Paper Review

Chang, BingChun, "A Running Time Improvement for Two Thresholds Two Divisors Algorithm" (2009). Master's Projects. Paper 42.

Robinson Raju
San Jose State University

Agenda

- Overview
- Background
  - Chunking
- BSW Algorithm
- TTTD Algorithm
- Experimental Comparisons
- New Improvement to TTTD Algorithm

Summary

- Chunking algorithms play a critical role in data de-duplication systems
- BSW - 1st prototype of content-based chunking algorithm
- TTTD - Proposed to improve BSW algorithm to control variations of chunk size
- Two values of the paper
  - Experimental evaluation of the 2 algorithms
  - Running time improvement of TTTD algorithm

BACKGROUND

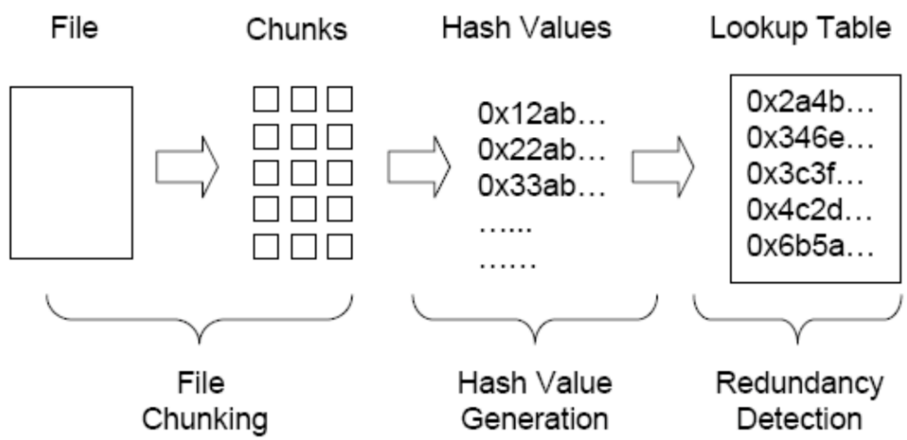
Data De-duplication

- **Data de-duplication**
  - Enterprises store a lot of data but most of them are redundant
  - Data de-duplication stores only one copy for all duplicate data, and creates logical reference to the copy so that users can access the data when needed
  - Reduces cost of storage, power consumption, maintenance cost
  - Makes data replication and recovery efficient

- **Two Schemes**
  - Hash-based
  - Content-aware

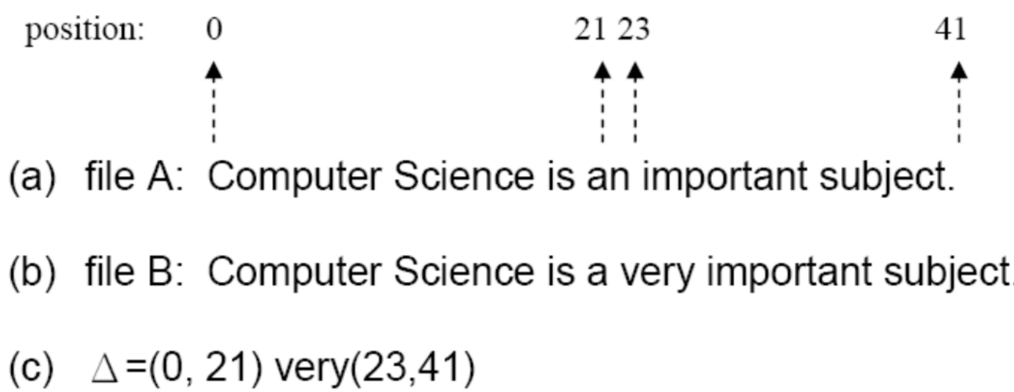
Hash-based Approach

- **File chunking** – When a new file arrives, the system breaks entire file into small blocks.
- **Hash value generation** – Uses SHA-1/MD-5 algorithms to generate unique signatures for each chunk.
- **Redundancy detection** – System looks up the hash and if not found, adds to lookup table.



Content-aware Approach

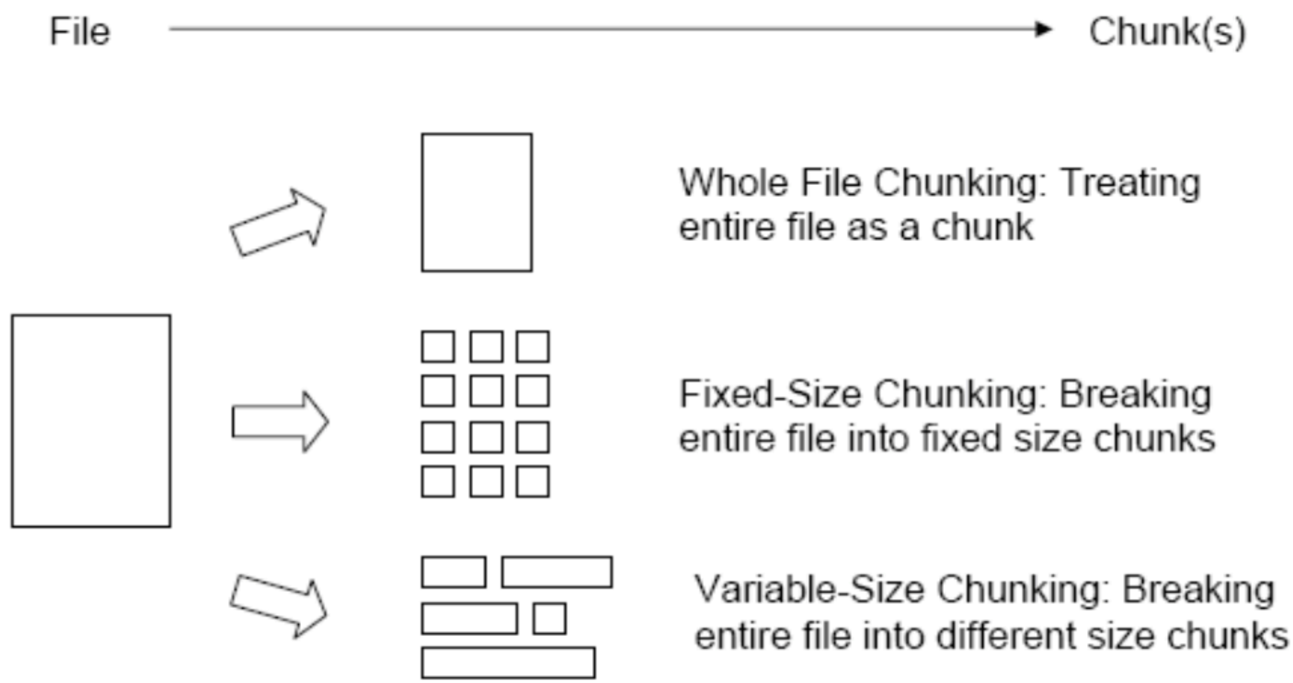
- **Identification** – Identifies the specific format of the incoming file and chooses a reference file.
- **Comparison** - byte-to-byte or block-to-block comparison with reference file.
- **Store delta** – Computes differences with reference file and stores it.



Chunking

- Process of partitioning a file into smaller chunks
- Time consuming since once has to traverse entire file
- Processing time depends on how the algorithm breaks the file
  - Smaller the size of chunk, better the de-duplication
  - Smaller the size, more number of chunks, more time to process
  - More number of chunks also would mean a larger lookup table (for hash-based) – If the lookup table is so large that it cannot be in memory, I/O calls to disk would mean increase in processing time.

Chunking Categories



- Whole file chunking : simplest and fastest but worst de-dedupe ratio

Boundary Shifting Problem

- Inserting/deleting just one byte could result in different hash values for chunks even though most of the data between new and old file is the same
- How to avoid ?
  – Variable-size chunking (content-based chunking)
  – Chunk boundaries may be determined by punctuation, end of line ..etc

Chunking Algorithms

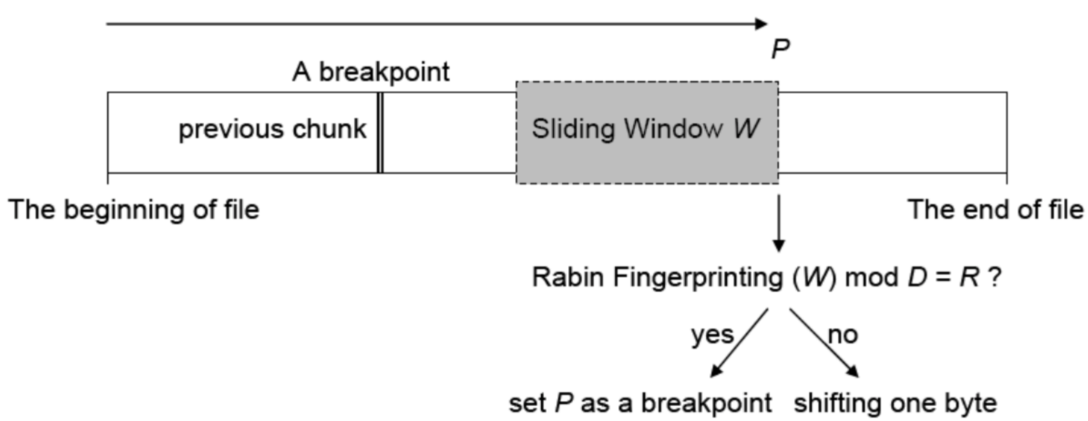
- Overlap
  - K-gram algorithm
  - 0 mod p algorithm
  - Winnowing algorithm
- Non-Overlap
  - Hash-breaking
- Basic Sliding Window (BSW) algorithm is the first prototype of the hash-breaking chunking algorithm
- The Two Thresholds, Two Divisors (TTTD) algorithm is the adaptation of the BSW to improve the problems in the BSW algorithm.

DETAILS OF

BSW AND TTTD ALGORITHMS

The Basic Sliding Window (BSW) Algorithm

- A fixed-size window W is shifting one byte at one time from the beginning of the file to end of the file.
- At every position p, uses Rabin Fingerprinting algorithm to compute a hash value h for the content of current window.
- If h mod D = R, the position P is a breakpoint for chunk boundary. Then the sliding window W starts at the position P. And repeats the computation and comparison.
- If h mod D ≠ R, the sliding window W keeps shifting one byte. And repeats the computation and comparison.



BSW Algorithm – Expected chunk size

- Parameter *D* plays an important role.
- In each shifting (one byte at a time), the probability of *h mod D = R is 1/D*
  - Expect to find a breakpoint at every D bytes
- E.g, if D = 1000 and 0 ≦ R ≦ 999, in the best case, expect to find a match for every 1000 bytes

BSW Algorithm - Problems

- Breakpoint in each shifting
  - This causes number of chunks to be equal (or more) to the number of bytes in the input.
- Does not find a breakpoint at all
  - This causes number of chunks to equal to 1 which defeats the purpose of chunking

The Two Thresholds Two Divisors (TTTD) Algorithm

- Proposed by HP Labs, Palo Alto, CA
- Uses the same idea as BSW
- In addition, uses 4 parameters –
  - The maximum threshold, The minimum threshold
  - The main divisor, The second divisor

Parameter	Purpose	Optimal Value
Maximum Threshold	To reduce very large chunks	2800 (bytes)
Minimum Threshold	To reduce very small chunks	460 (bytes)
Main Divisor	To determine breakpoint same as the BSW	540
Second Divisor	To determined a backup breakpoint	270

TTTD Algorithm

1. The algorithm shifts one byte at one time and computes the hash value.
2. If the size from last breakpoint to current position is larger than minimum threshold, it starts to determine the breakpoint by the main divisor.
3. Before the algorithm reaches the maximum threshold, if it can find a breakpoint by main divisor, then uses it as the chunk boundary. The sliding window starts at this position and repeats the computation and comparison until the end of file.
4. When the algorithm reaches the maximum threshold, it uses the backup breakpoint if it found any one, otherwise use the maximum threshold as a breakpoint.

TTTD Algorithm - Problems

- The eliminations of very large sized and small sized chunks cost the algorithm to increase the total number of chunks.
  - Increasing the total number of chunks also increases the amount of metadata, the size of lookup table, and the lookup table searching time.
- The second divisor is used only if max threshold is reached.
  - So it only prevents the algorithm from using the maximum threshold as the breakpoint and plays a trivial role in the TTTD algorithm.

EXPERIMENTAL COMPARISONS

Experiment Settings

- Configuration

Parameter \ Algorithm	BSW	TTTD
Window Size (bytes)	48	48
Main Divisor	1000	540
Second Divisor	N/A	270
Maximum Threshold	N/A	2800
Minimum Threshold	N/A	460

- Data set

Data Set	#1	#2	#3	#4
Data Name	Emacs	Emacs	GNU Manuals	GNU Manuals
Data Type	tar	source code	html	txt
No. of Files	5	16994	40	40
Total Size (MB)	171.3	607.2	36	22.3

Experiment Results – Running time, Chunk size

	Total Running Time (sec)		Total number of Chunks		Average Chunk-Size (bytes)	
Data Set	BSW	TTTD	BSW	TTTD	BSW	TTTD
#1	2910	2885	172874	182582	1040	985
#2	10568	11011	391036	481963	1629	1321
#3	617	639	24692	32364	1532	1169
#4	381	398	17803	19590	1316	1196
Average	3619	3733	151601	179125	1379	1168

	Max Chunk-Size (bytes)		Min Chunk-Size (bytes)	
Data Set	BSW	TTTD	BSW	TTTD
#1	16442	2800	48	412
#2	154075	2800	8	8
#3	97168	2800	48	68
#4	68224	2800	48	62
Average	83977	2800	38	138

Experiment Results – Chunk size distribution

- BSW

	Data Set #				
Interval (bytes)	**#1**	**#2**	**#3**	**#4**	**Average**
< 48 (%)	0	0.01	0	0	0.002
48 ~ 459 (%)	34.14	42.64	41.04	43.28	40.28
460 ~ 799 (%)	19.35	13.55	14.1	14.62	15.41
800 ~ 1199 (%)	15.3	9.23	10.3	11.2	11.51
1200 ~ 1599 (%)	10.29	6.35	7.24	6.94	7.71
1600 ~ 1999 (%)	6.93	4.93	5.27	5.63	5.69
2000 ~ 2399 (%)	4.51	3.79	4.09	3.65	4.01
2400 ~ 2799 (%)	3.07	3.07	3.06	3	3.05
>= 2800 (%)	6.41	16.41	14.91	11.67	12.35

- TTTD

	Data Set #				
Interval (bytes)	**#1**	**#2**	**#3**	**#4**	**Average**
< 460 (%)	0	1.03	0.05	0.05	0.28
460 ~ 799 (%)	47.0	32.17	42.88	39.75	40.45
800 ~ 1199 (%)	27.7	20.78	21.47	22.54	23.12
1200 ~ 1599 (%)	13.4	13.63	11.15	12.28	12.62
1600 ~ 1999 (%)	6.43	10.09	8.48	8.96	8.49
2000 ~ 2399 (%)	3.34	8.91	6.19	6.88	6.33
2400 ~ 2799 (%)	2.11	11.3	6.67	7.17	6.81
= 2800 (%)	0.03	2.09	3.11	2.36	1.9

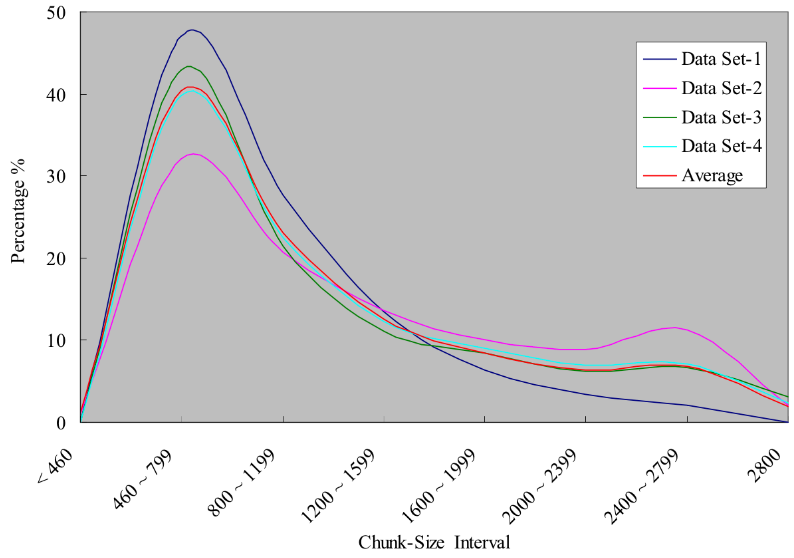
New Improvement of the TTTD Algorithm

TTTD-S ALGORITHM

TTTD-S Concept

- Key issue : The second divisor plays a trivial role in the chunking. As seen in the graph below, the second peak happens towards the end.
- **Idea** : What if we're able to bring the second peak earlier ?

Chunk Determined by	Data Set # #1	#2	#3	#4	Average
Main Divisor	180156 (98.67%)	391106 (84.04%)	28784 (89.04%)	17470 (89.36%)	90.3%
Second Divisor	2374 (1.3%)	64481 (13.86%)	2545 (7.87%)	1619 (8.28%)	7.8 %
Max Threshold	47 (0.03%)	9795 (2.1%)	997 (3.08%)	461 (2.36%)	1.9 %



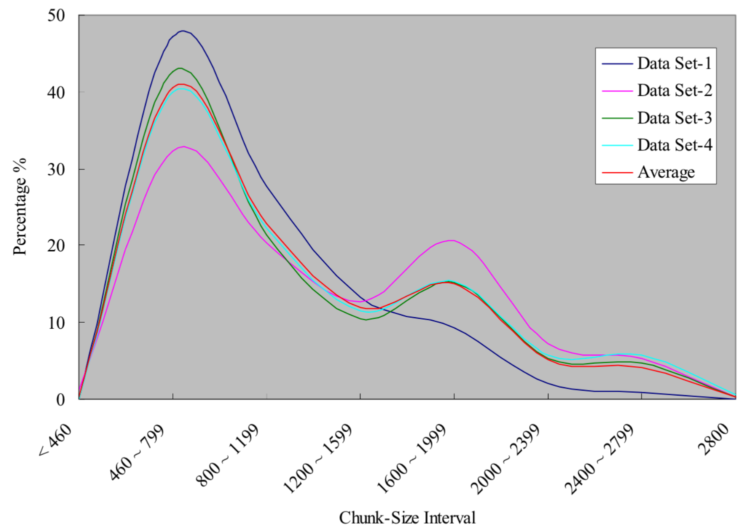
TTTD-S Algorithm

- Observation :
  - 70 % of the total chunks which are determined before 1600 bytes are determined by main divisor.
  - The second peak begins at 2400 bytes.
- The concept of new improvement :
  - Use a new parameter, where
  - 1500 < ***new parameter*** < 2400
- Algorithm :
  - When the control reaches the ‘new parameter’, the algorithm uses 1/2 of the original values as the new values for the main D and the second D.
  - After finding a breakpoint, it switches the new values back to the original values.
  - By increasing the probability of the mainD, the expectation is that the second peak will happen earlier.

Experimental Evaluation

- Reduced total running time from 3777 seconds to 3510 seconds in average case.
- Made the average chunk-size closer to the expected chunk-size from 1168 bytes to 1121 bytes.
- Reduced large-size chunks between 2400 bytes to 2800 bytes from 8.7 % to 4.4 %. The decreasing ratio is about 50 %.

	Total Running Time (sec)		Total number of Chunks		Average Chunk-Size (bytes)	
Data Set	**TTTD**	**TTTD - S**	**TTTD**	**TTTD - S**	**TTTD**	**TTTD - S**
#1	2885	2818	182582	186757	985	963
#2	11011	10242	481963	513330	1321	1241
#3	639	603	32364	33360	1169	1134
#4	398	379	19590	20385	1196	1149
Average	3733	3510	179125	188458	1168	1121



Possible improvement

- Instead of picking new parameter as “1500 < ***new parameter*** < 2400”, we could pick it dynamically based on past chunk sizes (In short, we could use a Machine Learning Technique)

- We might need different approaches for different file types (pdfs vs ppt, doc vs image)

APPENDIX

Rabin fingerprinting

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	
0	3																% 997 = 3 Q
1	3	1															% 997 = (3*10 + 1) % 997 = 31
2	3	1	4														% 997 = (31*10 + 4) % 997 = 314
3	3	1	4	1													% 997 = (314*10 + 1) % 997 = 150
4	3	1	4	1	5												% 997 = (150*10 + 5) % 997 = 508 RM R
5		1	4	1	5	9											% 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201
6			4	1	5	9	2										% 997 = ((201 + 1*(997 - 30))*10 + 2) % 997 = 715
7				1	5	9	2	6									% 997 = ((715 + 4*(997 - 30))*10 + 6) % 997 = 971
8					5	9	2	6	5								% 997 = ((971 + 1*(997 - 30))*10 + 5) % 997 = 442
9						9	2	6	5	3							% 997 = ((442 + 5*(997 - 30))*10 + 3) % 997 = 929 match
10 ← *return* i-M+1 = 6							2	6	5	3	5						% 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613

Rabin-Karp substring search example

Figure 1: Screenshot, SEDGEWICK, R., & Kevin, W. 2011, Algorithms (4th ed.).,