4 Basic experiments

Abstract

In this chapter we will present two types "clear" experiments: with a text file and a relational database. The reason is that they are wide used for storing semi-structured data.

4.1 Comparison with a text file

The need to compare NL-ArM access method with text files is determined by practical considerations – in many applications the text files are main approach for storing semi-structured data. Text files are kind of files of records which are still in the basis of modern database management systems. Because of this, it is important to determine whether there exists a benefit of structuring data in tries. For instance, if the records are connected to keywords (strings), the information may be recorded by burst tries (i.e. by NL-addressing) rather than as usually with a clear indication of the keywords in the records.

Important consideration in this case is that the sequential reading text file (to find concrete keyword) is very slow operation and every indexed approach will be quicker. Indexed text files are typical for relational data bases and this case will be analyzed further in this chapter.

Here, the goal is to investigate the *size* of the files and *speed* of their generation. In other words, we have to compare writing in a sequential text file with writing in NL-ArM archive.

To compare so different file structures we need to use a common record structure. The structure of text file is a record identified by a keyword, i.e.

<keyword><text information><CR>.

In ArM32 and, respectively, in NL-ArM archives, the information is structured in multi-way multi-layer hash structures and the content of record may be accessed by keyword as path to location of container where the text is stored, i.e.

$$<path> \Rightarrow <text>.$$

In this case the keyword (path) is not stored on the disk.

In both cases the <text> is the same.

We have to make choice for length of the keywords. It may be arbitrary but with fixed length.

Our choice for keywords' length is based on understanding that *the average word length in English is approximately **5.10 characters*** [Sigurd et al, 2004]. In other words, the most frequently used English words are up to 8 characters long (Figure 23).

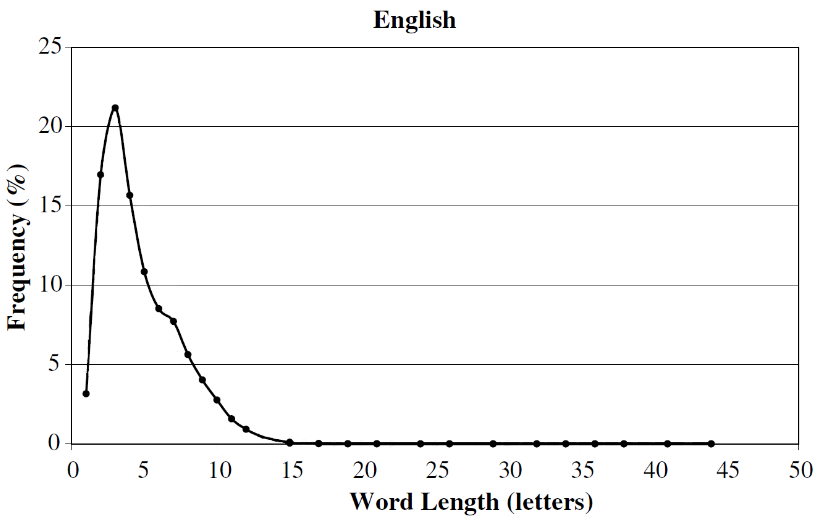

Figure 23. Word Length for English (extracted from [Sigurd et al, 2004])

The average word length in French is approximately 5.13 characters, in Spanish - 5.22 characters, in German - 6.26 characters [Sigurd et al, 2004], in Russian – 5.28 characters [Sharoff, 2001]. This means that length of 8 characters cover these languages, too.

Following this consideration, we chose 8 characters as length of the keywords in our experiments.

The basis of the experiments is a structure of 30 symbols:

– *Keyword* – artificial arbitrary string with maximal length of 8 ASCII symbols. Duplication of symbols is permitted;
– *String* of 22 arbitrary ASCII symbols,

which are stored (written) as follow:

– In the text file – as a structure consisting of keyword and string of length 30 bytes;
– In the NL-ArM archive - the same string (22 characters) stored in the elements with the specified by path of 8 characters (four ASCII symbols in one co-ordinate), i.e. in the archive will be written 22 bytes of the string and 8 symbols of the keyword will be assumed as path (NL-address).

For the experiments, the NL-ArM hash function was programmed to convert ASCII string to space path, i.e. four symbols to form one 32 bit co-ordinate word. The generated keyword string was extended with leading zeroes if it is needed. This way the 8 byte string keyword is converted in two 32 bit hash values and we have to use two layer hash structure. For instance, the string "ABCDEFGH" will be converted in two 32 bit words (ACBD) and (EFGH) where every letter's ASCII code occupies one byte.

The experiments follow the ones made in [Markov, 2006] and were provided on the same computer with a processor Celeron, 3.08 GHz, 512 MB RAM 160 GB HDD and operating system Microsoft Windows XP Professional Ver. 2002, Service Pack 2. The results of the experiments are given in Table 12, Table 13, Table 14, and Table 15. Corresponded graphic visualizations are given at Figure 24, Figure 25, Figure 26, and Figure 27. We received the same results as in [Markov, 2006] which means that the NL-addressing do not add significant complexity to one of the ArM32.

➢ *Comparison of time characteristics*

The Table 12 contains information about six experiments provided with different quantity of records from 2500 up to 100 millions. The table contains data in milliseconds (ms) about time for storing all data sets as well as the average time for storing of one record. The Figure 24 illustrates graphically the same data.

Table 12. Time (ms) for writing in text file and NL-ArM archive

writing in:	text file (ms)		in NL-ArM-archive (ms)	
number of records	all data	one record	all data	one record
2500	16	0.006400	47	0.018800
10000	62	0.006200	140	0.014000
250000	1625	0.006500	3328	0.013312
1000000	6703	0.006703	13359	0.013359
25000000	128719	0.005149	323407	0.012936
100000000	524281	0.005243	1314562	0.013146

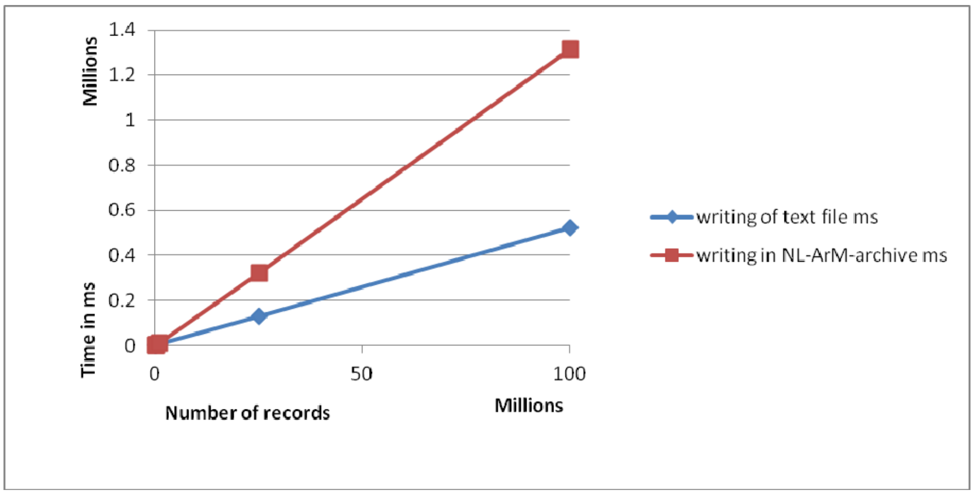

Figure 24. Time in milliseconds for writing in text file and NL-ArM archive

The conclusion is that the time of storing the information has expectable regularities: for great number of elements, writing in a NL-ArM archive became almost twice and half slower than writing in a text file. It is because of building the hash tables of the information in the NL-ArM archive.

Table 13 represents the time correlation between writing in text file and in NL-ArM archive. This correlation is illustrated on Figure 25.

Table 13. Time correlation for writing in text file and NL-ArM archive

number of records	text file	NL-archive
2500	1	2.94
10000	1	2.26
250000	1	2.05
1000000	1	1.99
25000000	1	2.51
100000000	1	2.51

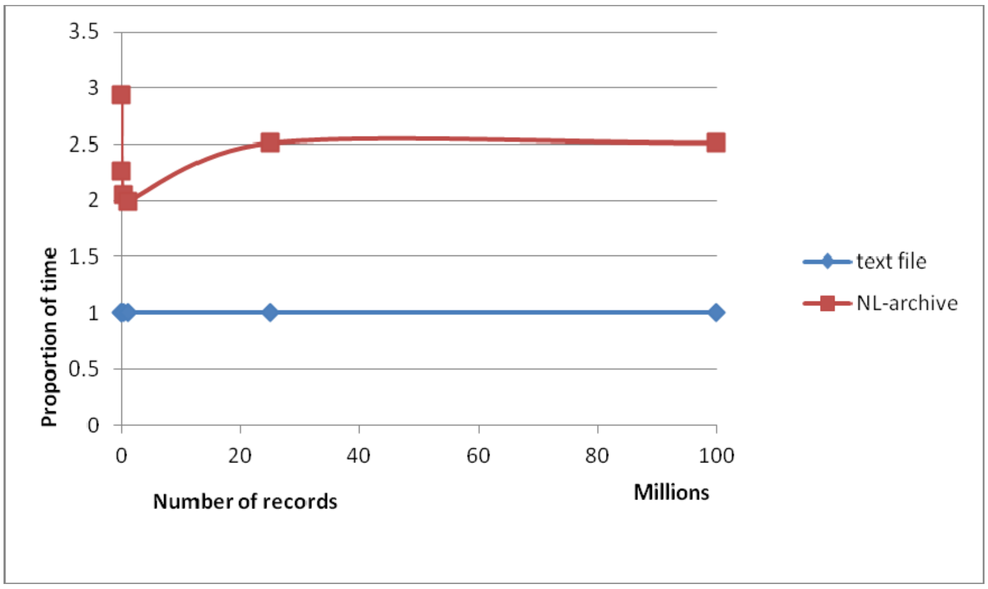

Figure 25. Time correlation between text file and NL-ArM for writing

It is important that the NL-ArM is *constantly* about two and half times slower. This means that the including new records in NL-ArM archive take the same time (about 0.013 ms) per record irrespective of the number of already stored records. For building very large data bases this is very crucial characteristic.

The speed of NL-ArM during storing the first 2500 – 5000 records is about three times slower. This is due to initial creating empty hash tables (information spaces) which takes additional time. This is illustrated on Figure 25 where in the beginning (most left part) of red curve there exists specific irregularity.

➢ *Comparison of size characteristics*

The comparison of file sizes shows that for great number of elements the text file became a little longer than NL-ArM archive. Adding indexes for speeding search in text file will increase the occupied memory because of:

- Duplicating the keywords in the index structures;
- Adding pointers to the records of the text file.

In the same time, after every writing operation, the NL-ArM archive is readies for immediate direct access (by arbitrary keywords) without need of additional indexing and duplication the keywords, which "annihilate" transmuting into paths (NL-addresses).

Table 14 contains data for sizes (in bytes) of the text file and the NL-ArM archive. The corresponded graphical visualization is shown on Figure 26.

The first column of Table 14 contains the number of records. The next two columns contain the size of the text file and the NL-ArM archive after storing the corresponded numbers of records, i.e. 2 500 records occupy 75 000 bytes in the text file and 130 048 bytes in NL-ArM archive.

It is important that in this case (8 bytes keywords) the NL-ArM ***takes about 5.5 bytes*** additional memory for every record to support hash tables' organization.

In other words, if we take the value of the size NL-ArM archive from the last row of the Table 14 (2 740 055 552 bytes) and subtract from it the real length of the stored 100 000 000 records of 22 bytes (2 200 000 000), we will receive the size of internal NL-ArM additional memory for hash indexes, which in this case is 540 055 552 bytes.

Now, dividing it on the number of records, i.e. 540055552/100000000 we receive the average of additional hash indexing memory for each record, i.e. 5.40055552 bytes.

Assuming this value as 5.5 bytes we may say that file with keywords longer than 6 bytes up to 8 bytes each will be stored by NL-ArM in a file with smallest size.

The same result is illustrated on the Figure 26 where the line of the size of the NL-ArM archive is under the line of the size of the corresponded text file (of records).

In the same time we receive one very important quality: *NL-ArM archive permits random direct access to all stored records immediately after writing it without any additional indexing.*

In experiments with text file we used artificially generated strings up to 8 symbols. If we use real English words, at the first glance, it will be more effective to use text file for storing couples (English word, definition), for instance, from a dictionary. If we want only to store the information, this conclusion is correct.

But if we want to use it via random read and/or update, we have to build indexes for quick access to the records which will duplicate the keywords and in addition will contain pointers to locations of keywords and definitions in the file. The external indexing structures used in modern databases need additional memory as well as time for realizing the same functionality. NL-ArM avoids such indexes.

Table 14. Size in bytes of the text file and the NL-ArM archive

number of records	text file	NL-ArM-archive
2 500	75 000	130 048
10 000	300 000	360 448
250 000	7 500 000	6 659 072
1 000 000	30 000 000	26 116 608
25 000 000	750 000 000	640 016 896
100 000 000	3 000 000 000	2 740 055 552

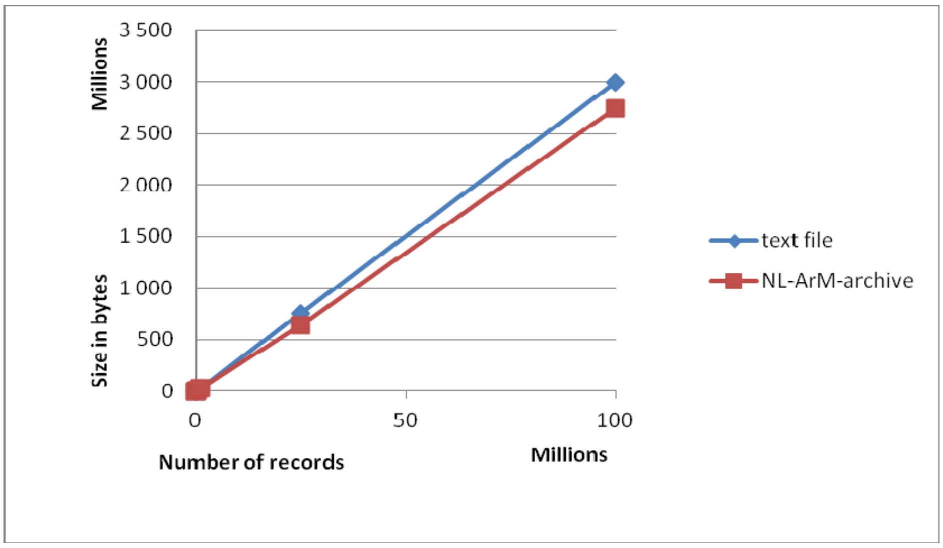

Figure 26. Size in bytes of the text file and the NL-ArM archive

Table 15 represents the relation between sizes of the text file and NL-ArM archive. This correlation is illustrated on Figure 27.

Table 15. Relation between sizes of the text file and NL-ArM archive

number of records	text file	NL-ArM archive
2 500	1	1.7
10 000	1	1.2
250 000	1.13	1
1 000 000	1.15	1
25 000 000	1.17	1
100 000 000	1.09	1

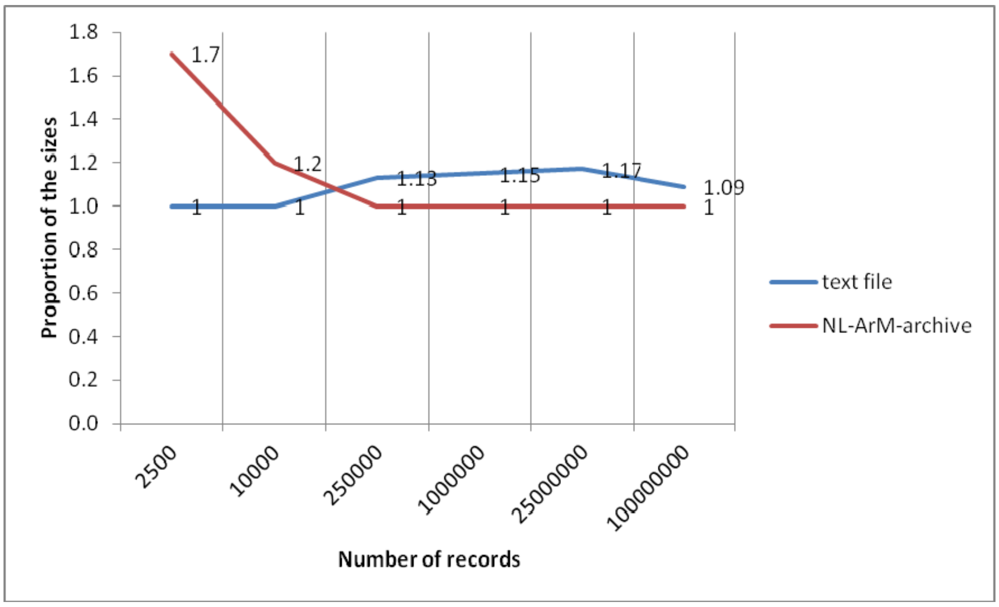

Figure 27. Relation between text file and NL-ArM for writing

The analysis of this relation indicates that, for 8 characters as length of the keywords and small quantity of records, the NL-ArM archive occupies more memory than text file but for the case of very large data the NL-ArM archive is smaller.

The explanation of this regularity is in the specific hash indexing in the NL-ArM archives. In the beginning, large empty hash structures are created which during the storing new records step by step are filled with internal pointers. This way, for great number of records, the hash indexing memory became about 5.5 bytes per record.

It is seen at the Figure 27. The graphics lines which represent sizes of text file and NL-ArM archive are crossed after 250 000 records. After 25 000 000 records the ratio comes to 1.09:1 which has to be examined in further experiments to find possible next cross point.

It is important to underline that these experiments were based on artificial data with fixed length (record of 30 bytes with 8 bytes artificially generated keyword of arbitrary ASCII symbols). If the length of the keywords is variable, the size of NL-ArM archive will be different according of length of the strings of keywords of stored information, i.e. according of number of layers of hash tables (depth of trie).

In sequential storing of records, NL-ArM access method is slower than same operation in text file. For applications where it is important in real time to register incoming information, the text files are preferable than archives with NL-Addressing.

We did not provide experiments to compare NL-access with searching and random reading/updating of records from text file because they are the slowest operations and every indexed approach will be quicker [Connolly & Begg, 2002]. Indexed files are typical for relational data bases and this case will be analyzed below.

4.2 Comparison with a relational database

To provide experiments with a relational database, we have chosen the system "Firebird" because:

- It is a relational database offering many ANSI SQL standard features that runs on Windows, Linux, and a variety of UNIX platforms;
- It offers excellent concurrency, high performance, and powerful query language;
- It has been used in production systems, under a variety of names, since 1981.

The Firebird Project is a commercially independent project of C and C++ programmers, technical advisors and supporters developing and enhancing a multi-platform relational database management system based on the source code released by Inprise Corp (known as Borland Software Corp, too) on 25 July, 2000 [Firebird, 2013].

Database management system "Firebird" is built on the code of Borland InterBase [InterBase, 2012]. It is a complete DBMS capable of managing databases in size from a few kilobytes to scores of gigabytes with excellent performance [Cantu, 2012].

Firebird supports all major operating systems including Windows, Linux, Solaris, MacOS, and there are multiple ways to access its database: native/API, dbExpress drivers, ODBC, OLEDB, Net provider, JDBC, Python module, PHP, Perl, and others. InterBase [InterBase, 2012] and Firebird [Borrie, 2004; ibphoenix, 2012] are widely distributed.

The experiments below were carried out with the version 2.0 of Firebird.

The experiments were provided in two steps: (1) *Writing* and (2) *Reading*.

The information was structured in the same manner as for experiments with sequential text files. The basis of the experiments is a table with two columns:

- *Keyword* - several (8 or 14) digital symbols;
- *String* - 22 arbitrary symbols,

which are stored (written) as follow:

- In the relational database – as a table structure consisting of two columns: keyword and string both encoded in ASCII;
- In the NL-ArM archive - the same string (22 characters) will be stored in the locations specified by keyword as path with two parts (digital ASCII symbols - 4+4 or 7+7, respectively).

For experiments with relational database NL-ArM hash function was programmed to convert ASCII digital strings of the two parts of keyword in two ***32 bit integer*** co-ordinate values, i.e. every part of keyword (digital string) is assumed as integer number and it is converted in a 32 bit integer value. In other words, string of 7 digit symbols is integer value which is less than 2^{32} and may be stored in 32 bits. This way we illustrate the possibility to have different hash functions for different specific cases of information.

To analyze performance of NL-ArM the keys were generated by special algorithm to test different variants of storing. For instance 10 000 rows may be stored via different combinations of the keys' two parts values:

- From 1 x 1 up to 10 x 1000;
- From 1 x 1 up to 100 x 100;
- From 1 x 1 up to 1000 x 10.

This way we have different "rectangular" varying the values of theirs vertex points. For Firebird keys were concatenated strings of the two parts:

- From 00010001 up to 00101000;
- From 00010001 up to 01000100;
- From 00010001 up to 10000010.

Special experiments were provided with "rectangular" which first point is shifted to point 1000000 x 1000000, i.e. for instance:

- From 1000000 x 1000000 up to 1000010 x 1010000;
- From 1000000 x 1000000 up to 1010000 x 1000010;
- From 1000000 x 1000000 up to 1001000 x 1001000.

For Firebird these keys looked as:

- From 10000001000000 up to 10000101010000;
- From 10000001000000 up to 10100001000010;
- From 10000001000000 up to 10010001001000.

➢ *Comparison of writing time characteristics*

In Table 16, several results from three variants of writing experiments are presented. The variants are based on a tables with two columns (key, string) and 10 000, 100 000, and 1 000 000 rows respectively.

The two columns of Table 16, marked as (X) and (Y), contain values of the two parts of the keywords which in the same time are coordinates of initial point of NL-ArM experimental rectangular. Its diagonal point is given by the corresponded offsets (ΔX) and (ΔY).

In the next column, the quantity of rows (ΔX∗ΔY) is given, respectively – 10 000, 100 000, and 1 000 000.

The writing time has been measured in milliseconds (ms) and the results are presented in the next two columns for Firebird and NL-ArM respectively.

In the last column, the ratio between Fireburd and NL-ArM for writing time is given. Graphical visualization of the ratio is on Figure 28.

Table 16. Writing time comparison of Firebird and NL-ArM

row No.:	initial values of co-ordinates		size of intervals of co-ordinates		Number of cells	writing time (ms)		ratio
						Firebird	NL-ArM	Firebird : NL-ArM
	(X)	(Y)	(ΔX)	(ΔY)	(ΔX*ΔY)	(ms)	(ms)	
1	1	1	10	1000	10000	21297	141	*151 : 1*
2	1	1	100	100	10000	14094	140	*100 : 1*
3	1	1	1000	10	10000	15438	156	*98 : 1*
4	1	1	10	10000	100000	160094	1563	*102 : 1*
5	1	1	100	1000	100000	145719	14062	*103 : 1*
6	1	1	1000	100	100000	141547	1265	*112 : 1*
7	1	1	10000	10	100000	155578	1719	*90 : 1*
8	1	1	1	100000	100000	292625	2907	*100 : 1*
9	1	1	100000	1	100000	289406	9390	*30 : 1*
10	1000000	1000000	10000	10	100000	156656	2109	*74 : 1*
11	1000000	1000000	10	10000	100000	162000	1672	*96 : 1*
12	1	1	10	100000	1000000	1740234	16640	*104 : 1*
13	1	1	100	10000	1000000	1591688	15187	*104 : 1*
14	1	1	1000	1000	1000000	1589734	14656	*108 : 1*
15	1	1	10000	100	1000000	1583906	13250	*119 : 1*
16	1	1	100000	10	1000000	1738047	17875	*97 : 1*
17	1000000	1000000	1000	1000	1000000	1778953	15750	*112 : 1*
				Total:	6830000	11577016	128482	*90.1 : 1*

The average of writing time data in milliseconds for the three groups (10 000, 100 000, and 1 000 000) are presented in Table 17.

In the last column, the average ratio of Firebird and NL-ArM is given. This relation is illustrated graphically on Figure 29.

Table 17. Average in milliseconds of writing time data

Number of cells	average writing time (ms) Firebird	ArM	ratio Firebird : NL-ArM
10000	16943.000	145.670	116.330 : 1
100000	187953.125	4335.875	88.375 : 1
1000000	1670427.000	15559.670	107.330 : 1

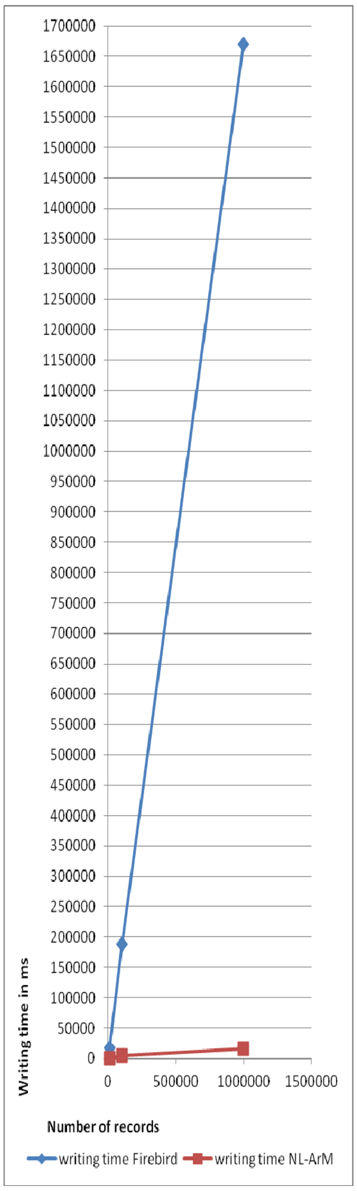

Figure 28. Time in miliseconds for writing by Firebird and NL-ArM

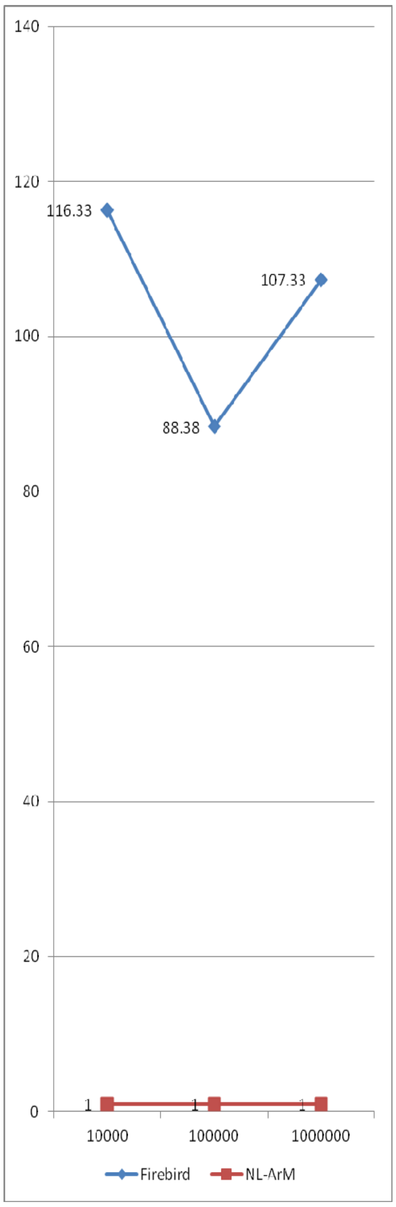

Figure 29. Time relation for writing by Firebird and NL-ArM

The results are expectable.

During initialization, Firebird take some additional time, than for rest records the consuming time has logarithmic regularity.

NL-ArM has no initialization procedures and has linear regularity for writing of all records (Figure 28). This relation may be seen at Figure 29, and more easily at Figure 30 where axes are logarithmic.

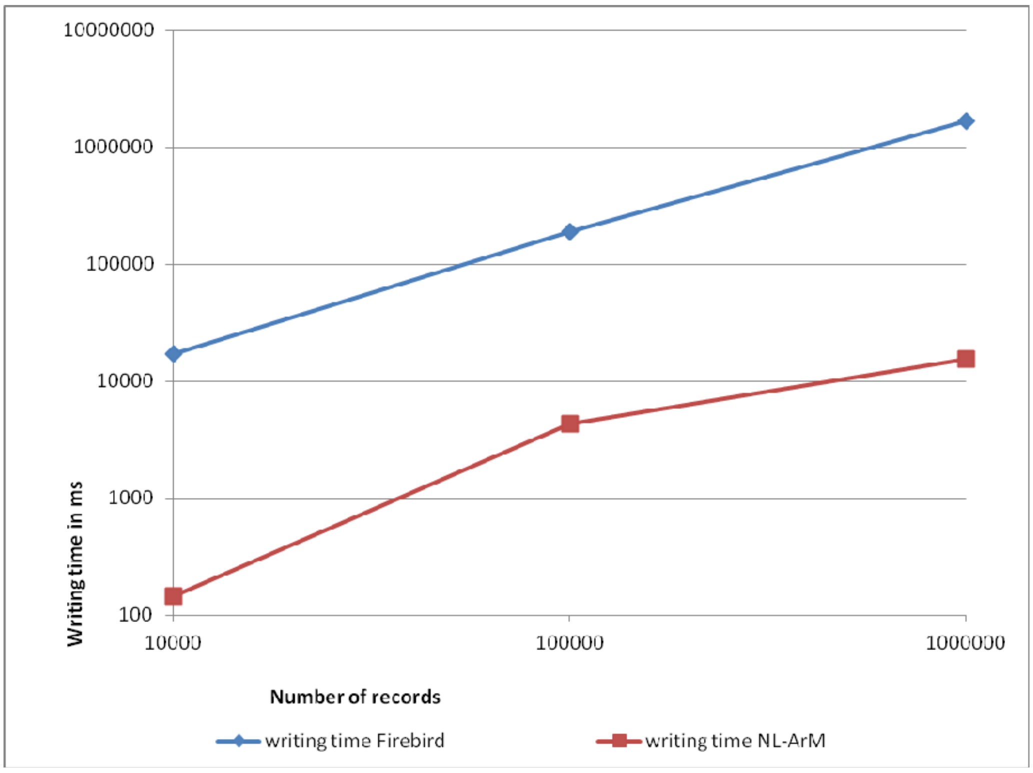

Figure 30. Logarithmic time relation for writing

In the same time we have to comment some disadvantages of NL-ArM in relation to Firebird. The keys used in the experiments were strings for the Firebird (relational) variants and two separate 32 bit (4-byte) co-ordinates for NL-ArM.

In relational model all keys have same influence on the writing time – they are written in the plain file by the same manner (as parts of records) and extend the balanced index in one or other its section which takes practically same time.

In NL-ArM the different values of co-ordinates cause various archive structures which take corresponded time for combinations of values. Practically, NL-ArM creates hyper-matrix and large empty zones need additional resources – time and disk space, which are not so great due to smart internal index organization but really exists.

Comparison of Firebird and NL-ArM writing times for the case of large empty zones in the matrix is given in Table 18. It is a sub-table from Table 16 and numbers of rows are the same.

Table 18. Comparison of Firebird and NL-ArM for the case of large empty zones in the matrix

row No.:	initial values of co-ordinates		size of intervals of co-ordinates		Number of cells	writing time (ms)		ratio
						Firebird	NL-ArM	Firebird : NL-ArM
	(X)	(Y)	(ΔX)	(ΔY)	(ΔX*ΔY)	(ms)	(ms)	
7	1	1	10000	10	100000	155578	1719	*90 : 1*
4	1	1	10	10000	100000	160094	1563	*102 : 1*
10	1000000	1000000	10000	10	100000	156656	2109	*74 : 1*
11	1000000	1000000	10	10000	100000	162000	1672	*96 : 1*

The influence of storing types is presented in Table 19. Visualization of ratios is shown on Figure 31.

For NL-ArM we have two cases:

1. Row oriented NL-ArM storing.
2. Column oriented NL-ArM storing.

Firebird is not so sensitive to the NL-ArM row and column oriented cases because these cases are only switched parts of keyword and the key length is the same. For NL-ArM the second case is more suitable because of column oriented hierarchical storing. Let remember, NL-ArM has multi-layer structure of perfect hash tables with 2^{32} entries. In addition, the NL-ArM hash tables have balanced internal indexes specially adapted for storing large data sets. Because of this, they are not so effective for small values of co-ordinates. For instance, the better ratio for the rectangle with starting point 1000000x1000000 is just due to special internal indexing of NL-ArM which is adapted to great co-ordinate values.

This conclusion is seen on Figure 31 where the Firebird ration line is practically horizontal but NL-ArM ratio line descends.

Table 19. Influence of storing types

case	**type of ratio**	**Firebird**	**NL-ArM**
1.	column to row oriented case (ratio of row 4 to row 7)	**1.029**	**0.909**
2.	column to row oriented case for the rectangle with starting point 1000000x1000000 (ratio of row 11 to row 10)	**1.034**	**0.793**

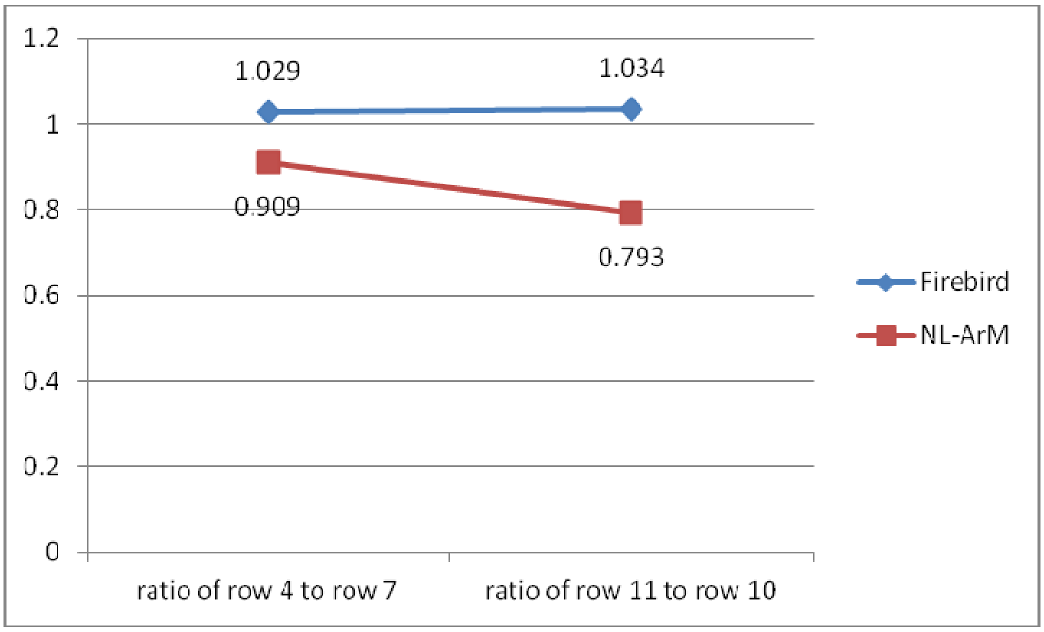

Figure 31. Ratios for NL-ArM row and column oriented writing

The influence of offset (1000000) is presented in Table 20. Visualization of ratios is shown on Figure 32.

Again, for NL-ArM we have two cases:

1. Row oriented NL-ArM storing.
2. Column oriented NL-ArM storing.

As in previous, Firebird is not so sensitive to the NL-ArM row and column oriented cases because they are only switched parts of keyword and the key length is the same. For NL-ArM the second case is more suitable because of column oriented hierarchical storing.

This conclusion is seen on Figure 32 where the Firebird ratio line is practically horizontal but NL-ArM ratio line descends.

Table 20. Influence of the offset from 1 to 1000000

case	**type of ratio**	**Firebird ratio**	**NL-ArM ratio**
1.	row oriented storing (ratio of row 10 to row 7)	**1.007**	**1.227**
2.	column oriented storing (ratio of row 11 to row 4)	**1.012**	**1.070**

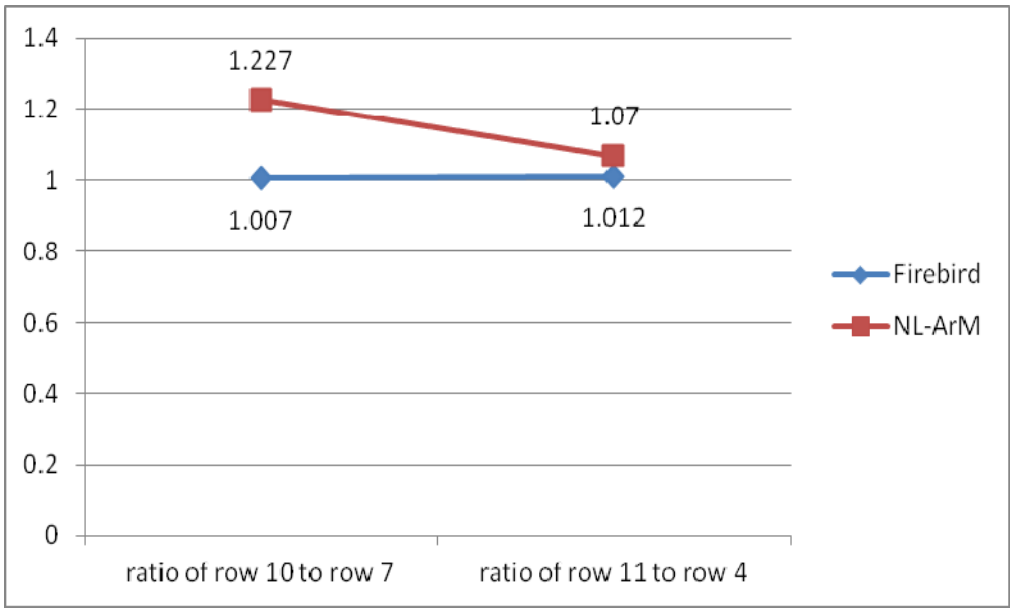

Figure 32. Ratios for the offset from 1 to 1000000

Concluding this part of experiments we have to note that the relations in Table 16 show that in writing experiments, regarding NL-ArM, Firebird is on average ***90.1 times slower.*** This result is due to two reasons. The first is that balanced indexes of Firebird need reconstruction for including of every new keyword. This is time consuming process. The second reason is the speed of updating NL-ArM hash tables which do not need recompilation after including new information.

Due to specific of realization, for small values of co-ordinates NL-ArM is not as effective as for the great ones.

Nevertheless, NL-ArM is always many times faster than Firebird.

If we need direct access to large dynamic data sets (via NL-path), than more convenient are hash based tools like NL-ArM. For instance, such cases are large ontologies and RDF-graphs.

➢ *Comparison of reading time characteristics*

The experimental data for reading time characteristics are given in Table 21, which has similar format as one for the writing time characteristics.

The experiments were done on the base of 6830000 queries with 8 or 14 byte numbers as keywords (to model two-dimensional 4-bytes binary co-ordinates), stored in a text file in order to maintain equivalence of Firebird with NL-ArM.

The columns of Table 21, marked as (X) and (Y), contain the co-ordinates of the initial point of the experimental rectangular, i.e. initial values of the first and second parts of the keywords.

The diagonal point is given by the corresponded offsets (ΔX) and (ΔY). In the next column the quantity of read elements is given, respectively – 10 000, 100 000, and 1 000 000.

The reading time has been measured in milliseconds (ms) and the results are presented in the next two columns. In the last column, the ratio between Fireburd and NL-ArM for reading time is given.

Table 21. Reading time comparison of Firebird and NL-ArM

	initial values of co-ordinates		size of intervals of co-ordinates		Number of elements generated in the specified interval	reading time for 10,000 elements		ratio of Firebird to ArM
						Firebird	ArM	
No.:	(X)	(Y)	(ΔX)	(ΔY)	(ΔX*ΔY)	(ms)	(ms)	
1.	1	1	10	1000	10000	532	156	*3:1*
2.	1	1	100	100	10000	406	172	*2:1*
3.	1	1	1000	10	10000	563	187	*3:1*
4.	1	1	10	10000	100000	1265	172	*7:1*
5.	1	1	100	1000	100000	234	188	*1:1*
6.	1	1	1000	100	100000	672	110	*6:1*
7.	1	1	10000	10	100000	2297	203	*11:1*
8.	1	1	1	100000	100000	13406	125	*107:1*
9.	1	1	100000	1	100000	15953	422	*37:1*
10.	1000000	1000000	10000	10	100000	1906	32	*59:1*
11.	1000000	1000000	10	10000	100000	1265	31	*40:1*
12.	1	1	10	100000	1000000	13547	188	*72:1*
13.	1	1	100	10000	1000000	2562	156	*16:1*
14.	1	1	1000	1000	1000000	1359	125	*10:1*
15.	1	1	10000	100	1000000	3719	250	*14:1*
16.	1	1	100000	10	1000000	21625	204	*106:1*
17.	1000000	1000000	1000	1000	1000000	750	32	*23:1*
				Total:	6830000	82061	2753	*29.8:1*

The average of *reading time* data in milliseconds for the three groups (10 000, 100 000, and 1 000 000) are presented in Table 22. This is illustrated on Figure 33.

Table 22. Average in milliseconds (ms) of reading time data

number of records	reading time		ratio	
	Firebird	NL-ArM	Firebird	NL-ArM
10000	500.33	171.67	*2.67 : 1*	
100000	4624.75	160.38	*33.5 : 1*	
1000000	7260.33	159.17	*40.17 : 1*	

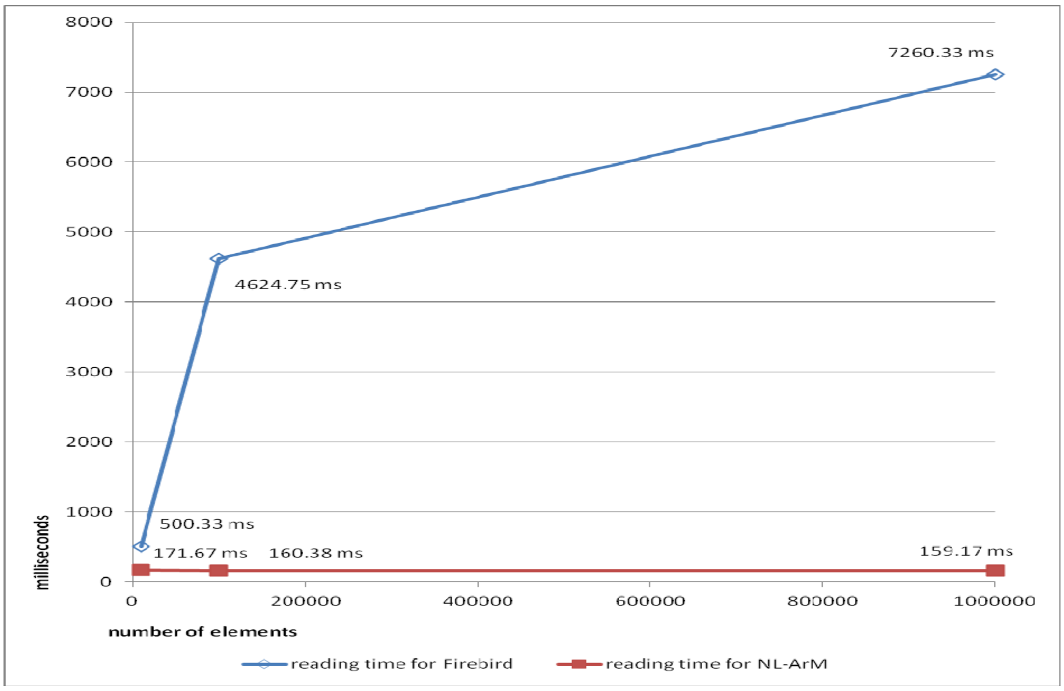

Figure 33. Time in milliseconds (ms) for reading by Firebird and NL-ArM

In the last column of Table 22, the *average ratio* of Firebird and NL-ArM for reading time is given. This relation is illustrated graphically on Figure 34.

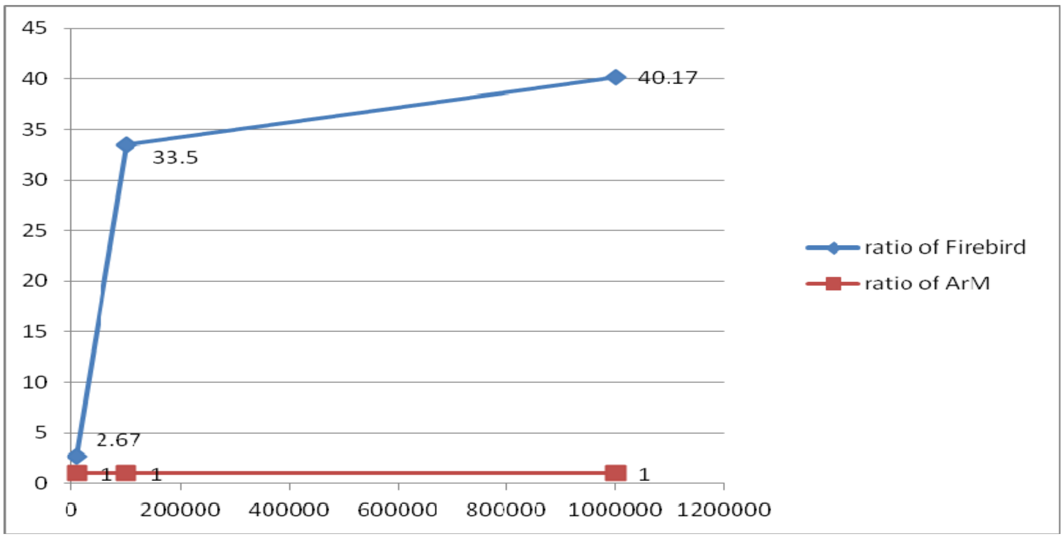

Figure 34. Time relation for reading by Firebird and NL-ArM

Concluding this part of experiments we have to note that the relations in Table 21 show that in reading experiments, regarding NL-ArM, Firebird is on average ***29.8 times slower.***

This result is due to the speed of access in NL-ArM hash tables which do not need search operations.

Again, note that if we need direct access to large dynamic data sets (via NL-path), than more convenient are hash based tools like NL-ArM. For instance, such cases are large ontologies and RDF-graphs.

➢ *Conclusion of chapter 4*

In this chapter two main types of basic experiments were presented. NL-ArM has been compared with (1) sequential text file of records and (2) relational database management system Firebird.

The need to compare NL-ArM access method with text files was determined by practical considerations – in many applications the text files are main approach for storing semi-structured data. To investigate the size of files and speed of their generation we compared writing in a sequential text file and in a NL-ArM archive.

For 8 characters as length of the keywords and small quantity of records, the NL-ArM archive occupies more memory than text file but for the case of very large data the NL-ArM archive is smaller. It is important to underline that these experiments were based on artificial data with fixed length (record of 30 bytes with 8 bytes artificially generated keyword of arbitrary ASCII symbols). If the length of the keywords is variable, the size of NL-ArM archive will be different according of length of the strings of keywords of stored information, i.e. according of number of layers of hash tables (depth of trie).

In sequential storing of records, NL-ArM access method is slower than same operation in text file. For applications where it is important in real time to register incoming information, the text files are preferable than archives with NL-Addressing.

To provide experiments with a relational database, we have chosen the system "Firebird". It should be noted that Firebird and NL-ArM have fundamentally different physical organization of data and the tests cover small field of features of both systems.

We did not compare the sizes of files of NL-ArM and Firebird because of difference of keywords – symbols for Firebird and integer values for NL-ArM.

In writing experiments, regarding NL-ArM, Firebird is on average ***90.1 times slower.*** *This result is due to two reasons. The first is that balanced indexes of Firebird need reconstruction for including of every new keyword. This is time consuming process. The second reason is the speed of updating NL-ArM hash tables which do not need recompilation after including new information. Due to specific of realization, for small values of co-ordinates NL-ArM is not as effective as for the great ones.*

In reading experiments, regarding NL-ArM, Firebird is on average ***29.8 times slower.*** *This result is due to the speed of access in NL-ArM hash tables which do not need search operations.*

If we need direct access to large dynamic data sets (via NL-path), than more convenient are hash based tools like NL-ArM. For instance, such cases are large ontologies and RDF-graphs.