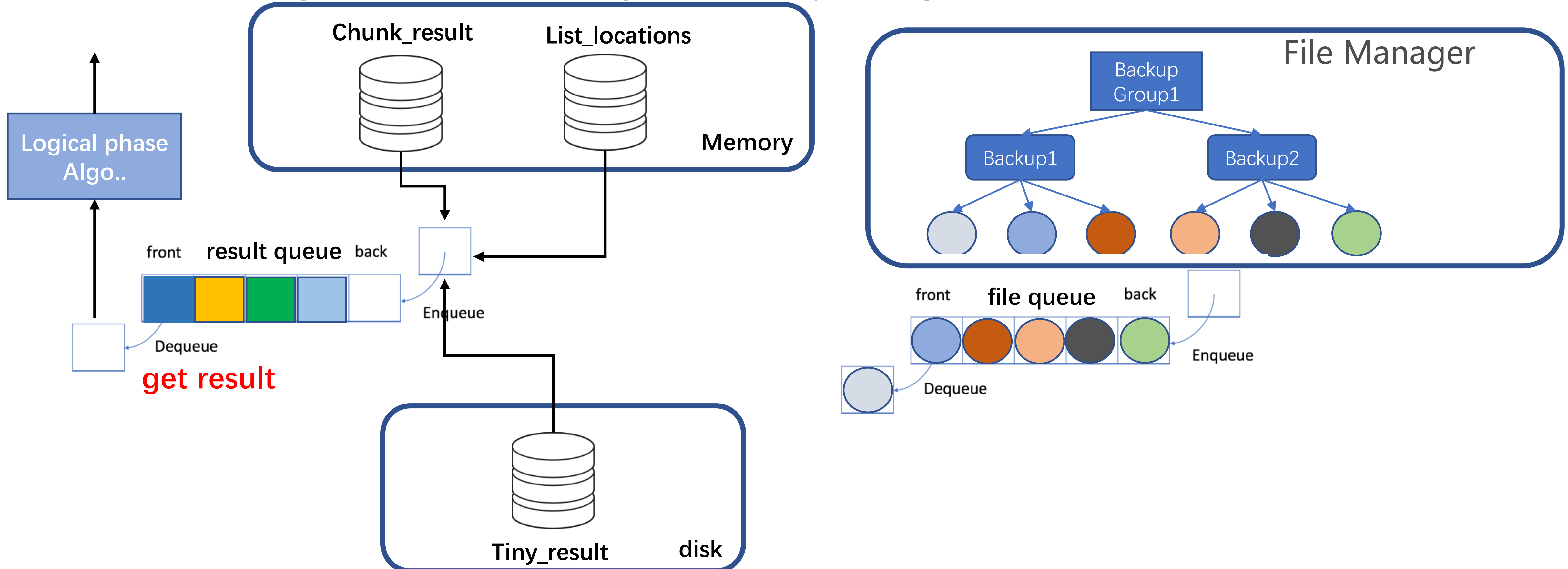
Seach file-recipe

Implementation

- **Platform**

Hardware：Xeon 4210 + 128GB DDR4 RAM + Dell R8DN1Y 1TB 2.5" SA TA HDD

Software：(Open-source deduplication system) Destor + Ubuntu 16.04.7



Implementation

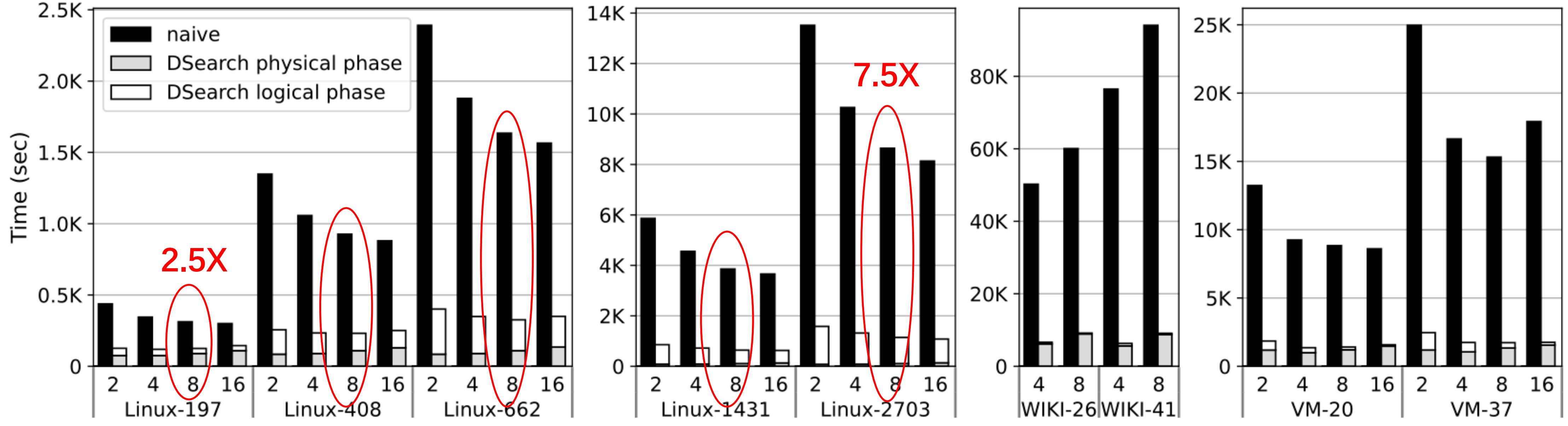
➢ Datasets

Dataset	Logical size (GB)	Physical size + metadata size (GB)			
		2KB	4KB	8KB	16KB
Wiki-26 (skip)	1692		667+16 40.4%	861+9 51.4%	
Wiki-41 (consecutive)	2593		616+22 24.6%	838+12 32.8%	
Linux-197 (Minor versions)	58	10+1 19%	10+1 19%	11+1 20.7%	13+1 24.1%
Linux-408 (every 10th patch)	204	10+4 6.9%	10+4 6.9%	15+2 7.4%	16+2 8.8%
Linux-662 (every 5th patch)	377	10+7 4.5%	11+5 4.2%	13+4 4.5%	17+3 5.3%
Linux-1431 (every 2nd patch)	902	10+18 3.1%	11+13 2.7%	10+13 2.5%	17+8 2.8%
Linux-2703 (every patch)	1796	10+34 2.5%	10+26 2.0%	13+20 1.9%	17+17 1.9%
VM-37 (1-2 days skips)	2469	145+33 7.2%	129+18 6.0%	156+10 6.7%	192+5 8.0%
VM-20 (3-4 days skips)	1349	143+19 12.0%	125+10 10.0%	150+6 11.6%	181+3 13.6%

➢ Keywords

Dictionary	Avg. pre/suf length	Avg. # pre/suf	Avg. # occurrences	Avg. keyword length
Wiki-high	1.09	85.3 M	722	8.4
Wiki-med	1.10	42.2 M	699	7.8
Wiki-low	1.08	5.7 M	677	6.0
Linux-high	1.09	64.8 M	653	10.5
Linux-med	1.20	32.8 M	599	10.4
Linux-low	1.13	5.7 M	583	10.4
Linux-line	1.22	31.4 M	63	25.9
VM-16	1.00	8.7 M	31	16
VM-64	1.00	8.6 M	29	64
VM-256	1.00	8.6 M	27	256
VM-1024	1.00	8.6 M	27	1024

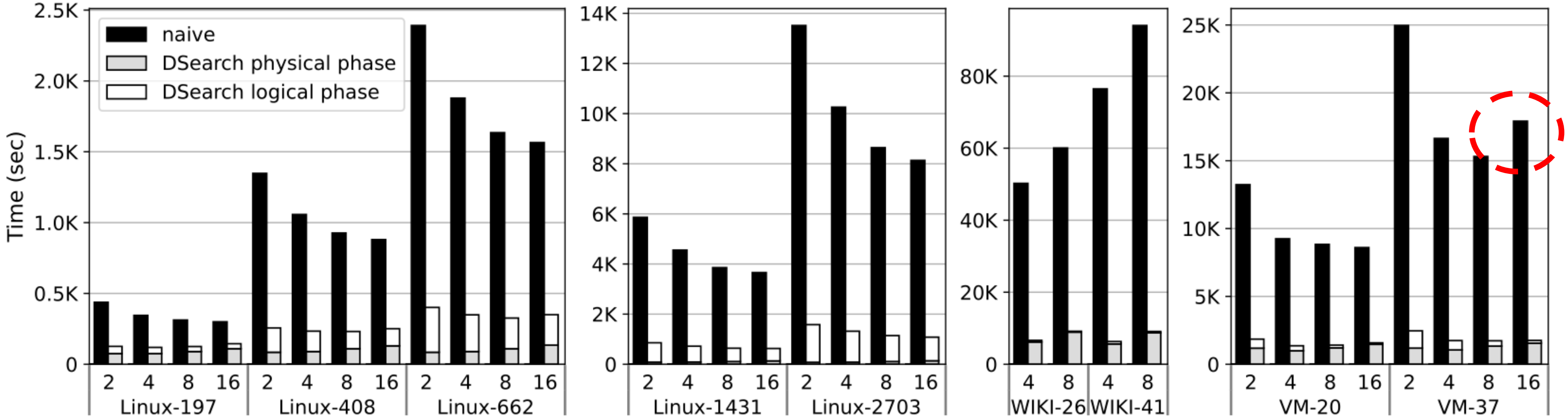
Evaluation —DedupSearch performance



➢ **Effect of deduplication ratio.**

- The difference between them increases as the deduplication ratio (the ratio between the physical size and the logical size) decreases.
- The increase occurs only in the logical phase, due to the increase in the number of file recipes that are processed.
- The time of the physical phase remains roughly the same, as it depends only on the physical size of the dataset.

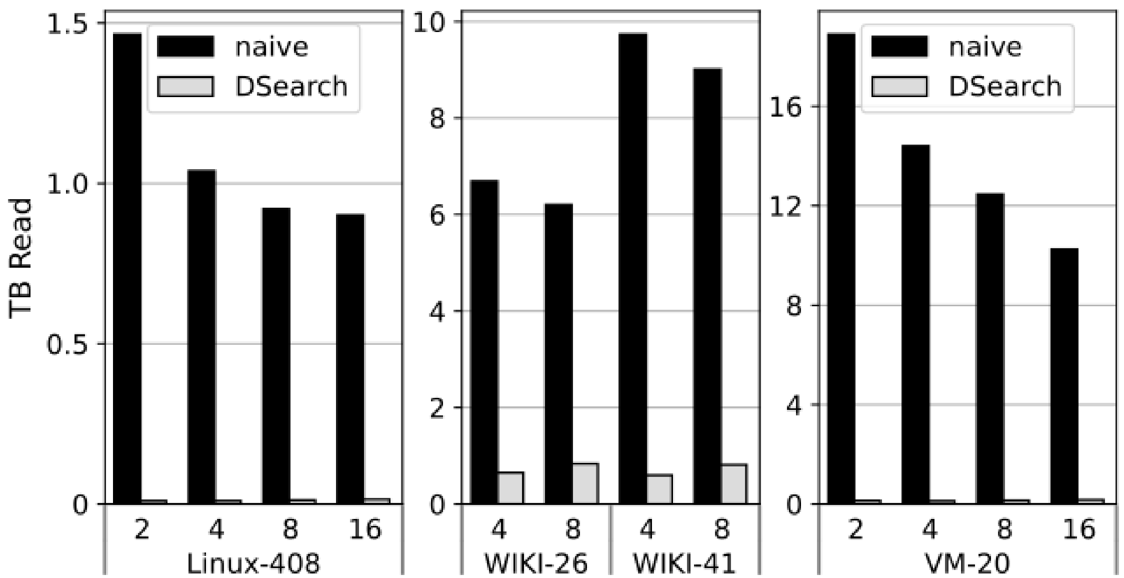
Evaluation —DedupSearch performance



- **Effect of chunk size.**

- Smaller chunks result in better deduplication but increase the size of the fingerprint index.

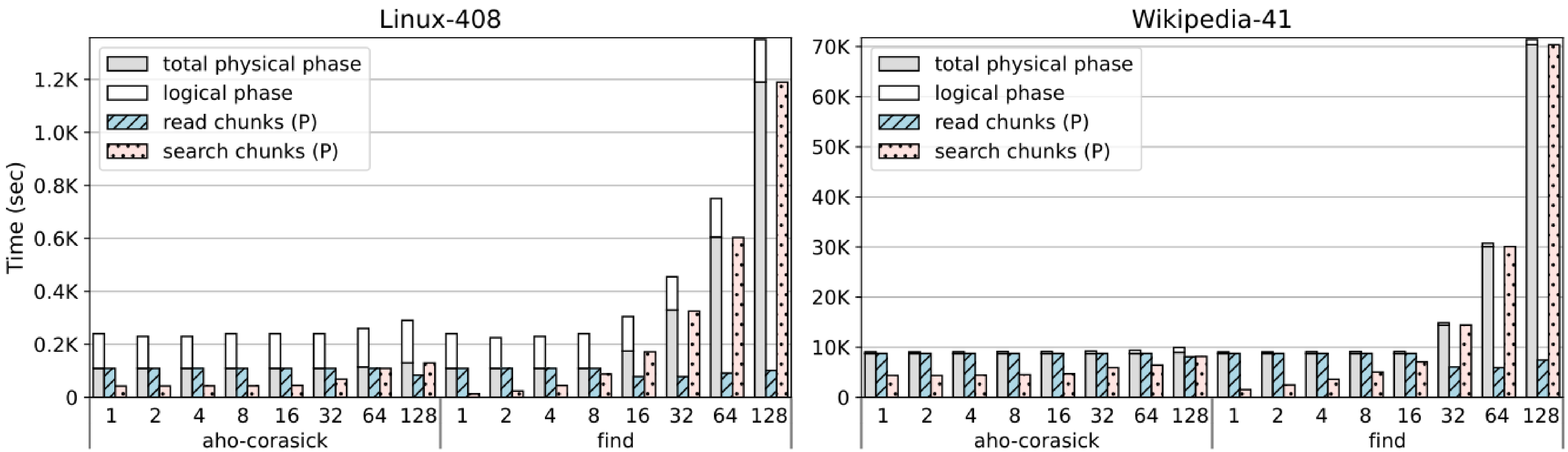
Evaluation —DedupSearch performance



➢ **Effect of chunk size.**

- For Naïve, the amount of data read increases with the logical size and decreases with the chunk size.
- For DedupSearch, the amount of data read is proportionate to the physical size of the dataset, regardless of its logical size.

Evaluation —DedupSearch performance

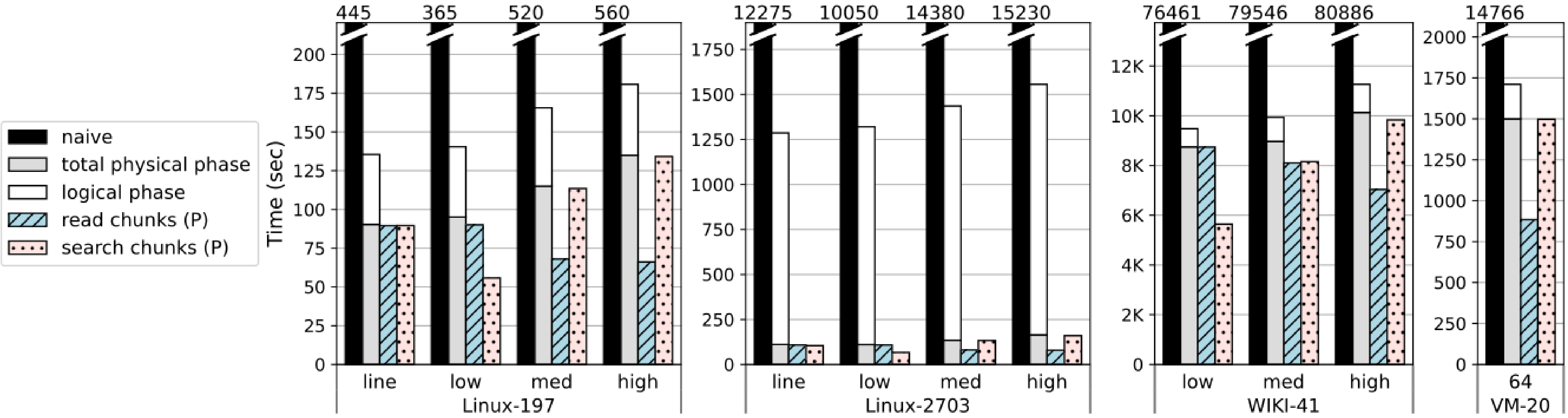


- **Effect of dictionary size.**

- When the Aho-Corasick algorithm is used, the chunks' processing time increases sublinearly with the number of keywords in the search query.

- The find is more efficient than Aho-Corasick when the number of keywords is small

Evaluation —DedupSearch performance



- ➢ **Effect of keywords in the dictionary..**

Evaluation —DedupSearch data structures

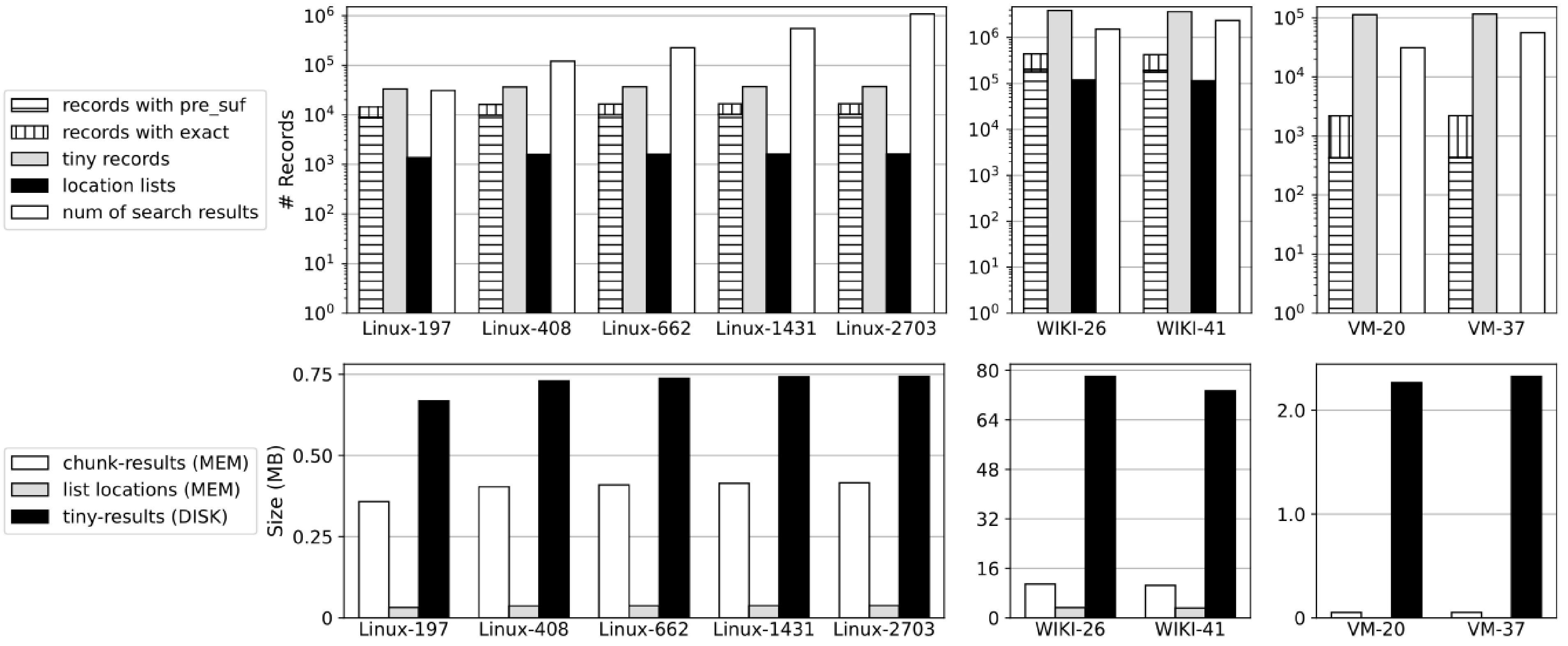
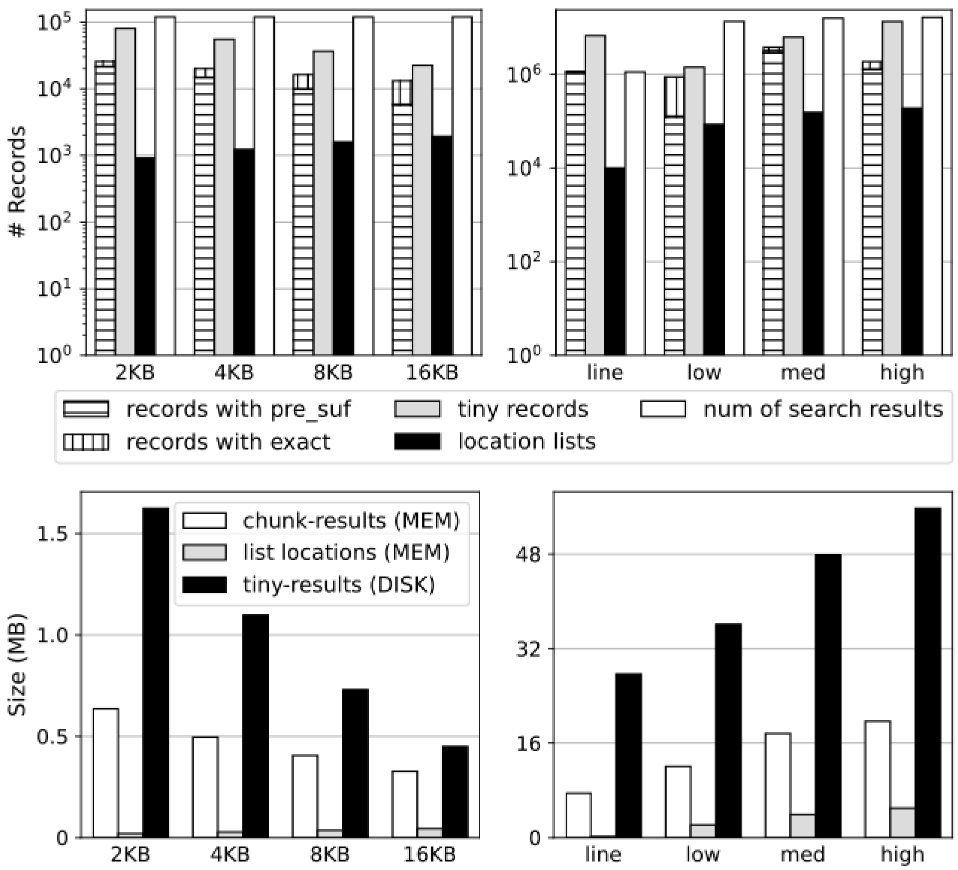


Figure 7: The number of search results and result objects during the search of a single keyword from the 'med' dictionary (top, note the log scale of the y-axis), and the corresponding database sizes (bottom).

- Index sizes

Evaluation —DedupSearch data structures



Dataset	# results (M)	% matches split	# tiny records (M)	# tiny accesses	tiny hit rate
Wiki-26	1.52 208.50	0.05 0.10	3.90 490.31	1167 44,719	0.10 0.94
Wiki-41	2.34 321.07	0.05 0.09	3.67 459.96	1780 69,094	0.10 0.94
Linux-197	0.03 5.08	0.19 0.12	0.03 4.19	59 1,665	0.08 0.73
Linux-408	0.12 16.08	0.19 0.11	0.04 4.57	197 5,986	0.15 0.71
Linux-662	0.23 29.16	0.19 0.11	0.04 4.63	360 11,101	0.16 0.70
Linux-1431	0.55 68.96	0.18 0.11	0.04 4.67	855 26,682	0.16 0.70
Linux-2703	1.08 134.65	0.18 0.11	0.04 4.68	1673 52,391	0.17 0.69
VM-20	0.03 4.02	0.00 1.61	0.11 14.62	0 0	N/A N/A
VM-37	0.06 7.24	0.00 1.61	0.12 14.96	0 0	N/A N/A

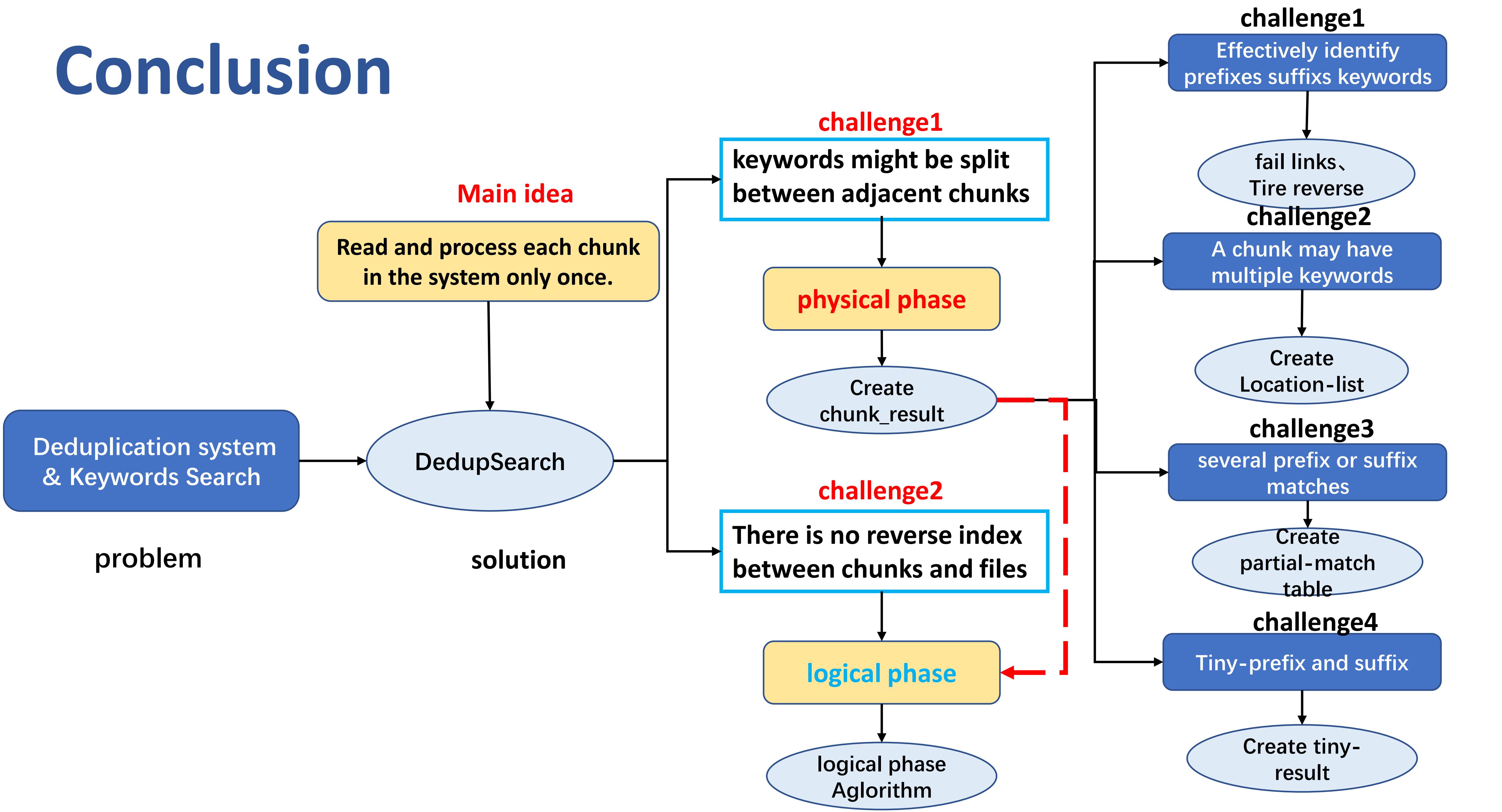
- Index sizes
- Database access

Evaluation — DedupSearch overheads

Dataset	Logical size	Physical size	Dedup ratio	Naïve time	DSearch time (logical)
Wiki-1	76	76	99.8%	616	620 (11)
LNX-1	1	0.80	80%	7.4	6.7 (0.6)
LNX-1-merge	0.82	0.78	95%	6.2	6.1 (0.1)
LNX-408	204	17	7.4%	926	231 (121)
LNX-408-merge	169	19	11.2%	768	203 (28)

The size (in GB) and dedup ratio of the datasets created from a single archived version with 8KB chunks, and the time (in seconds) to search a single keyword from the 'med' dictionary.

Conclusion



Main idea
Read and process each chunk in the system only once.
Deduplication system & Keywords Search
DedupSearch
problem
solution
challenge1
keywords might be split between adjacent chunks
physical phase
Create chunk_result
challenge2
There is no reverse index between chunks and files
logical phase
logical phase Aglorithm
challenge1
Effectively identify prefixes suffixs keywords
fail links、 Tire reverse
challenge2
A chunk may have multiple keywords
Create Location-list
challenge3
several prefix or suffix matches
Create partial-match table
challenge4
Tiny-prefix and suffix
Create tiny-result