CSCI4180 Tutorial-8:
Assignment 3 Review (Part-1)

Zuoru Yang

zryang@cse.cuhk.edu.hk

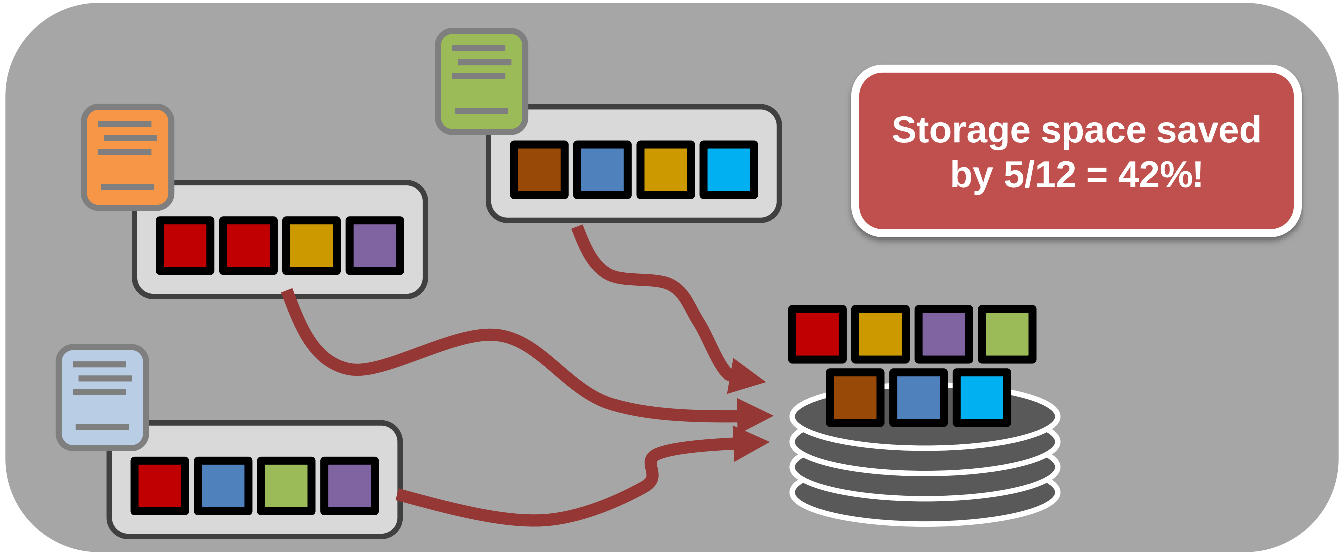
Nov. 18, 2020

Content

- Deduplication
- Variable-size chunking
- Checksum
- Indexing
- Hints
- Debug

Deduplication

- ➢ Deduplication → Coarse-grained compression
- ➢ Unit: chunk: fixed-size or variable-size
  - compute a fingerprint
    - Same fingerprint → same content
- ➢ Store only one copy of chunks with same content; other chunks refer to the copy by refences (pointers)



Deduplication (Cont.)

- Why “removing duplicate data”
  - Storage efficiency
  - Reducing data transfer
    - Removing the duplicate data in the client side
- Three main components
  - 1. Chunking: divide data into different chunks
  - 2. Checksum: compute checksum to identify each chunk
  - 3. Indexing: a search structure for efficient access of the information of chunks
    - Check whether a chunk is duplicated

Content

- Deduplication
- Variable-size chunking
- Checksum
- Indexing
- Hints
- Debug

Variable-size Chunking

- Why not using fixed-size chunking?
  - Boundary-shift problem
  - Vulnerable to data modification
    - Insert or delete
- Variable-size chunking
  - Content-defined chunking
    - Identify each chunk according to its content
  - Rabin fingerprint algorithm
    - An efficient rolling hash

https://en.wikipedia.org/wiki/Rabin_fingerprint

Rabin Fingerprint Algorithm

- What is fingerprint?
  - Fingerprint is the identifier of data
  - We use Rabin fingerprint algorithm to compute the fingerprint of the data
    - A method for implementing fingerprints using polynomials over a finite field.
  - Rabin fingerprint algorithm is a classical chunking algorithm, but it is not state-of-the-art.
    - Performance is not good
    - Faster chunking algorithm: FastCDC
    - But it is a good start point

https://en.wikipedia.org/wiki/Rabin_fingerprint

Rabin Fingerprint Algorithm (Cont.)

- Formula (How to compute Rabin fingerprint)

$$p_s(d,q) = \begin{cases} \left(\sum_{i=1}^{m} t_i \times d^{m-i}\right) \ mod\ q, & s = 0 \\ \left(d \times (p_{s-1} - d^{m-1} \times t_s) + t_{s+m}\right) \ mod\ q, & s > 0 \end{cases}$$

- Symbols
  - p_s: [result] The fingerprint we calculate.
  - t_i: [data] Usually t_i is 1 byte of data.
  - m: [parameter] Window size (bytes).
  - d: [parameter] base.
  - q: [parameter] modulus

https://en.wikipedia.org/wiki/Rabin_fingerprint

Rabin Fingerprint Algorithm (Cont.)

- Example
  - Start of the file

1	**9**	**6**	**4**	**8**	**6**	**3**	**5**	**…**

  - Parameters
    - m = 4 (window size)
    - d = 10 (base)
    - q = 13 (modulus)

Rabin Fingerprint Algorithm (Cont.)

- Example
  - index = 0

1	**9**	**6**	**4**	**8**	**6**	**3**	**5**	**…**

  - Calculate p_0

$p_0 = (t_1 * d^3 + t_2 * d^2 + t_3 * d^1 + t_4 * d^0) \bmod 13$

$p_0 = (1 * 10^3 + 9 * 10^2 + 6 * 10^1 + 4 * 10^0) \bmod 13$

$p_0 = 1$

Rabin Fingerprint Algorithm (Cont.)

- Example
  - index = 1

1	9	6	4	8	6	3	5	…

  - Calculate p_1

p_1 = (d * (p_0 – d^3 * t_1) + t_5) mod 13

p_1 = (10 * (1 – 10^3 * 9) + 8) mod 13

p_1 = 2

We can calculate the following fingerprint in the same manner.

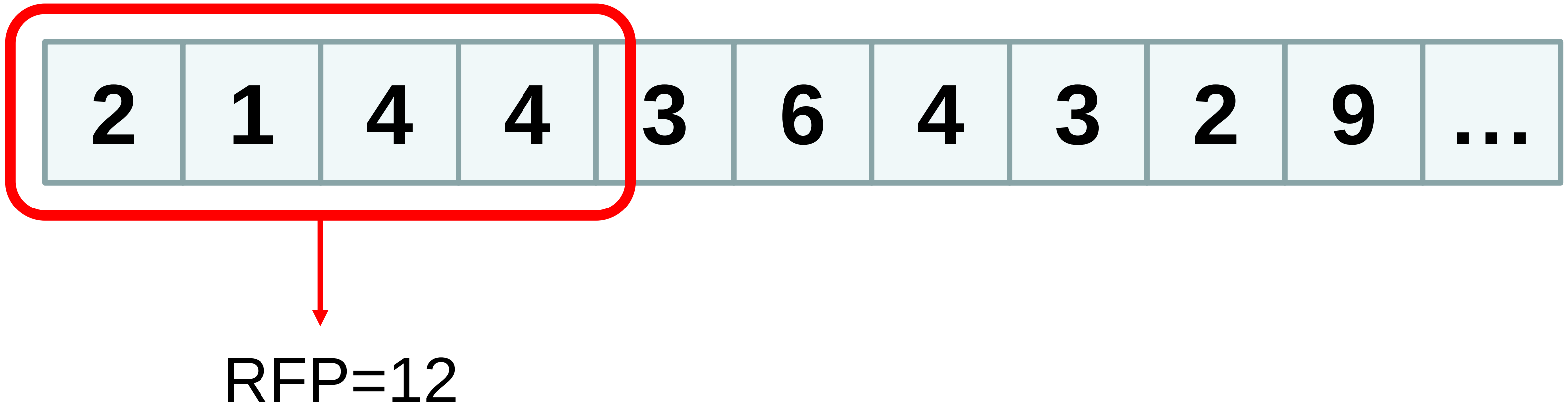
Chunking With RFP Algorithm

- How to do variable-size chunking via RFP algorithm?
  - We need these parameters
    - For calculate the robin fingerprint
      - m: window size
      - d: base
      - p: modulus
    - For chunking
      - mask: multiple 1-bits
        - To control the average chunk size

Chunking With RFP Algorithm (Cont.)

➢ How to do variable-size chunking via RFP algorithm?

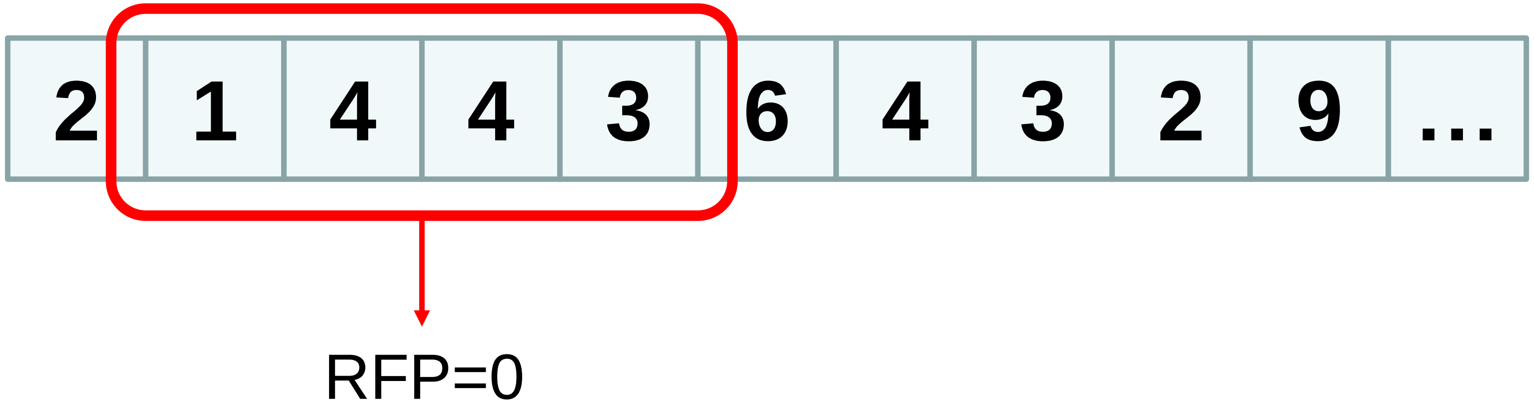
- Example: m=4, d=10, q=13, mask=$(1111)_2$=15



- RFP & mask != 0
- Think about: in which case will RFP & mask = 0 ?

Chunking With RFP Algorithm (Cont.)

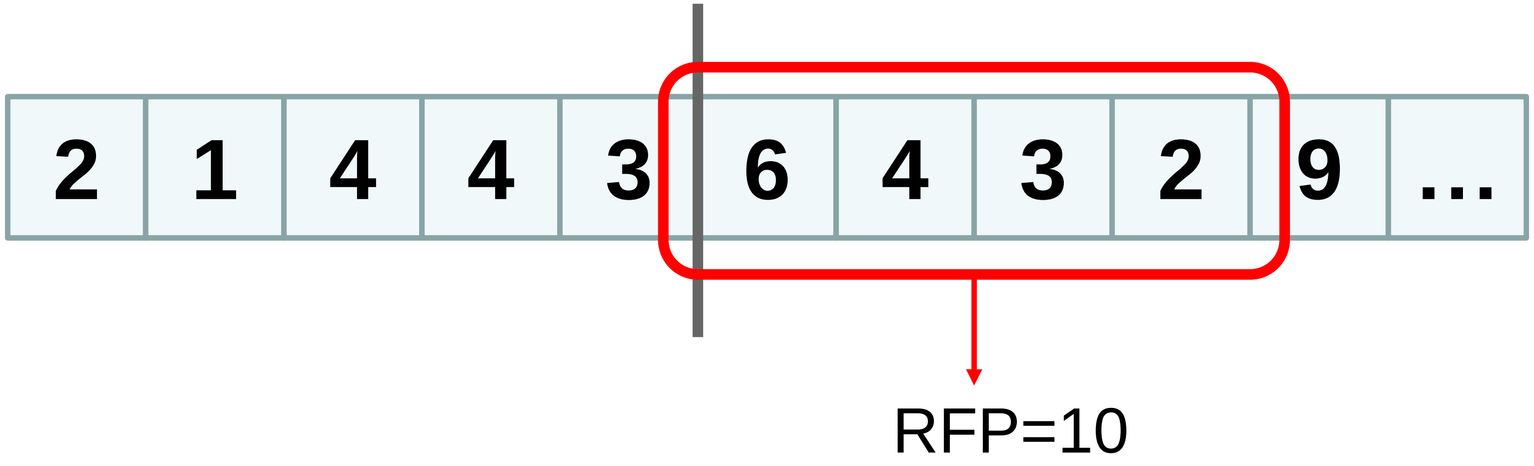
- How to do variable-size chunking via RFP algorithm?
  - Example: m=4, d=10, q=13, mask=$(1111)_2$=15



- RFP & mask = 0
- Set an anchor point at the end of current window

Chunking With RFP Algorithm (Cont.)

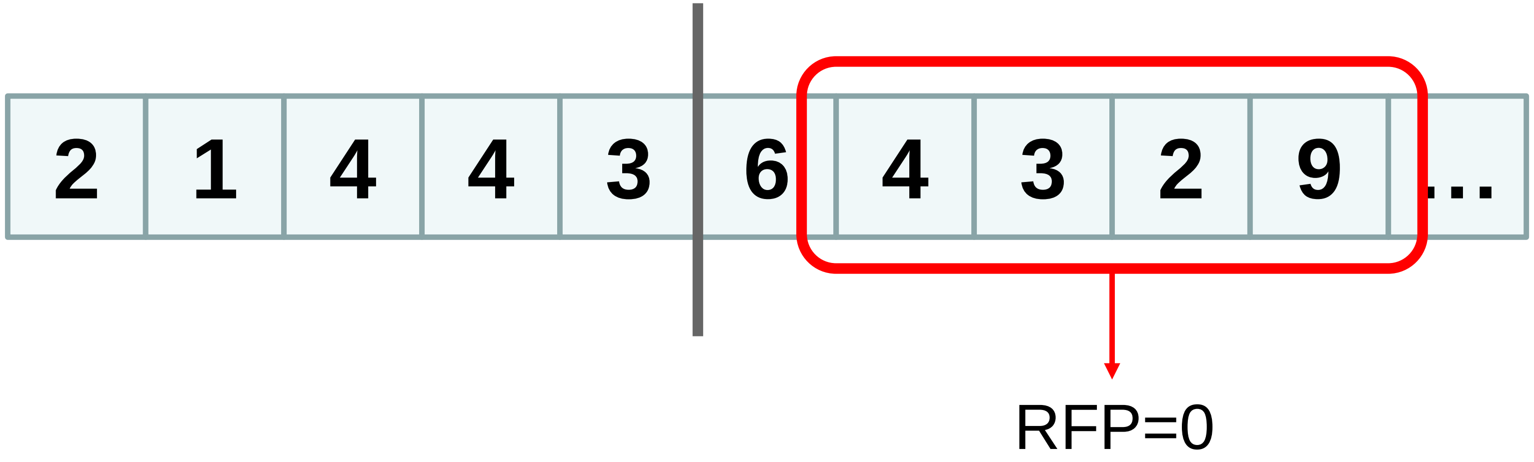
- How to do variable-size chunking via RFP algorithm?
  - Example: m=4, d=10, q=13, mask=$(1111)_2$=15



  - RFP & mask != 0

Chunking With RFP Algorithm (Cont.)

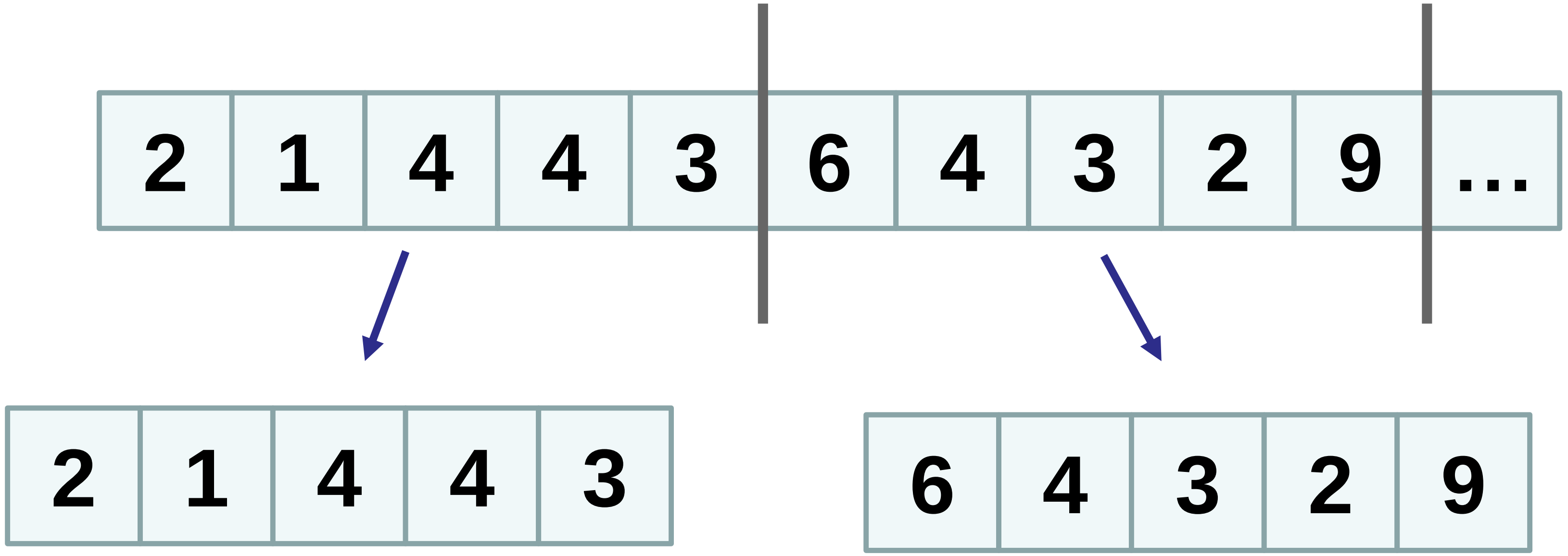
- How to do variable-size chunking via RFP algorithm?
  - Example: m=4, d=10, q=13, mask=$(1111)_2$=15



- RFP & mask = 0
- Set an anchor point

Chunking With RFP Algorithm (Cont.)

- How to do variable-size chunking via RFP algorithm?
  - Example: m=4, d=10, q=13, mask=$(1111)_2$=15



  - Up to now, we create 2 chunks with RFP algorithm

Zero Chunks

- We also need to recognize long runs of zeros when we do chunking
  - Special features of zero chunk
    - It can be longer than the maximum chunk size.
      - e.g. min_chunk = 5, max_chunk = 10. A file of 20 bytes contains all zeros. In this case, there will be only one logical chunk for this file.
    - It can be smaller than the minimum chunk size.
      - This case can only happen when the zero run is at the end of a file
      - e.g. |chunk1|chunk2|chunk3|…|chunkn|000|

Summary of chunking

- Basically, we start from a buffer whose size is minimum size of a chunk, except for EOF.

- Then we need to figure out to proceed with zero run, or to proceed with RFP algorithm for chunking

- Chunking
  - A zero run is found
  - RFP & mask = 0
  - Maximum size of a chunk is reached

Content

- Deduplication
- Variable-size chunking
- Checksum
- Indexing
- Hints
- Debug

Chunksum

- How to calculate checksum of a data chunk?
  - We can use **SHA-256** algorithm to create checksum, which is available in **Java MessageDigest Library**
    - Note that: for our assignment, we choose MD5 algorithm
  - Example:
    - There is a data buffer **data**, whose length is **len**

```
MessageDigest md = MessageDigest.getInstance(“SHA-256”);
md.update(data, 0, len);
byte[] checksumBytes = md.digest();
```

Content

- Deduplication
- Variable-size chunking
- Checksum
- Indexing
- Hints
- Debug

Indexing

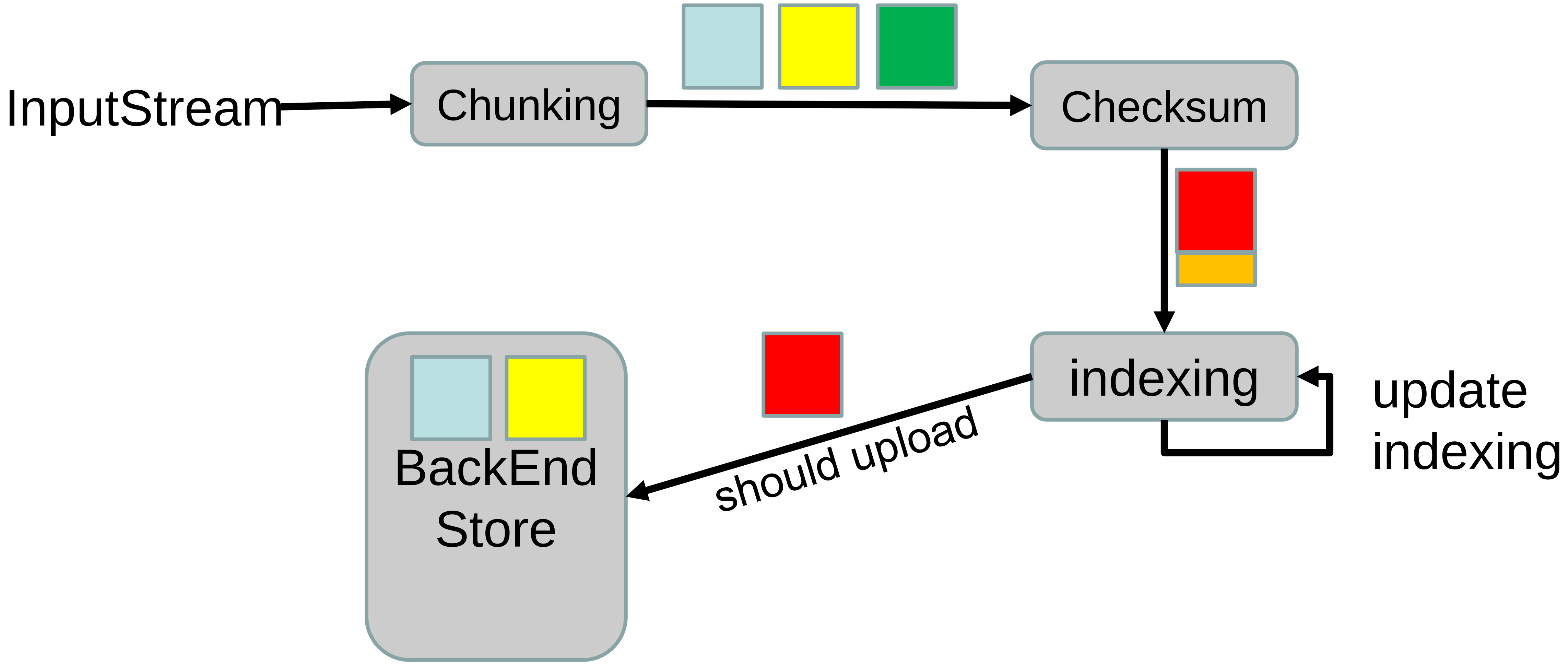
- Two kinds of indexes
  - 1. Fingerprint indexing: to manage chunks
  - 2. File recipe: to manager files
- Fingerprint indexing
  - It is a data structure
  - Given a fingerprint value
    - Return whether corresponding chunk exists
- File recipe
  - Given a file recipe
    - Return all chunks' information of this file

Content

- Deduplication
- Variable-size chunking
- Checksum
- Indexing
- Hints
- Debug

Hints

- Upload process
  - The whole workflow



Hints (Cont.)

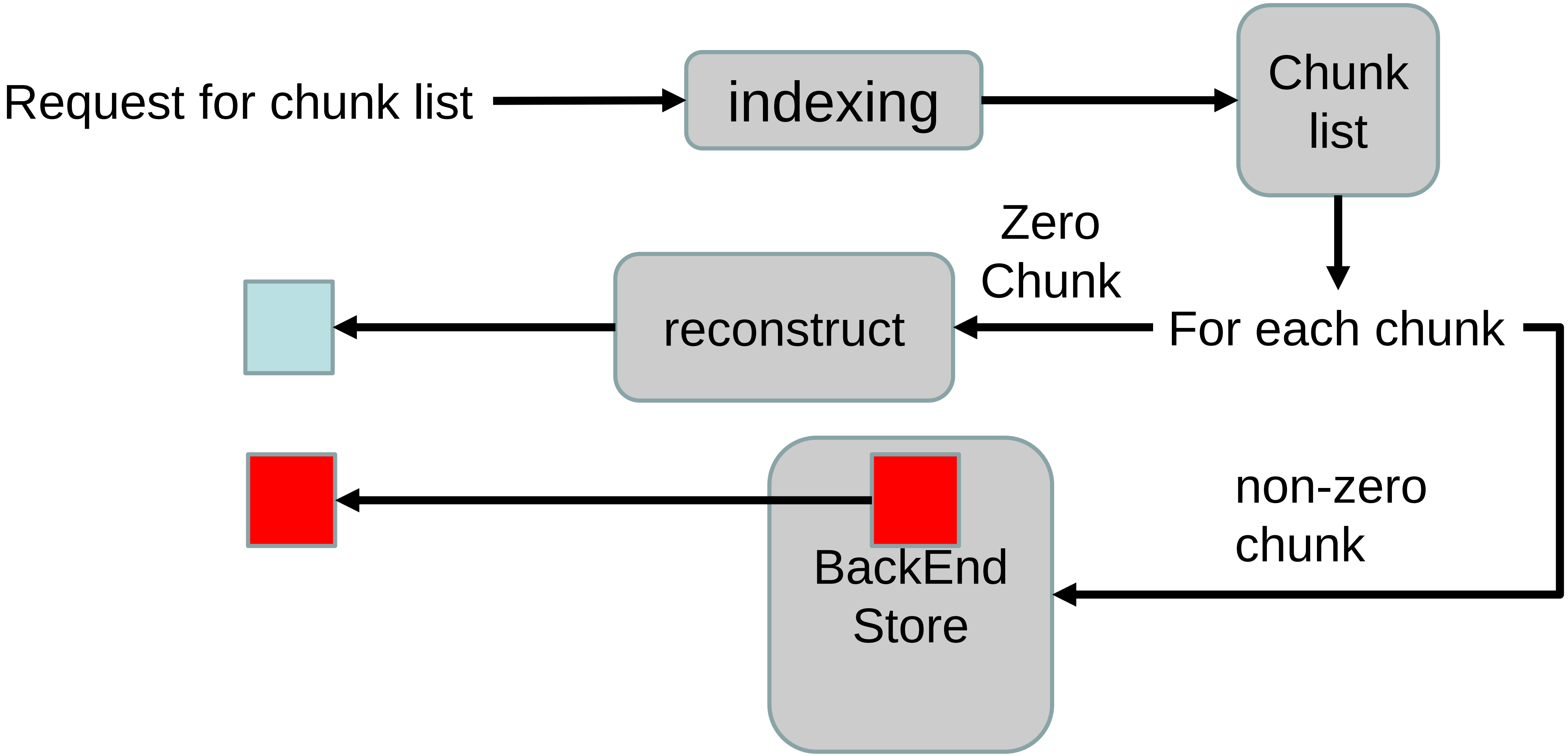
- Parameters needed in chunking provided
  - min_chunk: minimum size of a chunk -> window size
  - avg_chunk: tells modulus
  - max_chunk: maximum size of a chunk
  - d: base
  - Chunks are identified based on MD5
- How to decide whether a chunk needs to be uploaded to backend store?
  - When the chunk is unique
    - which means there is no duplicated chunk stored in backend store.
  - When the chunk is full of zeros
    - there is no need to upload it.

Hints (Cont.)

- Statistic for upload
  - Example
    - Parameter: min_chunk = 10, max_chunk=15, upload file1 which contains 20 bytes of zeros and 10 bytes random data.
  - Output:
    - Total number of files that have been stored: 1
    - Total number of pre-deduplicated chunks in storage: 2
      - A logical chunk of zero run.
    - Total number of unique chunks in storage: 1
      - No physical chunk in backend store
    - Total number of bytes of pre-deduplicated chunks in storage: 30
    - Total number of bytes of unique chunks in storage: 10
    - Deduplication ratio: 3.00

Hints (Cont.)

- Download process
  - The whole workflow

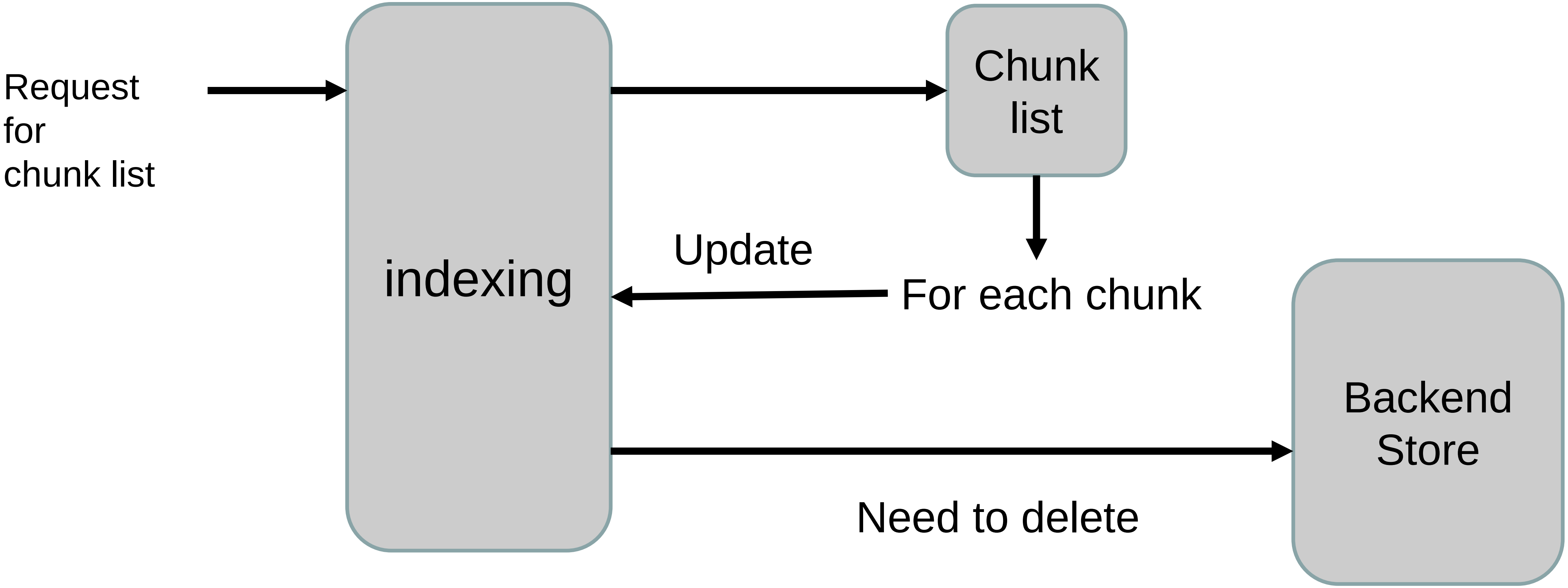


Hints (Cont.)

- How to get chunk list?
  - 1. We maintain a metadata file called “mydedup.index”
  - 2. At the beginning of each operation
    - we reconstruct the indexing data structure from this metadata file.
  - 3. At the end of each operation,
    - we update the indexing data structure and store it into our metadata file
    - we can rebuild the indexing data structure for the next operation.

Hints (Cont.)

- Delete process
  - The whole workflow



Hints (Cont.)

- Decide whether to delete the chunk physically.
  - There is no file in our storage that depends on this chunk.
  - Think about what do we need to maintain in our indexing data structure.

- Attention
  - RFP algorithm might overflow before modular operation
    - (a + b) mod q ≡ (a mod q + b mod q) mod q
    - (a - b) mod q ≡ (a mod q – b mod q) mod q
    - (a * b) mod q ≡ (a modq * b mod q) mod q

- Performance
  - Fast modular exponentiation algorithm
  - Leverage multi-threading to pipelining the workflow

Hints (Cont.)

- How to test your program?
  - Create test cases by yourself. This is also an important skill!
  - Can you deal with upload, download, delete operation in sequence without errors?
  - Can you download the file that contains the same contents with the original file that you uploaded?
  - Can you correctly deal with long runs of zeros?
  - Can you correctly deal with duplicated chunks? (e.g. different file contains same chunks)
  - Can you deal with large files? (e.g. linux image file)
- How about the performance of *your implementation?*

Thank you
Q & A