Design and Implementation of two Storage Models for Main Memory Database Systems

Bachelor Thesis

submitted at
Chair of Practical Computer Science III
Prof. Dr. G. Moerkotte
School of Business Informatics and Mathematics
University of Mannheim

by
Nick Weber
February 5, 2017

Contents

List of Figures

List of Algorithms

1 Introduction

These days databases store unprecedented volumes of data, mostly in the multi-terabyte or even petabyte range. Still, the data needs to be accessible and processible at any given time. To fulfill this requirement, performance is the number one priority of a database system. The performance of a database system highly depends on the system's efficiency at storing and retrieving data from primary storage. For this reason the database community is conducting continuous research in the field of physical data storage. Over the years two major approaches in storing data have been established, known as row-store and column-store. Both have situational advantages on each other and dominated for about 30 years; especially row-stores were the standard architecture for relational database systems. Row-stores are commonly used for transaction-oriented database systems and column-stores are usually chosen for analytical queries over large datasets. In the early 2000s Anastassia Ailamaki, David J. DeWitt, Mark D. Hill and Marios Skounakis have proposed a new data layout model which they named PAX[4]. This work of Ailamaki et al. is the foundation of this bachelor thesis.

The goal of PAX is to combine the advantages of row-based and column-based approaches in a hybrid model. Best of both worlds are combined to fully utilize cache resources and minimizing record reconstruction costs. Compared to a row-based implementation, the research team behind PAX measured 17% to 25% less elapsed time for the execution of range selection queries and updates. At the same time TPC-H queries are executed 11% to 42% faster. With more attributes involved in the query, PAX's execution time remains stable while column-based implementations have increasing execution time due to high record reconstruction costs. This elaboration will take a closer look at the different storage models and discuss the implementation of an in-memory database system. In preparation for that it is important to have basic knowledge about computer science and database systems. Therefore the next chapter will cover basics needed to understand the topic in more detail. In the third chapter, the history of storage models and some major milestones in research will be discussed and afterwards the storage models themselves will be described further. The fourth and fifth chapter cover the implementation and evaluation respectively. Last but not least the thesis will be concluded with a summarization of the findings and opportunities for further research.

2 Basics

2.1 Fundamental Terms

Random-Access-Memory: Random-access-memory or RAM is a type of computer storage where every byte can be accessed directly. The name emphasizes the difference to data storage media such as magnetic tape or magnetic disks which can only be accessed either sequentially or block wise. The two widely used forms of RAM are static RAM (SRAM) and dynamic RAM (DRAM). SRAM is more expensive to produce but is generally faster and therefore often used as *cache memory* for the CPU. DRAM on the other hand is cheaper and mostly used as main memory in computers. Both RAM types are volatile memory and lose their data when power is removed.

Cache Memory: Cache memory, in the context of this thesis simply refered to as cache, is a fast and relatively small memory type completely managed by the hardware. It stores the most recently accessed main memory contents in order to reduce access time to data. Modern caches are usually build directly into the central processing unit (CPU) and consist of three levels, namely L1-, L2- and L3 cache. Each cache level has a different size and access time. The L1 cache comes closest to the core and is the smallest but also fastest one. With increasing cache level comes decreasing access speed but increasing size. A cache's function is to reduce main memory access which is a major bottleneck of todays database systems (DBS). The difference between the layers of the memory hierarchy will be discussed in the next subsection.

Cache Hit/-Miss: When a processor requests data from main memory, it will first check whether the data is already in the cache or not. If the CPU finds the memory location of the data in the cache, a cache hit has occured. In this case, the processor immediately processes the data in the *cache line*. However, not finding the memory location in one of its caches results in a cache miss. Now the cache has to allocate a new entry and fetch the data from the main memory. Processing data in the cache is about 100 times faster than fetching data from main memory into the cache.

Cache Line: When the CPU fetches some data from main memory, it wont just load a single byte but a block of fixed size (64 byte on todays x86 architectures), called a cache line. If the CPU now needs additional data from the main memory and it happens that this data is in the same cache line as the previous loaded, this is a major performance increase since a cache hit occurs. Otherwise, if none of the other data in the cache line is ever used, this results in bad performance and is called overfetch. The concept of cache lines can be exploited to speed up execution by storing related data closely

together, known as *spatial locality*.

Prefetching: Sometimes a computer's hardware and software can predict which data the CPU will need in the future. In this case the processor can fetch data into the cache before it is actually needed. This is called prefetching. With prefetching the execution of a program can be significantly accelerated by reducing CPU stall times. Various hardware and software techniques exist and together they can significantly reduce wait states. An example for hardware prefetching would be to observe and detect memory access patterns and prefetch data based on this observation. Except for certain situations this process is quite difficult. An easier approach is software-based prefetching like inserting prefetching instructions into the program's source code. The compiler automatically predicts future cache misses and inserts prefetching instructions based on analysis of the code. This is widely used within loops over arrays.

Principle of Locality: An observation has shown that 90% of the execution time of a computer program is spent executing 10% of the code, known as the 90/10 rule. In the execution of a program it often occurs that the same values are accessed multiple times or access to an address space results with high probability in another access in close proximity. The principle of locality describes these phenomenons. There are two important types of locality, *temporal locality* and *spatial locality*. Temporal locality means that recently accessed data is likely to be accessed again in the near future. Spatial locality means that addresses of data which are near one another tend to be referenced close together in time. Following the principle of locality as a programmer is generally a good opportunity to improve cache utilization (cf. 2.3 *Cache Utilization*).

Data Alignment: As already mentioned earlier, a processor does not read from and write to memory in byte-sized chunks. The chunk size in which a CPU accesses memory (i.e., the cache line size) is called *memory access granularity*. Data alignment means storing the data at a memory address equal to a multiple of the system's word size (8 bytes on todays x86 architectures). To increase the system's performance or sometimes even to avoid crashes, the data in the memory has to be aligned. This is because of the CPU which may run into problems accessing unaligned data. It may be necessary to insert some padding bytes at the end of some data to have a valid alignment. For example in an 8 byte aligned memory, storing a single char (1 byte) followed by a double (8 byte), a 7 byte padding after the char would be necessary. The double, on the other hand, is already aligned since its size equals the alignment size.

Online Analytical Processing (OLAP): OLAP deals with the preparation and analysis of data in a useful way for decision making. It is characterized by read-intensive queries on large data repositories, often with a high degree of complexity and involving aggregation. The term OLAP is often used to distinguish a DBS from a relational, transaction-oriented one. These are commonly called *Online Transaction Processing* (OLTP) systems.

TPC-H Benchmark: The Transaction Processing Performance Council (TPC) is a non-profit organization founded to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry[1]. TPC-H is a decision support benchmark focused on OLAP style queries over large datasets. It consists of eight relations and 22 queries, chosen to have broad industry-wide relevance. The TPC-H benchmark simulates decision support systems that execute queries with a high degree of complexity, on large volumes of data. The dataset of this benchmark is used to evaluate this work's implementation.

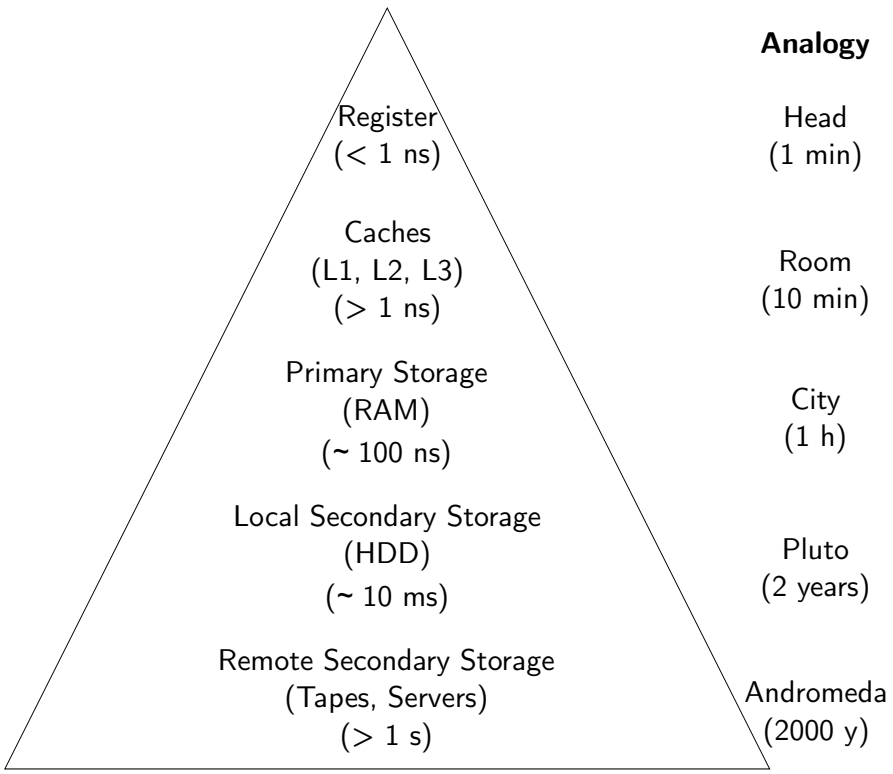


Figure 1: Visualization of the memory hierarchy. Adopted slightly modified from Kemper (2015), p. 213[12]

2.2 Memory Hierarchy

The large and affordable databases which can be seen today can only be realized because of the tremendous progress in storage technology. Early computers had a few kilobytes (KB) of main memory while today an average consumer desktop computer has around two terabytes (or $2 * 10^9$ KB) of disk storage. Until recently the amount of storage has been increasing by a factor of 2 every couple of years. Traditional DBS store their data on secondary storage, composed of HDDs. The swapping of pages between the main-memory-buffer and disk storage is the main bottleneck of conventional DBS. In Figure 1, we can see response times for different storage levels. There is a huge difference in access time between primary storage and secondary storage, about factor 10^5, which describes the previous mentioned bottleneck. The pyramidal shape of the image helps to visualize the difference in size between the storage levels. At the thin top the fastest type of memory, the registers, resides but in terms of capacity it is also the smallest one. Today a common register size is 64 bit but there are also special registers with up to 512 bit. As we can see, the lower the storage level goes the slower the response time becomes. A good analogy exists to realize the difference between storage levels in the memory hierarchy. By associating a register with our head and the access time to an information is about one minute, then the access of data stored on disk would be equivalent to the time needed for a flight to the dwarf planet Pluto. One can see that access to secondary storage should be prevented under all circumstances. Technological progress in recent years allowed for ever growing main memory capacities at a decreasing price tag. This had a big impact on the development and research of DBS. Now it became affordable to maintain the whole database in-memory, so called main memory databases, which led to chances and challenges equally. As we can see in Figure 2, processor performance had a much more rapid growth over time than memory performance had. Today the new bottleneck in the memory hierarchy is access to main memory. Challenges like *data storage layouts* and *cache utilization* became more important in order to reduce costly main memory access.

2.3 Cache Utilization

As elaborated in the previous section, for a main memory database the performance bottleneck was shifted to the area between RAM and cache. Since access to data in the cache is about 100 times faster than data in main memory (cf. 2.1 *Cache Hit/-Miss*), it is very important that needed data resides in the cache. By loading cache lines instead of single bytes the cache makes

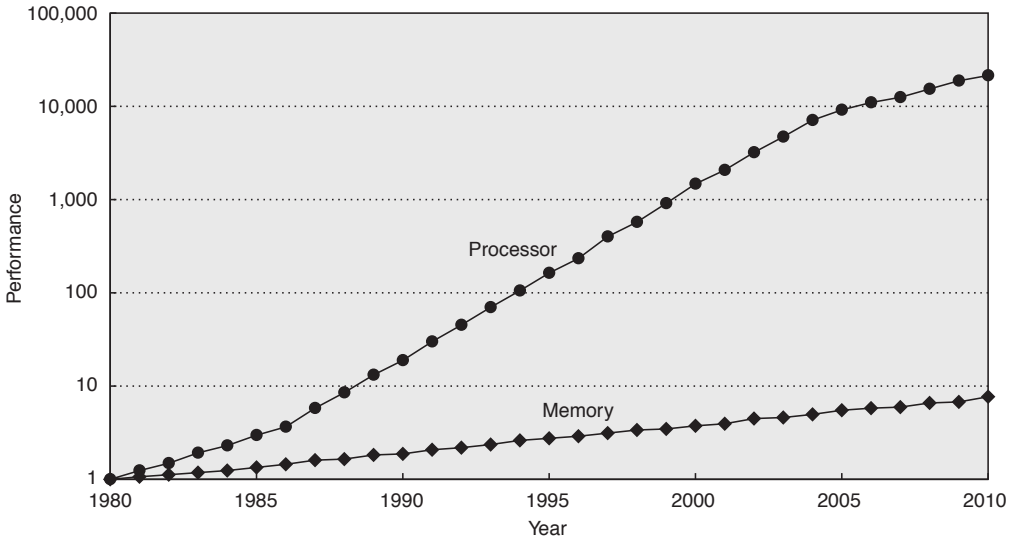


Figure 2: Gap in performance between memory and processors over time. Adopted from Hennessy and Patterson (2007), p. 289[10]

use of spatial locality (cf. 2.1 *Principle of Locality*). A good programmer will make use of this and try to pack related data closely together in main memory to achieve a high spatial locality and therefore a performance increase. A simple example demonstrating the dramatic speedup is the multiplication of large matrices. In Algorithm 1 we can see the pseudo code for a naive matrix multiplication without any cache utilization. The problem can be found in the deatails. The reads of $A[i][k]$ are in cache but $B[k][j]$ is not which results in a cache miss. Let's take a look at Algorithm 2. The only thing changed is switching the looping order for j and k. This will have no effect on the mathematical result but will greatly improve the cache locality. In this case both $C[i][j]$ and $B[k][j]$ are in cache while the read of $A[i][k]$ can be factored out. If we compare the two implementation's inner loops, we have one cache miss in Algorithm 1 against zero cache misses in Algorithm 2. A test on a year 2015 3.3 GHz processor (cf. 5.1 for setup information) and more than 3,000,000 elements in each matrix revealed a more than five times faster execution of Algorithm 2. The programs were written in C and compiled with *gcc-O*3.

Algorithm 1: Matrix multiplication without cache utilization

```
1 for i ← 0 to n do
2     for j ← 0 to m do
3         for k ← 0 to p do
4             C[i][j] ← C[i][j] +
              A[i][k] * B[k][j]
```

Algorithm 2: Matrix multiplication with cache utilization

```
1 for i ← 0 to n do
2     for k ← 0 to p do
3         for j ← 0 to m do
4             C[i][j] ← C[i][j] +
              A[i][k] * B[k][j]
```

2.4 Physical Algebra

The goal of a database system is not only to store accumulated data but also to retrieve information out of it. In order to extract new information out of old data, we try to find connections and make statements about the data. For this some kind of algebra (from the arabic 'al-jabr' meaning 'reunion of broken parts') is needed. Relational algebra (i.e., logical algebra) is used to state *what* should be done. An example for a query would be: *search in the data of employees and select all employees with an age lower than 35.* Relational algebra is the theoretical foundation for query languages like SQL. Physical algebra on the other hand is about *how* things should be done (i.e., the implementation). The relational and physical algebra consist of operators but each relational operator may have one or more physical operators associated which differ in their implementation. In Section 4.1 the implementation of physical operators for the database system implemented as part of this work are discussed. There are two ways to implement physical algebras, pull-based variants start at the output operator (i.e., root operator in an operator tree), also called *top* operator. The top operator requests one tuple at a time from the consecutive operator. Every operator forwards this request to the next operator. The last operator in this chain is the *scan* operator which reads the data from memory and returns a tuple to the previous operator. Each operator in the chain executes an operation on the tuple and returns the result back up until the top operator is reached. This is repeated until the whole query was processed. In this implementation the push-based variant is chosen. Here the procedure starts at the scan operator and each tuple is pushed upwards the operator chain.

3 Storage Models

3.1 History

Until the 1970s row-stores were the only architecture type for relational DBS. A common row-based implementation for storing records inside a page was the slotted-page approach. This approach is better known as the N-ary Storage Model (NSM) and will be discussed in detail in section 3.2. Row-stores store their data, as the name already indicates, in rows. Complete records are stored one after another as exemplified in Figure 3a. In the 1970s column-oriented storage techniques appeared and in 1985 Copeland and Khoshafian presented their work about the Decomposition Storage Model (DSM)[11][8], an alternative to NSM. The basic idea of DSM is to store each column of a table separately as in Figure 3b. More details on DSM will be discussed in Section 3.3. In comparison to NSM, DSM only had situational advantages at the cost of extra storage space. Even though further research on DSM improved its position towards NSM, column stores could not yet compete with the established row-stores. In the following years technological progress in

(a) Row Store (b) Column Store

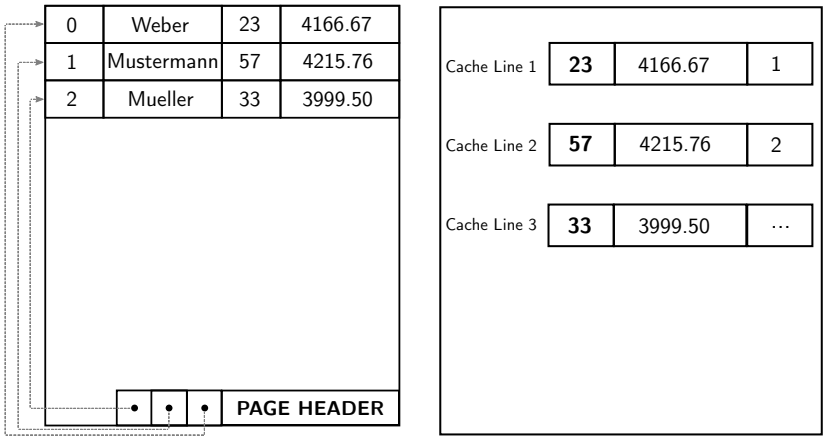
Figure 3: Physical layout of row-oriented and column-oriented databases

the hardware area had a big benefit for column-stores. The time required to access data in main memory and the execution of an instruction were about the same in 1980, as already seen in Figure 2. In the mid 1990s, however, memory latency had grown to hundreds of CPU cycles. As memory access became more expensive, column-stores gained popularity for large analytical queries over millions of records because of the better cache utilization (cf. section 2.3). In 1996, MonetDB, the first major column-store project of the academic community was created[7]. Also in 1996 SybaseIQ, often credited with pioneering the commercialization of colum-store technology[15], emerged as a powerful compressed column-oriented system. As already mentioned in

the introduction, Ailamaki et al. proposed the hybrid NSM/DSM alternative PAX in the year 2001. PAX stores the same data on each page as NSM but groups together all values of an attribute on so called mini pages (for further details see section 3.4). Projects like Data Morphing[9] and Clotho[13] continued research based on PAX. Hardware further advanced with large main memories and SIMD instructions. This in combination with architectural innovations such as late materialization and direct operation on compressed data throughout query plans extended the advantages of column-stores even more[2]. At the beginning of the 2000s column-stores got regained interest from both the academic community and industry. The introduction of the pioneering C-Store[15] and VectorWise[16] paved the way for new breakthrough column-oriented DBS.

3.2 The N-ary Storage Model

Traditional DBS store their data in rows. Each row in the table represents an entire record (cf. table *Employees* in Figure 3a). In implementations of row-stores the table's records are traditionally stored in slotted pages. This concept was introduced in section 3.1 as NSM. In NSM records are stored sequentially on pages like in an array of bytes. Because records can have a variable length, a pointer to each record is stored in a slotted page header at the end of the page. In order to acess record n the pointer in the n^{th} slot has to be followed. A sample NSM page is depicted in Figure 4a after inserting three records of the already known relation *Employees*. The relation



(a) NSM Page (b) NSM Cache Line

Figure 4: Sample NSM page and cache behavior

has four attributes: EID (employee id), name, age and salary. All attributes of a record are stored together sequentially on the same page, offering intra-record spatial locality, followed by the next record. New records are typically stored at the first available free space of the page. Beside pointers in the slots, the slotted page header contains the number of records stored, the available free space and a pointer to the end of free space on the page. After understanding the basic characteristics of NSM and row-stores, we can now think about strengths and weaknesses of NSM. In OLAP-style queries, which access only a few columns of a record, NSM exhibits poor cache performance. The following assumptions depict an example which will be used for further explanations. The sample system has a cache line size of 8 byte and attributes of table *Employees* have size 2 byte (EID), 8 byte (name), 2 byte (age) and 4 byte (salary). When running the query in Algorithm 3, the CPU will fetch

Algorithm 3: Simple select query

```
SELECT  salary
FROM    employees
WHERE   age < 35;
```

the first age value to evaluate the predicate. As described earlier the CPU will always load cache lines instead of single bytes. Each cache line starts at the age attribute and loads the following 8 bytes. Therefore, age and salary of the first employee as well as the EID of the next employee are stored in the same cache line. This behavior is visualized in Figure 4b, were we can see that one cache miss per record occurs. In the example for every needed value 6 bytes of memory bandwidth is wasted. Update transactions on the other hand, changing most attributes of the record, is where NSM performs well. With the same assumptions as before and the query in Algorithm 4, only 2 bytes cache space is wasted, even with this small cache line size. This

Algorithm 4: Simple update query

```
UPDATE  employees
SET     age += 5, salary += 1.1
WHERE   age < 35
AND     salary < 4000;
```

is because the *age* and *salary* attribute are stored close together. The CPU can load both attributes in one cache line and only the *EID* attribute is wasting cache space. Nevertheless, one cache miss per record occurs. With

the increase of predicates to evaluate and cache line size, the strengths of NSM in OLTP-style transactions further grow.

3.3 The Decomposition Storage Model

Early column-stores were implemented with DSM, storing all attribute values of a column together on a separate page. DSM partitions a relation with n attributes vertically into n sub-relations. All sub-relations are stored on separate pages containing exactly two attributes, a logical record id and an attribute value. The logical record id is used to join attributes together to reconstruct a record. DSM makes use of the slotted page concept to store the sub-relations, as seen in Figure 5. Instead of storing pointers to records,

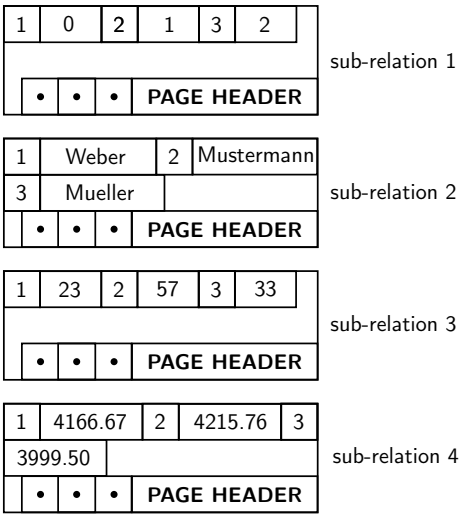


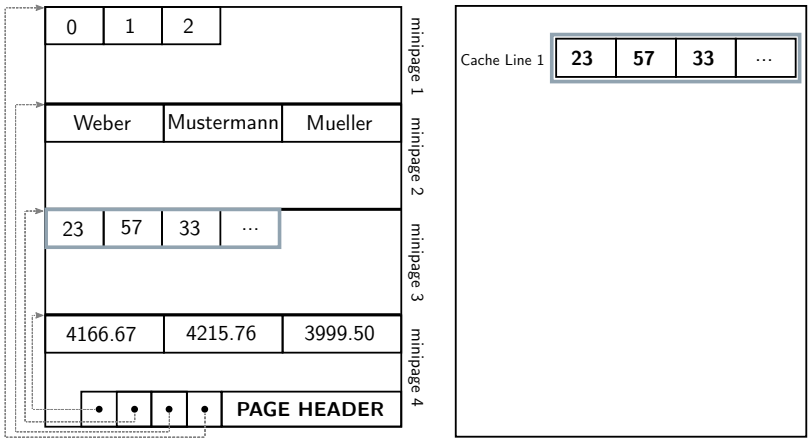
Figure 5: DSM sub-relations stored on separate pages

pointers to attributes are stored in the page header on each sub-relation. To access attribute m of record n one has to follow the n^{th} slot on sub-relation m. On the sample page the same data as previously from the table *Employees* is stored. Next to each attribute value, the logical record id is stored. It can already be seen that DSM needs more storage space to store the same data as NSM. This is because of the the logical record id, stored for every column and attribute. By storing values of the same attributes sequentially on the same page (inter-record spatial locality), DSM offers better cache behavior than NSM. We can observe this by running Algorithm 3 again, this time on the DSM implementation of our dataset. The CPU will load the first age value and a cache line of 8 bytes into the cache. With an attribute size of 1 byte for the logical record id, one cache line contains all attribute values needed of all

three records needed in order to process the query. With more records stored, this would result in one cache miss for every three records. Queries involving multiple attributes from different sub-relations is where DSM's performance deteriorates because it has no intra-record locality. The DBS has to join the involved sub-relations to reconstruct the original relation, since the records attributes are spread over multiple pages. The storage overhead and the time needed to join multiple sub-relations is what significantly limits the potential of originally DSM based implementation. An analysis (based on technology available at the time) showed that DSM could speed up certain scans over NSM when only a few columns were projected, at the expense of extra storage space[2]. Because performance is so heavily dependent on the queries, DSM was mostly considered a niche model.

3.4 Partition Attributes Across

In this section the detailed design of PAX and its goals are discussed. PAX aims to combine the intra-record spatial locality of NSM with the inter-record spatial locality of DSM. For inter-record spatial locality, PAX stores values of the same attribute together on one page. At the same time all attributes of a record are located on the same page for minimal impact on the intra-record spatial locality. This is achieved by vertically partitioning a page into so called *minipages*. Each minipage contains only values of the same attribute. Records are completely stored on the same page, spread over the minipages. To store a relation with m attributes, PAX partitions each page into m minipages. A PAX page with four attributes (i.e., four minipages) is depicted in Figure 6a. Slots in the page header point to the beginning of each minipage, the number of slots therefore equals the number of attributes. Values of the m^{th} attribute are stored in the m^{th} minipage and the n^{th} attribute values inside each minipage belong to record n. Therefore no logical record id is stored, making the storage overhead of DSM obsolete. To store a relation, PAX requires the same amount of space as NSM. Storing values of the same attribute together minimizes processor stall times when executing queries over few attributes. This is because cache resources are fully utilized by loading multiple relevant attribute values into the cache together. To get a more detailed insight on the cache behavior of PAX, Algorithm 3 is run again on the sample PAX page. Four 2 byte age values fit in a 8 byte cache line, resulting in only one cache miss for every four records. In the example, after the first cache miss all three age attribute values are present in the cache and therefore the query is processed with only one cache miss, as seen in Figure 6b. It is important to look into the record reconstruction costs of PAX, since DSM already had a good cache



(a) PAX Page (b) PAX Cache Line

Figure 6: Sample PAX page and cache behavior

behavior but suffered from poor performance when having to join multiple sub-relations. In contrast to DSM, no record reconstruction over multiple pages is needed. Because PAX stores all parts of a record on the same page only a *mini-join* among minipages has to be performed. This *mini-join* can be performed within the page boundaries, which incurs minimal record reconstruction costs.

4 Implementation

4.1 Class Overview

Both NSM and PAX were implemented as part of this work. The most important classes for the storage layout's implementation will be introduced and explained here. An overview of these classes can be seen in Figure 7, which is an illustration based on UML class diagrams. It should be noted that only a selection of few member variables and methods are depicted.

Relation: To store a database table, some kind of internal representation is needed. The system might need a logical schema, the segment were tuples are stored or information about attributes of the relation. For each table these information are stored within an object of class *Relation*. Instead of implementing one class for each table, with all the relevant information hard-coded into the class, a more generic approach was chosen. The class *Relation* represents all kinds of relations, independent of the later used storage layout. An object can either be created by passing the needed variables to a constructor or through the default constructor. In the second case a call to the *initRelation*-method has to be made before using the object any further. Sub-classes exist for storage layout dependent information such as the physical schema or information about the partitioning of a page.

Bulk Loader: In order to have any use of the database system, it must be filled with data. The process of loading data from some other source (e.g. a CSV-file) into the database is called *bulk-loading*. In this implementation the bulk-loading process was split into two steps. First the class *BulkLoader* handles the read-in of a data file and stores its content in an intermediate buffer. To construct an *BulkLoader*-object one must pass a filename, a relation-object and some variables for internal organization to the constructor. A call to the *bulk_load*-method starts the first step of bulk-loading: reading data from the associated file into the intermediate buffer. Until now everything is independent from the storage layout.

Bulk Insert: This class implements the second step of bulk-loading, storing data from the intermediate buffer inside memory pages. Since this process is not independent of the used storage layout, one class per storage layout has to be implemented. In our case we have the class *BulkInsertSP* (SP stands for slotted pages) to store data on a NSM page and the class *BulkInsertPAX* for a PAX page. In order to start the second step of bulk-loading, a call to *bulk_insert* with an *BulkLoader* and *Relation* object as parameter has to be made. The *BulkLoader* object is needed to get access to the intermediate buffer and one of the respective *Relation*'s subclasses is passed for storage

layout specific information. After a successful completion of the bulk-insert, the data is stored correspondingly to the used storage layout.

Page Interpreter: Because bulk-insert-classes have no page level logic implemented, a page interpreter is used. Either a *PageInterpreterSP* or *PageInterpreterPAX* object is used to access the respective page layout. The page interpreter manages page information such as number of records stored and available free space on the page. It also provides pointers to records or minipages respectively. New pages are initialized by the page interpreter, setting the initial values of the page header and, in the case of PAX, partitioning the page according to provided partition data.

Segment: Every memory page used by the database system, is managed in segments. A segment holds a vector of pointers to pages inside the memory chunks. Via a *Segment*-object one can access managed pages, get the total number of managed pages per segment or make requests for new pages. Upon request for a new and free memory page, the segment forwards the request to the *MemoryManager* and stores a pointer to the requested and allocated page in its vector.

Memory Manager: The memory manager's task is to allocate, deallocate (i.e., free) and manage chunks of aligned main memory. After the very first request for memory in the database system, the memory manager allocates a big chunk of memory. The chunk has a size of

$$MEMCHUNK\_SIZE * PAGE\_SIZE$$

bytes, both *MEMCHUNK_SIZE* and *PAGE_SIZE* are globally defined variables. After all free pages in a memory chunk have been assigned, the next chunk is allocated. The *MemoryManager* class keeps track of all allocated chunks with a vector of pointers to each chunk. Additionally a pointer to the first free page within a chunk of memory and a pointer to the end of a chunk is stored. These pointers are adjusted if a new memory chunk is allocated and point to the beginning and end of the new chunk respectively. The only methods visible to the rest of the DBS are *getPage* to request a new page and *freeMemory* and *freeAll* to either deallocate one specific or all chunks respectively.

Physical Algebra: For the implementation of the physical algebra operators a push-based variant is chosen. The scan operator has a built in selection and therefore serves as scan and select operator at the same time. In order to have storage layout independent operators, the scan with build in selection as well as the top operator were implemented as templates. The operators

offer three functions named *init*, *step* and *finish*. However, in this implementation, the first one is used as a placeholder without any functionality. With a call to *step* a tuple is forwarded to the next operator and the operation is executed. In the case of the top operator the operation is an optionally print of the forwarded tuple and an increment of some counter. A call to *finish* terminates the operation after the last tuple was forwarded.

Vectorized Processing: Instead of passing one tuple at a time, vectorized processing was implemented. Here not single tuples are forwarded to the next operator but a vector of N tuples is. Size N of a vector is determined before the query is run. The typical size for the vectors used in vectorized processing is such that each vector comfortably fits in L1 cache as this minimizes reads and writes throughout the memory hierarchy[2]. Vectorized is implemented as part of the selection. For each tuple the predicate is evaluated and tuples that satisfy the predicate are inserted into the vector. As soon as the vector is full the method *step* is called with the vector as argument. In turn the next operation then processes the full vector instead of single tuples.

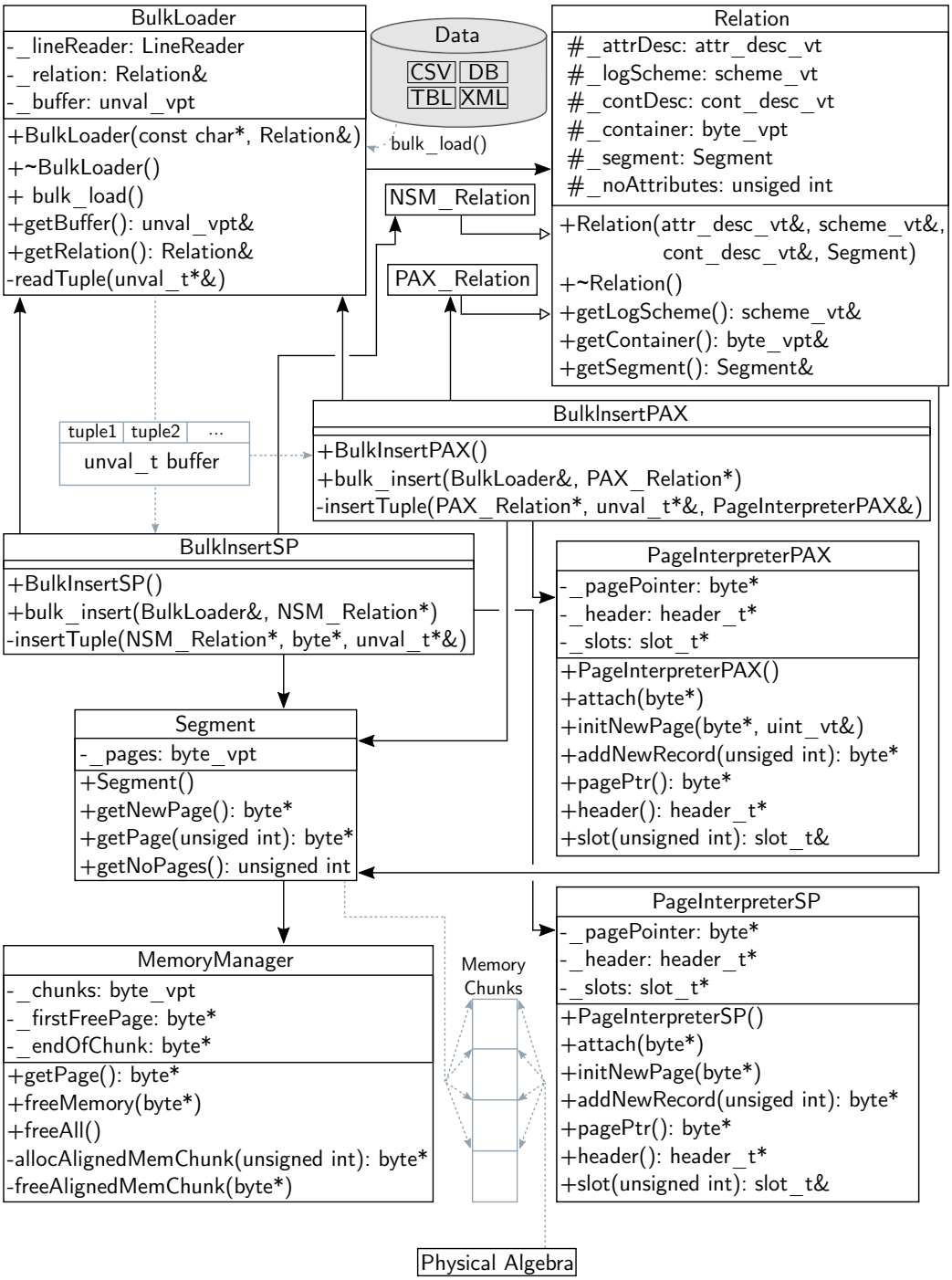


Figure 7: Class Overview

4.2 NSM

NSM specific implementation details will be discussed in this sub section.

Before storing data in a slotted NSM page, it has to be thought of data alignment, physical order of data and how single attribute values of a record can later be accessed efficiently. In NSM the physical schema of a relation differs from its logical schema in order to minimize data alignment padding bytes. Storing a record of table *Employees* in its logical order requires padding bytes for every attribute except for the name attribute. When storing a record in a physical order as depicted in Figure 8 on the other hand, no padding byte for the record is needed at all. A general rule is to store the largest attributes

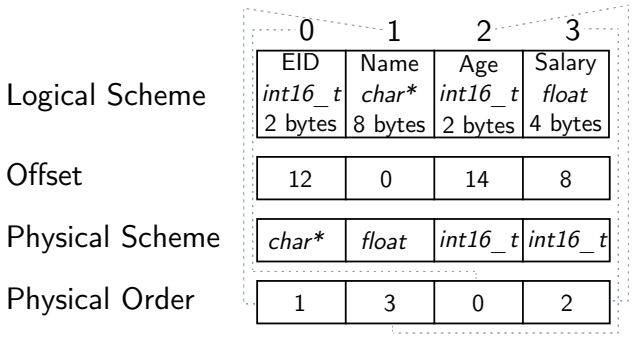


Figure 8: Logical Schema, Physical Schema and Offset of a NSM Relation

first, followed by the second largest, and so on. The *Physical Order*-vector indicates the physical order in which attributes are actually stored in memory. This vector can be thought of as a physical schema. Its values represent indices of stored attributes with respect to the logical schema. Its own index indicates the order within the memory page. Because only record pointers are stored in the slotted page, an *Offset*-vector is needed to calculate the location of a single attribute value. The offset vector is computed and stored in the **NSM_Relation**-class. The pseudocode for the offset calculation can be seen in Algorithm 5. The algorithm starts with searching for all 8 byte attributes. If an attribute with a size of 8 byte is found, the index of this attribute in the logical schema is stored in the *lPhys_schema*-vector. At the ninth line 8 byte is added to the previous offset and stored in the *lPhys_offset*-vector. After this procedure was executed for all 8 byte attributes, the algorithm continues with the same procedure for 4 byte-, 2 byte- and 1 byte attributes. At the end of the first while-loop, the physical schema and 'physical offset' is calculated. Now only the desired offset is missing which is calculated inside the last loop starting at line 26. Inside the loop the first logical attribute is searched for in the *lPhys_schema*-vector. If

Algorithm 5: Calculating the physical schema and offsets for the logical and physical schema

```
1  Function calcPhysSchemaAndOffset(){
2      index ← 0
3      lPhys_offset
4      lPhys_schema
5      lPhys_offset[index] ← 0
6      i ← 8
7      while (i > 0){
8          for (j ← 0 to no_of_attributes){
9              if (log_schema[j] has 8byte AND i == 8){
10                 lPhys_schema[index] ← j
11                 lPhys_offset[++index] ←
                   (lPhys_offset[index − 1] + i)
12             }
13             elif (log_schema[j] has 4byte AND i == 4){
14                 lPhys_schema[index] ← j
15                 lPhys_offset[++index] ←
                   (lPhys_offset[index − 1] + i)
16             }
17             elif (log_schema[j] has 2byte AND i == 2){
18                 lPhys_schema[index] ← j
19                 lPhys_offset[++index] ←
                   (lPhys_offset[index − 1] + i)
20             }
21             else{
22                 lPhys_schema[index] ← j
23                 lPhys_offset[++index] ←
                   (lPhys_offset[index − 1] + i)
24             }
25         }
26         i ← i/2
27     }
28     for (j ← 0 to no_of_attributes){
29         index ← 0
30         while (j != lPhys_schema[index]){
31             ++index
32         }
33         log_offset[j] ← lPhys_offset[index]
34     }
35 }
```

the attribute is found, its index within the vector is remembered. To get the offset of the attribute, the value inside the *lPhys_ offset*-vector at the remembered index location is stored. This procedure is repeated for all remaining attributes. Looking again at Figure 8 may help in better understanding the concept.

Now that the physical schema is known and the page addresses of attributes can be calculated with their respective offset, the required logic to store and access data is available. But the system can still not operate on a memory page level directly. As already stated in Section 4.1, the bulk-loading process has no knowledge of pages, their boundaries, free space on the page or where to store data within the page. To operate on a page level a page interpreter is needed. The class **PageInterpreterSP** realizes this page level logic for NSM pages. Newly allocated pages get initialized with a page header, setting the number of records, free space on page and an offset to the next free entry to default values. Each time the bulk-loading process has a new record to insert the *addNewRecord*-method of the page interpreter is called. It returns the address where to insert the record if enough free space for the record is available on the page. The pseudocode for this method is depicted in Algorithm 6. First the total size of the record needs to be calculated.

Algorithm 6: Compute address where to insert a new record

```
1  Function byte* addNewRecord(aRecordSize){
2      alignedSize ← ((aRecordSize + 7) & ∼ (uint) 0x07)
3      totalSize ← alignedSize + sizeof(slot_t)
4      byte* address ← 0
5      if (totalSize <= freeSpace){
6          address ← pagePointer() + nextFreeRecord
7          nextFreeRecord ← nextFreeRecord + alignedSize
8          freeSpace ← freeSpace − totalSize
9          slot(noRecords).offset ← address − pagePointer()
10         noRecords ← noRecords + 1
11     }
12     return address
13 }
```

Therefor the logical record size (passed as parameter) is adjusted to an 8 byte alignment. In Section 3.2 it was shown that for every stored tuple on the page a slot pointing to that tuple exists. Such a slot is generally implemented as offset instead of as a pointer to save storage space. An offset is then added to the address of a page to get the corresponding pointer. In

this implementation a slot is realized as 2 byte unsigned integer offset saving 6 bytes of storage space compared to a pointer. At line 3 of the pseudocode the size of a single slot is added to the aligned record size and saved as *totalSize*. The next step is to check whether there is enough free space available on the page to store the record and its slot. If that is not the case, *zero* is returned as address and the caller of the method knows a new page needs to be allocated. Otherwise the address is set to

$$address = pagePointer() + nextFreeRecord$$

and is returned after updating the page header. The caller of *addNewRecord* can now insert a record at the returned address.

Insertion of data is handled by the class **BulkInsertSP**. Its already mentioned method *bulk_ insert* is called with an *BulkLoader* and *NSM_ Relation* object as parameter (cf. Section 4.1, *BulkInsert*). Inside this method a page interpreter object is created for the previously discussed page level logic. Now the indermediate buffer of the bulk-loader is processed sequentially one tuple at a time. For every tuple the function in Algorithm 6 is called and the returned address is then passed to the method illustrated in Algorithm 7. For each attribute of the record this algorithm calculates the exact address location where to insert the attribute. This can be seen at line 4. The exact memory address is calculated by adding the physical offset of the respective attribute to the address passed as parameter. In the switch statement at the next line it is determined what data type the particular attribute has. This is done in two steps. First the physical schema is accessed using the attribute number (i.e., the i of the for-loop) as index. Then the value inside the physical schema is used as an index of the logical schema. In Figure 8 this daisy-chain concept is demonstrated by the dashed lines. The received data type is then used as argument for the switch statement and depending on the argument value the matching case label is executed. At line seven of the shown pseudocode a case label for the data type *char* is exemplified. The previously calculated pointer (cf. line four in Algorithm 7) is cast to *char*, dereferenced and then the respective *char*-value inside the intermediate buffer is assigned to it. Case labels for other data types work the same way. This procedure is repeated for all attributes of the record and last but not least the buffer pointer is incremented. After all tuples were inserted this way from the intermediate buffer into the memory pages, the program leaves the *bulk_ insert*-method and the database system is filled with data.

Algorithm 7: Inserting NSM tuple inside the main memory database

```
   parameter: rel: A NSM_Relation object
              address: The address where to insert the tuple
              buffer: A pointer to the intermediate buffer
 1 Function insertTuple(rel, address, buffer){
 2     byte* tempPointer
 3     for (i ← 0 to no_of_attributes){
 4         tempPointer ← (address + rel.getPhysOffset()[i])
 5         switch (rel.getLogSchema()[rel.getPhysSchema()[i]]){
 6             case (kCHAR){
 7                 *(char*)tempPointer ←
                   (buffer + rel.getPhysSchema()[i])− > c8()
 8             }
 9             case (...){
10                 ...
11             }
12         }
13     }
14     buffer ← buffer + rel.getNoAttributes()
15 }
```

4.3 PAX

In comparison to NSM, PAX requires no physical schema in terms of a physical attribute order. Also the alignment of records or attributes is already defined by the layout. Considering that in PAX only values of the same attribute are stored together, they will always align themselves if stored sequentially. The first topic to think about is how to partition a memory page into mini pages. Page partitioning is managed by the page interpreter. Attribute sizes based on which the partitioning is performed must be provided by the class **PAX_Relation**. To realize this, the method *getPartitionData* is implemented. The pseudocode for this method can be seen in Algorithm 8. A vector to store the partition data in is passed as reference and later passed to the method partitioning the page. The idea behind this algorithm is simple. For every attribute of the relation its attribute size is stored inside the passed vector and the total size is increased accordingly. After all attribute sizes are stored the total size is stored in the vector as well.

Algorithm 8: Computing the information needed for a page partition in PAX

```
   parameter: attrSizeVec: A reference to a vector to store the
                           partition data in
 1 Function getPartitionData(attrSizeVec){
 2     totalAttrSize ← 0
 3     for (i ← 0 to no_of_attributes){
 4         switch (logSchema[i]){
 5             case (logSchema[i] has 1byte){
 6                 attrSizeVec[i] ← 1
 7                 totalAttrSize ← totalAttrSize + 1
 8             }
 9             case (logSchema[i] has 2byte){
10                 attrSizeVec[i] ← 2
11                 totalAttrSize ← totalAttrSize + 2
12             }
13             case (...){
14                 ...
15             }
16         }
17     }
18     attrSizeVec[attrSizeVec.size() − 1] ← totalAttrSize
19 }
```

As in NSM a page interpreter is used for the page level logic. The **PageInterpreterPAX**-class offers methods to initialize new pages and to check whether a new record can be stored on the current page. Page partitioning can not be initiated from outside the class but is done during page initialization. Pseudocode for the page partitioning process is depicted in Algorithm 9. The vector containing the attribute sizes calculated in Algorithm 8 is passed as a parameter to the partitioning method. At line number 2, the page header is updated with the number of attributes. Besides the number of attributes the page header also contains the number of records, free space on the page and the maximum possible number of records fitting on the page. Each entry in the page header is a 2 byte integer leading to a total size of 8 bytes. At the next line the amount of free space on the page without any records is calculated as

$$PAGE\_SIZE - sizeof(header) - (noAttributes * sizeof(slot)).$$

Slots in PAX are 4 byte in size each containing a 2 byte mini page offset and a 2 byte integer indicating the attribute size stored on the mini page. Because mini pages must be properly aligned it is possible that the total number of records fitting on the page is lower than *pageSpace/recordSize*. The *totalRecords*-variable at line 5 represents this number. Inside the inner for loop a page partition with respect to *totalRecords* is computed. If the total size of all aligned minipages is greater than the available space on an empty page, *totalRecords* is consecutively decremented inside the do-while loop until a valid partitioning is found. Now only the page header needs to be updated for the page initialization to be complete.

Algorithm 9: Partitioning a PAX page into mini pages

```
   parameter: attrSizeVec: A reference to a vector storing the
                           partition data
 1 Function pagePartition(attrSizeVec){
 2     noAttributes ← attrSizeVec.size() − 1
 3     pageSpace ← (PAGE_SIZE − sizeof(header) −
       (noAttributes ∗ sizeof(slot)))
 4     recordSize ← attrSizeVec[noAttributes]
 5     totalRecords ← (pageSpace / recordSize) + 1
 6     sizeWithAlignment
 7     do{
 8         --totalRecords
 9         sizeWithAlignment ← 0
10         for (i ← 0 to noAttributes){
11             slot(i).miniPageOffset = sizeWithAlignment
12             slot(i).miniPageAttrSize = attrSizeVec[i]
13             sizeWithAlignment ← sizeWithAlignment +
               ((totalRecords ∗ attrSizeVec[i] + 7) & ∼ (uint) 0x07)
14         }
15     }
16     while (!(sizeWithAlignment <= pageSpace));
17     freeSpace ← freeSpace − (noAttributes ∗ sizeof(slot))
18     maxRecords ← totalRecords
19 }
```

To check if a given record fits onto the current page, *addNewRecord* can be called as in NSM. The difference is that no address is returned but only a integer value. If *-1* is returned a new page needs to be allocated (cf. return of address zero in NSM). Otherwise sufficient free space is available, the page

header is updated accordingly and the caller of the method can continue with copying the record into the page.

The class **BulkInsertPAX** also provides the method *bulk_ insert* to start the process of inserting data into the memory pages. This process was discussed in detail in Section 4.2 for NSM. For the most parts, it is the same for PAX and Algorithm 10 is the PAX counterpart of Algorithm 7. First major difference in PAX is the above mentioned *addNewRecord*-method. In NSM this method returns an address where to insert the record. In PAX the index at which the record can be inserted on the page is returned. Because in PAX a record has no single address we need to calculate the addresses for each attribute of a record. This is done in Algorithm 10 and explained in the following.

Algorithm 10: Inserting PAX tuple inside the main memory database

parameter: ***rel:*** A *PAX_ Relation* object
buffer: A pointer to the intermediate buffer
interpreter: A reference to an *PageInterpreterPAX* object
index: An index where to store attributes on the mini page

```
1  Function insertTuple(rel, buffer, interpreter, index){
2      byte* tempPointer
3      for (i ← 0 to no_of_attributes){
4          tempPointer ← (interpreter.pagePtr() +
           interpreter.getMiniPageOffset(i) +
           index * interpreter.getAttrSize(i))
5          switch (rel.getLogSchema()[i]){
6              case (kCHAR){
7                  *(char*)tempPointer ← buffer− > c8()
8              }
9              case (...){
10                 ...
11             }
12         }
13         ++buffer
14     }
15 }
```

As seen at line 4 there is a long expression calculating the address where to store attribute i at. It is helpful to remember Figure 6a in Section 3.4

to understand this expression. The method *getMiniPageOffset* returns an offset to the mini page where the attribute passed as parameter is stored. As one might expect the return of *getAttrSize* is the attribute size of the attribute passed as parameter. Now we can evaluate this expression step by step. First of all the mini page offset is added to the page address. The next step is to get the address where to store the attribute within the mini page. Since all attributes are stored sequentially all we have to do is multiply the attribute size with passed index. This gives us the attribute's offset inside the mini page. After this offset is added to the mini page pointer the exact memory address where to store attribute i at has been calculated. At line 5 the data type of attribute i is used as argument for the switch statement and as in NSM the matching case label is executed. The attribute's value within the intermediate buffer is then stored in the previously calculated memory address. After each inserted attribute the buffer pointer is incremented and the procedure is repeated for the remaining attributes.

5 Evaluation

5.1 Setup and Methodology

The evaluation was mainly conducted on an 2015 iMac 5K system running macOS Sierra. This computer features an Intel Core i5 (Skylake, i5-6600) processor which includes four independent cores running at 3.3GHz, 16GB of main memory, and a 8GT/s system bus. The processor has split 32-KB L1 data and instruction caches, for each core a 256-KB L2 cache, 6MB of shared L3 cache and the system has a cache line size of 64 bytes.

Other machines used for the evaluation (named after their hostname):

	fyndhorn	**olympia**	**centaurus**	**aries**
OS	Arch Linux	Arch Linux	Arch Linux	Arch Linux
CPU	Xeon E5-2620 v4, 8 cores @ 2.1 GHz	Xeon E5-2640 v3, 8 core @ 2.6 GHz	Xeon E5-2690 v2, 10 cores @ 3 GHz	Xeon X5680 6 cores @ 3.33 GHz
RAM	64GB	264GB	132GB	99GB
Bus	8GT/s	8GT/s	8GT/s	6.4GT/s
Cache	32k L1d & L1i, 256k L2, 20MB L3, 64B cache line	32k L1d & L1i, 256k L2, 20MB L3, 64B cache line	32k L1d & L1i, 256k L2, 25MB L3, 64B cache line	32k L1d & L1i, 256k L2, 12MB L3, 64B cache line

To evaluate bulk-loading, query speed and update performance, data from the TPC-H benchmark was used. The dataset of this benchmark was created with a scale factor of 1, resulting in a database size of approximately 1GB. Another test was conducted on a dataset of 10,000,000 tuples each with 100 int32 attributes. This dataset leads to a database size of 4GB. If not explicitly stated otherwise, a test was conducted on all the previously mentioned machines with a page size of 4096, 8192 and 16384 bytes respectively. A storage layout's name followed by a number in parenthesis indicates the used page size for this test (e.g. *NSM(8192)* refers to a test on the NSM database system with a used page size of 8192 bytes). The memory pages were aligned to 4096 bytes. Each test was run 15 times and then the average was calculated. Details to each test are described in their respective sub-section. To calculate the percentage PAX is faster (or slower) the following formula was used:

$$\left(\frac{Difference(NSM, PAX)}{ExecutionTime(NSM)}\right) * 100$$

Relation	NSM	PAX	Difference
Customer	0.136	0.136	0.000
Lineitem	8.020	8.221	-0.201
Nation	0.000	0.000	0.000
Orders	1.183	1.200	-0.017
Part	0.216	0.223	-0.008
PartSupp	0.627	0.634	-0.007
Region	0.000	0.000	0.000
Supplier	0.008	0.008	0.000
SUM	10.189	10.422	-0.233

Table 1: Measurement data of the bulk loading process (in sec)

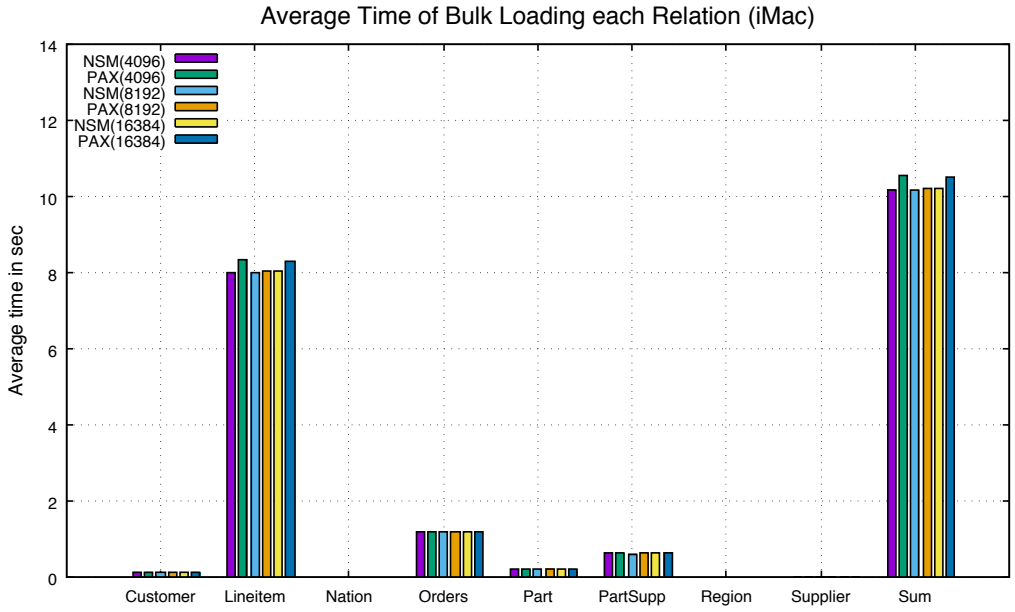


Figure 9: Plot illustrating the bulk loading performance of NSM and PAX

5.2 Bulk Load

The first test to be discussed is the measurement of bulk load performance. It was measured how NSM and PAX perform in bulk loading the complete TPC-H dataset into the database. Step two (cf. Section 4.1 *BulkInsert*) of the bulk load process was measured for each table of the TPC-H dataset and then

the sum was calculated. This was measured on every machine but the data in Table 1 shows only measurement data of the iMac system because outcomes were very similar on all machines. In the same table the average of the three page sizes can be seen and in Figure 9 a histogram with each page size is shown. Table *Nation*, *Region* and *Supplier* are so small compared to the others that they have no impact on the result. *Lineitem* is the biggest table of the TPC-H benchmark with about 6,000,000 tuples, taking approximately 80% of the total bulk load time. The overall performance of NSM and PAX are very much the same with a little advantage for NSM. A reason for this could be the slightly more complex pointer arithmetic and more function calls of PAX when inserting tuples into the database (cf. Algorithm 7 and 10).

5.3 Query

In this section a simple query executed on the *Lineitem*-table will be used for evaluation. This is depicted in Algorithm 11 in SQL syntax. The *X* in the *where*-clause ranges between 0 and 105,000 for a selectivity from 0% to 100%. The goal of this test is to measure the average query execution time for different selectivities.

Algorithm 11: Query comparing *L_EXTENDEDPRICE*-values of table *Lineitem* with different selectivity

```
SELECT   *
FROM     lineitem
WHERE    L_EXTENDEDPRICE < X;
```

In Table 2 one can see the measurement data for the fastest NSM and PAX execution time of all page sizes. NSM and PAX executed the fastest with page size 16384. At selectivity 0% and 100% the system can very effectively predict the branch of the comparison in Algorithm 11. Thereby pipeline hazards, resulting in stalls, are reduced and the CPU's prefetcher can prefetch data into the cache. NSM has a 1.6 times faster and PAX a 2.4 times faster execution at 100% selectivity compared to 50% selectivity. The slowest execution time of NSM and PAX is in selectivity range 40% to 60%. The less the system can predict branches the more branch mispredictions occur. If a branch was mispredicted the processor needs to flush the incorrect code path from the pipeline in order to resume execution at the correct location. This branch misprediction overhead is what deteriorates the performance in selectivity range 40% to 60%.

Selectivity	NSM(16384)	PAX(16384)	Difference	PAX in %-faster
0	23.84	12.12	11.72	49.15
10	25.87	16.89	8.98	34.73
20	30.22	22.19	8.03	26.58
30	35.63	27.91	7.72	21.67
40	40.33	33.40	6.93	17.19
50	42.48	33.03	9.46	22.26
60	41.32	31.35	9.98	24.15
70	37.21	25.23	11.98	32.19
80	30.82	23.51	7.31	23.72
90	26.69	18.49	8.20	30.72
100	26.67	13.80	12.87	48.26

Table 2: Query measurement data for the fastest NSM and PAX performance of all page sizes (in ms)

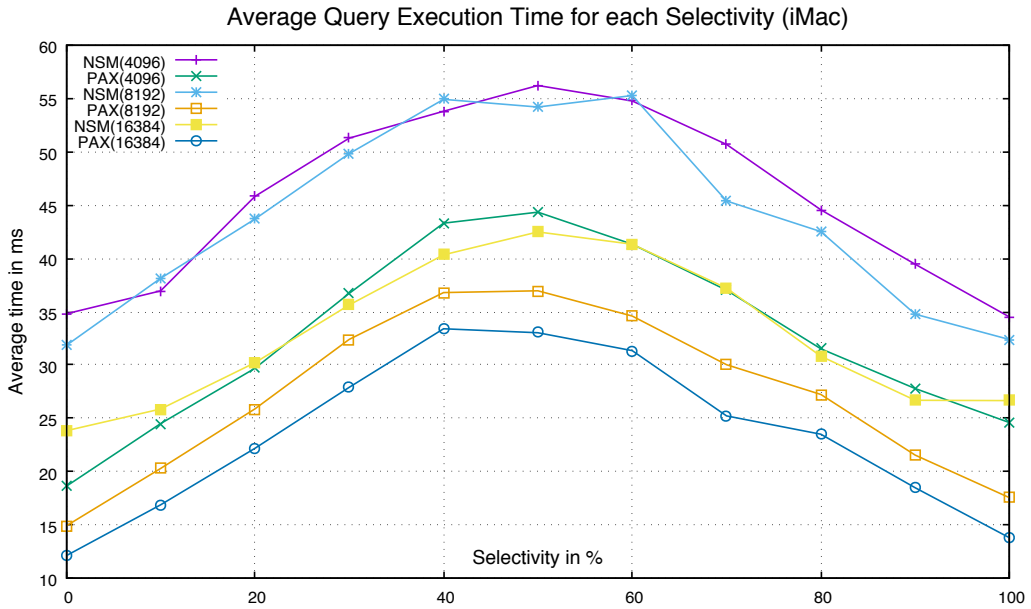


Figure 10: Performance comparison of NSM and PAX in a test query

In comparison to NSM, PAX has a clear advantage processing this query. It has a 17% to 49% faster execution depending on the selectivity. This is because of the better cache behavior of PAX (cf. Section 3.4 and Figure 6b). Figure 10 visualizes the performance difference between NSM and PAX. The slowest PAX performance with page size 4096 is still overall slightly faster

than the fastest NSM performance. In queries like the test query in Algorithm 11, PAX exhibits good cache performance because of its inter record spatial locality. With PAX multiple useful attribute values are fetched in each cache line. The *L_EXTENDEDPRICE*-attribute of relation *Lineitem* is of type *float* with 4 bytes in size. Per cache line 16 float attribute values can be fetched. In contrast, a cache line in NSM is polluted with attribute values not needed for the query. The size of one *Lineitem*-record is 62 bytes and therefore spanning a complete cache line. Previously mentioned performance increases near the very low and very high selectivity regions can be oberserved in the plot.

5.4 Update

To measure update performance of NSM and PAX the update statement in Algorithm 12 was used with a different number of attributes to change.

Algorithm 12: Update statement changing a different number of attributes in table *Lineitem*

```
UPDATE  lineitem
SET     L_ORDERKEY = value1, L_PARTKEY = value2, ...
```

The test starts with changing only one attribute of each tuple. After the performance of updating only single attribute values was measured, the number of attributes increased. This was continued until all attribute values of each tuple had to be updated. Remembering the update example discussed in Section3.2, it follows that NSM should perform better than in previous query. Because of NSM's intra-record spatial locality it does not pose a problem if several attributes of a tuple are changed. The measurement data of this test can be seen in Table 3. Again only the fastest NSM and PAX execution time of all page sizes is shown. At the start when only one attribute is changed PAX is significantly faster compared to NSM. It has a 77% faster execution time and finishes the update with a 27.66 milliseconds lead. This is significant compared to NSM's total execution time of 35.74 milliseconds but not suprising when considering PAX's properties. Its cache line is filled with relevant data when only one attribute is updated whereas in NSM the cache space is wasted with useless data incuring unnecessary cache misses. As the number of updated attributes increase the performance of PAX decreases. At twelve attributes the performance of PAX deteriorates significantly while NSM's performance remains stable. If all attributes of the relation are updated PAX performs 36% slower.

No. Attributes	NSM(16384)	PAX(16384)	Difference	PAX in %-faster
1	35.74	8.08	27.66	77.39
2	36.33	13.23	23.10	63.58
3	39.09	15.37	23.72	60.68
4	38.71	17.69	21.02	54.30
5	39.24	21.07	18.17	46.31
6	39.60	25.17	14.43	36.44
7	41.52	28.05	13.47	32.44
8	42.19	31.10	11.10	26.30
9	43.53	30.13	13.41	30.80
10	45.63	34.22	11.41	25.01
11	45.89	39.76	6.14	13.37
12	46.84	54.55	-7.71	-16.46
13	48.93	58.00	-9.07	-18.54
14	49.95	85.61	-35.66	-71.40
15	53.87	81.75	-27.88	-51.76
16	57.04	77.72	-20.68	-36.26

Table 3: Update measurement data for the fastest NSM and PAX performance of all page sizes (in ms)

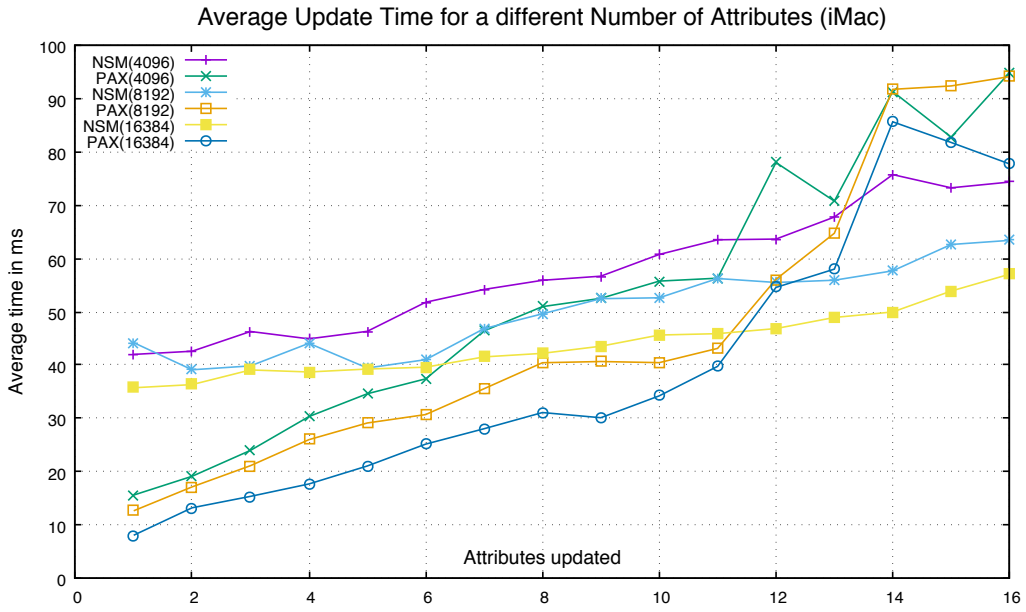


Figure 11: Performance comparison of NSM and PAX in a test update

This trend in execution time is plotted in Figure 11. In the beginning PAX is faster independent of the page size used. A steady increase in execution time for PAX can observed while NSM remains stable. In the end NSM is faster with every used page size and the performance gap would continue to increase as it can be seen in the next section. The reason for this is the same why PAX has an advantage at the beginning. With NSM's intra-record spatial locality the cache is fully utilized as soon as enough attributes are involved in the update. With PAX on the other hand the CPU has increasingly more problems to keep data it will later need in its high cache levels. A rising amount of attributes need to be accessed in low cache levels or even main memory.

5.5 Big Integer Chunk Update

The following test serves to further show the performance impact of updating a high number of attributes. For this test 10^7 tuples with 100 attributes each are stored in the database. Now the same procedure as in previous test is repeated. First only one attribute is updated, then two attributes, up to all 100 attributes. The test results are visualized in Figure 12. Note that the measurement was taken on the machine *olympia*. Because of the bigger L3 cache size of *olympia* it takes a higher number of attributes to see a clear difference. At about 25 updated attributes NSM is faster than PAX and the gap is only increasing. In the end the gap is in the range of seconds.

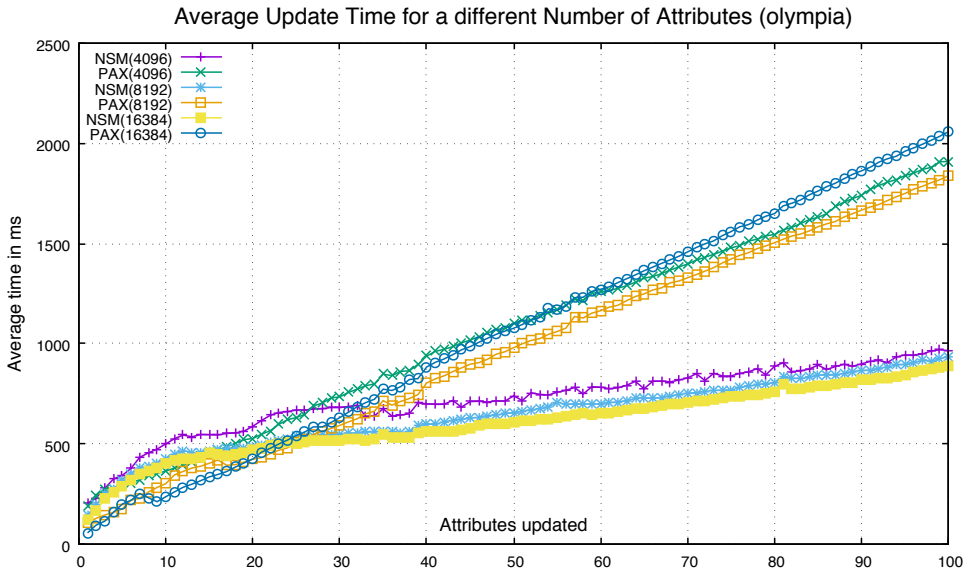


Figure 12: Difference between NSM and PAX when updating many attributes

5.6 Discussion

In this section the findings of this evaluation will be compared with the conducted evaluation in [4]. When reference to the work of Ailamaki et al. is made the term *original* is used (e.g. original paper, original implementation, and so on).

Bulk-Loading: Ailamaki et al. measured a 2% to 10% performance penalty for PAX compared to NSM in loading the TPC-H database into memory. Figure 13 shows a slightly modified version of the plot in their work. The original figure had measurement data for DSM plotted as well which was removed since the focus lies on the comparison between NSM and PAX.

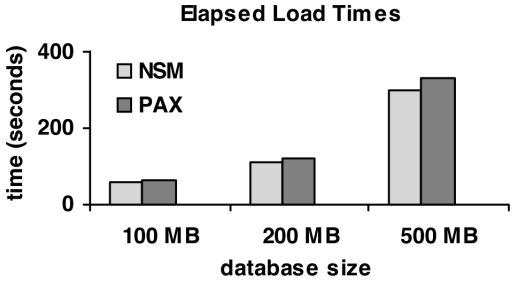


Figure 13: Elapsed bulk-loading times from the original paper (slightly modified)[4]

Measurements for three database sizes can be seen. Of course storage capacity increased by a multiple in recent years and there is no one-to-one comparison, but the performance difference between NSM and PAX is even less in the measurements discussed in Section 5.2. The reason for this could be the implemented page reorganization described in the original paper. This reorganization is needed when the original page boundaries calculated in the page partition would be exceeded by variable length attributes. If this is taken into account the results of the original measurements are very similiar to the results of this evaluation.

Queries: The used query in Section 5.3 is relatively simple compared to the queries of the TPC-H benchmark (Q1, Q6, Q12, Q14) used in the original evaluation. In Figure 14 the original results for query performance can be seen. Query six of the TPC-H benchmark comes closest to the query used in this evaluation (cf. Algorithm 11). As it can be seen in the plot this query has a speedup of about 17% to 34% over NSM. It should be noted that the used formula to calculate speedup in the original evaluation is

$$\left(\frac{ExecutionTime(NSM)}{ExecutionTime(PAX)} - 1\right) * 100$$

which is always higher than the one stated in Section 5.1. When calculated the speedup with the same data but the other formula a speedup of 14% to 25% is achieved. Again these results are a bit worse for PAX than the measurements shown in Table 2. A reason for a better performance of PAX today could be the progress in computer hardware, especially in caches. The used system in the original implementation had a 16 KB L1 cache, a shared 512 KB L2 cache and a cache line size of 32 B. All machines used for this evaluation had double the L1 cache size, an additional L3 cache and double the cache line size. This progress benefited PAX more because it has a better cache utilization than NSM.

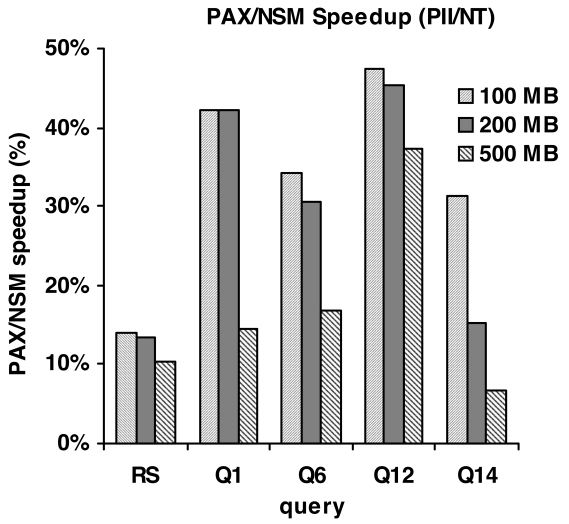


Figure 14: PAX/NSM speedup on read-only queries from the original paper (slightly modified)[4]

Updates: The major difference between the test update in this evaluation and the updates carried out in [4] is that the latter has a different number of fields in the predicate with different selectivities tested as well. In the update discussed in Algorithm 12, there was no predicate to evaluate and therefore a selectivity of 100%. A statement made by Ailamaki et al. is

> *"When executing updates PAX is always faster than NSM, providing a speedup of 10-16%"*

After evaluating the update performance of PAX in Section 5.4 and 5.5 this result could not be observed. When taking a look at update performances of the original evaluation, depicted in Figure 15, it can be seen that at most seven attributes were updated. The problem with this is that PAX has big

advantage if only a small number of attributes are changed. As described earlier with an increasing number of attributes comes decreasing performance of PAX (cf. Table 3). The example depicted in Figure 12 has shown a clear difference in performance when updating a high number of attributes.

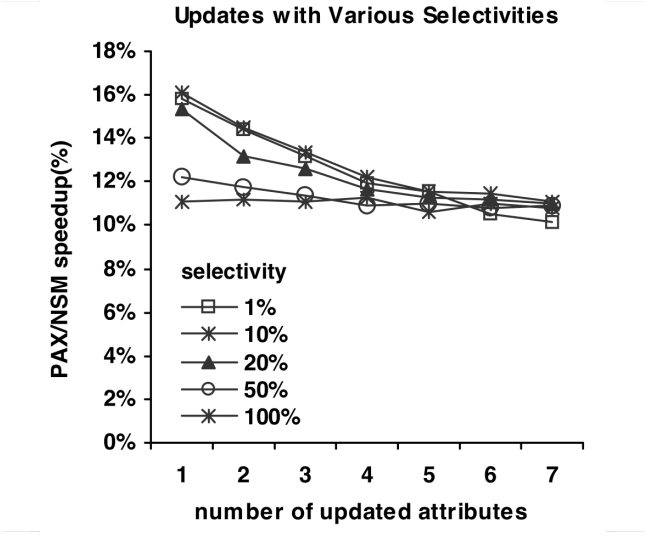


Figure 15: PAX/NSM speedup on updates from the original paper (slightly modified)[4]

6 Conclusion

6.1 Summary

All in all, this work covers the design and implementation of NSM and PAX and introduces the storage models to the reader. Some basics about computer science and database systems were introduced in order to understand subsequent sections. Before describing the storage models themselves, the history of different storage models as well as the placement of PAX in this chronology were discussed. Follow up projects continued research based on PAX. After comparing NSM with DSM and their respective strengths and weaknesses, PAX was described in more detail. PAX is an alternative data layout with the goal of combining the intra-record spatial locality of NSM with the inter-record spatial locality of DSM. Values of the same attribute are stored together on a page and all attributes of a record reside on the same page. When executing queries with a low number of attributes involved, cache resources are fully utilized as in DSM. At the same time the storage overhead of DSM and its high record reconstruction costs are avoided. This was shown and explained on examples.

Following that the design decisions around the implementation were discussed. The basic structure of the database system was described and the most important classes were explained in more detail. After getting a general overview, NSM and PAX specific implementation details were highlighted and core algorithms were presented.

Afterwards, the implentation was evaluated. Measurement data of different tests was introduced and plotted in graphs. The performance of NSM and PAX was compared in each test and the differences were analyzed and described. It was seen that bulk loading performance of NSM and PAX are similiar but with a small advantage for NSM. A possible reason for that is the more complex pointer arithmetic of PAX. PAX's advantage when running a simple query with only one comparison as predicate was observable. The reason for this is better cache behaviour of PAX with these kind of queries. In the end a comparison of update performance was undertaken. A simple update statement changing up to 16 attributes already indicated NSM's advantage as soon as multiple attributes are involved. In another update statement which changes up to 100 attributes this advantage was put into perspective even further. After evaluating the database system implemented as part of this work a comparison with the evaluation results of Ailamaki et al. was conducted. The measurements of bulk loading and query performance in the original work were similiar to the obtained results in this work's

evaluation. Update performance on the other hand differed in both evaluations. The original work stated that PAX is always faster than NSM when executing updates. This could not be confirmed as measurements showed an advantage for NSM when a certain number of attributes were updated.

6.2 Critical Review

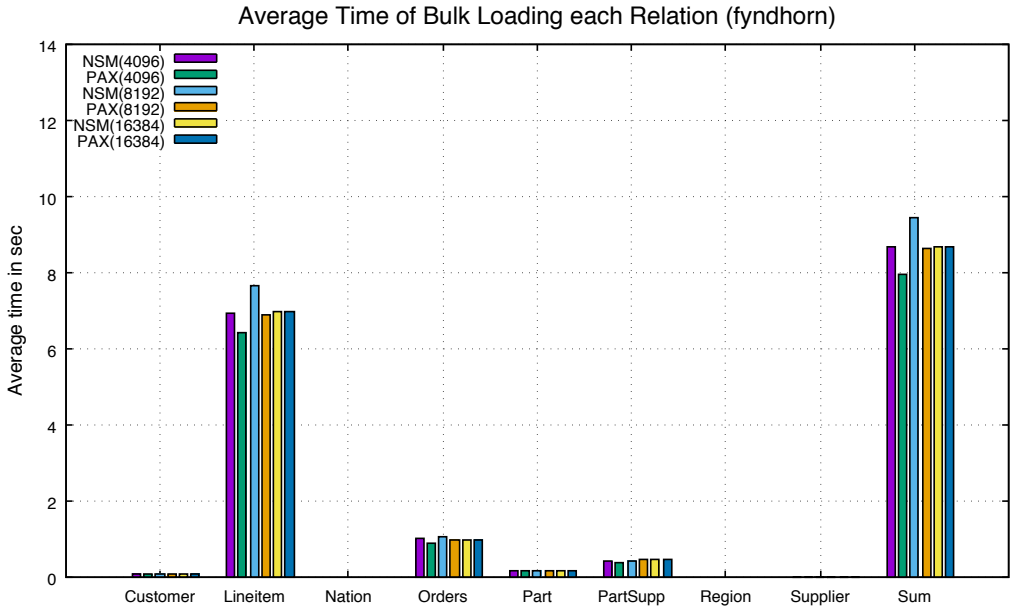
Ailamaki et al. proposed an important storage model, further advancing research progress in the field of physical data storage. The results of this work's evaluation confirmed the measurements of bulk loading and query performance in the original work. However, for update performance the results were different. Unfortunately update performance in the original evaluation was only measured for an amount of seven updated attributes. It is questionable if the performance of PAX in its original implementation would stay stable when the number of attributes updated is further increased. It would have also been interesting to see performance results for more OLTP-style workloads. The TPC-C benchmark could have been implemented to evaluate this. Also, no information about the used page size in the original evaluation is provided. Therefore, it is hard to say with which performance result (i.e., page size) a comparison should be conducted.

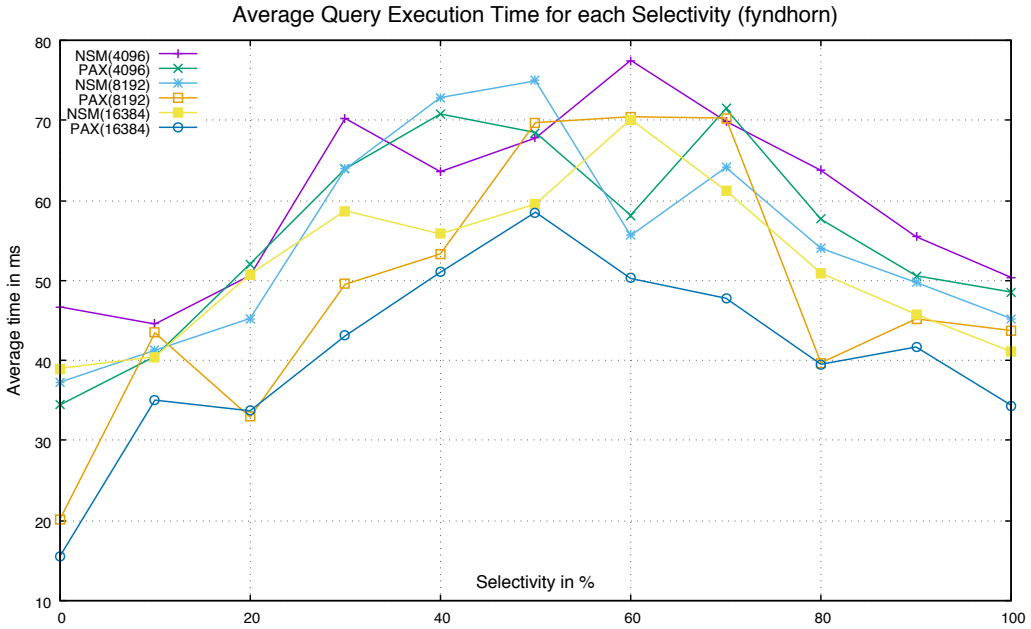
6.3 Future Work

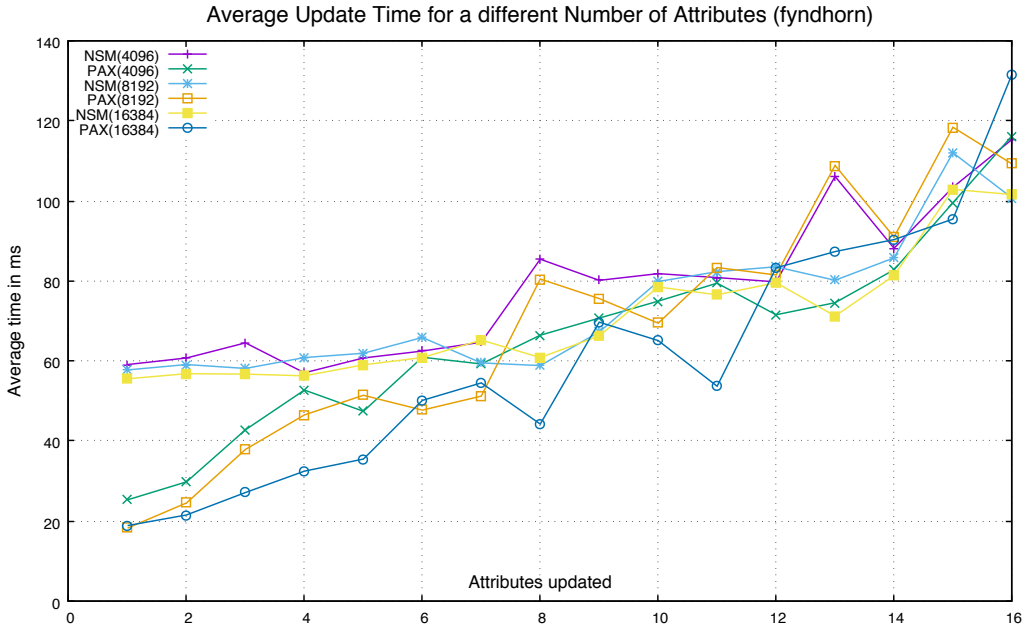
Because the only physical algebra operators implemented were scan and selection, query variety was limited. Therefore, behaviour of NSM and PAX was tested for simple queries and updates only. In order to better compare results, the same queries as used in the evaluation by Ailamaki et al. should be used. For a more realistic and comprehensive evaluation the queries of the TPC-H benchmark can be implemented. It should also be noted that the implementation was simplified by storing variable length attributes in container and only implementing inserts but no deletes. Therefore, all attribute values stored on a memory page have fixed length and no page reorganization is needed.

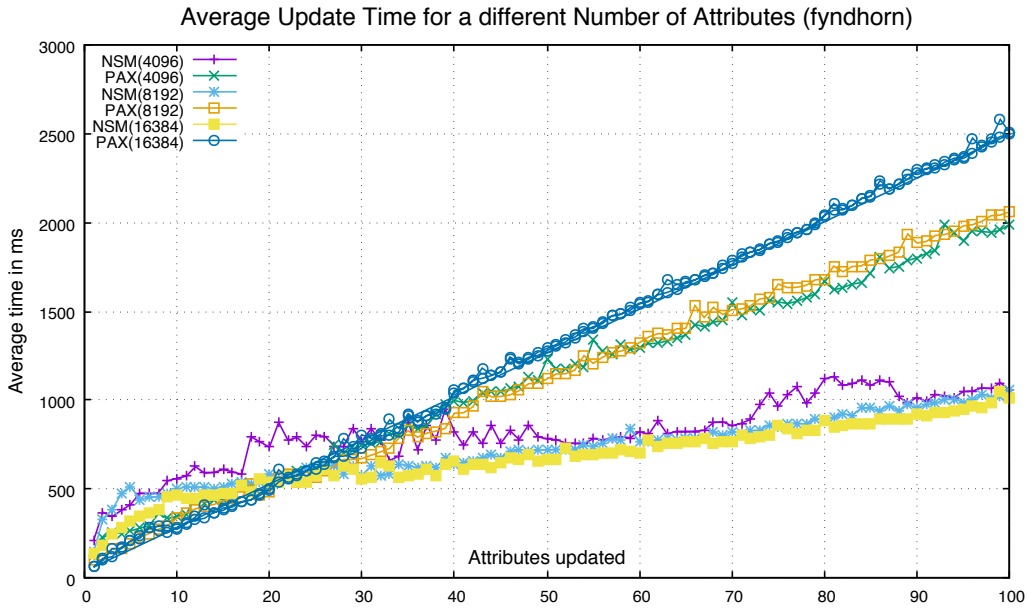
A Appendix

Fyndhorn Data Plots

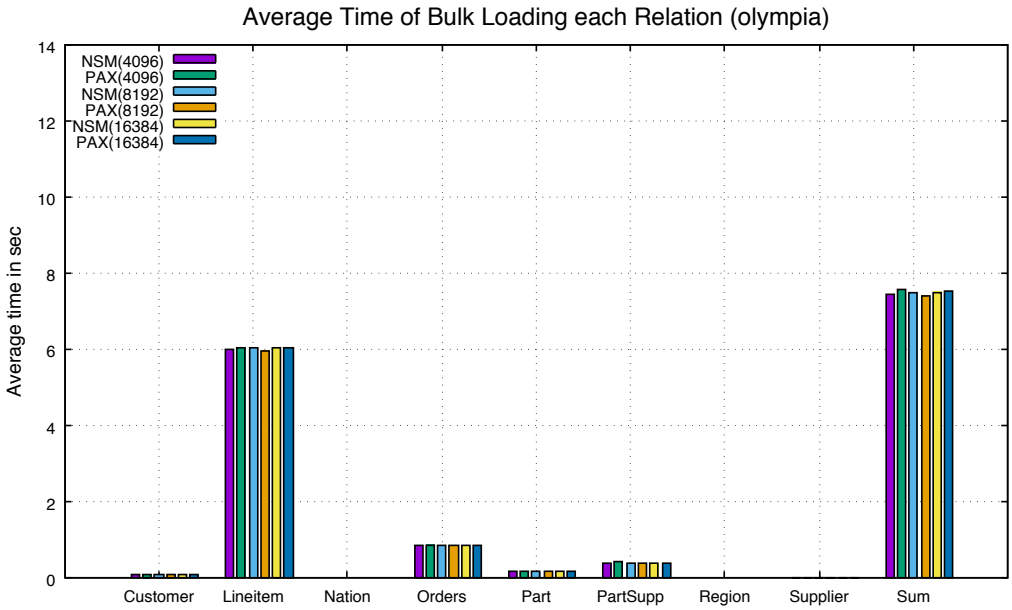


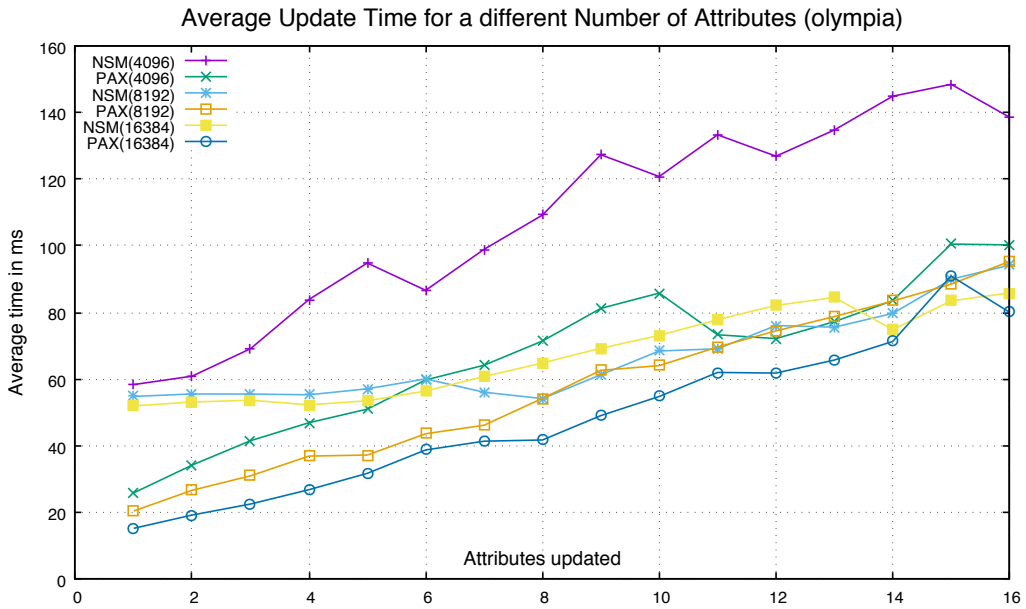


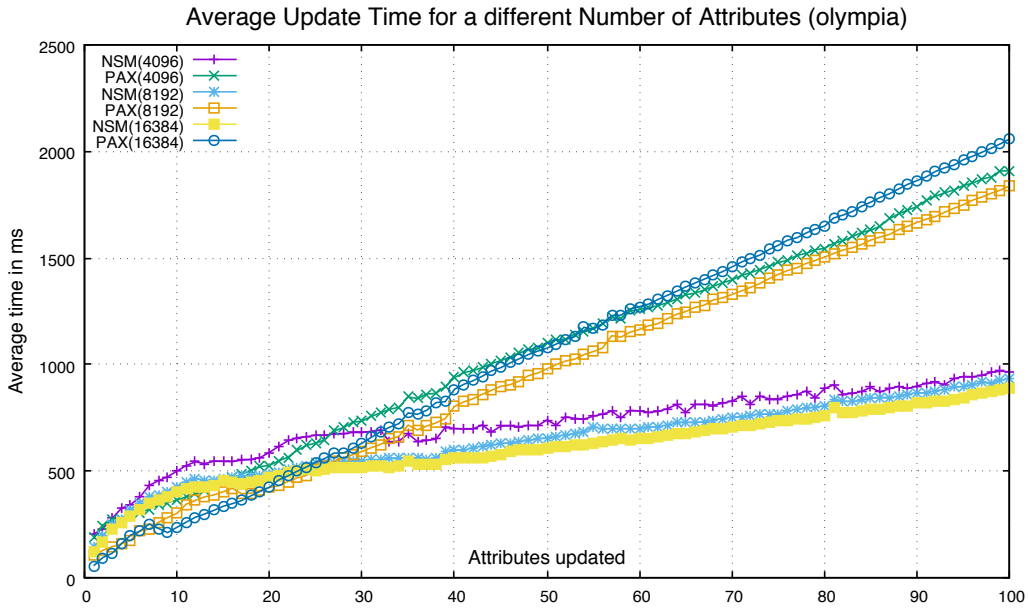

Average Update Time for a different Number of Attributes (fyndhorn)
NSM(4096)
PAX(4096)
NSM(8192)
PAX(8192)
NSM(16384)
PAX(16384)
Average time in ms
Attributes updated
0
20
40
60
80
100
120
140
0
2
4
6
8
10
12
14
16


Average Update Time for a different Number of Attributes (fyndhorn)
NSM(4096)
PAX(4096)
NSM(8192)
PAX(8192)
NSM(16384)
PAX(16384)
Average time in ms
Attributes updated
0
500
1000
1500
2000
2500
3000
0
10
20
30
40
50
60
70
80
90
100

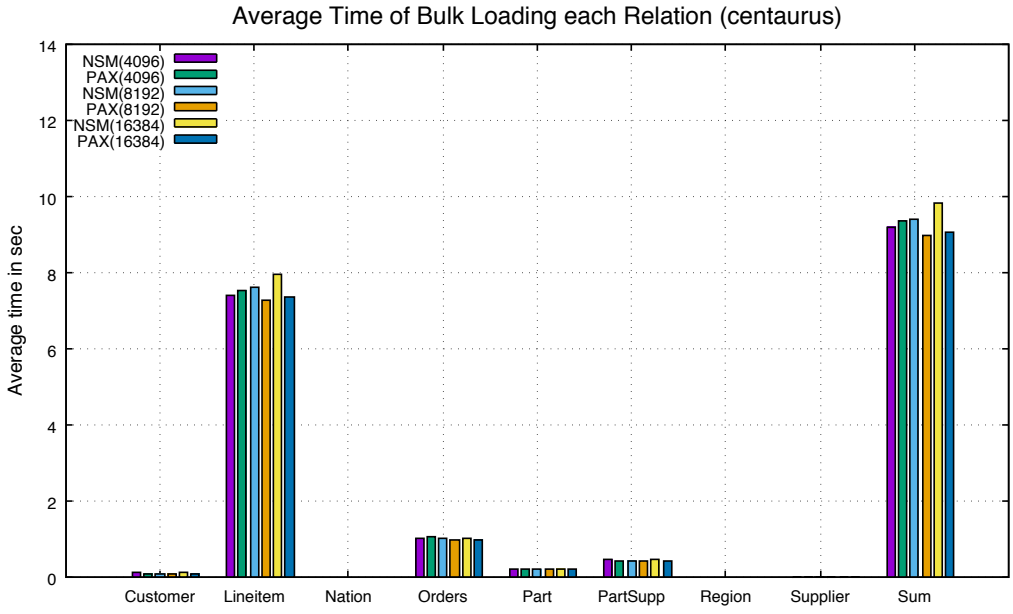
Olympia Data Plots

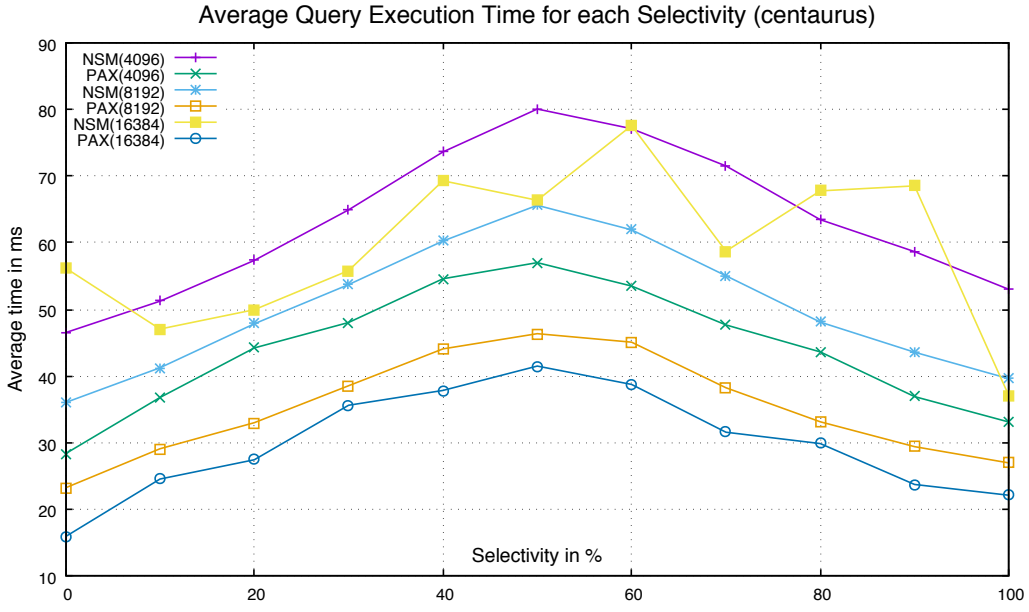


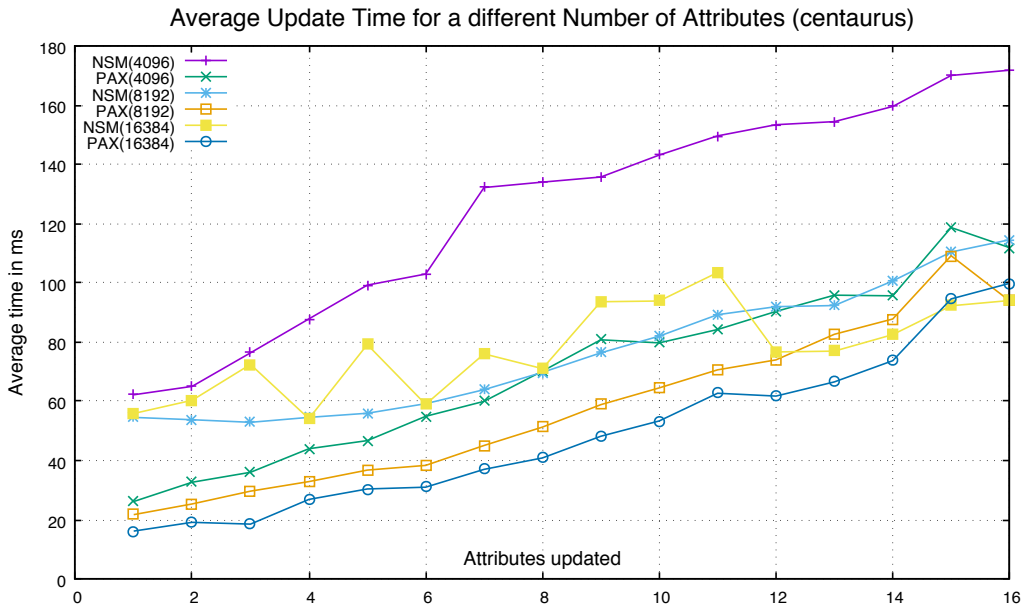

Average Update Time for a different Number of Attributes (olympia)
NSM(4096)
PAX(4096)
NSM(8192)
PAX(8192)
NSM(16384)
PAX(16384)
Average time in ms
Attributes updated
0
20
40
60
80
100
120
140
160
0
2
4
6
8
10
12
14
16

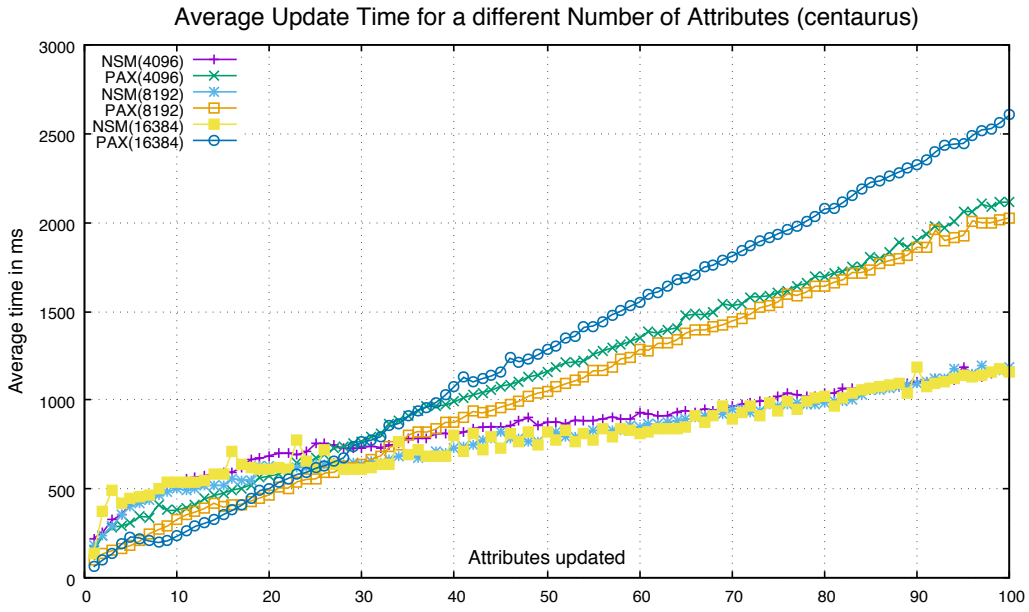

Average Update Time for a different Number of Attributes (olympia)
NSM(4096)
PAX(4096)
NSM(8192)
PAX(8192)
NSM(16384)
PAX(16384)
Average time in ms
Attributes updated
0
500
1000
1500
2000
2500
0
10
20
30
40
50
60
70
80
90
100

Centaurus Data Plots

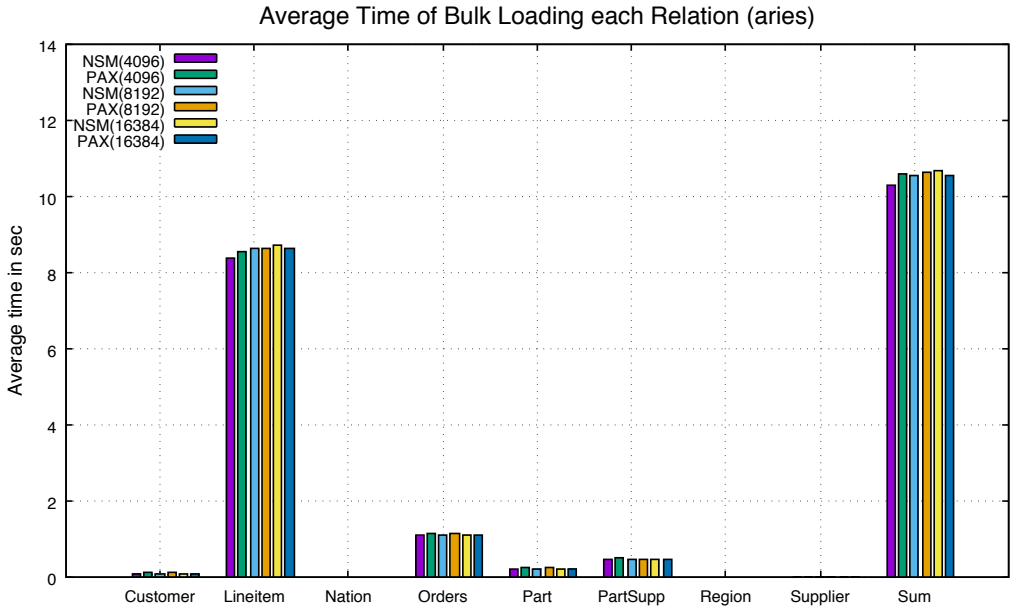


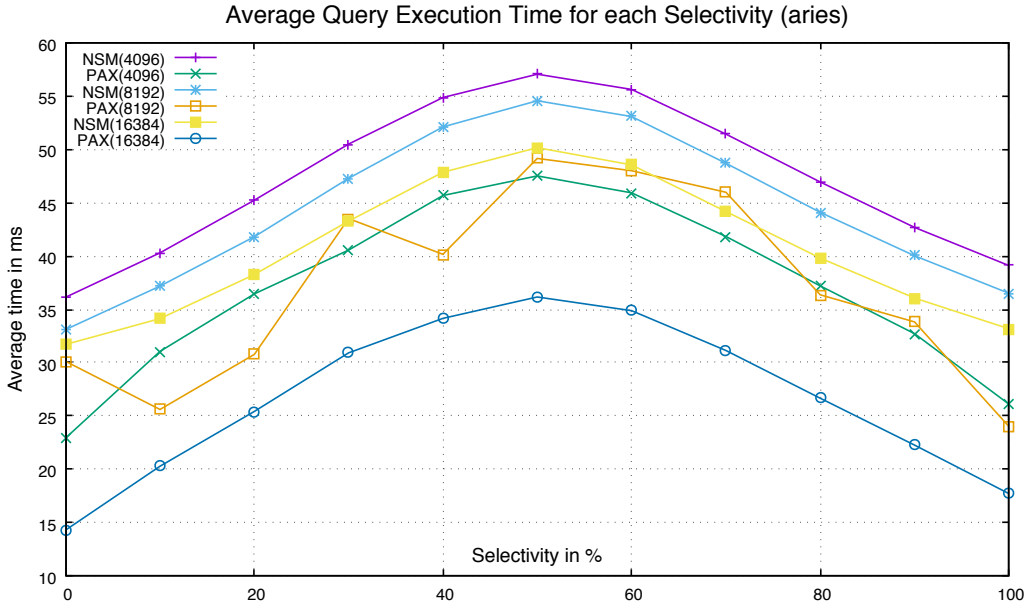


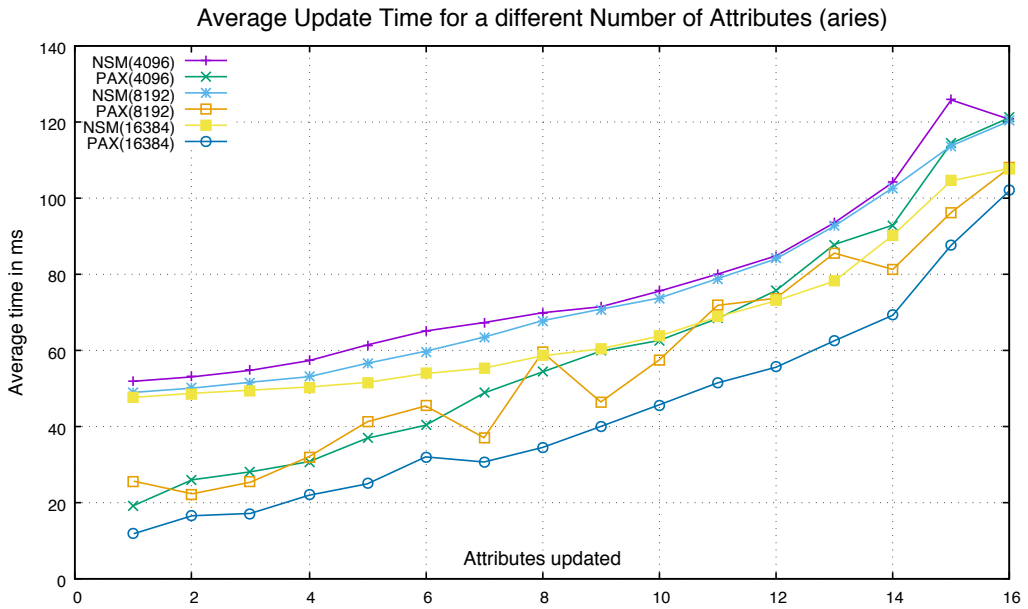

Average Update Time for a different Number of Attributes (centaurus)
NSM(4096)
PAX(4096)
NSM(8192)
PAX(8192)
NSM(16384)
PAX(16384)
Average time in ms
Attributes updated
0
20
40
60
80
100
120
140
160
180
0
2
4
6
8
10
12
14
16

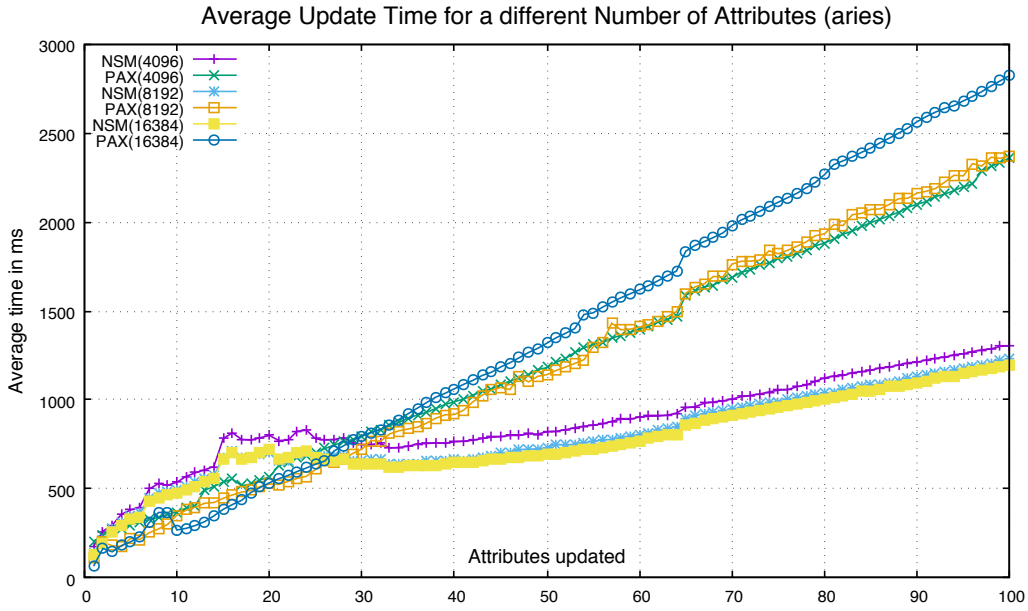

Average Update Time for a different Number of Attributes (centaurus)
NSM(4096)
PAX(4096)
NSM(8192)
PAX(8192)
NSM(16384)
PAX(16384)
Average time in ms
Attributes updated
0
500
1000
1500
2000
2500
3000
0
10
20
30
40
50
60
70
80
90
100

Aries Data Plots






Average Update Time for a different Number of Attributes (aries)
NSM(4096)
PAX(4096)
NSM(8192)
PAX(8192)
NSM(16384)
PAX(16384)
Average time in ms
Attributes updated
0
20
40
60
80
100
120
140
0
2
4
6
8
10
12
14
16


Average Update Time for a different Number of Attributes (aries)
NSM(4096)
PAX(4096)
NSM(8192)
PAX(8192)
NSM(16384)
PAX(16384)
Average time in ms
Attributes updated
0
500
1000
1500
2000
2500
3000
0
10
20
30
40
50
60
70
80
90
100

References

[1] TPC Mission. Website, January 2017 (accessed January 08, 2017). `http://www.tpc.org/information/about/abouttpc.asp`.

[2] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[3] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In Apers et al. [5], pages 169–180.

[5] Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors. *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 2001.

[6] P. Boncz. *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications*. PhD thesis, 2002.

[7] P. Boncz, W. Quak, and M. Kersten. Monet and its geographical extensions: A novel approach to high performance GIS processing. In *EDBT*, 1996.

[8] George P. Copeland and Setrag Khoshafian. A decomposition storage model. In Shamkant B. Navathe, editor, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985.*, pages 268–279. ACM Press, 1985.

[9] Richard A. Hankins and Jignesh M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *VLDB*, pages 417–428, 2003.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (4. ed.)*. Morgan Kaufmann, 2007.

[11] L. W. Hudson, R. D. Dutton, Mary Massara Reynolds, and W. E. Walden. Taxir: A biologically oriented information retrieval system as an aid to plant introduction. *Economic Botany*, 25(4):401–406, 1971.

[12] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 10., aktualisierte und erweiterte Auflage*. De Gruyter Oldenbourg, 2015.

[13] Minglong Shao, Jiri Schindler, Steven W. Schlosser, Anastassia Ailamaki, and Gregory R. Ganger. Clotho: Decoupling memory page layout from storage organization. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 696–707. Morgan Kaufmann, 2004.

[14] Juliusz Sompolski, Marcin Zukowski, and Peter A. Boncz. Vectorization vs. compilation in query execution. In Stavros Harizopoulos and Qiong Luo, editors, *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011*, pages 33–40. ACM, 2011.

[15] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 553–564. ACM, 2005.

[16] Marcin Zukowski and Peter A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.

Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Bachelorarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
Ich bin mir bewußt, dass eine falsche Erklärung rechtliche Folgen haben wird.

Mannheim, 5. Februar 2017

Nick Weber