On the Surprising Runtime Fragility Found when Querying a Single Table

[Experiments and Analysis Mini-Paper]

Endre Palatinus Jens Dittrich

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

1. PROBLEM STATEMENT

Given a table with two attributes and a simple query reading those attributes, which data layout (row, column or a suitable PAX-layout) should we choose in order to get the best possible performance? This sounds really simple, yet this paper shows it is not. We will explore the parameter space that has an impact on the performance of such queries including: (1) the data type used in the schema, (2) branches in the code, (3) the CPU architecture, (4) the compiler, (5) the optimization level, as well as (6) compile time vs. runtime layouts. In addition, we will show that a task like this is not easy to measure as 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. Our results indicate that a carefully or ill-chosen compilation setup can trigger a performance gain or loss of factor 1.6 to factor 28, depending on the setup. We also introduce robustness graphs displaying the impact of non-optimal layouts on runtime performance. Finally, we give a clear guideline on when to use which method.

2. DATA LAYOUTS IN MAIN MEMORY

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 [5]. However, for main-memory systems column grouped layouts have not proved to be of much use for OLTP workloads [3], unless the schema is very wide [6].

Another axis of partitioning a table is horizontal partitioning. This is usually based on the values of a column with low cardinality, e.g. geographical regions, but this is not necessarily a requirement. Therefore an alternative to laying out all records of a table in column layout is to do this *within* horizontal partitions, so-called chunks of the table. This can be done e.g. by taking repeatedly k records from the table and laying them out in column layout. We denote the special case when $k = 2^N$ by ColumnXN. Row layout is the same as ColumnX0, and column layout is equivalent to ColumnXN, where 2^N is larger or equal to the cardinality of the table. The chunks of these layouts are analogous to PAX pages [1], however, we can choose any chunk size that is a multiple of the tuple size, while for PAX we are restricted to multiples of the disk's block size. The possible horizontal partitionings of a table having 2 columns and 8 records, using column layout inside the partitions are illustrated in Figure 1. Here we can see the two extremes: row- and column layout, and chunked column layouts with a chunk-size of 2 (ColumnX1) and 4 (ColumnX2).

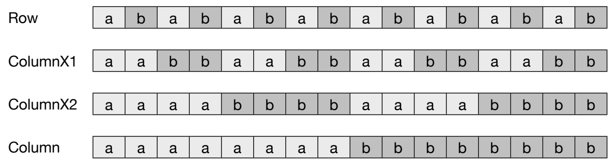
Row
a b a b a b a b a b a b a b a b
ColumnX1
a a b b a a b b a a b b a a b b
ColumnX2
a a a a b b b b a a a a b b b b
Column
a a a a a a a a b b b b b b b b

Figure 1: Horizontal partitionings of a table having 2 columns and 8 records, using column layout inside the partitions, and chunk sizes of powers of 2.

3. THE SIX-DIMENSIONAL PARAMETER SPACE OF OUR EXPERIMENTS

(1) The datatype used in the schema. Our dataset is a single table with two integer columns, with a total size of 2 GB. Depending on the data type chosen (1-byte, 4-byte, or 8-byte integers) we get the following scenarios:

Label	Schema	Tuple count
char	`Table_char (a int1, b int1)`	1024 * 1024 * 1024
int	`Table_int  (a int4, b int4)`	256 * 1024 * 1024
long	`Table_long (a int8, b int8)`	128 * 1024 * 1024

Table 1: The schemas used in our experiments

(2) The presence of branches in the query. We use two queries requiring the tuples to be reconstructed for processing as shown in Figure 2. We have chosen Q2 to factor out the possible runtime overhead caused by the branches in Q1 due to using the MIN function. Since Q2 has no branches, the measured query times are not affected by branch-mispredictions. We also tried a branch-free implementation of the min calculation in Q1 which, however, was consistently slower.

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

Figure 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, 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.

CPU	Architecture	RAM
Xeon 5150	Nehalem	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

Table 2: The machines used in our experiments

(4) The compiler. In our experiments we have chosen the three most commonly used compilers: clang++ (3.0-6.2), g++ (Debian 4.7.2-5), and icpc (15.0.0).

(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 three standard 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 the chunked column layout linearisation order described in Section 2. 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. In the first case a separate executable is compiled using templates for executing a query adjusted to a specified layout. This means the layout itself is hard-coded to allow for any compiler optimization to take place that might exploit the chunk size. If the chunk size is not known in advance, a single generic executable is generated that takes the chunk size as a runtime parameter when executing the query. This latter approach inherently has a higher query time for relatively small chunk sizes, caused by the CPU-overhead for executing the loops processing chunks that contain too few elements, yielding many short-lived loops.

4. METHODOLOGY

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 [4]. 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.

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 in natural language: "There is a 95% chance that the actual average runtime of System A is between 1.7 and 1.8 seconds."

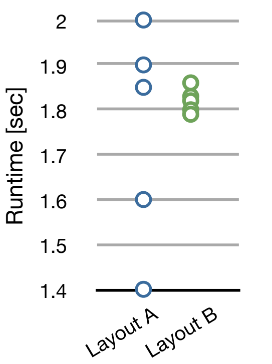

Figure 3: Query times for two different layouts, each measured five times

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 [4]. 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, we ran 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 [4] 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.

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, than we can use Grubb's test [2] to automatically detect outliers. The only problem is that this method tends to identify outliers too often for samples with less than eight elements. This explains why it detected an outlier in 27% of the cases in our experiments. To counter the error rate of the method we have included an additional condition for labelling an outlier: margin_or_error/$\overline{x} \geq 2.5\%$, where the margin of error is defined as the radius of the confidence interval. This reduces the detection rate to 3%, and those elements proved to be outliers after manual inspection.

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. It 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 [7]) can decide whether they are distinguishable, and if so, which one is better.

Machine	Table schema	Query	Compiler	Opt. level	Layout name	Chunk size provided at	Min. of Query time
Ivy Bridge	char	Q1	g++	-O3	columnX12	C	{0.221, 0.223}
			icpc	-O2	columnX12	C	{0.221, 0.223}
		Q2	g++	-O3	columnX3	C	{0.682, 0.688}
	int	Q1	icpc	-O1	columnX10	C	{0.257, 0.270}
			icpc	-O1	columnX10	C	{0.264, 0.265}
			icpc	-O1	column	R	{0.261, 0.264}
		Q2	g++	-O3	column	R	{0.225, 0.236}
	long	Q1	icpc	-O1	columnX9	C	{0.206, 0.210}
		Q2	g++	-O2	columnX9	C	{0.203, 0.205}
Nehalem	char	Q1	icpc	-O2	columnX6	C	{0.650, 0.654}
			icpc	-O3	columnX6	C	{0.651, 0.653}
		Q2	g++	-O3	columnX2	C	{0.930, 0.932}
			g++	-O3	columnX3	C	{0.930, 0.932}
	int	Q1	clang++	-O2	columnX3	C	{0.628, 0.632}
			clang++	-O3	columnX3	C	{0.629, 0.631}
		Q2	15 indistinguishable best solutions				{0.610, 0.635}
	long	Q1	8 indistinguishable best solutions				{0.595, 0.625}
		Q2	20 indistinguishable best solutions				{0.590, 0.619}
Sandy Bridge	char	Q1	icpc	-O2	columnX12	C	{0.224, 0.225}
		Q2	g++	-O3	columnX3	C	{0.838, 0.840}
	int	Q1	icpc	-O1	columnX10	C	{0.291, 0.294}
			icpc	-O1	columnX11	C	{0.292, 0.293}
		Q2	g++	-O3	column	R	{0.264, 0.265}
	long	Q1	icpc	-O1	columnX9	C	{0.204, 0.205}
		Q2	g++	-O2	columnX9	C	{0.197, 0.204}
Westmere	char	Q1	icpc	-O2	columnX12	C	{0.183, 0.185}
			icpc	-O3	columnX12	C	{0.182, 0.185}
		Q2	icpc	-O2	columnX0	C	{0.636, 0.637}
			icpc	-O3	columnX0	C	{0.636, 0.637}
	int	Q1	icpc	-O2	columnX2	C	{0.243, 0.244}
		Q2	g++	-O3	columnX10	C	{0.196, 0.198}
			g++	-O3	columnX10	C	{0.195, 0.199}
	long	Q1	icpc	-O1	columnX9	C	{0.176, 0.178}
		Q2	g++	-O1	columnX9	C	{0.176, 0.179}
			icpc	-O1	columnX9	C	{0.176, 0.179}

Table 3: The best layouts and most efficient ways of implementing a given query on a given table and executed on a given machine.

5. RESULTS

Table 3 displays our recommendations for choosing a data layout and implementation strategy for each machine, schema and query[1]. Notice that in some cases there are multiple best solutions. Looking at these results the question arises: what influences the choice of best solution? Thus, let us investigate the connection between the elements of the parameter space and the best layout.

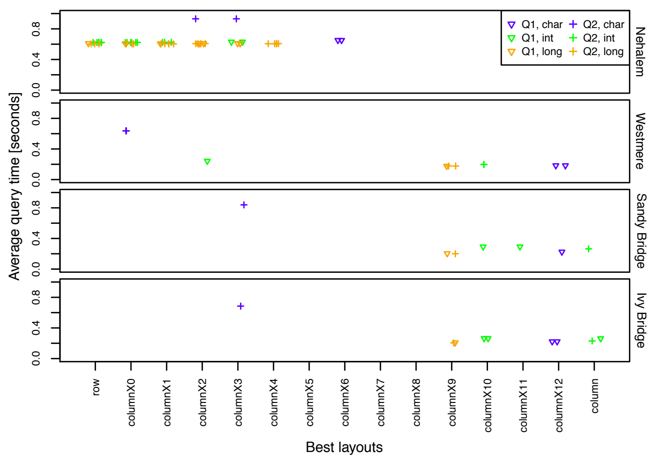

Figure 4: Best layouts and their query times. Drilled-down along machine, schema, and query.

In Figure 4 we can see the query times of the best layouts, drilled-down along machine, schema, and query. We can immediately notice the radical difference between Nehalem and the other three CPU architectures. The oldest one, Nehalem, 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: `2 * field_size * chunk_size = 4KB` — which is when the chunk perfectly fits the memory page: columnX12 for `char`, columnX10 for `int` and columnX9 for `long`.

Machine	char		int		long	
	Q1	Q2	Q1	Q2	Q1	Q2
Nehalem	11.2	6.6	2.7	2.6	2.0	2.0
Westmere	29.6	6.7	3.7	4.7	3.7	2.8
Sandy Bridge	20.6	5.2	4.3	4.5	2.9	2.8
Ivy Bridge	14.4	4.3	3.2	3.7	2.6	2.3

Table 4: The quotient of worst and best query times for each experiment. We observe up to a factor 29.6 difference in runtime.

So far we have seen the best solutions, but have not talked about the performance of the other ones. In Table 4 we show the ratio of the worst and the best query times, drilled-down along machine, schema, and query. For `char` we can get factor 4 to factor 29 worse by choosing the wrong layout and/or compiler. At this point it would be interesting to know, how does the best solution's runtime compare to the others solutions' runtimes?

[1]We have noticed that varying the chunk size of chunked column-layouts between 2^{12} and the biggest possible one (2^{30} for Table_char, 2^{28} for Table_int, and 2^{27} for Table_long) 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.

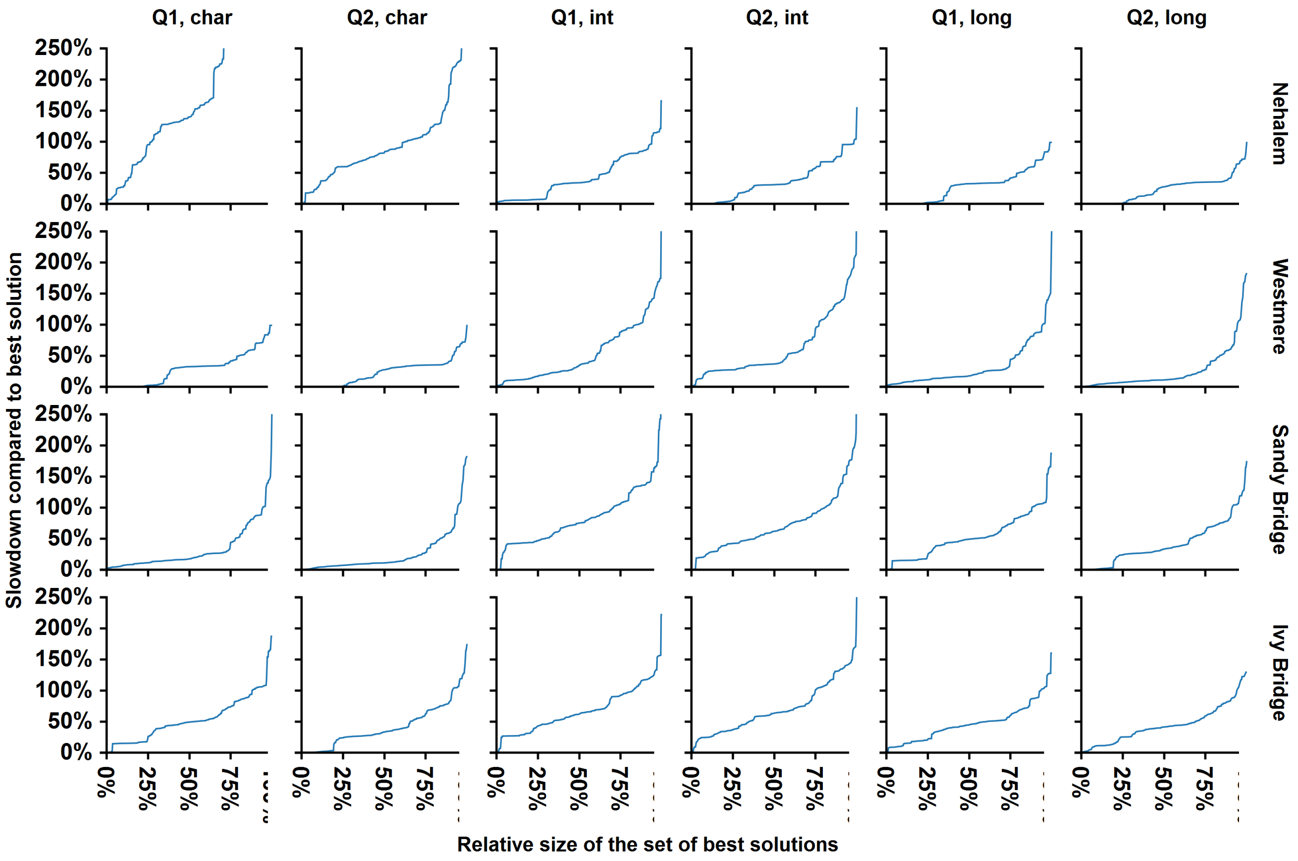

Figure 5: Robustness graphs for 24 different experiments displaying the impact of non–optimal layouts on runtime performance. The horizontal axis displays the k-th-best data layout picked (where k is in $1, \ldots, N = 252$; normalized to 100%). The vertical axis displays the performance overhead of that method over the best one (displayed till at most 250% overhead; notice that some curves leave their plot). The bigger the area under the curve, the more likely it is that a decision for a non-optimal data layout will trigger a performance loss.

6. ROBUSTNESS GRAPHS

In Figure 4 we could already observe that the set of best layouts for a particular experiment is not always considerably faster than the second or even the k-th-best method. A database architect may be willing to live with a data layout that is suboptimal for a very specific case, but does not incur too much performance overhead in the general case. To facilitate this decision, we introduce robustness graphs. Given runtime measurements for N different methods ($N = 252$ in our case), we depict the overhead of the k-th best method over the best method set. The result for 24 different experiments is shown in Figure 5. The robustness graphs show that for some situations, e.g. Q1 on `char` run on Sandy Bridge, one should be more careful in choosing the layout than in others, e.g. Q2 on `long` run on Westmere. We are planning to explore these kind of graphs as a general technique in a longer paper.

7. CONCLUSIONS AND GUIDELINES

Our guideline for choosing the best layout is as follows: For servers equipped with a Nehalem 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, but they are to be chosen carefully for the best performance.

We have shown how misleading it can be to choose the best solution along means. Take the case of Q1 on `long` run on Nehalem, where 20% of all possible solutions are statistically indistinguishable from best solutions with 95% confidence.

Acknowledgments. Research supported by BMBF.

8. REFERENCES

[1] A. Ailamaki et al. Weaving Relations for Cache Performance. In *VLDB 2001*, pages 169–180.

[2] F. E. Grubbs. Sample criteria for testing outlying observations. *The Annals of Mathematical Statistics*, pages 27–58, 1950.

[3] M. Grund et al. HYRISE: a main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[4] R. Jain. *The art of computer systems performance analysis*. John Wiley & Sons, 1991.

[5] A. Jindal, E. Palatinus, V. Pavlov, and J. Dittrich. A Comparison of Knives for Bread Slicing. *PVLDB*, 6(6):361–372, 2013.

[6] H. Pirk et al. CPU and cache efficient management of memory-resident databases. In *ICDE 2013*, pages 14–25, 2013.

[7] B. L. Welch. The generalization of Student's problem when several different population variances are involved. *Biometrika*, pages 28–35, 1947.