Maciej PENAR[1]

PERFORMANCE ANALYSIS OF WRITE OPERATIONS IN IDENTITY AND UUID ORDERED TABLES

Design of the database includes the decision about the physical storage. This is often overlooked as 1) this cannot be expressed in standard SQL and in result each Database Systems have their own way to specify the physical storage and 2) the decision is often made implicitly. This is dangerous situation as many of the databases use B+ trees as table implementation which stores the data physically sorted by some ordering attribute. The choice of the ordering attribute largely affects read and write operations. Commonly, IDENTITY/AUTO_INCREMENT constraint are being chosen as ordering attributes, due to their easy usage and monotonic nature. In some cases ordering tables by the attributes whose values are drawn from uniform distribution leads to better performance in terms of Transactions-Per-Second. Such cases includes situation when data does fit entirely in-memory or when we can limit the set of physical pages being accessed. In the end, however, We cannot entirely say that either monotonic or random attributes are superior. Both have their pros and cons. In this article We present (1) short description of the data structures in contemporary Database Systems, (2) the advantages and the disadvantages of the two common types which are used as the clustering attributes: GUID and IDENTITY, (3) performance analysis of write operation which compare both data types using B+ tree as primary storage and (4) evaluate the efficiency of these bulk load operation using heap files and B+ trees.

Keywords: database design, logical model, clustering, heap files, B + tree, UUID, GUID, IDENTITY, sequences, insert performance, batch loading

[1] Corresponding author: Maciej Penar, Rzeszów University of Technology, The Faculty of Electrical and Computer Engineering, Aleja Powstańców Warszawy 12, 35-959 Rzeszów; mpenar@kia.prz.edu.pl, https://orcid.org/0000-0002-4481-807X

1. Introduction

Few decisions should be made while designing the logical model of the database (DB). Firstly, DB designer should try to fulfil functional requirements, usually by creating tables with the appropriate data types. Secondly, constraints are put on the created schema as a result of normalization (i.e. FOREIGN KEY constraints) or by incorporating some business logic inside the database project (i.e. CHECK/UNIQUE constraints) [1]. Finally a good designer should consider context of the usage - how data is written, updated and read. At this stage indexes and partitioning schemes are created and the physical structure should be chosen – one of the most popular choices is the B + tree as physical implementation of table.

In this article we analyse the performance of ordered and unordered attributes as clustering key in B + tree. We use two most popular data types: IDENTITY and GUIDv4 [2]. Such analysis are regularly carried out on unofficial blogs and are subject of continuous discussion – usually ending up in overwhelming criticism GUID [3] [4] as they underperform in certain conditions. However, few articles happens to state otherwise [5] [6]. Unfortunately existing articles do not use scientific methods to evaluate performance. Also, it is common that DBs are compared to NoSQL solutions [7] [8] without stating which structure has been used as a storage and how the data was sorted (if it was). In this article we will consider the functional advantages and disadvantages of these types and will present the results of our experiment which assesses the effectiveness.

The article is organized as follows: section 2 provides a brief description of how modern database writes the data. Section 3 describes the advantages and disadvantages of GUID and IDENTITY. Section 4 presents the results of proposed experiments. Section 5 summarizes the article and discusses further research.

2. Storing the data

Typically while executing CREATE TABLE command, DB decides which structure should be used to as a table. DB organizes the data and metadata in blocks of bytes called *pages*. In this section we will give a short description of the data structures commonly used in DBs and we'll discuss the activities that Database Management System (DBMS) performs during INSERT command. We will finish the section with comment about transaction log. Two structures, which are commonly used in DBMS are *heap files* and a *B + tree*s [9].

2.1 Heap files

Heap files (also known as Sequential files) are DBMS equivalent of linked lists. Pages of heap files are linked together using pointer which are stored in a special sector of the page called *header*. This structure has a relatively low cost of INSERT command as it only requires appending it in the free space of the last page (see fig. 1). However, if the heap file is not indexed each SELECT statement requires scanning all blocks. This data structure is often used in Data Warehouses as it provides:

- Support for bulk operations as tables can be copied page-by-page. Afterwards the pointers are updated.
- Daily update of the reports may require scanning whole dataset. Therefore, the default method of accesing data in heap files are not drawback.

In order to index such heap files, one need a method that is used to identify the record regardless of the physical location on disk. DB must implement method to determine the logical ID of the record within the file. Usually component exists in DB which provides such identifier and in some cases it can be a bottleneck when many concurrent INSERTS are performed.

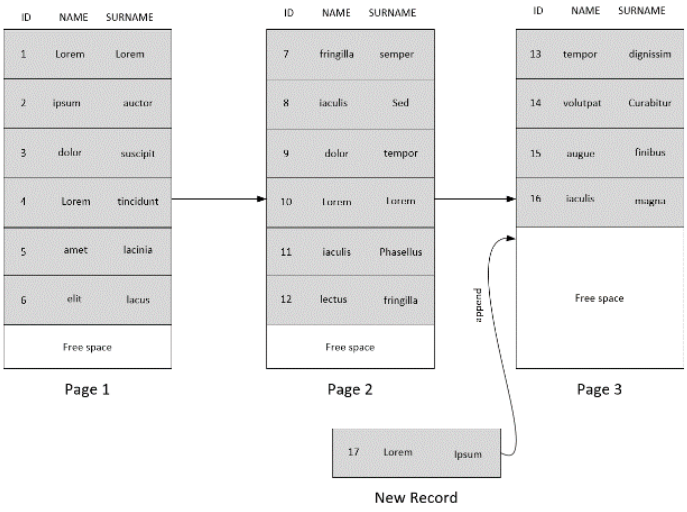

Figure 1. Allocating record in heap file with 3 pages

2.2 B+ tree

Also known as *Clustered Index* [10]. This structure requires an order $\leq_A$ over some attribute A (or a list of attributes). In B + tree there are two kind of pages:

- internal nodes – which contains values of attributes A and pointers to (1) either other internal nodes on the lower level of the tree (2) or to the leafs which contains raw data. Each node M contains n pointers p and $n-1$ keys $k \in A$. In each node, any pointer p_i leads to node N' so that: $N'(p_i) \coloneqq \{x \in N' \mid k_{i-1} < x.A \leq k_i\}$, for $1 < i < n$. In case when $i = 0$ or $i = n$ left and right side of inequality is omitted. This is shown in the figure 2.

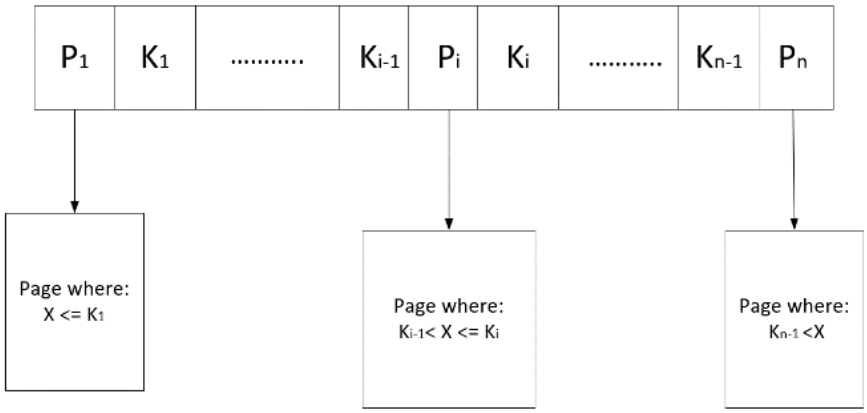

Figure 2. Internal node of B+ tree

- leaves (data pages) – which stores the raw data. Leaves layer is connected with pointers in a similar fashion as in the heap files. This feature and the fact that data is ordered by $\leq_{A_1 \ldots A_n}$ enables effective range queries given the value of A.

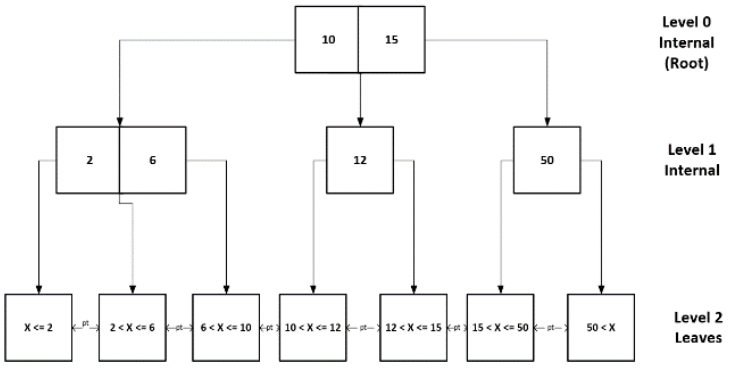

Figure 3. Example of B+ tree(here the data pages contains only information about ranges of the values)

SELECT statements which have different attributes than A in the WHERE claus requires a scan on the leaves. Also the INSERT to a B + tree is more complex than INSERT to the heap file. In B+ trees it requires finding the appropriate block so the order $\leq_{A_1 \ldots A_n}$ is preserved. If the leaf cannot hold any more data, it may require 1) splitting the page in half and 2) updating the internal node level above (if required). Example of B + tree is in the figure 3.

B+ tree implementations in DBMS have subtle details which influences the performance. Often the data in leaves is unsorted to minimize the requirement to reorganize the leaves after each INSERT. The order of the data is established based on the special *offset array* – which contains the offsets of the rows in the data page.

2.3 Additional comment

To minimize the expensive disk IO, DB store data in main memory - buffer pools. The size of the buffer pool is usually configurable. Whether or not a block of data is in the buffer is important not only for read but also for the write operations. In particular this is crucial for B + trees because the INSERT transaction must find the leaf where the new record should be put – one may think about this as implicit SELECT.

The author want to note that Durability of the transactions is achieved by logging the transactions. Write-Ahead logging (WAL) [11] is commonly used as logging scheme. In WAL transaction are firstly written to the log, then the transaction is executed. DB are properly utilized only when the transaction log becomes the bottleneck – therefore observing waits on transaction log can be indicator if some DB operations can be optimized.

3. Attributes

In this section, we describe two data types which are commonly used as ordering attributes of B + trees. We provide functional advantages and disadvantages of both types of data.

3.1 IDENTITY/SEQUENCE

This type is implemented as a 4-byte or 8-byte integer. Main idea is to provide the way to generate monotonic values. This means that DB allocates special counter for each column of this data type. It is incremented whenever the counter is accessed.

Extension of this concept is known as Database Sequences which are special objects which enable the precise control over the generated values. Often, when DB provides the Sequences, they wrap the IDENTITY. Also, other methods exist to

generate the sequential values i.e. SEQUENTIALID (which are GUIDv1/GUIDv2) or timestamp.

When monotonic value is used, then the “last page problem” occurs [1] – a large number of concurrent INSERT transactions may require accessing and modifying the last page (see Fig. 4). Which leads to lock contention there, as every transaction require exclusive lock on the page to perform actual transaction. Due to this, one can observe a significant drop of the database throughput. On the other hand, the pages where the payload of the INSERT should be put can be easily predicted – which results in minimal number of IO operations. Additionally, such pages are rarely deleted from the buffer pools.

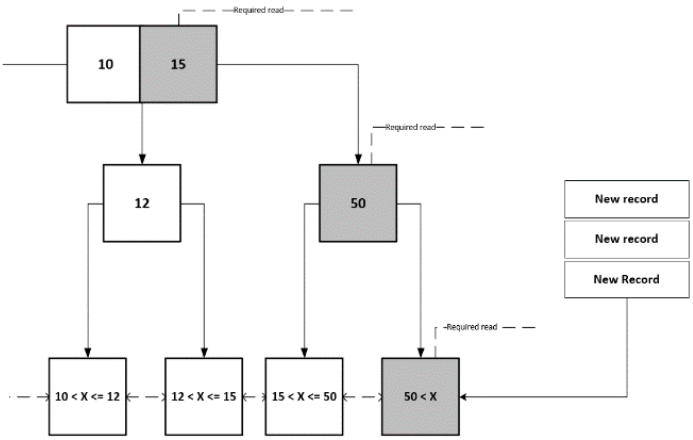

Figure 4. "Last Page Problem" when using IDENTITY. The path that is read by every transaction is highlighted

3.2 GUID

Globally Unique Identifier (GUID) is 16 byte integer. There are few subtypes of GUID which differs in way of generating the values. In this section we consider the GUIDv4 which values are drawn from the uniform distribution.

When using the GUID, "The Problem of the last page" disappears as random page is chosen for modification, thus reducing the probability of lock contention. As the values are random – in a distributed environment the clients themselves can generate them which eases the usage in the distributed environment. Unfortunately, when synchronizing the multiple datasets, some policy may be required when the duplicates are found.

As long as the whole dataset can be stored in main memory, the random types can be successfully used as ordering attributes. However, when the size of the table exceeds the size of the buffer pool, it is likely that the page that is affected by the transaction is not present in the cache. This means the additional disk IO which generally requires more time than waiting on a lock as stated in "The Last Page Problem". Table 1 presents a functional comparison of the data types.

Table 1. Functional comparison of IDENTITY and GUI

	IDENTITY	**GUIDv4**
Pros	• Last page is often in the cache, which reduces disk IO • Often, it is significantly smaller than GUID • Capabilities are extended with SEQUENCE	• Identifies entity in distributed DBs • Prevents the lock contention at the last page
Cons	• Generated by DB • Does not identify entities in distributed DBs • Introduces "Last Page Problem" • Does not prevent fragmentation of the pages • In distributed DBs the two-way synchronization requires synchronous flow	• Drops the throughput of the DB when the dataset does not fit in the memory • Often, it is significantly larger than IDENTITY • In distributed DBs the synchronization may require policy of identifying duplicate entries

4. Evaluation

The experiments were carried out on two nodes: the first contains DB, the second simulated clients performing concurrent transactions. DB on which the test was carried out was Microsoft SQL Server 2014 – Standard Edition installed on the Windows 10 Pro 64-bit. DB node had Intel (R) Core (TM) i7-6700 CPU @ 3.40 GHz, processor with 8 GB RAM and two hard drives. In order to examine the write performance to tables with random and monotonic clustering the DB was configured as in Table 2. The Disks were checked with winsat tool – the results are presented in Table 3.

Table 2. DB configuration parameters

Parameter	Max. log size	Initial log size	Max. DB size	Initial DB size	Buffer Pool size	Page size	DB threads
Value	10 GB	2 GB	20 GB	2 GB	4 GB	8 kB	4

Table 3. Referential measurement of disk performance using winsat

Disk/Type	Rand 16.0 Read	Seq 64.0 Read	Seq 64.0 Write	Avg. Seq Read	Max. latency	Avg. Rand Read
Disk 1 SSD	164.52 MB/s	447.71 MB/s	357.31 MB/s	0.169 ms	64.466 ms	0.186 ms
Disk 2 HDD	1.47 MB/s	109.67 MB/s	114.47 MB/s	5.334 ms	74.404 ms	12.182 ms

The following experiments were proposed to evaluate the performance:

- Experiment 1 – INSERT INTO performance evaluation when multiple several parallel connections are opened, assuming that the volume of data is in the buffer pool.
- Experiment 2 – multi-iteration INSERT INTO performance evaluation when multiple several parallel connections are opened. After each iteration, the data is preserved in DB. Time is measured for each iteration. At some iteration the data volume will exceed the capacity of the buffer.
- Experiment 3 - in which we evaluate the effectiveness of the batch load in different structures.

Below in the dedicated sections will be thoroughly discussed and the results of experiments.

4.1 Experiment 1

The first experiment examined the performance of write operations when many concurrent connections were executing the stored procedure. Pseudocode for the procedure is:

```
FOR i FROM 0 TO X
    BEGIN TRANSACTION
            INSERT INTO TABLE DEFAULT VALUES
    COMMIT
END FOR
```

Its workload simulates OLTP environment – where transactions have “point”/”by-id” flavour (insert, delete, update or read of a single row). In the experiment we defined the following variables:

- Number of parallel clients connected to the database
  $C \coloneqq \{1, 10, 25, 50, 75, 100\}$

- Number of rows inserted into the table – among all the connections
  $R \coloneqq \{0{,}5, 1, 2, 4\} * 10^6$
- Width of the row (in bytes)
  $W \coloneqq \{50, 104, 254, 504\}$

All tests were repeated 5 times and mean time and standard deviation was calculated (in seconds) – measurements are presented in table 4.

In the setup when a single connection was performing inserts, we cannot indicate which solution is better – mean difference is relatively small even for the largest volume ($W = 504$, $R = 4 * 10^6$), the relative difference equals to 8% in favour of the IDENTITY.

The situation changes when records are inserted in parallel. In each case this leads to significant reduction of time (37% of the base time for GUID, 39% for IDENTITY). When using GUID and 10 connections in every but two cases we observe faster INSERT with relative difference at level < 10%.

With 25 and more connections difference between the GUID and IDENTITY significantly differs. In case of smallest volume ($W = 50$, $R = 0{,}5 * 10^6$) GUID organized table finishes the task 1.48 to 2.24 times faster than IDENTITY. Similarly, the largest volume of data ($W = 504$, $R = 4 * 10^6$) can be inserted 1.24 to 1.83 times faster. The relative acceleration using 100 connections is presented in figure 5.

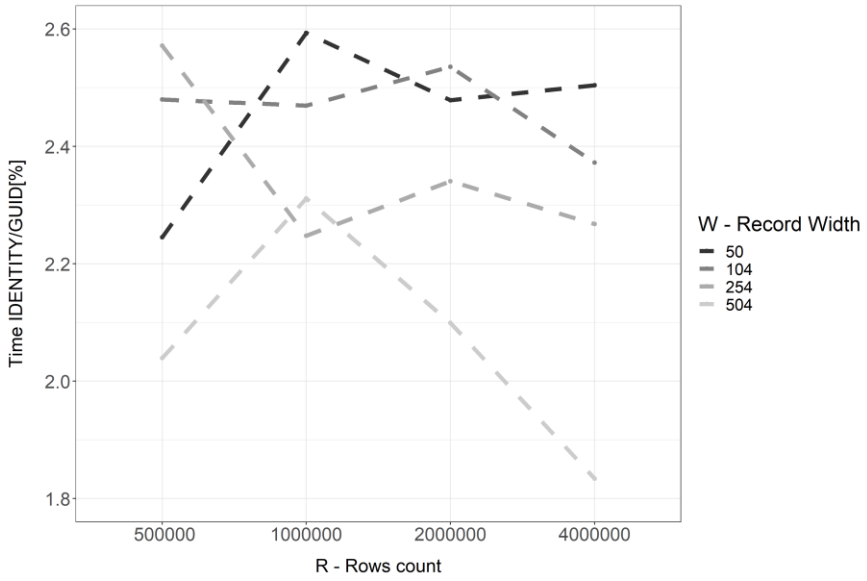

Figure 5: Relative comparison of mean Times using 100 parallel connections

Table 4. Comparison of mean time (in seconds) of INSERT operation on disk 1 for experiment 1. Standard deviation is shown in square brackets. Grey color indicates lower execution time

Inserted rows [R]	GUID				IDENTITY			
	Row width[W]				Row width[W]			
	50	104	254	504	50	104	254	504
	Connection: 1							
500000	36 [1.1]	35.3 [0.6]	38.3 [1.9]	36.8 [1.2]	34.5 [0.4]	36.4 [0.6]	38.9 [0.8]	35.9 [0.9]
1000000	69.1 [1.3]	72.8 [1.7]	76.5 [2.5]	73.9 [1.4]	69.9 [0.5]	71.9 [1.7]	77.6 [3.4]	71.2 [1.7]
2000000	138.6 [1.2]	149.3 [3.5]	147.2 [5.5]	150.5 [2.4]	140.3 [1.6]	147.3 [6.3]	144.9 [3.3]	142.4 [3.5]
4000000	290.8 [6.7]	293.6 [3.6]	293.8 [5.2]	312.7 [5.9]	288.7 [8.3]	291.3 [7.7]	286.7 [6.5]	288.1 [9.8]
	Connections: 10							
500000	10.5 [0.3]	10.8 [0.4]	12.2 [1.7]	12.9 [1]	12.4 [0.3]	12.7 [0.5]	13.2 [0.8]	14.5 [1.4]
1000000	21.5 [1.2]	21.8 [1.3]	37.5 [30.4]	26.9 [1.5]	24.9 [0.4]	24.9 [0.4]	26.5 [0.8]	28.4 [0.6]
2000000	41.2 [1.5]	44.2 [2.2]	47.1 [1.4]	53.1 [1.7]	49.5 [0.8]	51.1 [1]	52.5 [1]	57.1 [2.2]
4000000	84.1 [3.4]	91.1 [4.2]	96.4 [1.9]	117 [4.5]	99.7 [1.3]	101.5 [2.8]	105.3 [1.5]	114.1 [3.4]
	Connections: 25							
500000	6.3 [0.5]	6.5 [0.3]	7.7 [0.7]	8.7 [1]	9.3 [1]	9.8 [0.5]	11.1 [1.1]	13.2 [0.9]
1000000	12.9 [0.5]	13 [0.5]	15.3 [0.9]	18.2 [0.4]	18 [0.6]	19.2 [0.3]	21.3 [1.4]	25.8 [0.8]
2000000	33 [17.8]	26.8 [1.2]	30.4 [1.7]	37 [1.5]	36.2 [1.1]	37.8 [1.3]	42.6 [1.1]	50.9 [1.9]
4000000	50.4 [1.4]	53.2 [2]	64.3 [2.2]	83.1 [4.7]	70.8 [1.6]	74.9 [1.1]	85.2 [1]	103.2 [2.7]
	Connections: 50							
500000	5.2 [0.3]	5.6 [0.4]	6.7 [1]	7.3 [1.3]	9.3 [0.3]	10.2 [0.4]	11.8 [0.9]	13.7 [1]
1000000	10.2 [0.2]	10.4 [0.6]	12.9 [0.9]	15.5 [1.2]	18.2 [0.4]	20 [0.4]	23.5 [1.1]	28.1 [0.9]
2000000	20.3 [0.4]	21.2 [0.2]	25.3 [1.3]	32 [2.7]	36.7 [1.1]	40.1 [0.5]	47.2 [1.2]	55.8 [1.2]
4000000	41.5 [0.9]	45.1 [3.7]	51.2 [2.7]	71.5 [4.1]	73 [0.8]	79.8 [0.7]	93.3 [2.2]	113.9 [2.5]
	Connections: 75							
500000	5.2 [0.6]	5 [0.3]	14.6 [18.4]	7 [1]	10.4 [0.4]	11.3 [0.5]	13.1 [1.2]	14.9 [0.9]
1000000	10 [0.7]	9.8 [0.5]	11.7 [1]	14.5 [1.1]	20.1 [0.3]	22.2 [0.3]	24.9 [1.1]	29.1 [0.5]
2000000	19.1 [0.7]	19.7 [0.6]	23.3 [0.6]	31.8 [1]	40.3 [0.4]	52 [17.5]	49.9 [0.8]	57.7 [1.8]
4000000	37.7 [2.2]	41.3 [1.2]	48.2 [1.7]	68.9 [3]	80.1 [0.8]	104.3 [23.1]	99.7 [1.5]	115.9 [2.3]
	Connections: 100							
500000	5.3 [1]	5 [0.4]	5.6 [0.6]	7.6 [2]	11.9 [0.3]	12.4 [0.4]	14.4 [0.7]	15.5 [1]
1000000	9.1 [0.4]	9.8 [1.1]	11.7 [0.9]	13.5 [0.4]	23.6 [0.7]	24.2 [0.5]	26.3 [1]	31.2 [1.8]
2000000	18.6 [0.9]	19.8 [0.5]	22.9 [1.3]	29.2 [2.2]	46.1 [0.6]	50.2 [4.1]	53.6 [1.4]	61.3 [2.2]
4000000	36.5 [0.7]	40.8 [3.3]	46.7 [1.2]	66.3 [2.6]	91.4 [0.6]	96.8 [1.1]	105.9 [1.2]	121.6 [4.4]

"Last Page Problem" occurs more often when the row size is small because the greater number of the transactions is trying to write to the same page. This leads to the best relative performance of GUID organized table when ($W = 50, R = 4 * 10^6$) – one can observe 2.5 times speed up. It should be noted that IDENTITY will not gain performance boost when more than 25 connections are used – more number of connections leads to performance drop.

At the end of this subsection, we just note the fact that repeating this experiment on second disk gave similar results.

4.2 Experiment 2

The second experiment was derived from the experiment 1. Its idea was to load the data to DB within several iterations and after each, the records were preserved. The number of records in each iteration was fixed. The test was describe with the following parameters:

- Number of parallel clients connected to the database: $c = 100$
- Number of rows inserted into the table – among all the connections: $r = 4 * 10^6$
- Width of the row (in bytes): $w = 254$
- Number of iterations were set to 6. After last iteration, the database had 24kk record – which was roughly 5.68 GB of data.

Test was repeated 4 times on disk 1 (SSD). The time was measured in seconds.

Table 5: Comparison of mean and cumulated execution time of stored procedure (by iteration)

Method	Iteration	Time[s]	Cumulated time [s]
guid	1	60,55	60,55
	2	58,99	119,54
	3	87,38	206,92
	4	231,65	438,57
	5	371,84	810,41
	6	434,97	1245,38
numeric	1	114,89	114,89
	2	108,2	223,09
	3	108,72	331,81
	4	108,77	440,58
	5	109,54	550,12
	6	108,84	658,96

The results of this test highlights that when data in a table does not fit in the buffer pool, IDENTITY's predictible storage page becomes a positive property. In the first three iterations (about 3 GB of raw data, with a buffer pool of 4 GB) using GUID as clustering attribute is beneficial – as observed in the first experiment.Unfortunately, when DB cannot find pages in the buffer, it has to perform the expensive disk IO requests which causes a steep increase in the execution time. This can be seen in figure 5.

When using IDENTITY as clustering key the storing page does not disappears from the buffer pool (as a result of "cache thrashing"), thus making the INSERT operation work in the constant time, regardles of table size. Despite the fact that initially the IDENTITY configuration performs twice slower than the GUID – in long run, the cumulative time of all six iterations turned out to be almost twice lower (658 seconds using IDENTITY and 1245 seconds using GUID, as observed in Table 5).

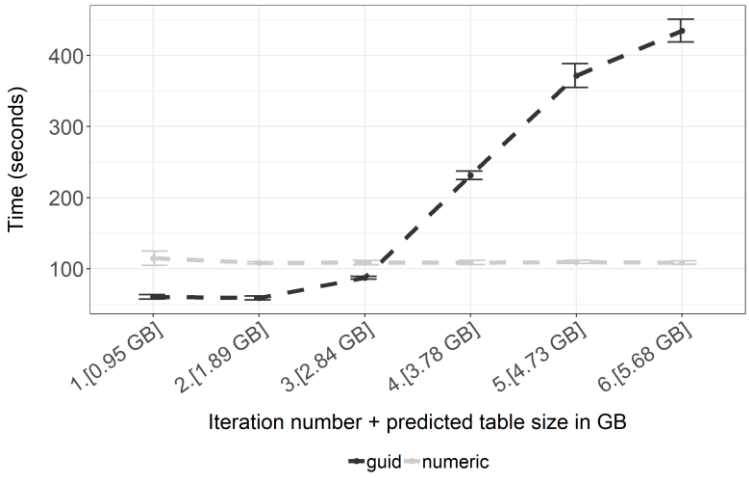

Figure 6: Comparison of mean execution time of stored procedure (by iteration)

4.3 Experiment 3

In the last experiment we use a single connection to perform batch operations (Bulk Inserts). In general DBs support special way to perform inserts of large volumes of data. In this experiment we compare Heap Files with B+ trees (with various clustering attributes).

For this test we prepared the file of 5.68 GB of raw data. The parameters in this experiment were:

- Number of parallel clients connected to the database: c = 1
- Number of rows inserted into the table: $r = 24 * 10^6$

- Width of the row (in bytes): $w = 254$
- Single commit batch size: 10^6
- Single commit batches were sorted

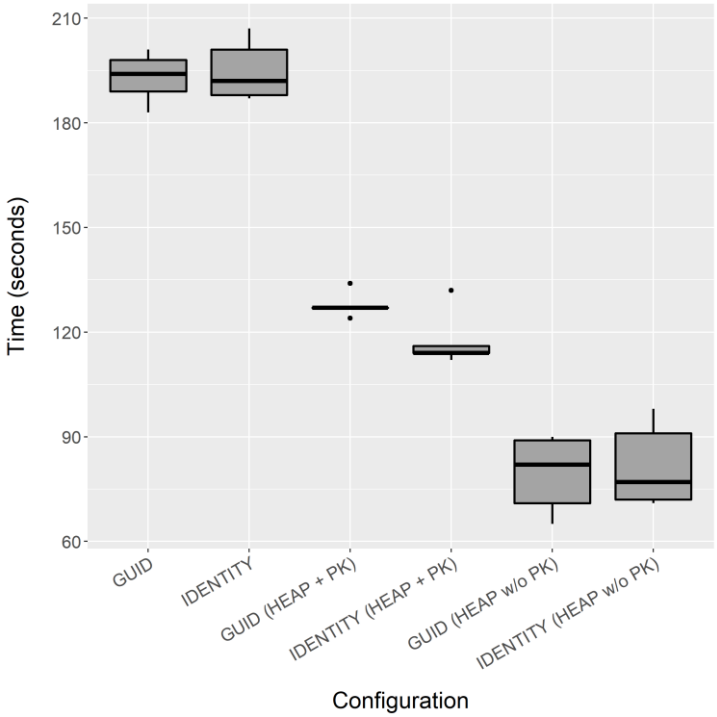

Figure 7: Comparision of batch load time

Test was repeated 5 times on first disk and the results are shown in the figure 7. When using B+ tree, the choice of the clustering attribute have not significantly influenced the load time – in both cases the mean time was around 195 seconds. Changing the table implementation to the Heap Files drastically cut down the time. When using Primary Key (PK) constraint, the load was done in about 120 seconds. When all constraints were drop, the load time achieved the astonishing 65 seconds (the lowest achieved load time). Interestingly enough – the choice of PK type did not influenced significantly the time.

5. Conclusion and further research

The study in this article shows the usage of random and monotonic attributes (represented by GUID and IDENTITY). We shows the conditions when choosing the right attributes leads to performance gain, measured as Transaction-Per-Minute. Also we presented the dramatic drop of the performance when data does not fit in the memory when random attributes are used. One should note however that some functionalities exists that can counter this negative effect – notably partitioning, Multi-Temperature Storages [12] or more control over the generated values (as in GUIDv1 or GUIDv2 [13]).

Further research should aim to:

1) Propose the formal model describing the performance
2) Demonstrate the ability to scale out the tables clustered on the random attributes beyond the size of the buffer pool
3) Test the performance of bulk load when dataset is not sorted
4) Measure the impact of page fragmentation

Bibliography

[1] Ullman D.J., Widom J.: A First Course In Database Systems, Helion Publisher, pages 110-129, 1997

[2] Leach P., Mealling M., Salz R.: RFC 4122: A Universally Unique Identifier (UUID) URN Namespace, https://tools.ietf.org/html/rfc4122 (Access: 9 September 2018)

[3] Nilsson J.: The Cost of GUIDs as Primary Keys, http://www.informit.com/articles/article.aspx?p=25862 (Access: 9 September 2018)

[4] Clayton R.: Do you really need a UUID/GUID?, https://rclayton.silvrback.com/do-you-really-need-a-uuid-guid (Access: 9 September 2018)

[5] Ricken U.: GUID vs INT/IDENTITY als Clustered Key, https://www.db-berater.de/2015/04/guid-vs-intidentity-als-clustered-key-2/ (Access: 9 September 2018)

[6] Penn J.: Taking It Further: GUIDs vs INTs as Primary Keys, https://scifisql.com/2017/05/07/guids-vs-ints-as-primary-keys/, (Access: 9 September 2018)

[7] Boicea A., Bucur I., Radulescu F., Truica C.A.: Performance Evaluation for CRUD Operations in Asynchronously Replicated Document Oriented Database, 20th International Conference on Control Systems and Computer Science, Bucharest, 2015

[8] Li Y., Manoharan S.: A performance comparison of SQL and NoSQL databases, IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing - Proceedings, 2013

[9] Elmasri R., Navathe S.: Fundamentals of Database Systems, Helion Publisher, pages 449 & 288-501, 2005

[10] Bača M., Grd P.: Analysis of B-tree data structure and its usage in computer forensics, Central European Conference on Information and Intelligent Systems, 2010

[11] Jhingran A., Khedkar P.: Analysis of Recovery in a Database System Using a Write-ahead Log Protocol, Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, 1992

[12] Brown D.P., Richards A: Managing access to data in a multi-temperature database, US Patent US9015146B2, 2015-04-21

[13] Marquardt A.: Generating Globally Unique Identifiers for Use with MongoDB, https://www.mongodb.com/blog/post/generating-globally-unique-identifiers-for-use-with-mongodb (Access: 9 September 2018)

ANALIZA WYDAJNOŚCI OPERACJI ZAPISU DLA TABEL UPORZĄDKOWANYCH ATRYBUTAMI IDENTITY ORAZ UUID

Projektowanie bazy danych wymaga podjęcia decyzji o fizycznej strukturze przechowującej dane. Często wpływ tej decyzji jest niedoceniany ponieważ 1) standard SQL nie precyzuje tego ograniczenia, przez co każdy dostawca Bazy Danych implementuje je po swojemu 2) wybór struktury jest podejmowany niejawnie. Na ogół domyślnymi strukturami są B+ drzewa które są strukturami posortowanymi. Wybór tej konkretnej implementacji tabeli wpływa zarówno na wydajność operacji odczytu jak i zapisu. Ze względu że częstą praktyką jest stosowanie atrybutów IDENTITY/AUTO_INCREMENT jako kluczy głównych, według tych wartości atrybutów ustalany jest fizyczny porządek tabeli. W pewnych przypadkach warto jednak korzystać z atrybutów o wartościach losowych w celu zwiększania przepustowości Bazy Danych (liczonej jako liczba transakcji na sekundę). Takie przypadki obejmują sytuację gdy dane mieszczą się w pamięci operacyjnej lub gdy możemy ograniczyć zbiór fizycznych stron do których Baza Danych będzie się odwoływać. W ogólnym przypadku ani atrybuty monotoniczne, ani losowe nie są lepsze od swoich konkurentów. W tym artykule (1) opisujemy struktury wykorzystywane we współczesnych Bazach Danych, (2) opisujemy zalety i wady dwóch najczęściej wykorzystywanych typów: GUID oraz IDENTITY, (3) prezentujemy analizę wydajności operacji zapisu porównującą oba typy w tabelach implementowanych jako B+ drzewo, (4) analizujemy wydajność operacji wsadowego ładowania zarówno w plikach sekwencyjnych jak i B+ drzew

Słowa kluczowe: projektowanie baz danych, model logiczny, porządkowanie, pliki sekwencyjne, B + drzewo, UUID, GUID, IDENTITY, sekwencje, wydajność wstawiania, ładowanie wsadowe