Improving Content Addressable Storage for Databases

Vandhana Selvaprakash Brandon M. Smith

Department of Computer Sciences
University of Wisconsin, Madison, WI 53706
E-mail: {vandhana,bmsmith}@cs.wisc.edu

Abstract

Recent trends have seen database clients use content addressable storage systems (CASs) for near-line storage. While CASs have many attractive properties such as storage space savings, data integrity, and low network bandwidth requirements, CASs are not well suited for databases, mainly because of the rigid structure of databases, and the way databases intersperse metadata with data.

In this paper, we evaluate where current CAS techniques fail for databases and identify properties of database systems that can be leveraged for potential improvements to CAS techniques specific to databases. We propose fives ways in which CAS systems can be made "database aware," and evaluate the potential strengths and weakness of each approach. We find that our techniques improve memory savings, but at the expense of coupling the solution too closely to particular database vendors.

1 Introduction

CAS divides files into *chunks* (other names include *segments* [3], *blocks* [1], *atomics* [7], or *fragments* [6]). Each chunk is indexed by a descriptor, or *fingerprint*, that uniquely identifies its contents. Files are hence content addressable, rather than location addressable. Identical chunks have the same fingerprint, which insures that redundant data is never stored twice. This technique substantially reduces the amount of storage space required, provided that duplicate information exists in the file system.

Virtually all the systems we study use the cryptographic SHA-1 hash function [5] to calculate the fingerprint for each chunk. One of the most desirable properties of this hash function is that it is very efficient to compute. Yet another desirable property is its strong collision resistance. With extremely high probability[1], each fingerprint will *uniquely* describe the contents of its corresponding chunk.

A three-way distinction can be made between chunking techniques. Data can be chunked based on: (1) whole file content, (2) fixed-size chunks, or (3) variable-size, content-based chunks. In (1) the chunk boundaries are simply determined by the beginning and ending of the files being stored. Techniques based on (2) determine chunk boundaries by fixed offsets from the beginning of each file. Techniques based on (3) are more complicated and typically involve discovering anchor points throughout each file based on its contents. This is usually accomplished using some variant of the Rabin fingerprinting technique described by Udi Manber in [4]. Although (1) and (2) are simple, in most situations they effectively make files stored on the system immutable. Variable-size chunking techniques do not suffer from this problem.

2 Motivation

Content addressable storage systems (CASs) are primarily used to store data that changes rarely, and is accessed infrequently (write once, read many). Databases increasingly use CASs for storing backup information, and snapshots. Unlike traditional CAS data (business data, MRI Images), which are loosely coupled with metadata(if any), databases are not well-suited for CAS. The problem with database storage structures is that metadata is A) substantial, and B) tightly interspersed with data. It is therefore difficult to exploit data redundancy for space savings.

Another limitation of current CAS techniques is that they only exploit identical files, blocks, or chunks. Typically, many pages within database storage structures are similar, but very few are identical.

Databases make up a significant portion of storage system clients. Simply said, they are too important to ignore. Given the increasing use of CASs by database systems, we aim to improve CAS techniques specifically for the database domain.

[1] Assuming random hash values and a uniform distribution, a fingerprint collision for an exabyte (10^18 bytes) of data divided into 8 kilobyte chunks will occur with probability less than 10^-20, which is many orders of magnitude less than hardware error rates [1].

Table 1: target chunk size vs. actual average chunk size

Data type	Target Chunk Size	Actual Average Size
	(bytes)	(bytes)
1 GB Music Data (mp3, wma, wav, m4a)	128	175
	256	303
	512	559
	1024	1070
	2048	2094
	4096	4137
	8192	8220
	16384	16413
	32768	32848
330 MB Sparse Oracle Tablespace	8192	21335
5 MB Sparse PostgreSQL Tablespace	8192	15443
560 MB Mix of Files (pdf, txt, doc, mpg, jpg, etc...)	4096	1100
	8192	1849
	16384	2062

Our approach to this problem is as follows. We implement several key features of state-of-the-art CASs and evaluate these features on different kinds of data, including music files, text data, and database data (Oracle [10] and PostgreSQL [9].) We determine quantitatively that recent variable-sized chunking techniques perform poorly on database data.

We then study the low-level storage structure of databases (mainly Oracle and PostgreSQL) and identify properties that can be leveraged for potential improvements to CAS techniques. We propose five ways in which CAS systems can be made "database aware," and evaluate the potential strengths and weakness of each approach. We find that our techniques improve memory savings, but at the expense of coupling the solution too closely to particular database vendors.

The rest of the paper is organized as follows. Section 3 current state-of-the-art CAS techniques; Section 4 discusses database semantics; Section 5 discusses five approaches to improve CAS techniques for databases; Section 6 discusses related work; finally, we end with some conclusions.

3 Evaluation of State-of-the-Art CAS techniques

3.1 Implementation

To evaluate state-of-the-art CASs [3], we implement the following:

- Variable-size chunking based on Rabin fingerprinting
- An efficient SHA-1 hash (chunk fingerprint) index storage structure based on a binary search tree
- A summary vector to further optimize the speed of chunk lookup

We implement a variable-size chunking tech-nique based on a widely cited paper by Udi Manber [4]. The technique is based on using Rabin fingerprints to discover anchor points within each file. A sliding 48-byte[2] window scans through each file. At each new byte, the fingerprint is updated and the last N bits are compared to zero. The average chunk size is then 2^N bytes based on the probability of the last N bits of the fingerprint being a certain value (zero is usually chosen). We then implement a binary search tree for efficient hash lookup.

Finally, we implement a summary vector to further speed up fingerprint lookup. It is not uncommon for archival or backup systems to store several terabytes of data. This requires gigabytes of fingerprints, the majority of which must be stored on disk. To minimize the number of actual fingerprint searches, we implement a summary vector based on a Bloom filter. When each new fingerprint is to be stored, it is first hashed using five independent functions. Each hash value maps to a single bit of the summary vector. If any one of the bits is zero, the system knows with 100% accuracy that the current fingerprint has not yet been stored and therefore it need not be search for. On the other hand, if all of the bits are one then the system must search for the fingerprint. Based on results from [3], a summary vector of sufficient size can increase overall throughput by ~20%. In practice, since our program does not store fingerprints to disk, we can only record the number of times the search algorithm would hypothetically go to disk.

[2] We actually allow the user to select an arbitrary window size, but 48-bytes is most common.

Figure 1: Duplicate music file discovery using variable-size chunking and fingerprint storage and lookup. Left column is a similarity measure.

```
0.99200000000  C:\Workspace\Alicia\Oasis_-_What's_The_Story_...
0.99078341013  C:\Workspace\Alicia\Incubus_-_Morning_View_-_...
0.99033816425  C:\Workspace\Alicia\Hard_Candy_-_Counting_Cro...
0.98913043478  C:\Workspace\Songs\Audioslave_-_Show_Me_How_T...
0.98802395209  C:\Workspace\Alicia\The_donavon_frankenreiter...
0.98726114649  C:\Workspace\Songs\Cornell-Rage_Demos_-_Track...
0.98373983739  C:\Workspace\Alicia\Eric_clapton_-_The_cream_...
...                                                       ...
0.09090909090  C:\Workspace\Songs\Blink_182_-_Reebok_Commerc...
0.08602150537  C:\Workspace\Songs\311_Wake_Your_Mind_Up-from...
0.08333333333  C:\Workspace\Songs\AUDIOSLAVE-getaway_car-.mp...
0.07826086957  C:\Workspace\Songs\Glassjaw_-_Lovebites_and_R...
0.07692307692  C:\Workspace\Songs\aladdin_-_i_can_show_you_t...
0.07594936709  C:\Workspace\Songs\Glassjaw_-_Trailer_Park_Je...
```

3.2 Results and Observations

We address several issues not discussed in the literature.

“How well does the average chunk size match the target chunk size in practice?” Our results are shown in Table 1. Two parameters that affect the average chunk size in practice are the min. and max. size limits. We reduce their impact on our implementation by setting them to very low and high values respectively.

First, we evaluate the performance for music data, where we find that the target size is almost exactly met. Next, we evaluate the performance for a mixture of data types (PDFs, text files, images, websites, etc.). We now find that the target size is not met, primarily due to the prevalence of small files, which restrict chunk size.

Finally, we evaluate the performance of database data. Interestingly, we find that the actual chunk sizes for database tablespaces are far from the desired chunk size. This is because tablespaces are mostly sparse, with long stretches of zeros. Very few anchor points can be found, and hence the chunk size tends to the maximum. The larger the chunk size, the lesser the commonality factoring – the result is that a mostly empty database that could be compressed down to fractions of its original size requires close to the original amount of space.

Using an 8 KB target chunk size, the mostly empty Oracle tablespace is only compressed by a factor of 1.000432:1, and the set of PostgreSQL tablespaces are only compressed by a factor of 1.070896:1. This clearly indicates that variable-size chunking performs very poorly on database data. Even with well-populated tablespaces, we do not achieve better performance due to the tightly interspersed metadata and data.

We then put our system to practical use. We find that our implementation is well suited for efficient discovery of duplicate (or similar) files. We test it on a directory containing 20GB of music data, and discover that ~60 duplicates exist in the music collection. An abbreviated list of results is shown in Figure 1. The process completed in roughly two hours, indicating a throughput of about 3 MB per second for our unoptimized program.

Finally, during the implementation stage, we discover an error in a key equation of [4], which initially made debugging our program extremely difficult. This discovery was surprising given that this paper is so widely cited. The problem is given further attention in Appendix A.

4 Database Semantics

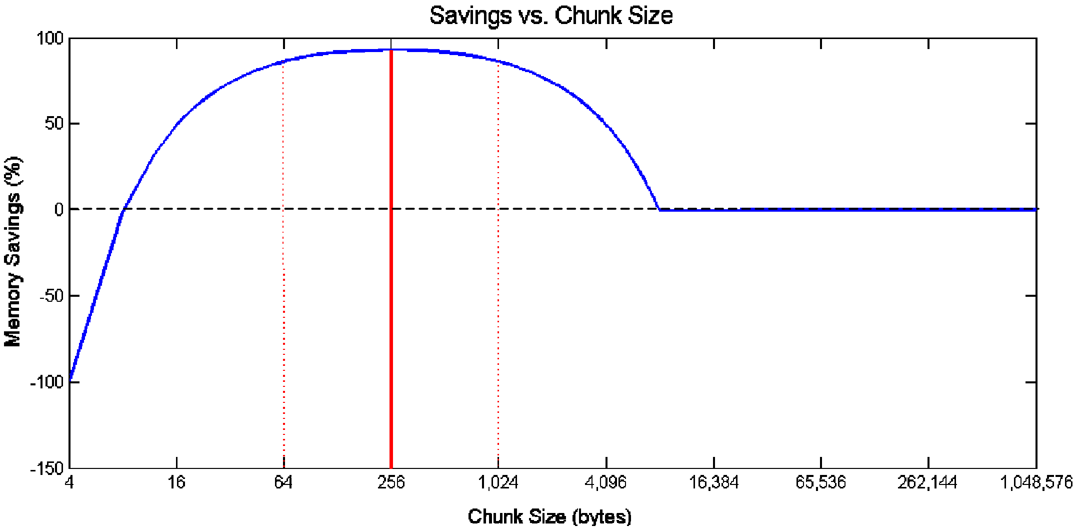
CAS systems can leverage database semantics to achieve greater memory savings for databases. We discuss specific information that can be used for this purpose below.

4.1 Database Structure

Databases store all information in structures of defined format. Knowledge of these formats allows the CAS system to separate the actual data from the metadata, and to better leverage commonality factoring algorithms.

All databases store data on disk in a structure called the slotted page. The disk block, which maps to a slotted page, is the basic unit of I/O.

Figure 2: Space savings for a range of chunk sizes using a naïve fixed size chunking technique on an empty 330 MB Oracle tablespace



4.2 Metadata

Metadata stored at the block level allow the CAS system to 'understand' the data stored in the block, to achieve finer grained commonality factoring. For instance, interpreting the row metadata allows the system to check for identical /similar rows. Interpreting per-row metadata allows for item level commonality factoring.

4.3 Database Parameters

DB configuration parameters allow CAS to set “thresholds” for memory saving, to balance saving with overhead incurred. For instance, the PCTFREE parameter of the Oracle database sets a lower bound on the size of the zeroed section of the block, which helps in lowering computational overhead.

5 Database Aware CAS

We seek lazy techniques, which attempt to optimize memory savings by commonality factoring while minimizing work done by the CAS system. All databases (their access methods) expect the storage system to use the database block as the unit of I/O. In the first two approaches, we retain the database block size as the fixed chunk size, and propose slight improvements. Faced with disappointing results, we then venture to break/deflate the database block into a representation more conducive to finer-grained com-monality factoring.

5.1 NAC for DB - Naive Chunking for DB

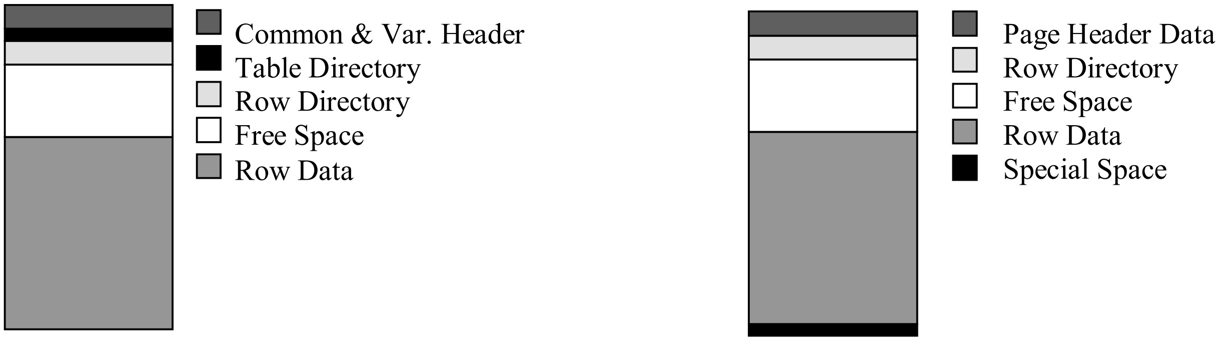
The laziest of all our approaches, this “test the waters” approach seeks to find if existing database block sizes are optimal for CAS systems, in which case our project would end without any further ado.

We implement a fixed-size chunking technique and evaluate the approach with different chunk sizes. We use database data: sparse tablespaces from Oracle and PostgreSQL. This represents a theoretical best case for storage savings, as redundancy is higher than in populated databases. Figure 2 shows observed memory savings relative to a broad range of chunk sizes. The graph shows a savings “sweet spot” at 256 bytes.

In theory, smaller chunk sizes better exploit data redundancy for space savings; however, each chunk requires a fixed amount of metadata. At a certain point, the metadata requirement for a larger number of chunks overwhelms any space savings due to chunking. As the chunk size increases past 8 Kbytes, we see no space savings either - all chunks include unique metadata and therefore no duplicate chunks exist.

The 8 Kbyte limit is no coincidence. On looking closer at the tablespace flat files, we find that sections of non-zero metadata occur exactly every 8 Kbytes. Oracle and PostgreSQL use 8 KB database blocks, as we discuss in a later section. Figure 3 shows the structure of pages used by Oracle and PostgreSQL.

Figure 3: page structure for Oracle (left) and PostgreSQL (right)



5.2 SCROUNGE - Savings via Chunk Redundancy with Overlays (UN-GEneric)

Our second approach takes the concepts of full and incremental backups from the database world and implements them at the database block level. We seek to improve commonality factoring by leveraging similarity between chunks of fixed sizes (which map to database blocks).

We compare a new *candidate* chunk that is to be stored in the system against a predefined set of *template* chunks. The candidate may then be stored as a *delta* chunk, which when overlayed with the template, will generate the candidate. The delta takes considerably less space than the original chunk, and data integrity is ensured by storing the fingerprint of the candidate along with the delta. The set of templates may be static or dynamic. They may include a few special-case chunks, or all existing chunks. We choose to optimize mostly zero database storage structures by defining a all-zero template. More templates can be defined, as special cases are identified.

We use a simple technique to compute deltas called *chunk windowing*. Each chunk is divided into smaller windows. A candidate chunk is deemed similar to a template chunk if it shares a sufficient number of identical windows with the template chunk. A threshold determines if the candidate is sufficiently similar to be stored as a delta of the template chunk. The threshold is determined, based on two factors: (1) Memory utilization - the delta structure must occupy at most a specified fraction of the chunk (2) Computational overhead of overlaying. We primarily focus on (1) and assume that given our problem, it takes precedence.

We find that block level commonality factoring performs only slightly better than NAC for DB, though it has a much higher computational overhead. More sophisticated delta techniques may be used to improve memory savings, but at the cost of even higher computational overhead. Figure 5 illustrates the dismal performance of SCROUNGE. Although SCROUNGE does marginally better at higher chunk sizes than NAC for DB, overlaying is sub-optimal when the chunk sizes approach the database block size.

We recognize that there are no significant savings to be had with commonality factoring at block-level granularity. We now look to improve memory saving by "breaking the disk block". By leveraging our knowledge of database block structure, we construct lossless representations of the block that take significantly less memory, while incurring slight computational overhead while "reconstructing the block".

5.3 FRESCO - FREe Space COmmonality factoring

FRESCO leverages a coarse “awareness” of the database block structure. The Storage system is “aware” that the block can be parsed into three sections: the header, which consists of the page header data, and the pointer structures, the free space section, and the actual data, along with any special trailer data/checksums.

The storage system reads per-block header information to identify the zeroed section of the block, and stores the non-zero sections alone. Computational overhead is incurred while restoring the block using zero padding at the required offset.

This technique is coarse-grained enough to de-couple the solution from a specific database vendor. Most databases follow similar slotted page structures, even if the actual data structures vary. We tested our implementation on Oracle. The PCTFREE configuration parameter is used to determine the lower bound (threshold) for saving. The technique shows significant memory savings - 73.6% savings for sparse tablespaces (mostly zeros),and 27.1% for denser tablespaces (mostly data).

Table 2: Memory savings for SCROOGE, tablespace size: 1 MB

Block Type	Memory Footprint (%)	Memory Savings
Sparse	0.73	99.27
Full	77.3	22.7
Average	8.97	91.03

5.4 SICCO - Savings via Intra Chunk row COmmonality factoring

SICCO takes a finer-grained approach to commonality factoring: row redundancy. The CAS system "knows" the database block structure, reads the metadata of each block, parses the data in the block into tuples, and leverages identical tuples.

Row commonality factoring is achieved by deflating the chunk into a *frame*. The Frame contains all details required to inflate the chunk when required by the database. Within each frame, a list of *tuple* structures store the actual data, along with tuple offsets. Both the row directory and the row data can be easily reconstructed from this list.

We chose PostgreSQL as it is open-source, and its data block structure much easier to understand than the enigmatic Oracle. Figure 4 shows the storage structures used in the implementation.

After much implementation effort, the results were depressing. The memory savings were marginally less than FRESCO. On examining the data, we found that the rowid attached to every tuple ensured that there was no commonality to be factored between duplicate rows. SICCO had the same memory savings as FRESCO, with the added overhead of the PGTuple data structures within the frame. The storage structures used in SICCO are shown in Figure 4

5.5 SCROOGE - Similarity based Chunk ROw redundancy, with Overlaying (GEneric)

In SCROOGE, we add the overlaying technique from SCROUNGE to SICCO, leveraging similar rows (tuples) instead of identical ones. We alter the PGTuple structure to include a flag that indicates if the tuple is a delta. It also adds a list of chunk windows, which store the delta. Overlaying is identical to SCROUNGE, only at the finer grained tuple level. Instead of static templates, like in SCROUNGE, overlaying in SCROOGE uses all valid tuples within a block, In a more generic approach to overlaying.

Evaluation of the SCROOGE technique on a PostgreSQL tablespace of 1 MB yielded promising results. The 8 KB PostgreSQL database block, on average, needed 736.5 bytes in the system. The memory savings per block varied greatly from almost 99% (for an empty block) to 23% (for an almost full block). Results are shown in Table 2.

6 Related Work

The system described in [15] chunks data based on whole-file content. The system described in [13] uses whole-files chunking only on small files that don't benefit from more complicated chunking techniques. Other than these two systems, chunking based on whole-file content does not seem to be used.

Fixed-size chunking has more support, as Venti [1] and the *rsync* algorithm [12] would attest. However, since the Venti paper was published in 2002, fixed-size chunking seems to have fallen out of favor due its more restricting nature. However, as we will discuss, CASs for databases might benefit from techniques inspired by fixed-size chunking due to the fixed structure imposed by databases on the data.

There have been many papers written in the last five years or so that discuss CASs based on variable-size chunking. Interestingly, LBFS [2], which uses variable-size chunking, is an exception. It was introduced in 2001 *before* Venti and represents the first true CAS in our opinion. The idea of using variable-size chunking really began to gain traction in 2003 with the introduction of systems like Avamar Technologies' CFS [7], *fingerdiff* [12], and CASPER [13]. These were followed in 2004 with systems like REBL [11] and papers like [14]. We discovered an especially informative recent paper from FAST 2008 describing DDFS [3], which also uses variable-size chunking along with several interesting optimizations to drastically improve throughput.

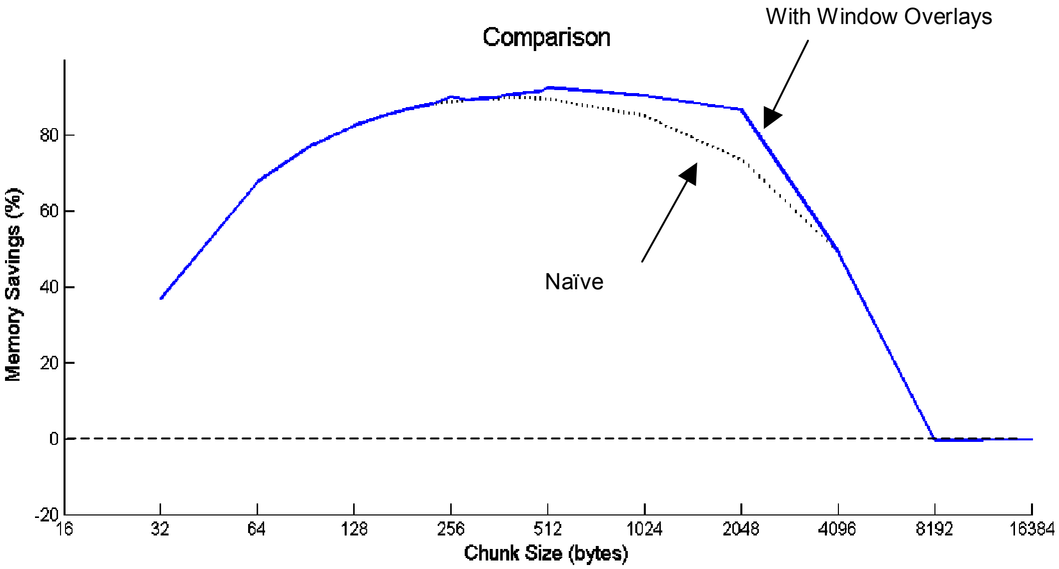
Figure 4: Storage structures for SICCO and SCROOGE

PGFrame {pageHeaderData, specialSpace, zeroBegin, zeroEnd, fingerprint, pTuple}

PGTuple {tupleData, offset} (SICCO)

PGTuple {flag, offset, tupleData (OR) delta} (Modified for SCROOGE)

Figure 5: Space savings for a range of chunk sizes using a windowing technique on an empty 330 MB Oracle tablespace



7 Conclusions & Future Work

We evaluated current CAS techniques and have found that current CAS techniques don't work well for databases. In this process, we observed that code involving bit-level operations was difficult to debug. As knowledge of database structure and semantics can help improve memory savings in CAS for databases. We then proposed five ways by which CAS systems can be made "database aware", and discussed the strengths and weakness of each approach. We emphasized that the database block needs to be “broken” to optimize memory savings, but that these savings are to be had at considerable computational overhead, and at the expense of coupling the solution too closely to a particular database vendor.

Future work involves investigating Inter Chunk Commonality Factoring, which would mean fragmenting, and redistributing block data. Finer grained Item-level Commonality factoring can also be explored.

Appendix A: Udi Manber’s Illusive Mistake

Udi provides an example in [4] in which a 50-byte substring is used to generate a fingerprint, F_1. He gives the equation

$$F_1 = (t_1 \cdot p^{49} + t_2 \cdot p^{48} + \cdots t_{50}) \bmod M ,$$

where t’s are byte values, and p and M are constants. Using Horner’s rule, this equation becomes

$$F_1 = (p \cdot ((\cdots (p \cdot (p \cdot t_1 + t_2) + t_3) \cdots)) + t_{50}) \bmod M$$

To find F_2, the fingerprint corresponding to the next byte minus the oldest byte, the following equation is given:

$$F_2 = (p \cdot F_1 + t_{51} - t_1 \cdot p^{49}) \bmod M$$

However, this is incorrect. The equation should be

$$F_2 = (p \cdot (F_1 - (t_1 \cdot p^{49}) \bmod M) + t_{51}) \bmod M ,$$

where every possible $(t_1 \cdot p^{49}) \bmod M$ is stored in a table.

References

[1] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In Proceedings of the USENIX Conference on File And Storage Technologies (FAST), January 2002.

[2] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. Symposium on Operating System Principles, pages 174-187, 2001.

[3] B. Zhu, K. Li, H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. FAST '08: 6th USENIX Conference on File and Storage Technologies, pages 269 – 282. 2008.

[4] Udi Manber. Finding Similar Files in A Large File System. Technical Report TR 93-33, Department of Computer Science, University of Arizona, October 1993, also in Proceedings of the USENIX Winter 1994 Technical Conference, pages 17-21. 1994.

[5] National Institute of Standards and Technology, FIPS 180-1. Secure Hash Standard. US Department of Commerce, April 1995.

[6] J. Zhang and T. Suel. Efficient Search in Large Textual Collections with Redundancy. Proceedings of the 16th International Conference on World Wide Web, pages 411 – 420, 2007.

[7] J. Hamilton and E. W. Olsen. Deisgn and Implementation of a Storage Repository Using Commonality Factoring. 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS '03), page 178, 2003.

[8] Val Henson. An Analysis of Compare-by-hash. Proceedings of the 9th conference on Hot Topics in Operating Systems – Volume 9, page 3, 2003.

[9] PostgreSQL 8.3.1. http://www.postgresql.org/

[10] Oracle Database 10g Express Edition. http://www.oracle.com/technology/products/database/xe/index.html

[11] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy Elimination Within Large Collections of Files. Proceedings of the annual conference on USENIX Annual Technical Conference, page. 5, 2004.

[12] A. Tridgell and P. Mackerras. The rsync algorithm. Joint Computer Science Technical Report Series, The Australian National University, TR-CS-96-05, 1996.

[13] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, and A. Perrig. Opportunistic Use of Content Addressable Storage for Distributed File Systems. USENIX Annual Technical Conference, 2003.

[14] C. Policroniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. Proceedings of the annual conference on USENIX Annual Technical Conference, page 6, 2004.

[15] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in windows 2000. In Proceedings of the 4th USENIX Windows Systems Symposium, August 2000.

[16] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-Aware Semantically-Smart Storage. Design, implementation, and performance of storage systems, pages 29 – 35, 2006.