Maciej PENAR[[1]](#footnote-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, heap files, B + tree, insert performance

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.

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 + trees [9].

2.1 Heap files

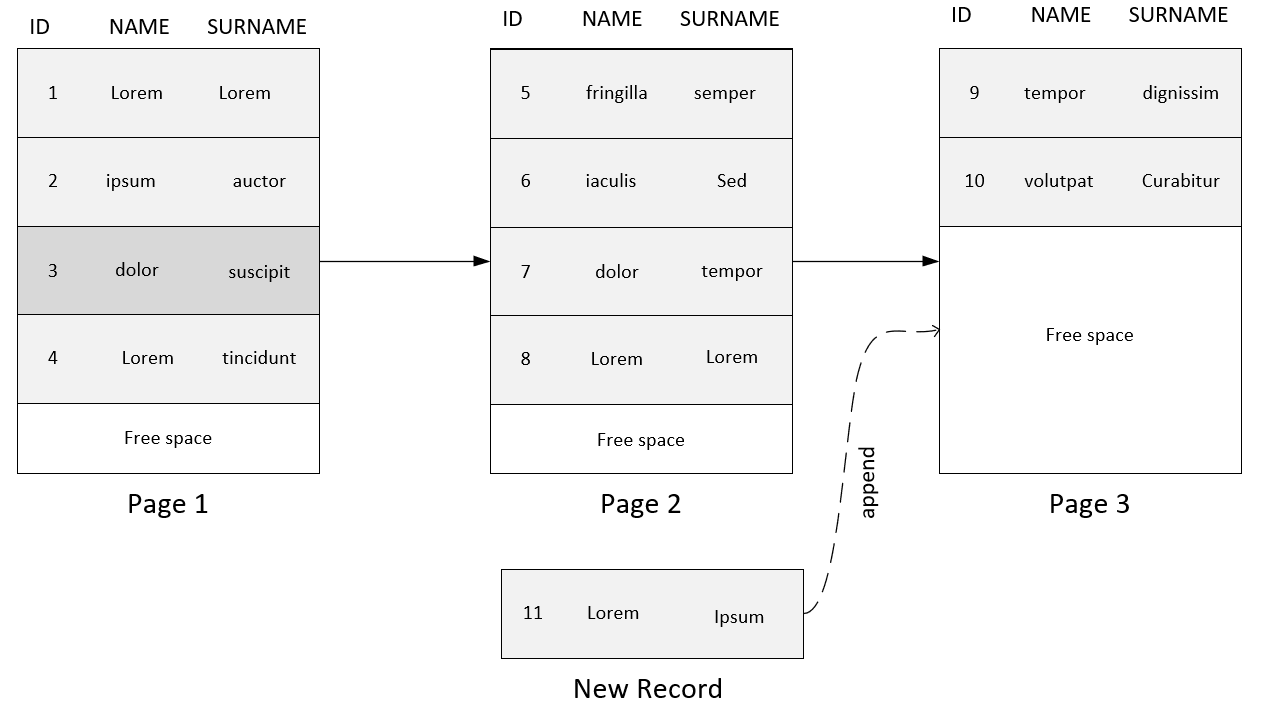
Heap files (also known as Sequential files) are DBMS equivalent of linked lists. Pages of heap files are linked together using pointer which are stored in a special sector of the page called header. This structure has a relatively low cost of INSERT command as it only requires appending it in the free 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 over some attribute A (or a list of attributes). In B + tree there are two kind of pages:

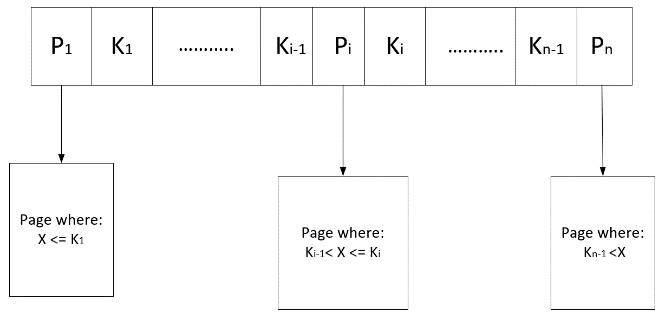
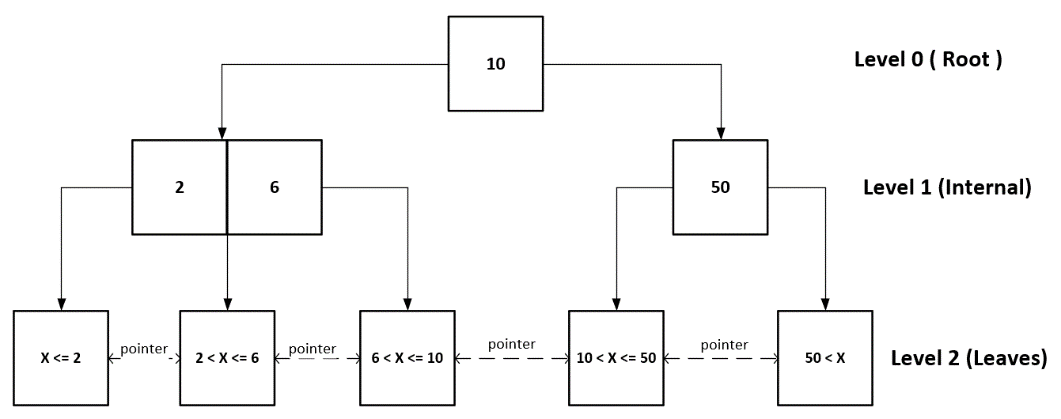
internal nodes – which contains values of attributes and pointers to (1) either other internal nodes on the lower level of the tree (2) or to the leaves which contains raw data. Each node contains pointers and keys . In each node, any pointer leads to node so that: , for . In case when or left and right side of inequality are omitted respectively. 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 enables effective range queries given the value of .

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 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 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.

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.

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 [↑](#footnote-ref-1)