DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing

Marcin Zukowski Niels Nes Peter Boncz

CWI, Amsterdam, The Netherlands
{Firstname.Lastname}@cwi.nl

ABSTRACT

Comparisons between the merits of row-wise storage (NSM) and columnar storage (DSM) are typically made with respect to the persistent storage layer of database systems. In this paper, however, we focus on the CPU efficiency tradeoffs of tuple representations inside the query execution engine, while tuples flow through a processing pipeline. We analyze the performance in the context of query engines using so-called "block-oriented" processing – a recently popularized technique that can strongly improve the CPU efficiency. With this high efficiency, the performance trade-offs between NSM and DSM can have a decisive impact on the query execution performance, as we demonstrate using both microbenchmarks and TPC-H query 1. This means that NSM-based database systems can sometimes benefit from converting tuples into DSM on-the-fly, and vice versa.

1. INTRODUCTION

As computer architecture evolves, and the "make the common case fast" rule is applied to more and more CPU features, the efficiency of an application program can no longer be measured by the number of instructions it executes, as instruction throughput can vary enormously due to many factors, among which: *(i)* CPU cache and TLB miss ratio, resulting from the data access patterns; *(ii)* the possibility of using SIMD operations (e.g. SSE) to process multiple data items with one instruction; *(iii)* the average amount of in-flight instructions unbound by code- or data-dependencies, thus available to keep the instruction pipelines filled.

While such factors and their (significant) impact on performance may be well-understood, even in the specific context of data management tasks, and a range of so-called *architecture-conscious* query processing algorithms has been proposed, our goal is to investigate how such ideas can be integrated in real database systems. Therefore, we study how architectural-conscious insights can be integrated into the (typical) architecture of query engines.

The central question addressed in this research is how tuple layout in a *block-oriented* query processor impacts performance. This work is presented from the context of the MonetDB/X100 prototype [5], developed at CWI. MonetDB/X100 uses the standard open-next-close iterator execution model, but its most notable characteristic is the pervasive use of block-oriented processing [15, 5], under the moniker "vectorized execution". In block-oriented processing, rather than processing a single tuple per next() call, in each iteration the operator returns a *block* of tuples. This block can contain from a few tens to hundreds of tuples, thereby striking middle ground between tuple-at-a-time processing and full table materialization. Typically, performance increases with increasing block size, as long as the the cumulative size of tuple-blocks flowing between the operators in a query plan fits in the CPU cache. The main advantage of block-oriented processing is a reduction in the amount of method calls (i.e., query interpretation overhead). Additional benefit comes from the fact that the lowest level *primitive* functions in the query engine now expose independent work on multiple tuples (arrays of tuples). This can help compiler and CPU – and sometimes the algorithm designer – to achieve higher efficiency at run-time.

While MonetDB/X100 is known as a column-store[1], our focus here is not persistent storage, rather the representation of tuples as they flow through a block-oriented query processing engine, which can be different from the storage format. In particular, we experiment with both horizontal tuple layout (NSM) and vertical layout (DSM) and also discuss indirect value addressing (to avoid tuple copying).

Our main research questions are: *(i)* what are the advantages and disadvantages of DSM and NSM for tuple representations during query execution? *(ii)* what specific opportunities and challenges arise when considering tuple layout in the context of block-oriented processing (SIMD, prefetching, block size)? *(iii)* can query executors be made to work on both representations, and allowed to (dynamically) switch between them, given that depending on the situation, and even depending on the query sub-expression, either DSM or NSM can be better?

1.1 Outline and Findings.

In Section 2 we first describe the NSM and DSM layouts considered. Section 3 starts with a number of microbenchmarks contrasting the behavior of DSM and NSM in sequential and random-access algorithms. DSM is significantly faster in sequential scenarios thanks to simpler ad-

[1]In fact, it also supports the hybrid PAX layout [4].

Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008), June 13, 2008, Vancouver, Canada.

dressing, larger vector sizes fitting in the L2 cache, whereas the higher spatial locality of NSM makes it the method of choice when operators access memory areas larger than the L1 cache randomly. SIMD instructions give an advantage to DSM in all sequential operators such as Project and Select, whereas in Aggregation only NSM allows to exploit SIMD (in some cases). Therefore, ideally, a query processing engine should be able to operate on data in both formats, even allowing tuple blocks where some columns are in NSM, and others in DSM. Further micro-benchmarks demonstrate that thanks to block-oriented processing, converting NSM tuples into DSM (and vice versa) can be done with high efficiency.

The consequences of these findings can be startling: we show in case of TPC-H query 1, that systems with NSM storage turn out to benefit from converting tuples on-the-fly to DSM, *pulling up* the selection operator to achieve SIMDized expression calculation, then followed by conversion back into NSM, to exploit SIMD Aggregation.

We wrap up by discussing related work in Section 4 and outlining conclusions and future work in Section 5.

2. TUPLE REPRESENTATIONS

To analyze different aspects of the DSM and NSM data organization, for the experiments presented in this paper we try to isolate the actual data access functionality from unnecessary overheads. This is achieved with following the block-oriented execution model, and analyzing the computationally simplest DSM and NSM data representations, presented in this section.

2.1 DSM tuple-block representation

Traditionally, the Decomposed Storage Model [9] proposed for each attribute column to hold two columns: a *surrogate* (or *object-id*) column and a *value* column. Modern column-based systems [5, 16] choose to avoid the former column, and use the natural order for the tuple reorganization purposes. As a result, the table representation is a set of binary files, each containing consecutive values from a different attribute. This format is sometimes complicated e.g. by not storing NULL values and other forms of data compression [21, 1]. In this case, some systems keep the data compressed for some part of the execution [1], and some perform a fully-transparent decompression, providing a simple DSM structure for the query executor [21]. Here, we choose a straightforward DSM representation, with columns stored as simple arrays of values. This results in the following simple code to access a specific value in a block:

```
value = attribute[position];
```

2.2 Direct vs. Indirect Storage

Variable-width datatypes such as strings cannot be stored directly in arrays. A solution is to represent them as memory pointers into a heap. In MonetDB/X100, a tuple stream containing string values uses a list of heap buffers that contain concatenated, zero-separated strings. As soon as the last string in a buffer has left the query processing pipeline, the buffer can be reused.

Indirect storage can also be used to reduce value copying between the operators in a pipeline. For instance, in MonetDB/X100, the Select operator leaves all tuple-blocks from the data source operator intact, but just attaches an array

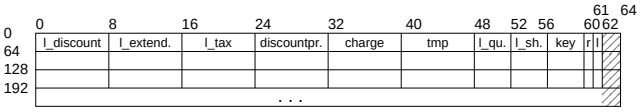

Figure 1: Diagram of the access-time optimized NSM data organization during computation of TPC-H Query 1

of selected offsets, called the *selection vector*. All primitive functions support this optional index array:

```
value = attribute[selection[position]];
```

Other copy-reduction mechanisms are also possible. For instance, MonetDB/X100 avoids copying result vectors altogether if an operator is known to leave them unchanged (i.e. columns that just pass through a Project or the left side of an N-1 Join). Note that the use of index arrays (selection vectors) is not limited to the Select operator. Other possibilities include e.g. not copying the build-relation values in a HashJoin, but instead storing references to them. In principle each column could have a different (or no) selection vector, which brings multiple optimization opportunities and challenges. In this paper, however, we focus on a simple, direct data storage.

2.3 NSM tuple-block representation.

Typically, database systems use some form of a slotted page for the NSM-stored tuples. The exact format of the tuples in this model can be highly complex, mostly due to storage considerations. For example, NULL values can be materialized or not, variable-width fields result in non-fixed attribute offsets, values can be stored explicitly or as references (e.g. dictionary compression or values from a hash table in a join result). Even fixed-width attributes can be stored using variable-width encoding, e.g. length encoding [17] or Microsoft's Vardecimal Storage Format [3].

Most of the described techniques have a goal of reducing the size of a tuple, which is crucial for disk-based data storage. Unfortunately, in many cases, such tuples are carried through into the query executor, making the data access and manipulation complex and hence expensive. In traditional tuple-at-a-time processing, the cost of accessing a value can be an acceptable compared to other overheads, but with block processing handling complex tuple representations can consume the majority of time.

To analyze the potential of NSM performance, we define a simple structure for holding NSM data, that results in a very fast access to NSM attributes. Figure 1 presents the layout of the tuples used for the processing of TPC-H Q1, visualized in Figure 4, and analyzed in Section 3.3. Tuples in a block are stored continuously one after another. As a result, tuple offset in a block is a result of the multiplication of the tuple width and its index. The attributes are stored in an order defined by their widths. Assuming attributes with widths of power of 2, this makes every value naturally aligned to its datatype within the tuple. Additionally, the tuple is aligned at the end to make its width a multiple of the widest stored attribute. This allows accessing a value of a given attribute at a given position with this simple code:

```
value = attribute[position * attributeMultiplier];
```

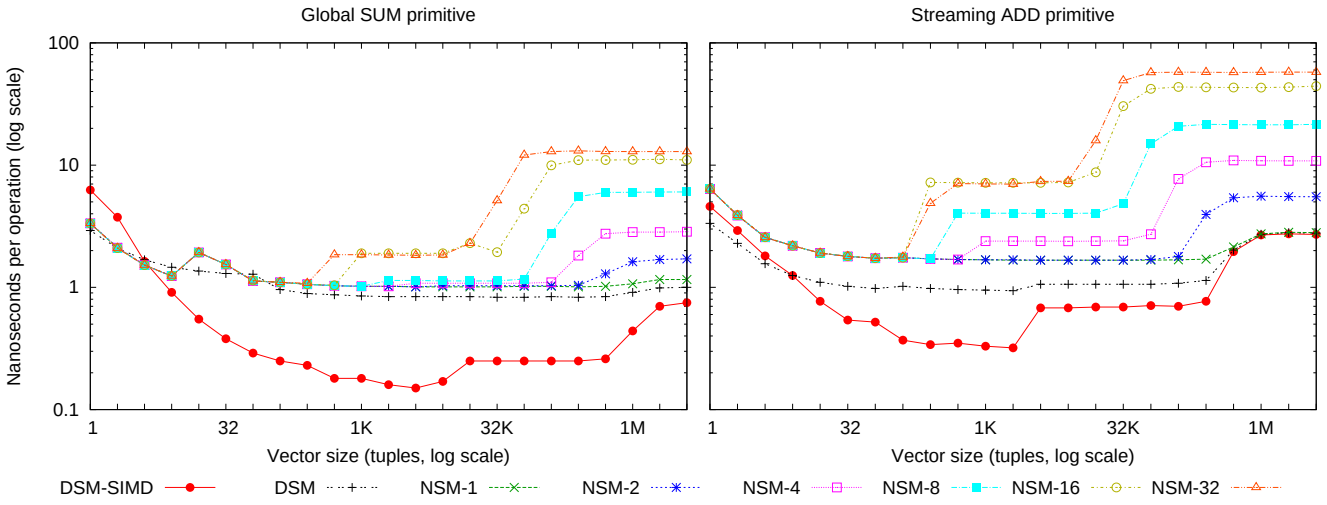

Figure 2: Sequential access: performance of the SUM and ADD routines with DSM and NSM and varying tuple widths.

3. EXPERIMENTS

In this section we analyze the performance of NSM and DSM data organization schemas on database performance. We start with a series of micro-benchmarks, presenting the baseline performance on some basic data access and manipulation activities. Then we demonstrate how these microbenchmark results are confirmed during the processing of the TPC-H Query 1. There, we also discuss some optimization techniques, that depend heavily on data organization, as well as on-the-fly data conversion.

3.1 Experimental setup

The experimental platform used is a Core2 Quad Q6600 2.4GHz with 8GB RAM running Linux with kernel 2.6.23-15. The per-core cache sizes are: 16KB L1 I-cache, 16KB L1 D-cache, 4MB L2 cache (shared among 2 cores). All experiments are single-core and in-memory. We used 2 compilers, GCC 4.1.2[2] and ICC 10.0[3].

We have performed similar experiments on a set of other CPUs: Athlon 64 X2 3800+ (2GHz), Itanium 2 and Sun Niagara. For Athlon and Itanium the results were mostly in line with the Core2 results. On Niagara the performance benefit of DSM was typically higher, and the impact of the data location was lower. This is caused by lower performance of Niagara in terms of sequential execution: it has a lower clock speed and in-order execution pipeline. Since Niagara was designed with multi-threaded processing in mind, it would be interesting to see how the presented, currently single-threaded, benchmarks perform when running in parallel. This might be a topic for future research.

3.2 Microbenchmarks

In this section we analyze the baseline performance of the DSM and NSM models in typical data-processing operations: sequential computations, random-access, and data copying.

3.2.1 *Sequential data access*

The left part of Figure 2 present the results of the experiment in which a SUM aggregate of a 4-byte integer column is computed repeatedly in a loop over a fixed dataset. The size of the data differs, to simulate different block sizes, which allows identifying the impact of the interpretation overhead, as well as the location (cache, memory) in block-oriented processing. We used GCC, using standard processing, and additionally ICC to generate SIMD-ized DSM code (NSM did not benefit from SIMD-ization). In the NSM implementation, we use tuples consisting of a varying number of integers, represented with *NSM-x*.

To analyze the impact of the data organization on CPU efficiency, we look at the performance of *NSM-1*, which has exactly the same memory access pattern and requirements as the DSM implementation. The GCC results show that for a single-integer table the performance of the DSM and NSM-1 is very close. The small benefit of DSM, ca. 15% in the optimal case, comes from the fact that thanks to a simpler data access the compiler is able to generate slightly more efficient code. However, with ICC-generated SIMD instructions, DSM is a clear winner, being almost 5 times faster in the optimal case. Note that SIMD can only be applied if the same operation is executed on adjacent memory locations, therefore it can only be used in DSM.

The other aspect of this benchmark is the impact of the interpretation overhead and data location. While for small block sizes the performance is dominated by the function calls[4], for larger sizes, when the data does not fit in the L1 cache anymore, the data location aspect becomes crucial. Performance of NSM-1 and DSM without SIMD is relatively flat, since even for main-memory sized data (1M+ tuples), the sequential bandwidth is close enough to balance the CPU activity. However, with the highly efficient (sub-cycle cost) SIMD DSM implementation, it operates fastest while the block still fits in the L1 CPU cache, then goes to an intermediate plateau when its fits L2, to become mem-

[2]compilation: `gcc -O6 -Wall -g -mtune=core2`
[3]compilation: `icc -O3 -Wall -axT`

[4]In a real DBMS, function call overhead is significantly larger [5] – this was a hard-coded micro-benchmark.

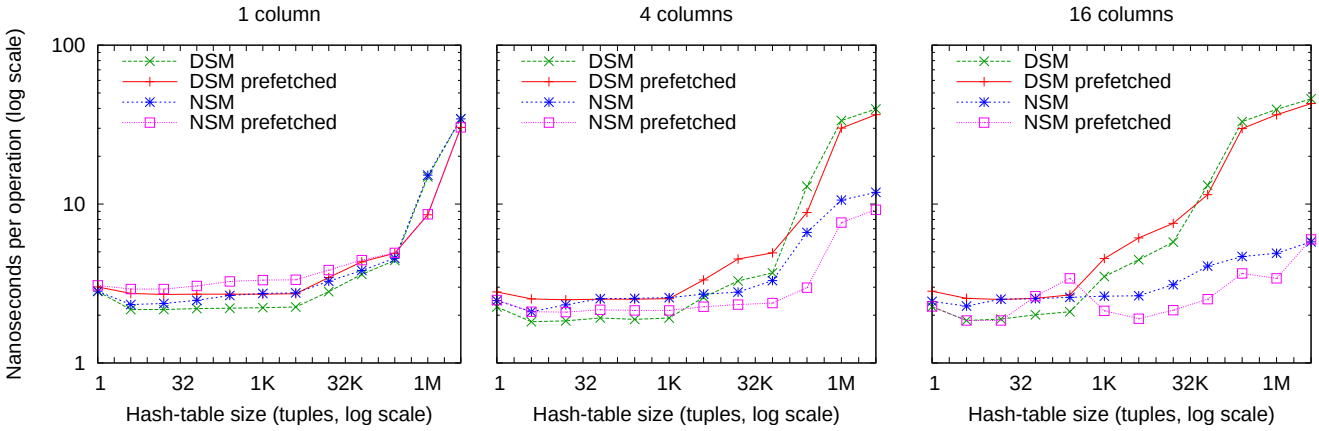

Figure 3: Random access: running a grouped SUM aggregate on DSM input data, using a DSM or NSM hash table, with or without prefetching, and a varying number of GROUP BY keys (X-axis)

ory bandwidth limited for larger sizes (i.e. L1 bandwidth exceeds L2 bandwidth which exceeds RAM bandwidth).

Looking at the performance of wider NSM tuples, we see that the performance degrades with the increasing tuple width. As long as the tuples are in L1, all widths are roughly equal. However, for NSM-16 and higher (64 byte tuples or longer) once the data shifts to L2, the impact is immediately visible. This is caused by the fact, that only a single integer from the entire cache-line is used. For NSM-2 to NSM-8, the results show that the execution is limited by the L2 bandwidth: when a small fraction of a cache-line is used (e.g. NSM-8) the performance is worse than when more integers are touched (e.g. NSM-2). Similar behavior can be observed for the main-memory datasets.

The SUM primitive has a relatively low memory demand compared to the CPU activity, as it only consumes a single attribute. The right part of Figure 2 presents a similar experiment, that uses an ADD routine which consumes two attributes and produces a new result attribute. Results follow the trends of the SUM operation, but there are some important differences. First, the higher number of parameters passed to the NSM routine (pointers + tuple widths VS only pointers) results in a higher interpretation overhead. Secondly, comparing DSM and NSM-1 for L1-resident data, shows that multiple more complex value-access computations in NSM have a higher impact on the CPU performance. Finally, with a higher memory demand, the impact of data locality on performance is significantly bigger, making even the DSM implementation fully memory-bound and in par with the NSM-1 version.

Concluding, we see that if access is purely sequential, DSM outperforms NSM for multiple reasons. First, the array-based structure allows simple value-access code. Second, individual primitive functions (e.g. SUM,ADD) use cache lines fully in DSM, and L2 bandwidth is enough to keep up. As mentioned before, during query processing, all tuple blocks in use in a query plan should fit the CPU cache. If the target for this is L2, this means significantly larger block sizes than if it were L1, resulting in reduced function call overhead. Finally, the difference in sequential processing between DSM and NSM can be huge if the operation is expressible in SIMD, especially when the blocks fit in L1, and is still significant when in L2.

3.2.2 *Random data access*

For this experiment, we use a table that consists of a single *key* column and multiple *data* columns. The table contains 4M tuples, is stored in DSM for efficient sequential access, number of *data* columns varies, and the range of the *key* column differs from 1 to 4M. We perform an experiment equivalent to this SQL query:

```
SELECT SUM(data1), ..., SUM(dataN)
FROM TABLE GROUP BY key;
```

To store the aggregate results, we use an equivalent of a hash-table in the hash-aggregation, but instead of the hash-value processing, we use the *key* column as a direct index. In DSM, the result table it is just a collection of arrays, one for each *data* attribute. In NSM, it is a single array of a size equal to the number of tuples multiplied by the number of *data* attributes. We apply block-oriented processing, using a block size of 256 tuples. In each iteration, all values from different *data* attributes are added to the respective aggregates, stored at the same vertical position in the table.

Figure 3 presents the results of this experiment for 1, 4 and 16 *data* columns. For a single column, the faster access code of the DSM version makes it slightly (up to 10%) faster than NSM as long as the aggregate table fits in the L1 cache. Once it enters L2 or main memory, the results of DSM and NSM are equal as they are memory-latency limited.

For wider tuples, DSM maintains its advantage for L1-based *key* ranges. However, once the data expands into L2 or main-memory, the performance of DSM becomes significantly worse than that of NSM. This is caused by the fact, that in DSM every memory access is expected to cause a cache-miss. In contrast, in NSM, it can be expected that a cache-line accessed during processing of one *data* column, will be accessed again with the next *data* column in the same block, as all the columns use the same key position. With the increasing number of computed aggregates, the same cache-line is accessed more often, benefiting NSM over DSM.

Figure 3 also shows experiments that use software prefetching, that is, we interspersed SUM computations with explicit prefetch instructions on the next tuple block. On Core2 we used the `prefetcht0` instruction. We also made sure the ag-

gregate table was stored using large TLB pages, to minimize TLB misses. In general, our experience with prefetching is that it is highly sensitive to the platform, and prefetch distances are hard to get right in practice. The end result is that prefetching does improve NSM performance when the aggregate table exceeds the CPU caches, however in contrast to [8] we could not obtain a straight performance line (i.e. hide all memory latency).

These simple random and sequential DSM vs. NSM micro benchmarks echo the debate on cache-conscious join and aggregation between partitioning and prefetching. In partitioning [7], the randomly accessed (e.g. aggregate) table is partitioned into chunks that fit the L1 cache. This table can be stored in DSM and probed very efficiently. The disadvantage of this approach is the cost of partitioning, possibly needing multiple sequential passes to achieve a good memory access pattern. The alternative is to have a NSM hash table exceed the CPU cache sizes, and pay a cache miss for each random probe. Unlike DSM, where random access generates a huge bandwidth need that cannot be sustained using prefetching, random probing in NSM benefits from prefetching.

In the following, we study on-the-fly conversion between DSM and NSM tuples. Using tuple conversion, it would e.g. become possible for a DSM-based system like MonetDB/X100 to use NSM (and prefetching) inside certain random-access query processing operators.

3.2.3 Data conversion

Many column stores use a traditional query processor based on NSM, calling for on-the-fly format conversion during the Scan operator. In case of C-Store [16][5], this is done using (slow) tuple-at-a-time conversion logic. We rather perform conversion using block-oriented processing, avoiding loop and function call overhead, where a single function call copies all values from one column in a block of NSM tuples into DSM representation (and vice versa):

```
NSM2DSM(int input[n], int width) : output[n]
 for(pos=0; pos<n; pos++]
  output[pos] = input[pos * width]
```

We performed micro-benchmarks, in which an NSM/DSM layout conversion is performed for datatypes of different widths. Table 1 shows that this can be done very efficiently, typically below 1 nanosecond per data value (ca. 2 CPU cycles on our test machine).

Therefore, given the different strengths and weaknesses of DSM and NSM, it becomes conceivable for a query optimizer to select the most appropriate storage format for certain sub-expressions in the query plan, and insert conversion operators to change the representation on-the-fly, potentially even multiple times. This could even lead to a situation where a query processing operator gets some input columns in DSM, and some in NSM (and the same for produced columns), similar as the persistent data is organized in the data-morphing technique [12].

We also measured the performance of copying a full NSM tuple into a different NSM representation. Such situation can occur e.g. during the merge join, where rows from both inputs need to be combined[6]. In this situation there are two choices: value-by-value copying and full-tuple copying (e.g. with `memcpy` equivalent). In tuple-at-a-time processing, the first choice will be typically significantly slower, due to high overheads of function calls and attribute-list iteration. However, Table 1 shows that in block-oriented processing, with the overheads amortized over a set of tuples, value-by-value copying can be very efficient. Full tuple copying, while fast, still suffers from overheads, as seen with a minimal performance difference between copying 8-byte and 128-byte wide tuples. As a result, for many types of tuples, attribute-by-attribute copying can be more efficient. This is especially useful, if copying includes only a subset of attributes, or if the field order in the result tuple needs to be different than in the source.

Data unit	conversion speed (ns / operation)		
	NSM⇒DSM	DSM⇒NSM	NSM⇒NSM
8-byte tuple, block size 1024			
1-byte char	0.85	0.85	0.65
4-byte int	0.74	0.66	1.06
full tuple	-	-	8.42
16-byte tuple, block size 512			
1-byte char	0.86	0.87	0.67
4-byte int	0.82	0.65	1.07
full tuple	-	-	9.62
32-byte tuple, block size 512			
1-byte char	0.93	0.87	0.72
4-byte int	0.79	0.71	1.11
full tuple	-	-	9.76
64-byte tuple, block size 256			
1-byte char	0.90	0.90	1.57
4-byte int	0.89	0.74	1.62
full tuple	-	-	10.09
128-byte tuple, block size 128			
1-byte char	0.97	0.90	1.43
4-byte int	0.87	0.76	1.45
full tuple	-	-	13.17

Table 1: Data conversion speed for different tuple widths and different conversion units. For each tuple width, the best block size was chosen.

3.3 TPC-H Query 1

To see the impact of data organization in a more realistic scenario, we have evaluated the performance of the TPC-H Query 1 with different settings. A sketch containing the main primitives used in its query plan is presented in Figure 4. For simplicity, it does not include selection computation, connections between `count` and `sum` primitives to the aggregates table, and post-processing of the aggregate results. As Figure 4 shows, the computation in Query 1 consists mostly of 2 phases: sequential computation of input for aggregation, and random-access computation of aggregates. The microbenchmarks presented above suggest, that the best data organization for the first phase is DSM, and for the second phase it is NSM.

In this query plan we exploit the fact that `l_returnflag` and `l_linestatus` are `char` datatypes. This makes the possible key combinations limited to 65536 values (in fact, there are only 4 used). In this situation, instead of following the

[5]see `Operators/TupleGenerator.cpp` in C-Store 0.2

[6]Naturally, with more complex tuple representation the copying can be avoided, but we assume simple (hence fast) NSM organization

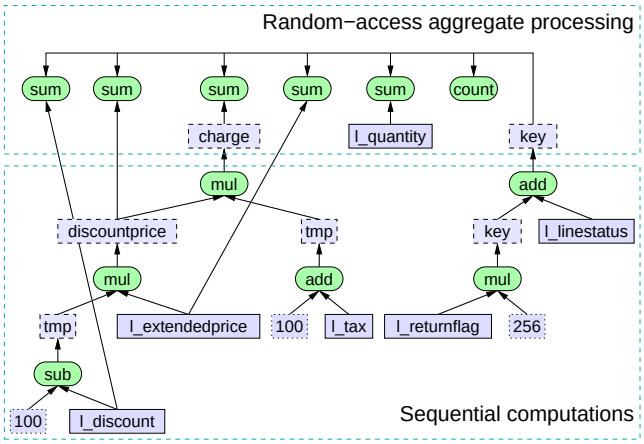

Figure 4: Simplified plan of TPC-H Query 1 (omitted date selection, links to the aggregates table, and post-aggregation computations)

traditional hash-table based processing, we use *direct aggregation* [5], in which the position in the aggregates table is computed directly from the key attributes. Since the original Query 1 only uses 4 `GROUP BY` key combinations, we also tested a slightly modified version of it that adds a 3rd `extrakey` column, and artificially fills all three key columns to simulate different numbers of `GROUP BY` combinations.

3.3.1 Pull Selections Up

In DSM, calculations on simple directly addressed arrays (i.e. without selection vector) are amendable for SIMD-ization, hence execute significantly faster. Therefore, if a Select does not eliminate many tuples and is followed by computation (e.g. a Project), it becomes beneficial to first do the calculations with SIMD, and the selection only afterwards. This counter-intuitive "pull selections up" strategy is in fact applicable to TPC-H Q1. Note that for this optimization, it is not strictly necessary to put the Select on top of the Project. In MonetDB/X100, the Select is still executed first, but for each tuple-block by looking at the selectivity (the length of the selection vector `m`) Project primitives may choose to ignore the selection vector `sel` and compute results for all `n` tuples, benefiting from SIMD:

```
ADD(long a1[n],a2[n]; int sel[m]): int output[n]
 if (m > n/2) // if many selected, compute on all
   for(pos=0; pos<n; pos++)
     output[pos] = ADD(a1[pos],a2[pos]) // SIMD
 else
   for(idx=0; idx<n; idx++)
     pos = sel[idx]
     output[pos] = ADD(a1[pos],a2[pos])
```

In fact, this performance boost makes it beneficial for a plan that starts with NSM tuples to switch to DSM.

3.3.2 SIMD Aggregation

Another SIMD optimization concerns grouped aggregation in NSM. If multiple identical aggregate functions must be computed (e.g. TPC-H Q1 has 5 grouped SUMs), we can SIMD-ize the aggregate update operation. This means that we have a primitive SUM2 function, that sums two adjacent NSM columns of 64-bit longs (its start pointer is denoted `col2` here) with two adjacent 64-bits aggregate totals (`tot2`):

```
SUM2(long tot2[m],col2[n]; int grp[n],w1,w2)
 for(pos=0;pos<n;pos++)
   simd_t* dst = (simd_t*) (tot2 + w2*group[pos])
   simd_t* src = (simd_t*) (col2 + w1*pos)
   *dst = SIMD_ADD2_LONG(*dst, *src)
```

As grouped aggregation takes more than half of the execution time in TPC-H Q1, applying SIMD here significantly affects performance. In fact, SIMD Aggregation makes it beneficial to switch back from DSM to NSM after the calculations to profit from SIMD.

3.3.3 On-the-Fly NSM/DSM Conversions

We run TPC-H on data that is in both NSM and DSM storage layouts, but consider switching layout before and after doing the calculations (i.e. Select and Project). Also, the format of the aggregate table can be DSM or NSM.

The results of the experiment, presented in Figure 5, confirm the trends from the micro-benchmarks. The DSM-formatted input (A,B,C) achieves significantly better performance. However, the DSM-formatted hash table suffers from random memory accesses (A,D,E). Using an NSM hash table removes this problem (B,G), and converting the DSM data on the fly into NSM allows to perform SIMD-based aggregation, further improving the performance (C,H). For NSM input we see that converting it into DSM allows faster sequential computation (E,G). For additional analysis of the performance, Table 2 presents the profiling of different scenarios for a case with 32K unique `GROUP BY` keys. It shows, that the extra data conversion before doing the projection and the aggregation phase can be in some cases more than balanced by the performance improvement gained in the following computation.

The performance benefits presented in this section are limited due to a fact that most of the computation is based on 8-byte integers. The currently available SSE3 SIMD instruction set provides only 128-bit SIMD operations, allowing just 2 operations to be executed at once. Since SIMD functionality is continuously improving, we expect these gains to become more significant in the future.

4. RELATED WORK

Block-oriented processing [15, 5] recently gained popularity as a technique to improve query processor efficiency. Traditionally, its main goal was to reduce the number of function calls [15]. Further research demonstrated that it also can result in a much higher CPU instruction cache hit-ratio [19]. Block-oriented processing is also an enabling technique for different performance optimizations that require multiple tuples to work on: exploiting SIMD instructions [18], memory-prefetching [8], and performing efficient data (de)compression [20].

The trade-offs between NSM and DSM as disk storage formats were analyzed in [13], where it is demonstrated that DSM performs better when only a fraction of the attributes is accessed. In contrast to what our paper proposes, the system described in [13] forces a conversion of DSM data on disk into NSM before entering the block-oriented iterator pipeline, allowing DSM layout only in the early scan-select stages. Avoiding early materialization of NSM tuples in column stores also was the topic of [2], but this work still requires forming NSM tuples at some moment in the

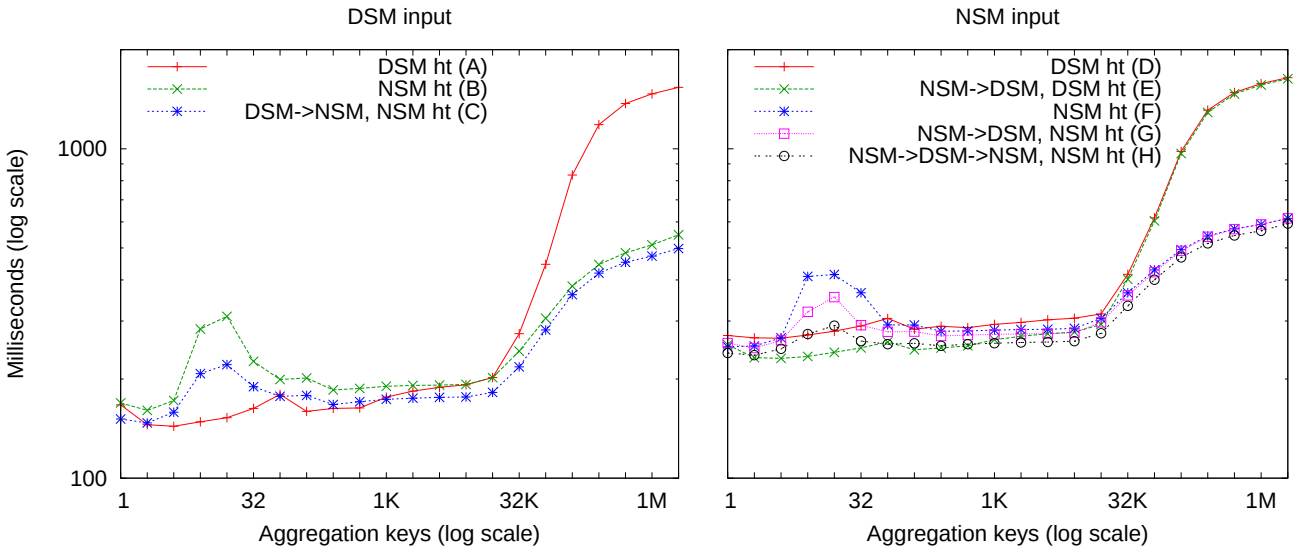

Figure 5: TPC-H Q1, with a varying number of keys and different data organizations (ht – hash table)

	Time (millions of CPU cycles)							
Source data	DSM	DSM	DSM	NSM	NSM	NSM	NSM	NSM
Projection phase	DSM	DSM	DSM	NSM	NSM	*DSM	*DSM	*DSM
Aggregation input	DSM	DSM	*NSM	NSM	NSM	DSM	DSM	*NSM
Aggregation table	DSM	NSM	NSM	DSM	NSM	DSM	NSM	NSM
Primitive	(A)	(B)	(C)	(D)	(E)	(F)	(G)	(H)
nsm2dsm_discount	0.00	0.00	0.00	0.00	0.00	325.07	340.47	337.72
nsm2dsm_extendedprice	0.00	0.00	0.00	0.00	0.00	17.73	18.24	17.99
nsm2dsm_tax	0.00	0.00	0.00	0.00	0.00	25.17	20.03	19.64
nsm2dsm_quantity	0.00	0.00	0.00	0.00	0.00	16.84	17.14	16.93
nsm2dsm_shipdate	0.00	0.00	0.00	0.00	0.00	19.70	20.02	19.45
nsm2dsm_linestatus	0.00	0.00	0.00	0.00	0.00	22.30	19.21	19.10
nsm2dsm_returnflag	0.00	0.00	0.00	0.00	0.00	22.76	19.31	19.12
select	28.40	27.98	27.96	330.37	338.77	38.93	39.77	39.00
tmp = 100 - discount	53.57	52.36	52.07	30.31	30.17	14.85	14.14	13.89
discountprice = tmp * l_extendedprice	47.52	47.17	47.33	40.67	40.99	18.52	17.56	17.80
tmp = 100 + l_tax	50.08	50.18	49.89	27.76	27.83	13.64	13.93	13.70
charge = tmp * discountprice	18.04	17.10	17.35	40.08	44.63	20.89	18.18	17.83
key = 256 * l_returnflag	20.66	20.09	20.12	28.85	28.69	19.43	19.77	19.39
key = key + l_linestatus	22.43	21.84	21.92	39.42	39.42	21.11	21.54	21.42
key = 256 * key + extrakey	33.66	32.95	33.07	64.39	64.46	32.38	32.78	32.24
dsm2nsm_charge	0.00	0.00	18.73	0.00	0.00	0.00	0.00	20.41
dsm2nsm_discountprice	0.00	0.00	20.07	0.00	0.00	0.00	0.00	20.09
dsm2nsm_discount	0.00	0.00	17.50	0.00	0.00	0.00	0.00	17.54
dsm2nsm_extendedprice	0.00	0.00	17.54	0.00	0.00	0.00	0.00	17.55
COUNT()	50.71	51.75	50.56	56.80	72.81	52.67	58.65	50.24
SUM(charge)	55.05	80.24	0.00	59.49	0.00	56.98	104.75	0.00
SUM(l_discount)	52.49	56.47	0.00	62.92	0.00	51.69	62.81	0.00
SUM(discountprice)	52.24	48.02	0.00	58.46	0.00	52.01	47.47	0.00
SUM(extendedprice)	55.43	48.65	0.00	69.11	0.00	56.97	48.71	0.00
SIMD-SUM	0.00	0.00	107.68	0.00	226.44	0.00	0.00	130.98
SUM(l_quantity)	64.05	52.23	53.87	70.64	68.01	58.03	50.20	48.66
TOTAL	603.98	608.17	553.65	981.47	981.47	956.30	1006.63	931.14

Table 2: Primitive function profile of a modified TPC-H Q1 (SF=1) with different data organizations. Star (*) denotes an explicit format conversion phase. Block size 128-tuples, 32K unique aggregation keys.

query plan. Comparing DSM and NSM execution is also the focus of a recent paper [10], though the methodology is a high-level systems comparison without investigating the interaction with computer architecture.

Some performance analysis of DSM and NSM data structures has been presented in [11] where the authors propose "super-tuples" for both rows and columns. Related research is PAX [4], a storage format that combines low NSM costs for getting a single tuple from disk with good cache behavior of "thinner" DSM storage. In memory PAX is almost equivalent to DSM (the only difference is a possible impact on the possible block sizes), it has the same properties during the processing. The PAX idea has been generalized in [12] in the *data-morphing* technique, that allows part of the attributes in a given disk page to be stored in DSM and part in NSM, depending on the query load. This research, focused only on persistent data reorganization based on the query load. Our technique goes further, by proposing dynamic reorganization of transient, in-flight data.

On many architectures, SIMD instructions expect the input data to be stored in simple arrays, as in DSM. Since most database systems work on NSM, the potential of SIMD can often not be used. Notable exceptions include [18], as well as the family of MonetDB processing kernels [6, 5]. SIMD instructions are also becoming more relevant due to appearance of architectures such as Cell that provide SIMD only [14]. Interestingly, in this context it was already shown that grouped aggregates can only be SIMD-ized when using a NSM-like data organization (*array of structures*).

5. CONCLUSIONS AND FUTURE WORK

We have shown how different tuple layouts in the pipeline of a block-oriented query processor can strongly influence performance. For sequential access, DSM is the best representation, usually as long as the tuple blocks fit L2; DSM also wins for random access inside L1. If a sequential operator is amendable for SIMD-ization, this causes DSM to strongly outperform NSM; the difference sometimes even making it profitable to pull selections *upwards* to keep data densely organized for SIMD. NSM, on the other hand, is more efficient for random access operations (hash join, aggregation) into memory regions that do not fit L1. Unlike DSM, random access memory latency to NSM can be hidden using software prefetching. Finally, grouped Aggregation allows SIMD calculations only in case of NSM.

This means that it depends on the query which data layout is the most efficient in a given part of the plan. With the conversion between NSM and DSM being relatively cheap, we show that query plans such as TPC-H Q1 can be accelerated by using both formats with on-the-fly conversions. Therefore, we think that this work opens the door for future research into making tuple layout planning a query optimizer task. Additionally, more complex data representations should be investigated, including mixing NSM and DSM in one data block, as well as using indirect data storage.

6. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, 2006.

[2] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, 2007.

[3] S. Agarwal and H. Daeubler. *Reducing Database Size by Using Vardecimal Storage Format*. Microsoft, 2007.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, Rome, Italy, 2001.

[5] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[6] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, May 2002.

[7] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, pages 54–65, Edinburgh, 1999.

[8] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3):17, 2007.

[9] A. Copeland and S. Khoshafian. A Decomposition Storage Model. In *SIGMOD*, Austin, TX, USA, 1985.

[10] N. H. Daniel J. Abadi, Samuel R. Madden. Column-Stores vs. Row-Stores: How Different Are They Really? In *SIGMOD*, Vancouver, Canada, 2008.

[11] A. Halverson, J. L. Beckmann, J. F. Naughton, and D. J. DeWitt. A Comparison of C-Store and Row-Store in a Common Framework. Technical Report TR1570, University of Wisconsin-Madison, 2006.

[12] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, pages 417–428, Berlin, Germany, 2003.

[13] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized Databases. In *VLDB*, 2006.

[14] S. Heman, N. Nes, M. Zukowski, and P. Boncz. Vectorized Data Processing on the Cell Broadband Engine. In *DAMON*, 2007.

[15] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE*, Heidelberg, Germany, 2001.

[16] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, Trondheim, Norway, 2005.

[17] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, September 2000.

[18] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, Madison, USA, 2002.

[19] J. Zhou and K. A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *VLDB*, Berlin, Germany, 2003.

[20] M. Zukowski, S. Heman, and P. Boncz. Architecture-Conscious Hashing. In *DAMON*, Chicago, IL, USA, 2006.

[21] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, Atlanta, GA, USA, 2006.