Runtime Fragility in Main Memory

Endre Palatinus, Jens Dittrich

Information Systems Group, Saarland University
http://infosys.cs.uni-saarland.de

Abstract. In this paper we investigate the following problem: Given a database workload (tables and queries), which data layout (row, column or a suitable PAX-layout) should we choose in order to get the best possible performance? We show that this is not an easy problem. We explore careful combinations of various parameters that have an impact on the performance including: (1) the schema, (2) the CPU architecture, (3) the compiler, and (4) the optimization level. We include a CPU from each of the past 4 generations of Intel CPUs.
In addition, we demonstrate the importance of taking variance into account when deciding on the optimal storage layout. We observe considerable variance throughout our measurements which makes it difficult to argue along means over different runs of an experiment. Therefore, we compute confidence intervals for all measurements and exploit this to detect outliers and define classes of methods that we are not allowed to distinguish statistically. The variance of different performance measurements can be so significant that the optimal solution may not be the best one in practice.
Our results also indicate that a carefully or ill-chosen compilation setup can trigger a performance gain or loss of factor 1.1 to factor 25 in even the simplest workloads: a table with four attributes and a simple query reading those attributes. This latter observation is not caused by variance in the measured runtimes, but due to using a different compiler setup.
Besides the compilation setup, the data layout is another source of query time fragility. Various size metrics of the memory subsystem are round numbers in binary, or put more simply: powers of 2 in decimal. System engineers have followed this tradition over time. Surprisingly, there exists a use-case in query processing where using powers of 2 is always a suboptimal choice, leading to one more cause of fragile query times. Using this finding, we will show how to improve tuple-reconstruction costs by using a novel main-memory data-layout.

Keywords: main-memory databases, data layouts, robust query processing, tuple reconstruction

1 Introduction

The two most common data layouts used in todays database management systems are row and column layout. These are only the two extremes when vertically partitioning a table. In-between these extremes there exists a full spectrum of

column-grouped layouts, which under certain settings can beat both of the aforementioned traditional layouts for legacy disk-based row-stores [7]. However, for main-memory systems column grouped layouts have not proved to be of much use for OLTP workloads [5], unless the schema is very wide [9].

Another axis of partitioning a table is horizontal partitioning, where the partitions are created along the tuples instead of along the attributes. This is usually based on the values of an attribute with low cardinality, e.g. geographical regions, but this is not a strict requirement. Forming horizontal partitions can also be done by simply taking repeatedly k records from the table, which we will call chunks in the following. Within a horizontal partition we can have any vertically partitioned layout, including row and column as well. One notable example in disk-based database systems is the PAX-layout [1], where the horizontal partitions have a size that is the multiple of the hard disk's block size, and inside these partitions the tuples are laid out in column layout. Another notable example is MonetDB X/100 [2, 12], which chooses the chunk size such that all column chunks needed by a query fit into the CPU cache.

We can apply a strategy similar to PAX in main memory as well, however, we have more freedom in choosing the size of the horizontal partitions. Therefore in main-memory we can simply form so-called chunks of the table by repeatedly taking k records from the table and laying them out in column layout *within* the chunk. We denote this layout by memPAXk. In this sense, row layout is the same as memPAX1, and column layout is equivalent to memPAXn, where n is larger or equal to the cardinality of the table. The chunks of these layouts are analogous to PAX pages [1], however, there are two important differences: (1) we can choose any chunk size (in bytes or tuples) that is a multiple of the tuple size, while for PAX we are restricted to multiples of the disk's block size, and (2) we neither store any helper data structures per chunk, nor use mini-pages as in the disk-based PAX-layout. The possible memPAX layouts of a table having 2 columns and 8 records, and using chunk sizes of powers of 2 are illustrated in Figure 1. Here we can see the two extremes: row- and column layout, and memPAX layouts with a chunk size of 2- and 4 tuples.

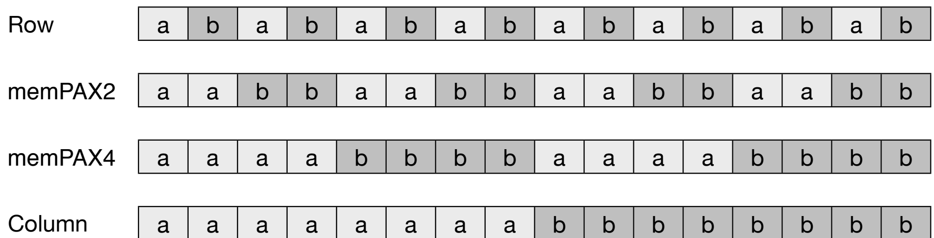


Fig. 1. memPAX layouts of a table having 2 columns and 8 records, considering powers of 2 chunk sizes.

2 The six-dimensional Parameter Space of our Experiments

We are going to explore a six-dimensional parameter space of a fairly simple workload: a table with four attributes and two simple queries reading those attributes. The whole experiment is conducted on memory resident tables, and using hand-coded queries implemented in C++. We are going to refer to this workload as our micro-benchmark. In the following we specify the dimensions:

(1) The datatype used in the schema. Our dataset is a single memory-resident table with four integer columns, with a total size of 10 GB. Depending on the data type chosen (1-byte, 4-byte, or 8-byte integers denoted by `int1`, `int4`, and `int8`, respectively) we get the following scenarios:

Table 1. The schemas used in our experiments

Label	Schema	Tuple count
char	`(a int1, b int1, c int1, d int1)`	$2560 * 1024^2$
int	`(a int4, b int4, c int4, d int4)`	$640 * 1024^2$
long	`(a int8, b int8, c int8, d int8)`	$320 * 1024^2$

(2) The presence of conditional statements in the query code. We use two queries requiring all tuples to be reconstructed for processing as shown in Figure 2. Q1 performs a minimum-search on the sum of all attributes of a tuple, which being a conditional expression yields a branch in the implementation. We have tried out a branch-free implementation of the min[1] calculation as well, which, however, was consistently slower. Q2 on the other hand performs a branchless calculation: it sums up the product of the attribute values of each tuple. Since Q2 has no branches, the measured query times are not affected by branch-mispredictions.

```
Q1: SELECT MIN(a+b+c+d) FROM T;
Q2: SELECT SUM(a*b*c*d) FROM T;
```

Fig. 2. The queries used in the experiments

(3) The CPU architecture. The performance characteristics of a main-memory database system are influenced the most by the machine's CPU. As there are usually significant changes between the subsequent CPU architectures,

[1] min = min XOR ((temp XOR min) AND NEG(temp < min));

we have chosen machines equipped with Intel CPUs of four subsequent architectures, all running Debian 7.8.0 with Linux kernel version 3.2.0-4-amd64 as shown in Table 2, with hyper-threading either disabled or not supported.

Table 2. The machines used in our experiments

CPU	Architecture	RAM
Xeon 5150	Core	16 GB DDR2 @ 266 MHz
Xeon X5690	Westmere	192 GB DDR3 @ 1066 MHz
Xeon E5-2407	Sandy Bridge	48 GB DDR3 @ 1333 MHz
Xeon E7-4870 v2	Ivy Bridge	512 GB DDR3 @ 1600 MHz

(4) The compiler. In our experiments we have chosen the three most commonly used compilers[2]: clang (3.0-6.2), gcc (Debian 4.7.2-5), and icc (15.0.0). clang and gcc are both open-source, while icc is proprietary software. clang is actually a C-compiler front-end to the LLVM compiler infrastructure. It compiles C, Objective-C, and C++ code to the LLVM Intermediate Representation (IR), similar to other LLVM front-ends, which allows for a massive set of optimizations to be performed on the IR before translating it to machine code. GCC is short for GNU Compiler Collection, a compiler supporting among others the C/C++ language. It support almost all hardware platforms and operating systems, and it is the most popular C/C++ compiler, and also the default one in most Linux distros. Intel's C/C++ compiler can take advantage of Intel's insider knowledge on Intel CPUs. It is said to generate very efficient code especially for arithmetic operations.

(5) The optimization level. We intuitively expect to get higher performance from higher optimization levels, yet there is no guarantee from the compiler's side that this will also hold in practice. Thus, we have decided to evaluate all three standard optimization levels: `-O1`, `-O2`, and `-O3`.

(6) Compile time vs. runtime layouts. The tables in our dataset are physically stored in a one-dimensional array of integers, using a linearisation order conforming to one of the layouts described in Section 1. Any query fired against this dataset needs to take care of determining the (virtual) address of any attribute value, and possibly reconstructing tuples as well. To do this it is required to know the chunk size, which can either be specified prior to compiling a given query, i.e. at compile time, or only provided at runtime.

To allow for any compiler optimization to take place, we have been extensively using templates to create a separate executable for each element of the parameter space, i.e. we have an executable for every dataset, query, machine, compiler, O-level, and layout. In case of compile time chunk sizes we have created a separate executable for each chunk size, while for runtime memPAX layouts only a single

[2] More precisely their C++ front-ends: clang++, g++, and icpc

generic one. The generic executable processes the query chunk by chunk, for which it needs the chunk size provided the latest at runtime. For smaller chunk sizes this approach has an inherent CPU-overhead caused by the short-living loops.

3 Methodology

3.1 Motivating Example

The most common way of measuring the performance of algorithms, systems, or components in the database community is to report the average runtime out of 3 or 5 runs. Let's look at an example: assume we measured runtimes of a query when executed against two different layouts. Layout A has an average runtime of 1.75 seconds and Layout B of 1.82 seconds. In this case we would clearly declare Layout A as superior to Layout B.

However, if we take a look at the runtimes of all 5 runs in Figure 3, we can see that Layout A has a high variance (0.06), whereas the query time for Layout B is rather stable (it's variance is 0.00075). Most system designers would probably prefer Layout B, due to its performance being more predictable. This example demonstrates that reporting the average runtime alone is not sufficient for comparing two solutions [6, Chapter 13]. Therefore at a minimum the variance or standard deviation of the sample should be provided along with the average to get a proper description of the sample.

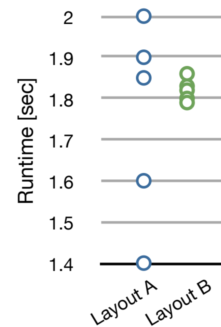


Fig. 3. Query times for two different layouts, each measured five times

We should keep in mind that when experimentally comparing multiple systems, we only get a *sample* of their performance metrics which can only be used to *estimate* the populations' performance metrics. Thus, there is always a level of uncertainty in our estimates, which renders the necessity of expressing this uncertainty in some way. One possible way to do this is to use confidence intervals, which express the following in natural language: "There is a 95% chance that the actual average runtime of System A is between 1.7 and 1.8 seconds."

3.2 Confidence Intervals

To create a confidence interval we first have to choose our confidence level, typically 90%, 95% or 99%, denoted by $1-\alpha$, where α is called the significance level. We require the sample size n, the sample mean $\overline{x}$, sample standard deviation σ, and the significance level α. Then the confidence interval is defined as follows: $(\overline{x} - C \times \frac{\sigma}{\sqrt{n}}, \overline{x} + C \times \frac{\sigma}{\sqrt{n}})$, where C is the so-called confidence coefficient. The choice of the confidence coefficient is determined by the sample size [6, p. 206].

If we have a large sample ($n \geq 30$), we can use the $1-\alpha/2$-quantile of the standard normal distribution for the confidence coefficient: $C = Z_{1-\alpha/2}$. However, in experiments we usually run only 5 measurements, thus we have a sample size of $n = 5$. Therefore, we should only use the $1-\alpha/2$-quantile of the Student's t-distribution with $n-1$ degrees of freedom: $C = t_{[1-\alpha/2,n-1]}$. The prerequisite is that the population needs to have a normal distribution, which is a fair assumption for our runtime measurements. For instance, the 95% confidence intervals for our example in Figure 3 are: (0.23, 3.27) for Layout A, and (1.65, 1.99) for Layout B. This makes Layout B a safer choice, if predictability is of great importance for the system designer. (See [6, Chapter 13] for details.) When looking at the measured query times on Layout A in Figure 3, we can see that the relatively wide confidence interval for Layout A is due to the large variance of the sample: the points are scattered out across the (1.4, 2.0) interval. However, a sample can have a large variance even if most measured values are "near" to each other, and only a few of them having a higher or lower value than the rest. These latter are called outliers.

3.3 Outlier Detection

An outlier is an element of a sample that does not "fit" into the sample in some way. It is hard to quantify the criteria for labelling an element as an outlier, and it also depends heavily on the use-case. Therefore, the most common technique used for detecting outliers is plotting the sample on a scatter plot, and visually inspecting the plot by a human. If we assume, that there is only one outlier in the sample, and it is either the minimum, or the maximum value, then we can use Grubbs' test [4] to automatically detect outliers. The only problem is that this method tends to identify outliers too often for samples with less than eight elements. To counter the error rate of the method we have included an additional condition for labelling an outlier: $\text{margin_of_error}/\overline{x} \geq 2.5\%$, where the margin of error is defined as the radius of the confidence interval.

3.4 Choosing the Best Solution when there is no Single Best Solution

Choosing the best solution using the average runtime is easy, we simply take the one with the smallest one. We have also seen that this can be arbitrarily wrong, and that is why confidence intervals provide a better basis of comparison than the sample mean. However, comparing confidence intervals is not that straight-forward as comparing scalars. If two intervals are disjoint, they are easily comparable. If they are not disjoint, and the mean of one sample is inside the other sample's confidence interval, they are indistinguishable from each other with the same level of confidence, as that of the intervals. Finally, if they are not disjoint, but their means do not fall into the other sample's interval, an independent two-sample t-test (Welch's t-test [11]) can decide whether they are distinguishable, and if so, which one is better.

4 Micro Benchmark Results

In this section we investigate the connection between the query time and the elements of the parameter space. We will consider all dimensions mentioned in Section 2, and show their effect on performance. We have executed all executables single-threaded, and have pinned the process to a given CPU core to avoid runtime variance cased by data- and thread shuffling. We have noticed that varying the chunk size of the memPAX layouts between 2^{16} and the biggest possible one does not make a significant difference in the query times, regardless of the query, machine, and compiler. Thus, we have excluded those results from our discussion.

4.1 Runtime Fragility

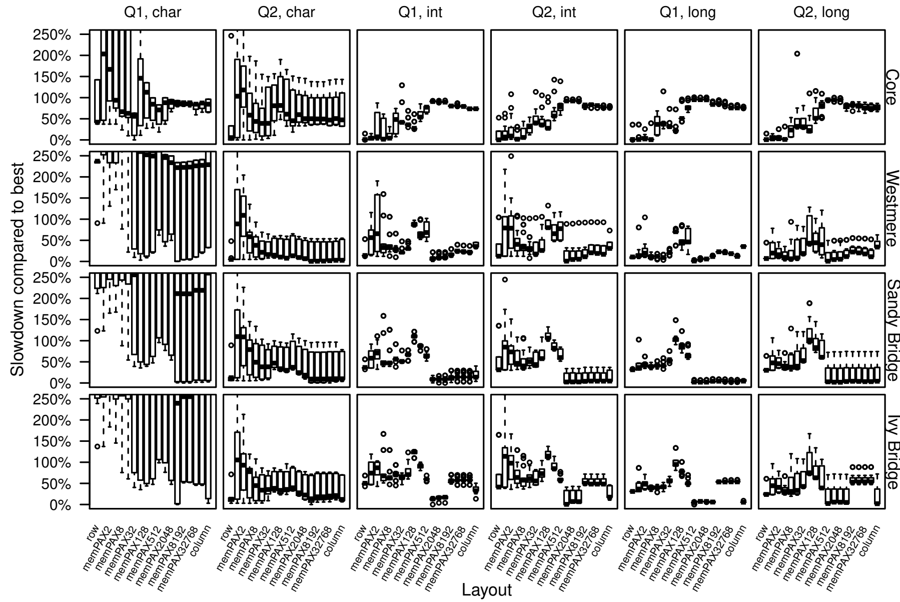


Fig. 4. Runtime fragility of the various data layouts in our micro benchmarks

We start our discussion with runtime fragility, by which we denote the performance difference caused by using another layout, compiler, O-level, etc. Note the difference between query time variance and fragility: fragility is *not* caused by query time variance, but by using another parameter combination that simply yields a different runtime, potentially a factor better or worse.

Let us first consider the runtime fragility of compiling, i.e. the effects of changing the compiler setup which consists of the compiler, O-level, and runtime

vs. compile time layouts. In Figure 4 in each subplot we show the median query times, when fixing only the machine, query, and the dataset, but changing the compiler setup and the data layout. The fragility is presented by using box plots, which show the minimum, first quartile, median, third quartile, and the maximum value of the median query times for each data layout. To help us compare the fragility of compiling across the different scenarios, the vertical axis displays the performance overhead of each compiler setup over the best one (displayed till at most 250% overhead; notice that some boxes leave their plot).

The most apparent finding is that the query time of both queries on the `char` schema is extremely fragile compared to that of the `int` and `long` schemas. This is quantified in Table 3, where we show the performance drop between the worst and the best query times, drilled-down along machine, schema, and query. We observe up to a factor 25 difference in runtime. For `char` we can get factor 3.3 to factor 25 worse by choosing the wrong layout and/or compiler setup. Furthermore, for `char` there is not only a much larger fragility across the different data layouts as seen in Table 3, but even inside a given layout as well. In the majority of the cases the compiler setup can make a factor 0.5 to 1.0 more performance drop compared to the best layout and compiler setup for Q2 on `char`.

Table 3. The performance drop between worst and best query times caused by changing the compiler setup and the layout.

Machine	**char**		**int**		**long**	
	Q1	**Q2**	**Q1**	**Q2**	**Q1**	**Q2**
Core	6.8	3.3	1.3	1.4	1.1	2.0
Westmere	25.0	3.8	1.9	2.5	1.0	1.3
Sandy Bridge	13.2	3.6	1.6	2.4	1.5	1.9
Ivy Bridge	14.4	3.3	1.7	2.6	1.3	1.7

Let us now investigate how exactly the compiler setup determines performance. In Figure 5 we can see the query times on the `char` schema for runtime-layouts (plus row and column). Please note that the query times of compile time layouts are not shown to enhance readability. For Q1 we can see that `g++ -O1` and `-O2` consistently yield a very bad performance, which is at least 3 times worse for chunk sizes above 8. The worst query times are produced by `g++ -O3` on memPAX1 and memPAX2. It is clear that the short-living loops of these two layouts incur a large CPU overhead, yet it is surprising to see that `g++ -O3` makes the layouts with chunk sizes not bigger than 16 even more inefficient. Considering the other two compilers, `clang++` performs in between the other two, and `icpc` consistently yields the best runtimes. On the other hand, for Q2 `icpc -O2` and `icpc -O3` perform the worst for chunk sizes above 2.

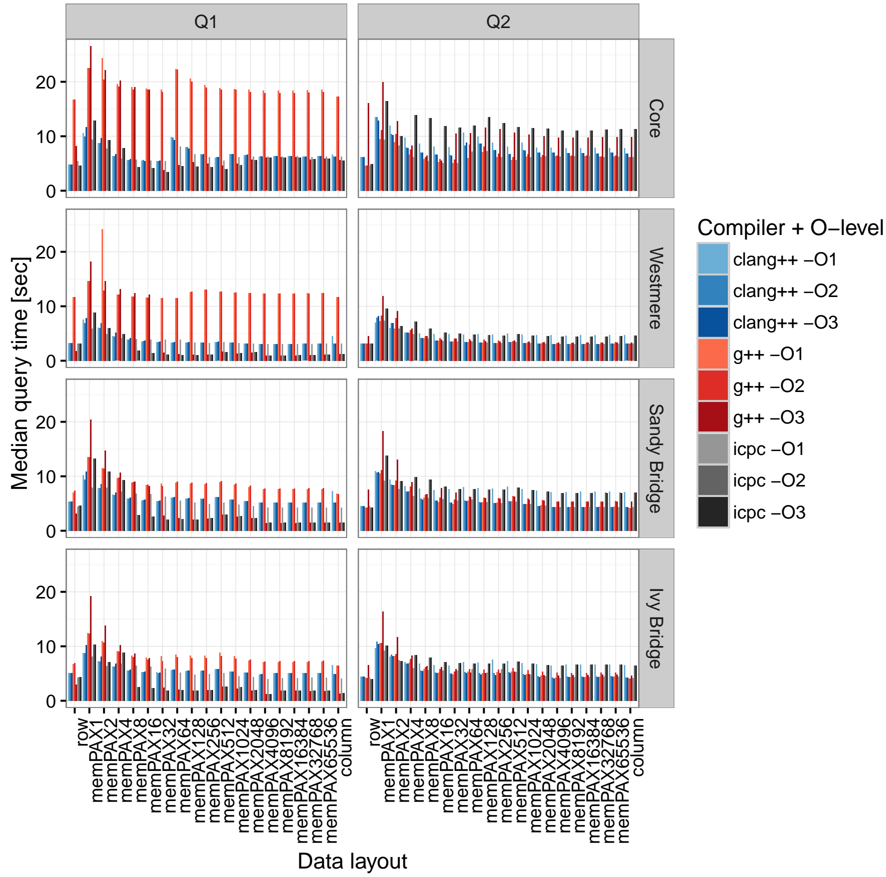


Fig. 5. Query times on the `char` schema across compiler setups and runtime-layouts in our micro benchmarks.

4.2 Best Solutions

Our second major finding is not about fragility, but a substantial difference between the effectiveness of the different layouts depending on the machine the query is executed on. To highlight this we show in Figure 6 the query times of the best layouts, drilled-down along machine, schema, and query. We can immediately notice the radical difference between Core and the other three CPU architectures. The oldest one, Core, prefers layouts with smaller chunk sizes, i.e. close to row layout. The three newer ones on the other hand prefer larger chunk sizes, i.e. close to column layout. For the latter CPUs we can further notice that the best layout for a dataset is often the one, where the following holds: k `* attribute_size * tuple_count = 4KB,` $k \in \{1 \ldots$ `attribute_count` $\}$ — which is when the chunk or an attribute's column inside a chunk perfectly fits the

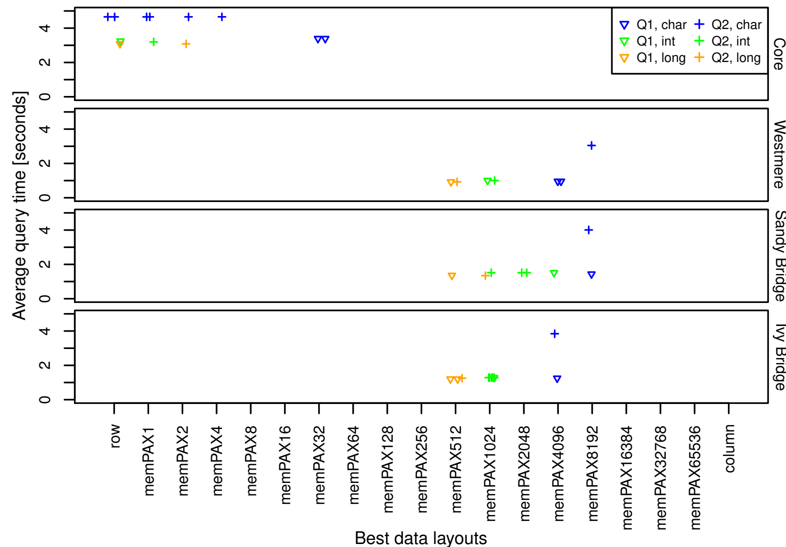


Fig. 6. Best layouts and their query times. Drilled-down along machine, schema, and query.

memory page: memPAX4096 for `char`, memPAX1024 for `int` and memPAX512 for `long`.

4.3 Conclusions and Guidelines

We can conclude that when using the `int` and `long` schemas, we can focus on choosing a proper data layout only, since the compiler setup is not expected to cause significant fragility. However, for the `char` schema care has to be taken to choose *both* layout and compiler setup wisely.

Our overall guideline for choosing the best layout is as follows: For servers equipped with a Core CPU it is a safe bet to use row layout, while for machines with the subsequent Westmere, Sandy Bridge, and Ivy Bridge architectures it is just fine to use column layout. For the latter machines we can exploit the schema for some fine-tuning, by creating PAX-blocks with the same size as the virtual memory pages. Having branches in the query is an additional argument for this optimization. The compiler, O-level, and compile time vs. runtime layouts will not change the choice of best layout (see Q2 on `int` run on Ivy Bridge), but they are to be chosen very carefully for the best performance. In cases like the `char` schema, for the optimal compiler setup, however, one has to try out all possible combinations, since it highly depends on the target system.

We have also shown how misleading it can be to choose the best solution along means. Take the case of Q2 on `char` run on Core, where the 6 best solutions are statistically indistinguishable from each other with 95% confidence, yet they differ either in the layout, the compiler, or the optimization level.

5 Revisiting Strided Memory Access

5.1 Motivation

Various size metrics of the memory subsystem are round numbers in binary, or put more simply: powers of 2 in decimal. System engineers have followed this tradition over time. Some well known examples of objects with powers of 2 sizes: cachelines, caches, RAM modules, HDD blocks, virtual memory pages, and even HDFS blocks. Surrounded by this flood of round binary numbers a data engineer feels pressed to develop data structures with similarly ,,round" sizes. So did we feel, until one day we started to question the optimality of this tradition, and dared to look at memPAX layouts with chunk sizes in between powers of 2.

5.2 Background

One of the CPU events debunking the random-access nature of main memory is the memory bank conflict. To understand this event, we first have to explain interleaved memory. DRAM and caches are both organised into banks. In case of DDR3 there are typically 4 banks. Caches on the other hand can have a varying number of banks, depending on the actual CPU generation. Interleaved memory means that the memory addresses are split among the banks in a round-robin fashion, i.e. $\text{membankID} = \text{address} \mod 4$, which allows for requests to different banks to be fetched — though not transferred — in parallel, thereby improving the bandwidth utilisation. (See [8, Section 5.2] for more details.)

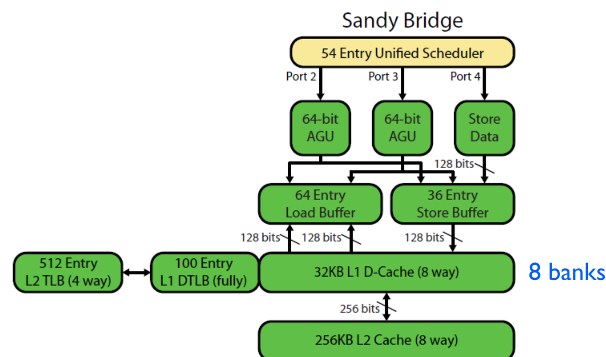


Fig. 7. The architecture diagram of Intel Sandy Bridge. Image source: http://www.realworldtech.com/sandy-bridge/7

In Figure 7 we can see the part of the Sandy Bridge architecture diagram that is related to the memory subsystem. There are two important improvements over previous generations [3]. Firstly, the Sandy Bridge architecture has two memory read ports where previous Intel processors had only one. The maximum

throughput is now 256 bits read and 128 bits write per clock cycle. The flip side of this coin is that the risk of contentions in the data cache increases when there are more memory operations per clock cycle. It is quite difficult to maintain the maximum read and write throughput without being delayed by cache bank conflicts. The second improvement is, that there is no performance penalty for reading or writing misaligned memory operands, except for the fact that it uses more cache banks so that the risk of cache conflicts is higher when the operand is misaligned.

Getting back to memory bank conflicts, the Intel Architecture Optimization Manual [3, Sections 2.2.5.2 and 3.6.1.3] gives a precise description on this event for the Sandy Bridge architecture: "A bank conflict happens when two simultaneous load operations have the same bit 2–5 of their linear address but they are not from the same set in the cache (bits 6–12)." Thus, in contrast to our expectations, it is actually not beneficial for the performance of load bandwidth-bound code to perform a strided access of addresses with a stride that is a multiple of the cache line size. In that case the addresses will have the same bits 5–0, but different bits 12–6, thus a bank conflict will occur.

5.3 Performance Implications on Tuple-reconstruction

To demonstrate the effects of bank conflicts on the performance of an application, lets consider Q1 and Q2 executed on Sandy Bridge on `char` fields, compiled with `g++ -O2`, and the chunk sizes being provided at compile time. Let us take a look at the query times for all chunk sizes [measured in tuples] between 2 and 1024, considering multiples-of-2 chunk sizes as well, in Figure 8. The black symbols on the left show the query times for row layout, while the ones on the right show the query times for column layout. The red line shows the query times for powers-of-2 chunk sizes, while the blue line shows the runtimes for multiples-of-2 chunk sizes, which is more fine granular. This exemplifies the details that can get overlooked when not performing a fine-granular exploration of the parameter space. Interestingly, there is a periodic spike in the query time, with a period size of 64, which happens to be the cache line size. Recall, that when executing Q1 we have to reconstruct the tuples for computing the aggregate value. As we have two attributes only, the stride of the memory access equals to the chunk size multiplied by the field size. Thus, for `char` fields the stride equals the chunk size. From the above discussion we know that a strided access of memory addresses with a multiple of 64 stride should result in a bank conflict.

Therefore, we have decided to validated this claim by letting VTune find the hardware events responsible for the spikes in the query time. We have taken a sample of the experiments, those with a chunk size between 448 and 512. Both endpoints of this interval are multiples of 64, and where the query time has its spikes. We have measured all existing PMU events and looked for those that have a linear correlation with the query time. We have found out that out of the ca. 200 PMU events available for Sandy Bridge, only three correlate significantly with the query time:

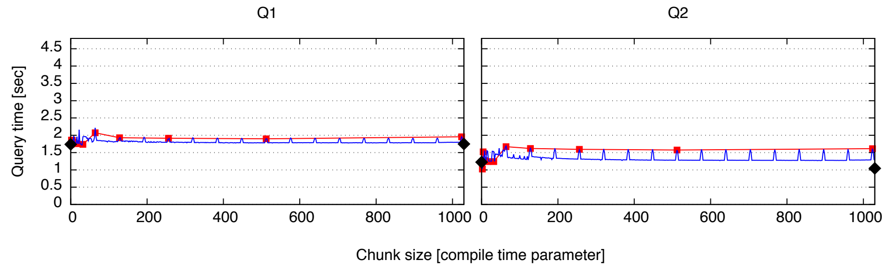
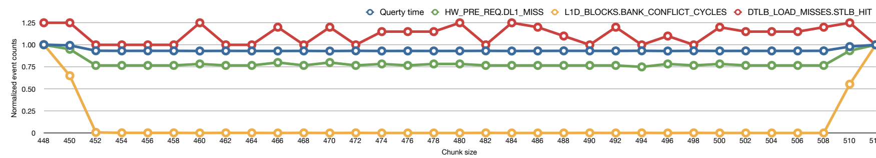


Fig. 8. Query times of Q1 and Q2 executed on Sandy Bridge on `char` fields, compiled with `g++ -O2`, and the chunk size being provided at compile time.

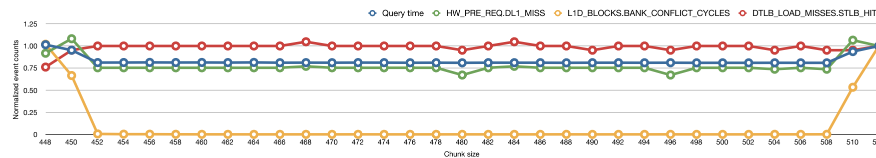
DTLB_LOAD_MISSES.STLB_HIT: data TLB load misses that hit in the second level TLB

HW_PRE_REQ.DL1_MISS: hardware prefetch requests that miss in the L1 data cache

L1D_BLOCKS.BANK_CONFLICT_CYCLES: memory bank conflict in the L1 data cache



(a) Q1



(b) Q2

Fig. 9. PMU events of Q1 and Q2 executed on Sandy Bridge on `char` fields, compiled with `g++ -O2`, for chunk sizes in $\{448, 450, \ldots, 510, 512\}$.

We have plotted these three PMU events and the query time in Figure 9, normalised to the respective values measured for chunk size 512. As we have the same spikes in the query time for the two endpoints of the chunk size interval, the normalised query times equal 1 at these points, and are below 1 for all other points. We can see that the memory bank conflicts in the L1 data cache have a very strong linear correlation to the query time. Basically, both the query time and the latter metric have only 3 different values. The query time is the lowest

when there are no L1D bank conflicts at all, and it increases together with the metric just next to the chunk sizes where the spikes are, and reaches its maximum together with the metric. The other two events also show a strong correlation, however, they do not drop to 0 inside the considered chunk size intervals.

As we can see in Figure 8, for Q2 choosing a memPAX layout which is not a power of 2 improves the query time by approximately 20%. This is definitely a significant improvement in the spectrum of what can be expected from data layouts. Q2 is a typical example of tuple-reconstruction, and thus memPAX layout can also be used for improving the tuple-reconstruction part of more complex queries.

6 TPC-H Experiments

Real world analytical workloads are significantly more complex, than our micro-benchmarks. They have a wider schema with different attribute types, and the queries use more expensive operators as well, including aggregation and joins. In order to investigate the runtime fragility of more complex workloads, let us consider the TPC-H benchmark [10].

6.1 Experimental setup

We have implemented Q1 and Q6 of the TPC-H benchmark as hand-coded applications written in C++. These two queries are single-table queries touching only the Lineitem table. We have implemented two variants of the Lineitem table: one matching the schema described in the benchmark, which we will refer to as *uncompressed*. The second version, on the other hand, is a *compressed* table. We have applied some compression schemes to the Lineitem table, as explained in Table 6.1, using the information in Section 4.2.3 "Test Database Data Generation" of the TPC-H Standard Specification.

Table 4. The compression schemes applied to the TPC-H Lineitem table

Field name	DDL-compliant data type	Compressed type	Encoding	Reason
L_LINENUMBER	int32_t	uint8_t	domain	in [1..7]
L_QUANTITY	int64_t	uint8_t	domain	random value [1..50]
L_DISCOUNT	int64_t	uint8_t	domain	random value [0.00 .. 0.10]
L_TAX	int64_t	uint8_t	domain	random value [0.00 .. 0.08]
L_SHIPINSTRUCT	char[25]	uint8_t	dictionary	random string from list Instructions
L_SHIPMODE	char[10]	uint8_t	dictionary	random string from list Modes
L_COMMENT	char[44]	uint32_t	dictionary	random text [10,43]

6.2 Runtime Fragility

We show the runtime fragility of the various data layouts for Q1 and Q6 in the TPC-H benchmarks in Figure 10, for both the uncompressed and the compressed Lineitem table. For the uncompressed Lineitem table, column layout is the clear winner in terms of performance. What is more important, is that it also has the lowest fragility across the different compiler setups, and for Q6 it has almost no fragility compared to the other layouts.

On the other hand, for the compressed Lineitem table column layout is not a clear winner. If we consider the median query times inside a given layout — depicted by the strong dash inside the boxes — for Q6 column layout is significantly worse than the memPAX layouts with larger chunk sizes. There is one very interesting difference when comparing to the query times on the uncompressed Lineitem table: the layouts of the compressed Lineitem table are much less fragile, as for Q6 the boxes are 2–5 times narrower than that of the uncompressed table.

(a) Over uncompressed tables

(b) Over compressed tables

Fig. 10. Runtime fragility of the various data layouts for TPC-H queries

7 Conclusions

In this paper we have identified various sources of query time fragility – implementation factors that can change the performance of a query by factors in an unpredictable way. We have investigated the fragility of both micro-benchmarks and complex analytical benchmarks. We have considered the CPU architecture, the compiler, and the compiler flags as important factors. We have introduced the memPAX layout and compared its fragility to column layout and row layout.

We have shown that when querying tables with 1–byte integer columns a very high fragility is to be expected, in our case leading to a performance drop of up to factor 25. In case of more complex schemas and queries the inhomogeneity of the schema has a direct effect on the fragility. Applying dictionary- and domain encoding to the columns have reduced fragility by 50% to 80% in our experiments on the TPC-H benchmark.

We have found a use-case in query processing where using powers of 2 is always a suboptimal choice, leading to one more cause of fragile query times. We have shown how to choose the chunk sizes of the memPAX layouts to improve tuple-reconstruction costs by 20%.

Acknowledgments. Research supported by BMBF.

References

1. A. Ailamaki et al. Weaving Relations for Cache Performance. In *VLDB 2001*, pages 169–180, 2001.
2. P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, volume 5, pages 225–237, 2005.
3. I. Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual.*
4. F. E. Grubbs. Sample criteria for testing outlying observations. *The Annals of Mathematical Statistics*, pages 27–58, 1950.
5. M. Grund et al. HYRISE: a main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.
6. R. Jain. *The art of computer systems performance analysis.* John Wiley & Sons, 1991.
7. A. Jindal, E. Palatinus, V. Pavlov, and J. Dittrich. A Comparison of Knives for Bread Slicing. *PVLDB*, 6(6):361–372, 2013.
8. D. Patterson and J. Hennessy. *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface.* The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2008.
9. H. Pirk et al. CPU and cache efficient management of memory-resident databases. In *ICDE 2013*, pages 14–25, 2013.
10. TPC-H. http://www.tpc.org/tpch/.
11. B. L. Welch. The generalization of Student's problem when several different population variances are involved. *Biometrika*, pages 28–35, 1947.
12. M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100-A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.