Design and Implementation of META and Log Record Reduction in GT.M Database

A Project Report
Submitted in partial fulfilment of the
requirements for the Degree of

Master of Engineering

in

Systems Science and Automation

by

Srikanth B.

Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

January 2002

Contents

Acknowledgments **ii**

Abstract **iii**

1 Introduction **1**
- 1.1 M language . . . 1
- 1.2 Greystone Database Structure - GDS . . . 2
- 1.3 Journalling . . . 4

2 Design and Implementation of META **6**
- 2.1 Search in GDS . . . 6
- 2.2 Design of META . . . 7
- 2.3 Search in META . . . 8
- 2.4 Adaption in META . . . 10

3 Log Record Reduction **11**
- 3.1 Compression algorithms . . . 12
- 3.2 Compression of Log records . . . 13
- 3.3 bzip . . . 14
- 3.4 gzip . . . 15
- 3.5 Optimizations made to libraries of *bzip* & *gzip* . . . 16

4 Data Sets used for META **18**

5 Performance Evaluation **20**
- 5.1 META . . . 20
  - 5.1.1 Discussion of Results . . . 20
- 5.2 Journalling . . . 22

6 Future work **24**

Bibliography **25**

A Analysis of META **26**

B Performance graphs of META **28**

Acknowledgments

I am greatly indebted to my project advisor, *Dr. Jayant R. Haritsa*, for his ceaseless support and encouragement throughout the course of the project. The stimulating words which inspire me towards perfection, his untiring enthusiasm to spare anything for the sake of work are admirable. I am fortunate enough to work under the guidance of such a teacher who loves to mould his students to excel and outperform himself and his expectations. Database Systems Lab, with its conducive atmosphere to learn and work and great facilities lacking in nothing, aided a lot in realizing this work.

I am grateful to *Prof. Narasimha Murty M.* and *Prof. Narahari Y.* for their guidance and backing during my entire course.

I would like to thank my lab-mates bandi, leela, nagesh and pankaj for their enjoyable friendly company and help. I would also like to thank amit, maya, srikanta, suresha and vikram for their cooperation, suggestions and valuable critiques which directed me properly.

Special thanks go to my friends ashok, baba, deepesh and subramanyam for making my stay in IISc a memorable one.

Finally, I would like to convey my gratitude to my parents and my sister, for their care and concern that made me what I am.

Abstract

GT.M is a commercial database system for which the observed bottlenecks are in access method of the database and in journalling of log records. This report proposes schemes to improve the system performance regarding the above two issues. Regarding the access method, the existing block structure in the database, **GDS**, uses a linear search to look for a key in a GDS block of variable length records, with prefix-compressed keys sorted in a collating sequence. This data structure is redesigned to improve the search time without the loss of prefix-compression. The new block structure, **META**, is designed and implemented that dynamically adapts itself to the variations in the contents of the block due to key insertions. The average improvement in search time got by META is approximately 30% of that of the original GDS block structure.

Journalling (logging) with before-imaging of blocks causes degradation of the system performance due to increase in I/O contention during normal database operation. The system performance can be improved by *on the fly* compression of log records that contain before-images of blocks, which reduces I/O time more than the increase in CPU time spent in compression. Different compression techniques are looked at and analysed for on the fly log record compression. The standard library versions of the compression methods *bzip* and *gzip* are modified to suit the compression of these log records which are relatively small in size (8 KBytes). Both these methods of compression are compared and found that *gzip* is better than *bzip* with respect to space and time in most cases. The compression ratio of log records achieved, on average, is 50%. Performance improvement of 30% of overall execution time is achieved.

Chapter 1

Introduction

Greystone Technology M - GT.M [1] is a database engine and development platform for transaction processing database applications with compiler of ANSI/ISO standard M language [2]. GT.M's data is type-less (unstructured array of bytes), hierarchical, multi-dimensional sparse arrays of content-associative memory. It is an open architecture database system that can support relational, object oriented and hierarchical conceptual models simultaneously on the same data. The database is accessible to concurrent host-based and client-server applications via C/C++, SQL and M. It is owned by Sanchez Computer Associates, Inc., Pennsylvania, USA. It is now available as open source. It is largely used in health-care centers and financial institutions like ING Group, American Express Membership Banking, etc. It is a package of 10MB 'C' code of 1700 files.

1.1 M language

M language is a simple general purpose procedural programming language oriented towards database applications. M is an alias for MUMPS which stands for Massachusetts general hospital Utility Multi-Programming System. It has built-in constructs for multi-tasking and multi-user support. It has untyped variables with automatic type conversions between numeric and string types. Syntactically, MUMPS has only one data-type : strings. Semantically, the language has many data-types: text strings, binary strings, floating point values, integer values, boolean values. Interpretation of strings is done inside functions, or implicitly while applying mathematical operators. It supports string subscripts for multi-dimensional arrays. The language does not need the user to use file opening or record declarations for file manipulations. They are implicitly done using persistent variables called Global variables. The main database operations supported are:

Set : sets a global variable, i.e., writes into database

Example: Set ^Ranking("IISc","CSA") = 1

Write : prints the value of a global variable, i.e., displays on the console

Example: Write ^Weather("Bangalore","IISc","26^{th} Jan")

Kill : deletes a global variable from database

Example: Kill ^Terrorist("Osama","Afghanistan")

1.2 Greystone Database Structure - GDS

GT.M environment allows the variables in a M program [4] to be made persistent. They are called Global variables. The global variables can be un-subscripted (simple) or subscripted (arrays). The subscripts can be of any data type like integer, real, character and string. A key is a global variable name including the subscripts (if any) in the form of encoded type-less bytes with a collating sequence. For example, consider 'Student' as global variable and "Srikanth", 604700080 as subscripts of the variable 'Student'. 'Student' is a global variable name of a multi-dimensional array and 'Student("Srikanth",604700080)' is a key which is a subscripted global variable (an element of the array).

The records, which are (key, data) pairs, are organized as balanced hierarchical B-trees of blocks called Greystone Database Structure (GDS) blocks. Each node in a B-tree is a GDS block. These GDS blocks are stored in the database in ordinary Unix files. Each un-subscripted global variable (array name) and all the keys (with/without subscripts) of that variable are organized as a B-tree called *Global Variable Tree* (GVT). All these un-subscripted global variable names (array names) are in turn arranged in another B-tree called *Directory Tree* (DT). The GDS blocks in the B-tree can be index blocks whose fan-out is the number of variable length records in it containing pointers to child blocks or leaf (data) blocks that actually contain the data. The data blocks in the directory tree contain un-subscripted global variable names as keys and pointers to the roots of the GVTs of these global variables as the data. The keys are stored with prefix-compression in comparison with their previous key in each GDS block [3].

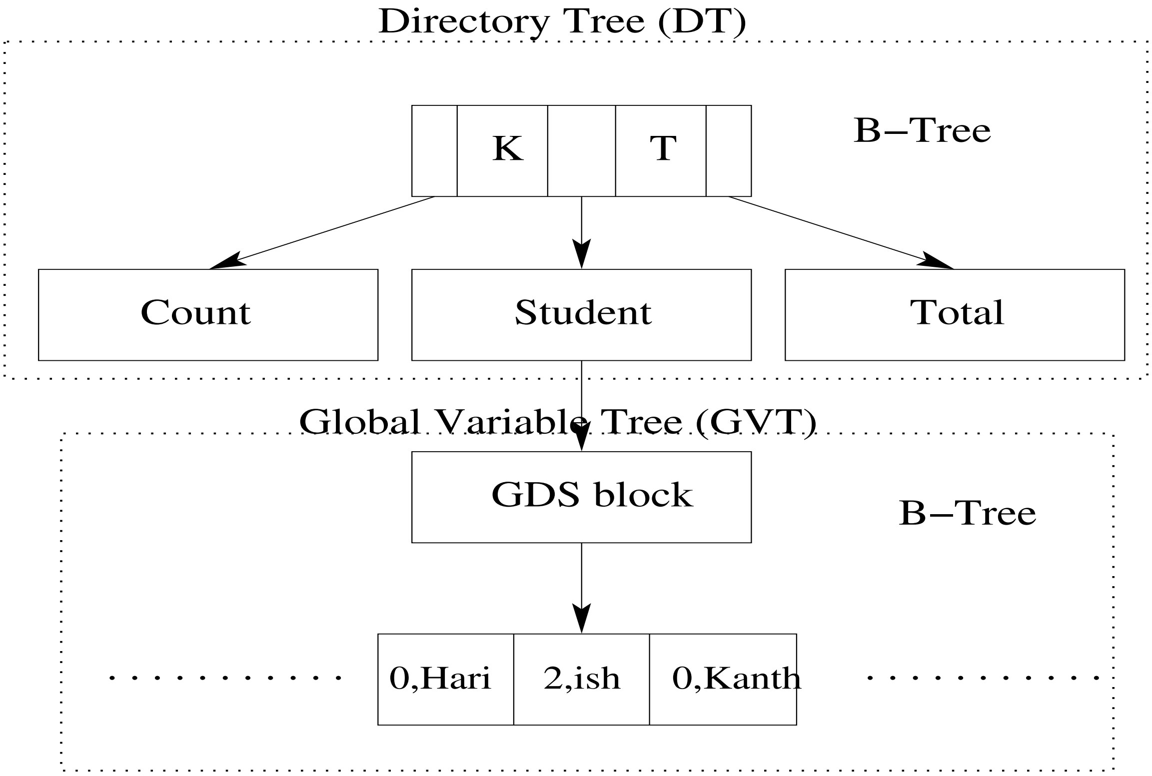

Figure 1.1: Greystone Database Structure

The Figure 1.1 shows an example of Greystone Database Structure. In this, 'Count' and 'Total' are un-subscripted global variable (simple). 'Student' is a subscripted global variable (multi-dimensional array). The Directory Tree (DT, upper half in Figure 1.1) contains details of all global variables (un-subscripted and subscripted) in the form of B-tree. The Global Variable Tree (GVT, bottom half of Figure 1.1) of 'Student' global variable is organized as a B-tree containing keys like 'Student("Hari")', 'Student("Harish")' and 'Student("Kanth")', etc. These keys are prefix compressed with previous key and stored as compression (prefix) count and the rest of the key. For example 'Student("Harish")' is stored as '2,ish' (see the Figure 1.1).

In a GDS block of variable length records with prefix-compressed keys, given a search key, search has to start all the way from beginning of the block for finding the nearest match to it. This linear search is because of the prefix-compression of the keys. This search time needs to be improved (see Chapter 2).

1.3 Journalling

The second issue is logging (journalling). There are two types of log records in GT.M journalling. The M-level journalling uses log records that contain the information of logical updates (of the order of 20-30 bytes). This journalling logs all the commands which update the database. The Before-Image journalling uses log records that contain the before-images of the blocks of updation. They store both the logical and physical update of the GDS blocks (both commands and the before-images of updated blocks). These second type of log records consume more space (of the order of 8 KBytes). This hinders the performance of the system, when logging is enabled, due to more I/O contention. To reduce the I/O contention, these log records are to be compressed on the fly during normal database operation. Here is a brief overview about Information Theory and Compression.

In Information Theory [6], the term **entropy** is used to quantify the amount of information content in a given data. The term entropy is used with many different interpretation as measure of uncertainty or randomness associated with a probability distribution, expected value of $\log(1/p(X))$ (p is the probability distribution of random variable X), thermodynamic entropy of a system, etc. In data compression, the entropy of a symbol is the negative logarithm of its probability of occurrence. The entropy of a data set is sum of the entropies of all individual symbols in it, provided the occurrence of one symbol is independent of its context (neighboring symbols). In an alphabet like ASCII, the number of bits used to represent any symbol is same (8 bits), assuming that all the symbols are equally probable. Practically, in a given data set, the occurrence of symbols of an alphabet may not be so uniformly distributed, with some symbols having high probability of occurrence and some low having probability of occurrence. This helps in compressing the data set by mapping the symbols or strings of symbols to code based on the information content or entropy (probability of occurrence) of the symbols.

Data compression, when used offline, like in archiving, file compression, etc., is mainly concerned about achieving high compression (ratio). The compression or decompression overhead (CPU time) is not very critical. When used for compressing log records (or data in database) on the fly, apart from compression ratio, the compression and decompression overheads are also critical. The algorithm chosen should neither be simple giving less compression nor be very complex with more overhead of (de)compression. Hence, a trade-off has to be considered between speed and compression ratio. Log records, being relatively small when compared with normally used input files for compression, cannot be compressed with a simple algorithm, since it yields poor results. The existing standard compression methods like *bzip*, *gzip*, etc. cannot be used directly, since they consume more time, space and have redundant features. Hence, the standard libraries of *bzip* and *gzip* are tailored to suit our requirements.

The rest of the report follows this pattern :
Chapter 2 gives the details of the design and implementation of the new block structure META. Chapter 3 deals with the issue of log record reduction. Chapter 4 covers the types of data blocks used for performance analysis of META. Chapter 5 gives the performance evaluation of system. Chapter 6 shows the chances of further improvement in performance.

Chapter 2

Design and Implementation of META

In a GDS block, the prefix-compressed keys are stored in sorted non-decreasing order of a collating sequence. The search for a key in a given block is to find the position of the key in the block that is equal to or just greater than the search key, i.e., to find the position where the search key should have been in the block.

2.1 Search in GDS

The algorithm for search is as follows. Here, match of two keys is number of bytes common between them; compression of a key is the number of bytes of key compressed with respect to its immediate predecessor.

1. The current key in the block for comparison is first key with 0 compression
   current match $= 0$
2. *if* (reached end of block) *stop*
3. *if* (current key compression $>$ current match) *goto* Step 6 //skip to next key
4. *else if* (current key compression $<$ current match) *stop* //position found

5. compare search key with compressed current key
   if (search key < current key) *stop* //position found
   else current match = match of search key with current key

6. current key = current key + record length //next key

7. *goto* Step 2

This is a linear search that does not utilize the property of keys being sorted in GDS block, due to prefix-compression of variable length keys.

2.2 Design of META

The problem concerned here is to improve the search time in a GDS block of variable length records having compression, without loss of compression. This problem is similar to the problem of organizing variable length records in a typical database problem. The general alternatives are storing offsets of the starting of records in a fixed location in the block either in the form of an array or as a tree. But, this is not feasible here, since the keys have to be uncompressed to do so, which is against our above requirement.

Another possibility is to pick up a subset of keys from the block and store them in a reserved space as a coarse index. These *meta*-keys should span the entire range of the block. Here, the keys in the GDS block need not be uncompressed. The meta-keys can in turn be prefix-compressed either with respect to parent-key of the block or with respect to the preceding meta-key. If they are prefix-compressed with respect to parent-key, the search within a block will not be independent of other blocks and allows to traverse only down the B-tree of blocks but not the other way, which is needed in some cases. Hence, the method chosen prefix-compresses meta-keys with respect to their immediate predecessor in the GDS block. This new structure is termed as **META** in this report.

There are many alternatives for picking up meta-keys that can span the entire GDS block :

i) Choosing Optimal subset of meta-keys (small, many and well-spanned) that can fit in the reserved space available by linear programming method

ii) Picking up meta-keys at regular intervals of *bytes*

iii) Picking up meta-keys at regular intervals of *records*

The first choice of linear programming turns out to be expensive when compared to the in-memory block search we are concerned about. The reason for this and performance analysis are stated in Appendix A. The second choice of picking up meta-keys at regular intervals of bytes is improper since these positions may not be the beginning of keys due to variable length records. Hence, the meta-keys are chosen at regular intervals of *records* proportionate to the reserved space available.

2.3 Search in META

In the new structure, a small part of each block is reserved for storing *meta-keys*. Meta-keys are chosen from the keys that appear in the data segment of the block. They are chosen such that there is (approximately) an equal number of keys within each interval defined by neighboring meta-keys. Along with each meta-key, a pointer to the beginning of the data record in which that meta-key appears is stored. Each meta-key, barring the first, is stored in compressed format, maintaining only its difference with respect to the previous key. This is exactly identical to the way that keys in the data segment are stored in compressed format with respect to their predecessors. A pictorial representation of this new structure is shown in Figure 2.1.

Given the above structure, searching for a key proceeds as follows :
First, the key is sequentially compared with the set of *meta-keys* and the pointer associated with the largest meta-key that is less than the search key is followed into the data segment. The subsequent search process is identical to the above mentioned algorithm for searching keys in a GDS block, except that the starting record is the one pointed to by the meta-key, rather than the first physical record in the block. In short, instead of always searching

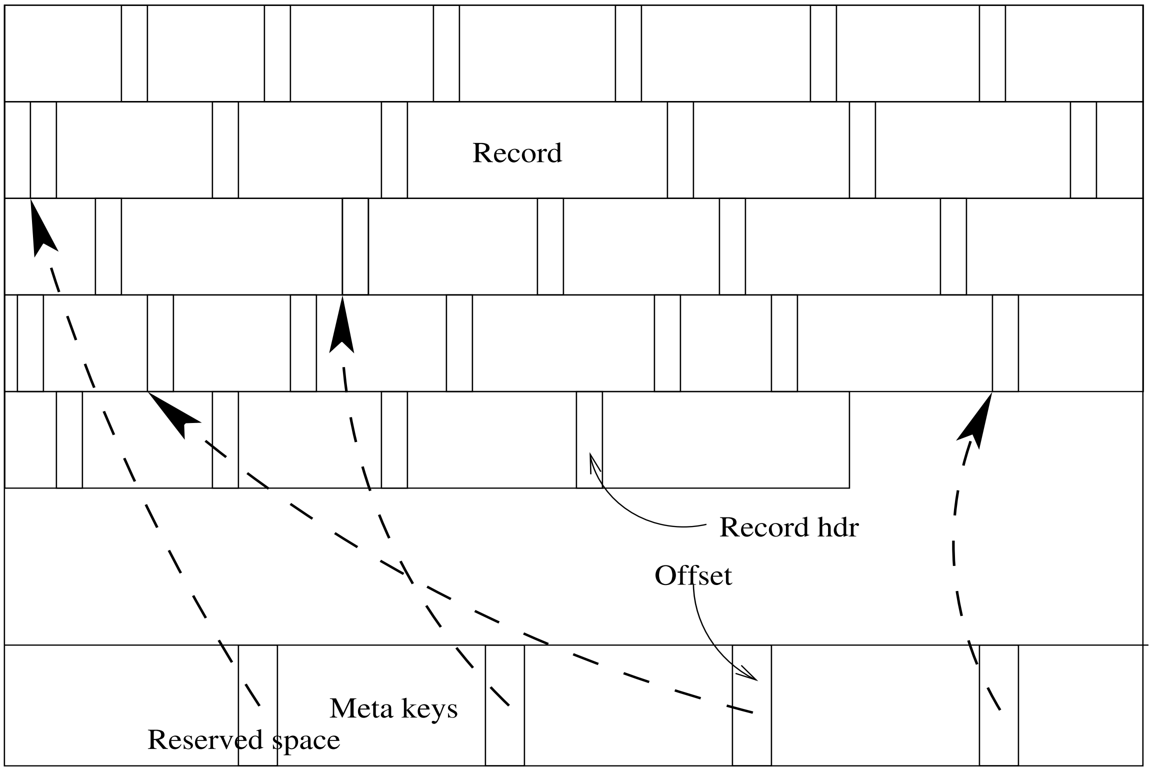

Figure 2.1: GDS block with meta-keys

from the beginning of the block, the meta-keys are used to quickly identify the appropriate portion of the block in which to search.

META structure resembles that of skip-lists [12]. Skip-list is a type of linked list having nodes that have levels, level being the number of pointers each node has. The level of a node is probabilistically determined when inserted. A level i pointer will point to next node of level i or more that is, on an average, 2^i nodes ahead. The search for a node starts at the highest level node and its level being current level. It proceeds to search nodes at that level, skipping the nodes in between, till the sub-range of nodes where the search node might be found is reached. The current level is reduced and the search is repeated till the current level is reduced to 1, i.e., till the node or next greater node is found. META structure is similar to a skip-list of having only two levels, the level two nodes (meta-keys) skipping nodes (keys) at regular intervals.

2.4 Adaption in META

Whenever there is an insertion of records in the blocks, the meta-keys have to be rebuilt. This will degrade the performance of the system, instead of improving it, since re-choosing the meta-keys for every update is quite expensive. Hence, the meta-keys are dynamically rebuilt (updated) whenever the search behaves worse than expected.

Initially, there will be no meta-keys in the GDS block. Whenever a search for a key takes much more number of key comparisons than the size of the interval of meta-keys at which they are picked from the block, the meta-keys are (re)chosen, expecting that more number of keys were inserted (in that interval) in the block after the previous choice of meta-keys. This makes it to self-organize itself to the new insertions dynamically only when needed. This new structure is suitable for the type of queries that have more searches than updates, which is the typical case with GT.M database.

Chapter 3

Log Record Reduction

GT.M uses journalling as a logging scheme for database recovery [3]. There are two schemes of journalling :

i) M-level journalling :

In M-level journalling, the logical updates (commands) are logged in the journal file before doing the physical updates to the database. In case of a crash, once a backup of the database is restored, the logical updates are played forward from the journal to the database.

ii) Before-image journalling :

In Before-image journalling, the logical updates as well as the physical updates (before-images) of the GDS blocks are logged into the journal file. This facilitates us to do backward recovery which is much faster than forward recovery. The database and journals are periodically synchronized (around 5 minutes) by check-pointing. This is called an EPOCH. An epoch journal record is written in the journal file and the database and journals are synchronized as of that point. Whenever a GDS block is updated for the first time in an epoch, a copy of the block is written into the journal as a record called **PBLK** journal record, before the actual update. This is called Before-image of the block. In case of a crash, as part of backward recovery, the PBLKs (before-image blocks) are played back upto the last EPOCH point. At this point, the database is consistent with the journals. Then, the logical records (that are inter-mixed with the PBLKs in the journal file) are played forward until the end of the journal file.

Since a before-image record is almost of the size of a GDS block (which is of the order of 8 KBytes), and since as many before-image records get written as the number of GDS blocks that change during the course of one epoch, the journal file becomes huge compared to M-level journalling. Hence, it is desirable to get these records compressed before writing the PBLK records into the journal-buffer (from where it is then flushed to the journal file on disk). This way, it causes less I/O contention as part of normal database activity due to reduced journal I/O requirements. But the compression algorithm will now result in a bit more CPU work. Here, the compression speed of the algorithm is of more concern than that of decompression.

3.1 Compression algorithms

There are two types of compression algorithms [7] :

i) Lossless compression :

   The input can be exactly restored after decompression eg. archiving, ordinary file compression, GIF, etc. There are many lossless compression algorithms like Huffman coding, Arithmetic coding, LZW algorithm, etc.

ii) Lossy compression :

   The input cannot be exactly restored after decompression. These are used to compress efficiently at less accuracy eg. multimedia file compression, JPEG, etc. There are many lossy compression algorithms like lossy LZ algorithm, fractal transform, wavelet transform, discrete cosine transform compression, etc.

Here, the log records are to be compressed while logging and decompressed without any loss during recovery. Hence, we are concerned about lossless compression of PBLK journal records.

There are two types of compression methods :

i) Statistical methods :

   These methods require accurate estimation of probability distribution of occurrence of the symbols of an alphabet in the input string *. The proper mathematical model (order of the model) chosen should reflect the dependencies between the symbols.

ii) Dictionary based methods :

   These methods store a dictionary of strings in the file and replace all occurrences of the same string with (length, distance) pair, where length is that of the string and distance is the number of symbols between the string and its last occurrence.

There are two types of compression models :

i) Static models :

   These models need prior knowledge of the probabilities of occurrence of the symbols or the dictionary of strings of symbols. The codes for all the symbols are static over an entire given input data set.

ii) Adaptive models :

   These models decide the codes for the symbols or strings of symbols dynamically as the input data is processed. The codes change based on the variations of the probabilities of occurrence or dictionary contents.

3.2 Compression of Log records

In static statistical methods, like Huffman coding, Shannon coding, Arithmetic coding, etc., either the probabilities or the codes of all the symbols are to be stored along with the compressed file. Since the size of the log records to be compressed are relatively small when compared to normally used input files for compression, it is not affordable to store these extra information, which might even increase the size of record instead of compressing it. In case of adaptive methods, the log records are not long enough that a good accurate model is learnt from it and hence compression may not be to the

*string is taken as sequence of any bytes (not only printable) in general desirable extent. Similarly, in case of dictionary based methods, like LZ77, LZW, etc., the dictionary has to be stored along with the compressed block. Hence, it is not feasible to use a simple method to compress these PBLK records (Before-Images).

The headers themselves can be compressed or eliminated or combinations of different methods can be used to give better compression but with more (de)compression overhead. The standard libraries of the compression methods **bzip** or **gzip** give good compression in general, as they use hybrid combinations of different methods, but have more (de)compression overhead. *bzip* uses Burrows-Wheelers transform, Move-To-Front, Run-Length-Encoding, and Huffman coding. *gzip* uses LZW method, Run-Length-Encoding and Huffman coding. These libraries cannot be directly used, since they consume more space and time and have functionalities not so relevant for our purpose. Hence, they are modified and made suitable to the compression of PBLK records. Here is a brief outline of the methods employed in *bzip* and *gzip*.

3.3 bzip

bzip [10] uses Burrows-Wheelers block sorting text compression algorithm and Huffman coding. BW block sorting algorithm [11] transforms the input string in such a way that the symbols with similar context are grouped, which makes it better for compression. Since the transform is invertible, the original input string can be obtained back. Briefly, the algorithm transforms a string S of N characters by forming the N rotations (cyclic shifts) of S, sorting them lexicographically, and extracting the last character of each of the rotations. A string L is formed from these characters, where the i^{th} character of L is the last character of the i^{th} sorted rotation. In addition to L, the algorithm computes the index I of the original string S in the sorted list of rotations. There is an efficient algorithm to compute the original string S given only L and I. The sorting operation brings together rotations with the same initial characters. Since the initial characters of the rotations are adjacent to the final characters, consecutive characters in L are adjacent to similar strings in S. If the context of a character is a good predictor for the character, L will be easy to compress with a simple locally-adaptive compression algorithm like Move-To-Front

(MTF) encoder. This property is exactly the one needed for effective compression by Move-To-Front encoder, which encodes an instance of a character by the count of distinct characters seen since previous occurrence of that character. The output of MTF encoder is dominated by low numbers, which can be compressed by Huffman coding.

The most important factor in compression speed is the time taken to sort the rotations of the input block. A simple application of quick-sort requires little additional space (one word per character), and works fairly well on most input data. However, its worst-case performance is poor. First the string S_0 is formed, which is the concatenation of input string S with EOF, a new character that does not appear in S. Because EOF is different from all other characters in S_0, the suffixes of S_0 sort in the same order as the rotations of S_0. Hence, to sort the rotations of S, it suffices to sort the suffixes of S_0. Thus, the problem of sorting rotations of input string S is reduced to the problem of sorting the suffixes of S_0, which can be solved in O(N log(N)) time. The suffixes of string S_0 are sorted efficiently using a combination of quick-sort and exponential radix sort.

3.4 gzip

The compression algorithm used by *gzip* [8] is a variation of LZ77(Lempel-Ziv 1977) algorithm [9]. It finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to its previous occurrence, in the form of a pair (distance, length), where length is that of the string and distance is the number of symbols between the string and its last occurrence. Distances are limited to 32 KBytes, and lengths are limited to 258 bytes. When a string does not occur anywhere in the previous 32 KBytes, it is emitted as a sequence of literal bytes. Literals or match lengths are compressed with one Huffman tree, and match distances are compressed with another tree. The trees are stored in a compact form using Huffman coding at the start of compressed data. It efficiently decides on the fly whether to apply static Huffman coding or dynamic Huffman coding. Duplicated strings are found using a hash table. All input strings of length 3 are inserted in the hash table. A hash index is computed for the next 3 bytes. If the hash chain for this index is not empty, all strings in the chain are compared with the current input string, and the longest match is selected. The hash chains are searched starting with the most recent strings, to favor small distances and thus take advantage of the Huffman encoding. There are no deletions from the hash chains, the algorithm simply discards matches that are too old.

To avoid a worst-case situation, very long hash chains are arbitrarily truncated at a certain length, determined by a compression level (-1 to -9). So gzip does not always find the longest possible match but generally finds a match which is long enough. gzip also defers the selection of matches with a lazy evaluation mechanism. After a match of length N has been found, gzip searches for a longer match at the next input byte. If a longer match is found, the previous match is truncated to a length of one (thus producing a single literal byte) and the longer match is emitted afterwards. Otherwise, the original match is kept, and the next match search is attempted only N steps later.

The lazy match evaluation is also subject to a compression level. If the current match is long enough, gzip reduces the search for a longer match, thus speeding up the whole process. If compression ratio is more important than speed, gzip attempts a complete second search even if the first match is already long enough. The lazy match evaluation is not performed for the fastest compression levels (speed options -1 to -3). For these fast modes, new strings are inserted in the hash table only when no match was found, or when the match is not too long. This degrades the compression ratio but saves time since there are both fewer insertions and fewer searches.

3.5 Optimizations made to libraries of *bzip* & *gzip*

The libraries of *bzip* and *gzip* cannot be directly used, since they consume more space and time and have functionalities not so relevant for our purpose of compressing relatively smaller input data. Hence, they are modified and made suitable to the compression of PBLK records. The modifications made are :

The dynamic memory allocations (*malloc calls*) done when every PBLK log record is compressed are removed and allocated only once during the initialization.

Memory requirements are reduced suitable to range of input sizes (maximum block size of 64 KBytes).

Headers and Correction codes (CRC) are removed.

For *bzip*, a combination of exponential radix sort and quick-sort is used for log record compression instead of 3-way quick-sort used for large inputs. This version is termed as *o-bzip*.

For *gzip*, a medium speed searching of duplicate strings is used without any lazy matches instead of a slow speed searching with lazy matches. This version is termed as *o-gzip*.

The *o-bzip* and *o-gzip* versions are tested with compression of different types of input data (text, binary and mixed). These results are mentioned in Table 3.1. The compression times taken by the modified versions are almost half of that of their standard library counterparts. The average compression ratio (percentage of compression to the original size) of *o-bzip* is 45% and that of *o-gzip* is 50%. In the results shown, *bzip* performs worse than *gzip* because of the expensive sorting of all possible rotated strings in *bzip*. This is the same reason why compression time of *bzip* increases non-linearly.

Block size (bytes)	*bzip*		*o-bzip*		*gzip*		*o-gzip*	
	c. size† (bytes)	c. time‡ (ms)	c. size (bytes)	c. time (ms)	c. size (bytes)	c. time (ms)	c. size (bytes)	c. time (ms)
1k	510	3.41	496	1.78	489	0.72	483	0.39
2k	958	6.53	944	3.11	946	1.25	940	0.61
4k	1723	8.21	1709	4.43	1595	1.71	1589	0.87
8k	3271	13.19	3257	6.779	3102	2.06	3096	1.05

Table 3.1: Optimization results

†Average compressed size
‡Average compression time

Chapter 4

Data Sets used for META

To evaluate the efficacy of the META design, a range of tests are conducted. For the test data, the blocks used are from the real test set [5] provided by the company. The blocks are of 4 KBytes with 10% of the block size left as reserved space for the meta-keys in each block. For ease of comparison, it is assumed that this space is left vacant in the original GDS blocks as well. Here key refers to uncompressed key, unless mentioned.

The blocks are categorized into the following five types based on their key and data characteristics (Representatives of each type are evaluated in our study):

Type 1 Short keys (small $\Longrightarrow$ < 20), few records in GDS block (few $\Longrightarrow$ < 100)

Type 2 Short keys, many records

Type 3 Long keys, low compression ratio(low $\Longrightarrow$ < 0.5), few records

Type 4 Long keys, high compression ratio, few records

Type 5 Long keys, high compression ratio, many records

The reserved space was filled with meta-keys until either the space ran out or until 10% of actual keys had been inserted, whichever occurred earlier. The complete details of the data blocks are given in Table 4.1.

1	2	3	4	5	6	7	8	9	10
Meta-Keys				**Keys**				**Data**	
# of keys	**key size**	**Comp. key size***	**Comp. ratio†**	**# of keys**	**key size**	**Comp. key size***	**Comp. ratio†**	**% space‡**	**data size**
(1) Short keys, few records									
6	9	3	0.67	59	9	3	0.67	88	54
(2) Short keys, many records									
30	11	3	0.73	305	11	3	0.73	41	5
30	12	4	0.67	305	12	3	0.75	41	5
(3) Long keys, low ratio, few records									
6	67	59	0.12	54	67	59	0.12	7	5
6	62	54	0.13	54	67	59	0.12	7	5
5	74	66	0.11	55	67	58	0.13	7	5
(4) Long keys, high ratio, few records									
6	30	3	0.90	54	30	3	0.90	77	26
6	27	5	0.81	51	26	3	0.88	78	28
6	53	8	0.85	64	52	3	0.94	72	21
(5) Long keys, high ratio, many records									
23	91	8	0.91	262	91	5	0.95	36	5

Table 4.1: Details of Data blocks

*average compressed (meta)key size
†compression ratio = (key size - compressed key size)/key size
‡percentage space occupied by data to total space

Chapter 5

Performance Evaluation

5.1 META

The performance of the routine which looks for a match to a given search key for each of the data blocks described in the Chapter 2 has been evaluated. To ensure complete and accurate results, the search is iterated 1000 times for each and every key in a block. With respect to the average search performance, the results are given in Table 5.1 (the detailed performance graphs are given in the Appendix B).

5.1.1 Discussion of Results

The observations from the results show that for most of the data block types, META provides significant improvement over the original version, especially when the number of records (hence, meta-keys) are more. For example, for Type 2 blocks ('short keys, many records' type blocks), the percentage gain in search time is over 60% (see last column of Table 5.1). We also found that for most search key locations, META's performance is better than or as good as the original. However, searching for a key located in the *first* interval of keys will *always* be more expensive in META since the search-key has an additional meta-key comparison without any compensating reduction in the search space as compared to the original. For blocks where the number of meta-keys is small (recall that we have limited the number of meta-keys to be approximately 10% of the set present in the data block), the size of the first interval is significantly large, and therefore the overall average performance may not turn out to be appreciably better.

ORIG time (ms)	**META time (ms)**	**Time gained (ms)***	**% gain†**
(1) Short keys, few records			
2.04	1.36	0.68	33.33
(2) Short keys, many records			
5.20	1.79	3.41	65.58
5.27	1.95	3.32	63.00
(3) Long keys, low ratio, few records			
3.07	2.63	0.44	14.33
3.07	2.65	0.42	13.68
2.87	2.63	0.27	08.36
(4) Long keys, high ratio, few records			
2.47	1.83	0.64	25.91
2.08	1.76	0.32	15.38
2.77	2.32	0.45	16.25
(5) Long keys, high ratio, many records			
6.09	3.37	2.72	44.66

Table 5.1: Observations in the tests (Search time)

Type 1 Short keys, few records :

This type of blocks have a good improvement of search time since the key size is small (see Figure B.1).

*ORIG time - META time

†(gain * 100)/ (ORIG time)

Type 2 Short keys, many records :

For this type of blocks, the performance improvement is very good since both the favorable conditions (small sized keys and more meta-keys) are in operation (see Figures B.2(a)-B.2(b)).

Type 3 Long keys, low compression ratio, few records :

In this case, the improvement is less when compared to all other cases. Since the key size is more, the first (meta)key is long. This increases the search time for keys in first interval of keys in the block. In addition to that, as the number of meta-keys are less (number of records being less), the *average* performance improves only marginally (see Figures B.3(a)-B.3(c)).

Type 4 Long keys, high compression ratio, few records :

Blocks of this type have considerable improvement. The keys in the first interval take more search time. For the rest of the keys, as the compression is more, there is improvement nullifying the effect of less number of meta-keys equivalent to that of Type 1 (see Figures B.4(a)-B.4(c)).

Type 5 Long keys, high compression ratio, many records :

This type of block will have search time more for the initial keys because of the long key stored at beginning of both the reserved space and the actual data. Since number of records are more, number of meta-keys are more. Except for search-keys in the first interval, the performance is as good as that Type 2 for the rest of the search-keys (see Figure B.5).

5.2 Journalling

The Before-image journalling is tested with PBLK records compressed with *o-bzip* and *o-gzip*. In the these tests, the data blocks of size 8 KBytes are used. The test data set contained 2500 PBLK records which are to be compressed and 6,45,000 M-level (logical) records. The tests are run on a Pentium III 800 MHZ Linux machine with 40 GBytes of

IDE disk storage and 512 MBytes of main-memory. The machine is kept in single-user mode and the running process is given higher priority (nice value of -4), so that there is no much disturbance from other process contending for resources (CPU+I/O). The same test is run 40 times to find the average execution time for comparison. Here we have evaluated only the compression times of the PBLK log records.

The average time taken for the compression of 8 KByte blocks by *o-bzip* is 2.5ms and that of *o-gzip* is 0.9ms. There is considerable amount of reduction in I/O, since the effective size of the records are reduced to 50% of it.

The original method without log record compression took 181.400 sec for execution while the method with log record compression using *o-bzip* took 223.100 sec and the method with log record compression using *o-gzip* took 123.950 sec. There is performance degradation in case of method using *o-bzip*, because the time gained due to reduction in I/O contention is nullified by the high overhead (CPU time) for compressing the PBLK log records. This very high compression overhead for *o-bzip* is because of the expensive sorting of strings (as mentioned in Section 3.5). The net improvement due to the reduction of log record size is not appreciable because of the larger number of uncompressed log records when compared to the compressed PBLK records.

Chapter 6

Future work

There are many other issues to be improved for better system performance. The block splitting criteria currently employed in GDS is to split the block at the point where the key causing the split is inserted. This may result in uneven distribution of keys between the splitting blocks. Splitting such that the number of keys are equally divided the split blocks may result in performance improvement due to reduction in number of splits.

There is a scope for improvement by efficient compression of all types of log records so as to reduce I/O contention with minimal expense CPU time.

The concurrency control mechanism used in the system is Optimistic method. It uses Backward validation of committing transactions. This may be experimented by replacing it with Forward validation method.

Bibliography

[1] http://www.sanchez-gtm.com/

[2] http://www.mtechnology.org/

[3] GT.M Administration and Operations Guide, *Sanchez Computer Associates*, February 2000

[4] GT.M Programmer's Guide, *Sanchez Computer Associates*, April 2000

[5] GT.M Version 4.2 Profiling the M programs & Testing Utilities, Technical Bulletin 55-010-016, *Greystone Group, Sanchez Computer Associates*, December 2000

[6] M. Cover and A. Thomas, "Elements of Information Theory", Wiley-Interscience, New York, 1991

[7] M. Nelson and J. Gailly, "The Data Compression Book", M&T Books, New York, 1995

[8] J. Gailly and M. Adler, "Zlib general purpose compression library version 1.1.3", July 1998

[9] A. Lempel and J. Ziv, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp. 337-343, May 1977

[10] J. Seward, "bzip2 and libbzip2", A program and library for data compression, March 2000

[11] M. Burrows and D. Wheeler, "A Block sorting Lossless Data Compression Algorithm", SRC Research Report, *Digital Equipment Corporation*, May 1994

[12] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees", *Communications of the ACM*, Vol. 33, No. 6, pp. 668-676, June 1990

Appendix A

Analysis of META

Input : A GDS block of n keys and **M** bytes of space for *meta*-keys

Output : Sequence of *meta*-keys

Optimization criterion : Minimize Average search time for all search keys

Constraint : Output should fit in **M** bytes

Let $k_1, k_2, ..., k_n$ be the lengths of keys in the GDS block in sequence and $c_1, c_2, ..., c_n$ be the compression counts of the respective keys, compression count being the number of prefix-bytes of a key common to its previous key in the block. The search time of a search key can be analysed in terms of two metrics

1) Number of byte comparisons

2) Number of key comparisons

Byte comparisons as metric : When number of byte comparisons is used as metric for search time analysis, the analysis of the complexity of the searching algorithm will be dependent on all the values of k_i and c_i, $1 \leq i \leq n$. The search time complexity will involve many min, max functions which depend on value distributions of key lengths and compression counts of the keys in the block. Even for a given search key, the search time calculation will be complex. Hence, objective function of this optimization problem (Minimize average search time over all search keys) will itself turn out to be complex.

With number of byte comparisons as metric, the optimal sequence of *meta*-keys with minimum average search time cannot be chosen using linear programming techniques since the objective function itself is non-linear. Any programming technique needs to assign weights to different keys based on their length, longer keys given more weightage over shorter keys, since they need more byte comparisons. Solving this programming problem with a complex objective function and constraints would be very expensive. Solving this problem using heuristics might result is near to optimal solution but at the expense of time. Hence this way of solving the optimization problem is not feasible

Key comparisons as metric : When number of key comparisons is used as metric, the assumption is that all the keys in the block take same individual comparison time. All keys are equally important except for consideration of space limitation of the variable-length *meta*-keys.

Let us consider the problem without space limitation first. Let p fraction of n keys are picked at regular key intervals of size $1/p$ as *meta*-keys. Worst case search time will be $np + 1/p$, where np is the number of *meta*-keys compared and $1/p$ is the number of keys compared in that sub-interval. Best case search time will be $\Theta(1)$. Average search time will be $np/2 + 1/2p$. Minimizing this function (average search time) will result in $p = 1/\sqrt{n}$. This means that picking $\sqrt{n}$ keys at regular key intervals of size $\sqrt{n}$ will result in optimal average search time of $\sqrt{n}$ key comparisons. This is the optimal solution without any restriction on space for *meta*-keys.

Consider the given problem with limitation of $\mathbf{M}$ bytes of space for *meta*-keys. Let $\mathbf{M}$ be 10% of GDS block size. The optimal solution without limitations has to be compromised with space limitations. When $\sqrt{n} < 10$, i.e, $n < 100$, picking up *meta*-keys at regular intervals of 10 keys (proportional to fraction of reserved space to size of GDS block) till the reserved space gets filled will give a near to optimal solution when the key length distributions are not much skewed. Fewer *meta*-keys are picked than that of optimal interval size of $1/\sqrt{n}$ keys, due to limitation of the space. When $\sqrt{n} > 10$, i.e, $n > 100$, picking up *meta*-keys at regular intervals of $1/\sqrt{n}$ keys, which is the optimal, will almost surely result in fitting these in the space of $\mathbf{M}$ ($1/10^{th}$ of block size), since the fraction $1/10$ is larger than the fraction $1/\sqrt{n}$ of optimal *meta*-keys, unless the key lengths are very skewed. Hence, a simple method of choosing *meta*-keys at regular intervals gives near to optimal solution in most cases at a very low cost. Hence, a complex linear programming optimization method would be of no help even when the metric of search time is number of key comparisons.

Appendix B

Performance graphs of META

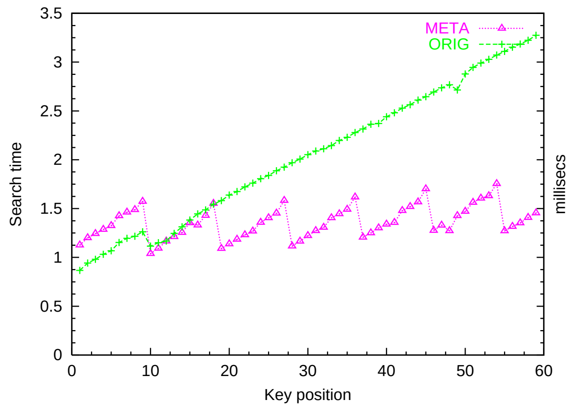

Figure B.1: **Type 1** Short keys, few records

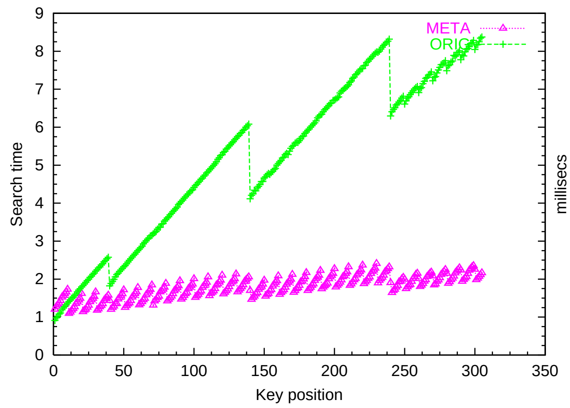

Figure B.2(a): **Type 2** Short keys, many records

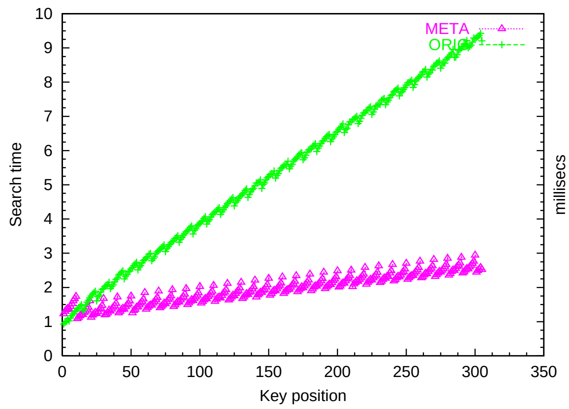

Figure B.2(b): **Type 2** Short keys, many records

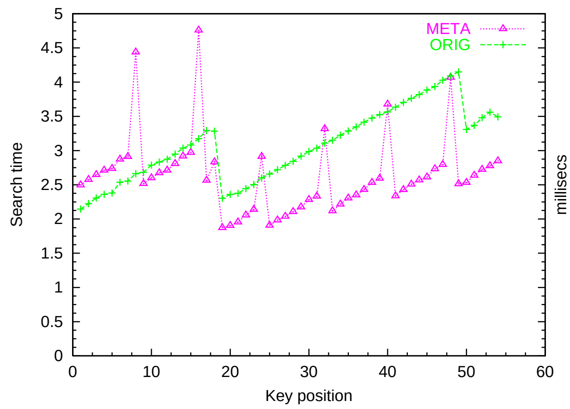

Figure B.3(a): **Type 3** Long keys, low compression ratio, few records

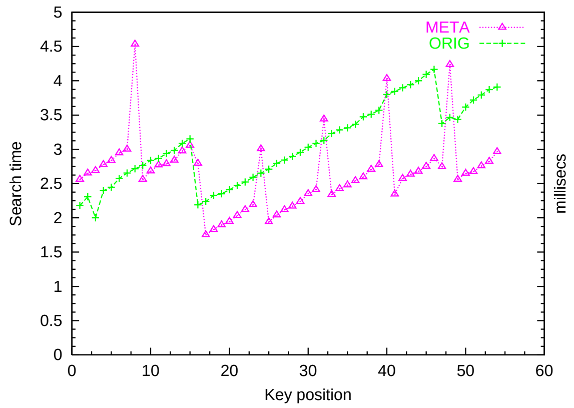

Figure B.3(b): **Type 3** Long keys, low compression ratio, few records

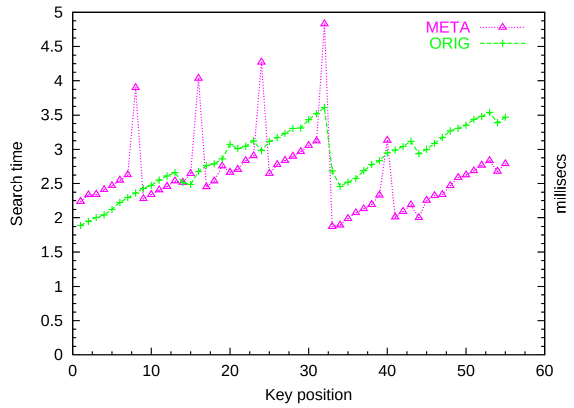

Figure B.3(c): **Type 3** Long keys, low compression ratio, few records

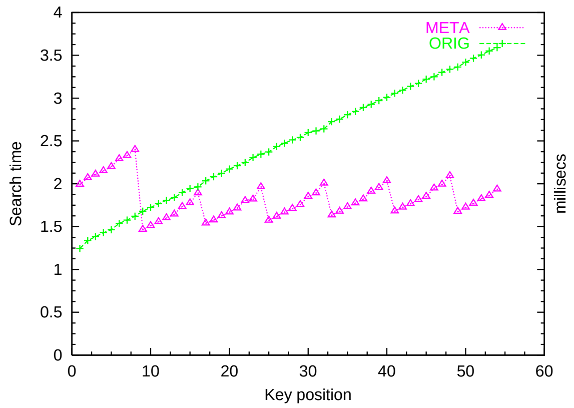

Figure B.4(a): **Type 4** Long keys, high compression ratio, few records

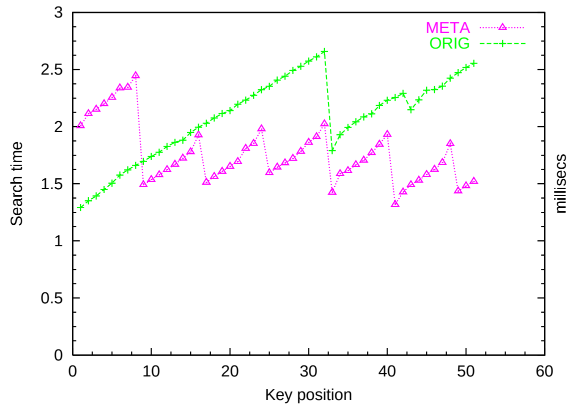

Figure B.4(b): **Type 4** Long keys, high compression ratio, few records

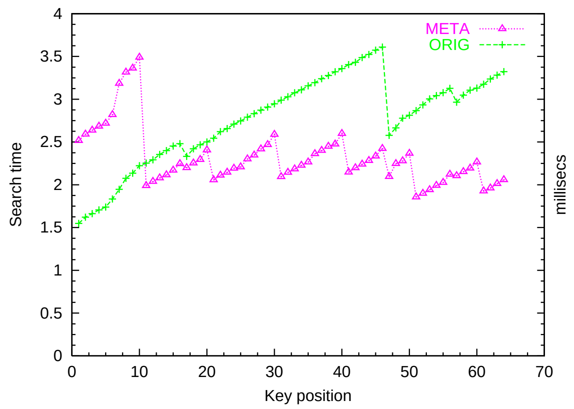

Figure B.4(c): **Type 4** Long keys, high compression ratio, few records

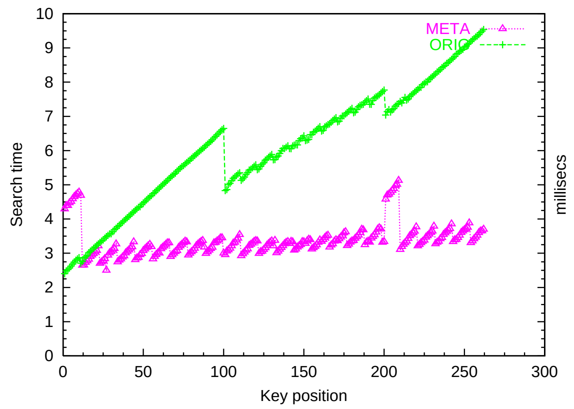

Figure B.5: **Type 5** Long keys, high compression ratio, many records