Exploring Query Execution Strategies for JIT, Vectorization and SIMD

Tim Gubner
CWI
tim.gubner@cwi.nl

Peter Boncz
CWI
peter.boncz@cwi.nl

ABSTRACT

This paper partially explores the design space for efficient query processors on future hardware that is rich in SIMD capabilities. It departs from two well-known approaches: (1) interpreted block-at-a-time execution (a.k.a. "vectorization") and (2) "data-centric" JIT compilation, as in the HyPer system. We argue that in between these two design points in terms of granularity of execution and unit of compilation, there is a whole design space to be explored, in particular when considering exploiting SIMD. We focus on TPC-H Q1, providing implementation alternatives ("flavors") and benchmarking these on various architectures. In doing so, we explain in detail considerations regarding operating on SQL data in compact types, and the system features that could help using as compact data as possible. We also discuss various implementations of aggregation, and propose a new strategy called "in-register aggregation" that reduces memory pressure but also allows to compute on more compact, SIMD-friendly data types. The latter is related to an in-depth discussion of detecting numeric overflows, where we make a case for numeric overflow *prevention*, rather than detection. Our evaluation shows positive results, confirming that there is still a lot of design headroom.

1. INTRODUCTION

In the past decades, query processing architectures have evolved from using the tuple-at-a-time iterator model to specialized analytical systems that use block-at-a-time execution, either producing entire materialized intermediate table columns as in the case of MonetDB [3], or small columnar fragments ("vectorized execution") as in the case of VectorWise [4]. Two important advantages of columnar and block-at-a-time processing are: (1) reduced interpretation overhead (interpretation decisions are made per block rather than per individual tuple) and (2) enabling a direct mapping onto SIMD instructions.[1] Interpretation overhead in tuple-at-a-time execution of analytical queries is typically more than 90% of CPU work [4], hence block-at-a-time often delivers an order of magnitude better performance.

An alternative way to avoid interpretation overhead is Just-In-Time (JIT) compilation of database query pipelines into low-level instructions (e.g. LLVM intermediate code). This approach was proposed and demonstrated in HyPer [9], and has also been adopted in Spark SQL for data science workloads. The *data-centric compilation* that HyPer proposes, however, leads to tight *tuple-at-a-time* loops, which means that so far these architectures do not exploit SIMD, since for that typically multiple tuples need to be executed on at the same time. JIT compilation is not only beneficial for analytical queries but also transactional queries, and thus better fits mixed workloads than vectorized execution. Also, with high-level query languages blending in with user code in frameworks such as Spark, JIT-compilation provides a standing opportunity to compile and optimize together in a single framework both user and database system operations. A notable related project into this direction is Weld [11] which offers a low-level DSL for data science workloads that can be JIT-compiled onto multiple hardware platforms.

Overall, it seems that a mix of vectorization and JIT [14] is needed, specifically because certain tasks, where processing is data-dependent (e.g. fast data decompression) or where adaptivity is important cannot be efficiently JIT-ted. HyPer uses vectorized interpreted processing in its data scans, and passes blocks of decompressed columns to its JIT-compiled query pipelines [8], so it is already using a mix.

In the past two decades, SIMD instructions in common x86 have evolved from 64-bit width (MMX) to the current 512-bit width (AVX-512). With MMX, the decision of common database architectures to ignore SIMD was logical, but the gap between the standard ("scalar") CPU instruction set which operates at the 64-bit machine width and AVX-512 is now almost an order of magnitude and arguably should not be ignored any longer. Admittedly, the database research community has studied the potential of SIMD instructions in database operations (a non-exhaustive list is, [12, 17]), but these typically focus on SIMD-izing particular operations rather than looking at database architecture as a whole. Arguably, in this time and age, we may have to re-evaluate database architecture on a much broader level in order to truly leverage modern processors with SIMD.

Many of the characteristics of SIMD are also found in GPU and APU processors, which also flourish (only) if the same set of instructions is to be applied on many independent data items without divergence in control flow [5]. The

[1] Sec 4.1 of [1] lists more advantages of vectorized execution (e.g. adaptivity, algorithmic optimization, better profiling).

Proceedings of ADMS 2017
.

aversion for divergent control flow is also found in FPGA and ASIC implementations of query operators [16, 10].

In this paper we do not aim yet to propose a new architecture for data processing on modern, heterogeneous, computing architectures, but rather pursue the more modest goal to explore parts of its design space in order to inform future choices. Specifically, we think that columnar execution as in MonetDB and data-centric execution as in HyPer can be seen as two designs at extreme points in terms of the granularity dimension (full-table versus tuple-at-a-time) as well as compilation (fully interpreted versus fully compiled). The question is what trade-offs lie in between. In order to shed light on this question, we decide here to focus on a restricted set of operations (we essentially exclude join) and for this purpose focus on the well-known TPC-H query 1 (Q1, Figure 1). This query measures computational efficiency of a database system and consists of an aggregation with few groups and significant expression calculation. Join optimization is not important as it contains no joins, nor is indexing, since the selection predicate of Q1 selects $> 95\%$ of the tuples.

Thus, in this paper we explore the execution options that combining vectorization with JIT brings, specifically also trying to use SIMD instructions, by generating many different implementation variants and benchmarking these.

The main contributions of this paper are as follows:

- we show that there is a large design space for combined vectorized and JIT compilation by illustration, in the form of different *implementation flavors* [13] for Q1.
- we show that query executors on SIMD significantly benefit from using compact data types and discuss design options for database storage that maximize opportunities for using thin data types. Specifically, this involves both compact storage and associated statistics, but also what policy is used for numeric overflow.
- we contribute a new adaptive "in-register" aggregation strategy, that adaptively triggers in aggregations with highly group locality (few neighbouring distinct groups in the tuple stream).

In our experiments (among others on a Knights Landing processor that support AVX512), we see that using compact data types has become important for efficiency, as the flavors using these consistently improve performance. Our "in register" aggregation, is the generic execution strategy that performs best, showing that on modern architectures there is still considerable headroom beyond the fastest system up to date on this query (HyPer). Hence, we hope that this paper will inspire future database query engine designers to propose innovative vectorized query compilers.

The remaining paper is structured as follows: in the next section we discuss TPC-H Q1 in detail, covering three aspects of its execution (operating on compact data types, the perfect identity hash optimization and guarding against numerical overflow in aggregates). In the following section, we describe all the different implementation flavors we created for Q1, including in Section 3.1 our new In-register aggregation method, and in Section 3.2 our fully-vectorized implementation of Q1 in AVX-512. In Section 4 we experimentally evaluate our multiple Q1 implementation flavors and compare them against each other. Section 5 discusses future work and our conclusions.

```
SELECT
 l_returnflag, l_linestatus,
 count(*)                                AS count_order
 sum(l_quantity)                         AS sum_qty,
 avg(l_quantity)                         AS avg_qty,
 avg(l_discount)                         AS avg_disc,
 avg(l_extendedprice)                    AS avg_price,
 sum(l_extendedprice)                    AS sum_base_price,
 sum(l_extendedprice*(1-l_discount))     AS sum_disc_price,
 sum(l_extendedprice*(1-l_discount)*(1+l_tax))
                                         AS sum_charge,
FROM
 lineitem
WHERE
 l_shipdate <= date '1998-12-01' - interval '90' day
GROUP BY
 l_returnflag, l_linestatus
ORDER BY
 l_returnflag, l_linestatus
```

Figure 1: TPC-H Query 1 (Q1)

2. EXECUTING Q1, IN DETAIL

The case of Q1 was initially used [4] to motivate the VectorWise system (then called MonetDB/X100 - we will use "**X100**" here to shortly denote VectorWise-style query execution). It showed the large interpretation overhead that the then-common tuple-at-a-time iterator model was posing for analytical queries. The vectorized approach iterates through the table at a vector-size of, typically, 1024 tuples. For each operator in an expression (e.g. `l_tax+1`), its execution interpreter calls a *primitive* function that accepts inputs either as constants or a small array of the vector-size. The primitive implementation typically is a simple loop without dependencies that iterates through the vector (e.g. `for(int i=0; i<1024; i++) dst[i] = col[i]+1`).

VectorWise also proposed "micro-adaptivity" [13], in which different equivalent execution strategies *inside* an operator are tried using a *one-armed bandid* approach, where the different strategies are sometimes *explored* and otherwise the best alternative is *exploited*. Depending on the operator, one can often find different ways of doing the same (e.g. evaluating a filter using if-then branches or with predication) and different compilation options (different compiler, different optimization options, different target machines) that also deliver equivalent "flavors" of doing the same thing. A more complex example of micro-adaptivity is changing the order of expensive conjunctive filter-predicates depending on the observed selectivities. Relevant for `l_tax+1` is "full-computation" [13], in which computation operations that follow a filter operation can be executed *without* taking the filter into account, i.e. are executed on all data, to get simple sequential memory access and SIMD. In a vectorized system, a filter typically results in a selection-vector (or bitmap), that identifies the qualifying tuples in a batch. Subsequent operations normally take this selection-vector into account in order to execute subsequent operations only on a subset of tuples. However, if the filter was not very selective, it may be faster to compute on all values. This decision to compute on all values or not can be made micro-adaptively, based on observed performance (or using a heuristic based on the selection percentage). Because in Q1, more than 95% of the tuples survive the `l_shipdate` predicate, and there are multiple SIMD-friendly computations (such as `l_tax+1`), this optimization provides significant payoffs.

For aggregates (e.g. `sum(l_quantity)`, an aggregation table is updated (e.g. `for(int i=0; i<1024; i++) aggr[groupid[i]]`

`+= col[i]`). Aggregation using such an array we call "**Standard**" aggregation here: it implies a load/compute/store sequence for every aggregate function for every tuple. As described in the sequel, the group-ID column is computed before that, often using a hash-table.

Interestingly, the paper that (re-)popularized JIT for query compilation, in the HyPer system [9] also used Q1 as its leading example, showing that a careful tuple-at-a-time compilation procedure that fuses all non-blocking operators, can produce a very tight loop for that pipeline consisting of efficient LLVM instructions that clearly beats the VectorWise approach. This paper confirms those results, however it introduces a new aggregation strategy called "in-register" aggregation that allows vectorized execution to claim back the performance crown on Q1. One of the reasons this new aggregation strategy is faster is that it allows to operate on *compact* data types and is SIMD-friendly. Our point is not to argue that vectorization is better than JIT compilation, on the contrary. The idea of "in-register" aggregation could also be applied with JIT (though we do not do so yet, here) and is an example of the rich and still open design space of query compilation, beyond just data-centric compilation.

2.1 Compact vs. Full data types

The database schema typically influences the data representation chosen for query execution. For example in Q1, we notice that column `l_tax` is of SQL type decimal(15,2). The more efficient, and often used, way of implementing decimal(x,y) types in SQL is to represent them as integers $x.10^y$, where the system remembers y for each decimal expression. As such, we can deduce that `l_tax` will fit into a 64-bit integer ($log_2(10^{15+2}) < 64$).

In terms of real data population, though, the actual value domains are much more restricted:

- `l_tax` only contains values (0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8), which represented as integers (value range [0,80]) fit a single byte. Similarly, `l_discount` only contains values 0.0 and 0.09, so the integer range is even more restricted: [0,9].
- `l_returnflag` and `l_linestatus` contain single-character strings that range respectively between 'A'-'R' and 'F'-'O' (in Unicode, the code ranges are [65,82] and [70,79]). It is possible to represent both as bytes.
- `l_extendedprice` varies between [901.00,104949.50], so its integer domain [90100,10494950] fits a 32-bit integer (it actually needs 24 bits).
- the expression `(1+l_tax)` produces values in the domain [1.0,1.8], which in the integer representation of decimal(15,2) becomes [100,180]. This still fits a single unsigned byte.
- `(1-l_discount)*(1+l_tax)` produces values in the domain [0.91,1.8], which in the integer representation of decimal(15,2) becomes [91,180]. This still fits a single unsigned byte. Arguably, to guarantee no loss of precision and exact answers, in a multiplication the return type could be decimal(30,4)[2], which would lead to [9100,18000], which still fits a two-byte short.
- following the latter, `l_extendedprice*(1-l_discount)` produces integer values in the domain [8199100, 1049495000], fitting a 32-bit integer.
- `l_extendedprice*(1-l_discount)*(1-l_tax)` produces integer values in the domain [819910000,188909100000], for which a 64-bit integer must be used.

We can see that the two additions and subtractions in Q1 can be done using single-byte logic. The three different multiplications (that remain after common-subexpression elimination) should be done with 16-bit, 32-bit and 64-bit precision, respectively. The often-followed policy of using the implementation type implied by the SQL type (i.e. 64-bit integer due to decimal(15,2)) wastes SIMD effort. We can calculate this as doing two (add/sub) plus three (mult) hence 5 operations on 64-bit = 320 bits, whereas doing two on 8-bit, one on 16-bit, one on 32-bit and one 64-bits leads to 128-bit (3x less "SIMD-work").

With that in mind, a natural question is: how can the query compiler know about these column domains? Obviously, there are statistics that analytical database systems already keep, for instance ZoneMaps (also called MinMax) indexes specifically provide this information, not only for whole table columns, but also for specific stretches.

A more critical variant of the question is how a system can guarantee tight [Min,Max] bounds if the system can be updated. A single (inserted) large value could hence force all computation onto the widest possible integer type, which seems a serious performance vulnerability given that real data (unlike TPC-H) is noisy.

This paper does not aim to provide a definite answer to that question, rather aims to instill research towards achieving systems that both keep most of their data tightly represented, yet can handle updates. We do observe that columnar compressed systems typically handle updates out-of-place, in C-store this is called Read-Store and Write-Store [15]. VectorWise similarly performs updates in a differential structure called Positional Delta Tree (PDT [6]). HyPer stores its DataBlocks [8] for cold data in compressed form, whereas hot data is stored uncompressed. Until now, these systems move data periodically towards the compressed format based on data hot-ness. However, it is envisionable to do this not solely based on data temperature, but to also let data distributions play a role. That is, one could let outliers stick around in the Write-Store longer, or move them to an uncompressed block (HyPer terminology) such as not to disrupt the tight [Min,Max] ranges that can be kept on the compressed Read-Store.

One can alternatively see execution on compact data types as a case of "compressed execution" [2]. The compression schemes of VectorWise use "patching" [18], which means that the compression scheme stores outliers in a separate location, and they are decompressed in a separate "patching phase". One could envision systems where execution on a range of tuples is split up in disjunct subsets of tuples that within each subset have a homogeneous (compressed) representation – which is very similar to patching. This way one can JIT-generate specific code for each such specific subset.

2.2 Perfect Identity Hash Optimization

Continuing our in-detail observation of TPC-H Q1, we note that after these calculations, data gets aggregated. This means that GROUP BY expressions must be calculated

[2]Please note that this is not specified by the SQL standard. An alternate policy could be to deliver the maximum precision of both * operands, i.e. decimal(15,2) again, or decimal(30,2).

and transformed into group-IDs, where we assume these IDs to be array indexes of an aggregation table that contains the aggregate results ("Standard" aggregation). In this query (where there are no interesting orders, as TPC-H prohibits creating indexes on columns `l_returnflag` and `l_linestatus`), most systems will use hash-based group-ID calculation. However, exploiting the limited domain of these columns, a specific optimization would represent their single-character values as Unicode integer numbers. A single such column could directly be used as the array-index group-ID, leaving the non-occurring character slots in the aggregation table unused. This wastes memory, but as this memory is not touched this may be acceptable as the size of the array is not large. The memory waste can be made smaller by only reserving 1+Max-Min slots and subtracting the Min value (`groupID[i] = col[i]-Min`). The advantage of this is that no hash-table is needed at all, in fact the identity hash is trivially a perfect hash. This perfect identity hash optimization can also be applied with multi-column group-by using addition and multiplication by the domains: e.g. for two columns `groupID[i] = (col`$_1$`[i]-Min`$_1$`) + (1+Max`$_1$`-Min`$_1$`)*(col`$_2$`[i]-Min`$_2$`)`.

In case of Q1, the GROUP BY columns happen to be all representable in single-bytes, and the results of all computations as well (as (1+82-65) * (1+79-70) = 180). Therefore the perfect identity hash optimization in Q1 provides fertile ground for SIMD execution: seven AVX-512 instructions (load,-,*,load,-,+,store) compute 64 group-IDs.[3]

2.3 Guarding against Overflow in Aggregates

Finally, we turn our attention to the task of calculating the count() and sum() aggregates (avg() is a combination of these two), specifically focusing on the problem of overflow in sum(). A database query processor is required to produce correct answers. It might occur that due to enormous data sizes (very many, large values) a sum() result can no longer be represented by the system. Rather than giving a wrong answer in such a case, the system should give a runtime error: it should check for numeric overflow.

Checking for overflow is an un-sexy database engine topic that has received scant attention in literature. Many database systems are implemented in C or C++ and these programming languages do not expose the overflow flags that modern CPUs have for their scalar instructions. This means that overflow-checking involves rather convoluted extra if-then-else tests, where the if-branch must perform various comparisons. This represents extra work that needs to be executed for every addition in the sum(). Given the simple work a sum() performs, this can impose significant overhead.

Interestingly, the use of the LLVM environment by HyPer gives that system access to CPU overflow flags, as these are exposed by LLVM. This significantly reduces the checking overhead, but there still is a necessity to perform an if-then based on the overflow flag, as an error needs to be raised in case of overflow. An optimized method of guarding for overflow is to continue execution but just OR the overflow flag results of all operations together. Only at the end of the query (or morsel) an error is raised if the flag got set. This further reduces overhead, but due to the OR work some overhead still remains.

In the previous sections, we discussed how SIMD-friendly systems might be very aware of the domains of the columns involved in (sum) expressions, in order to choose compact representations for them. Such systems could in many cases move from overflow-checking to *overflow prevention*. In fact, for the simple additions, subtractions and multiplications we discussed before, the knowledge about the domains not only allowed us to choose compact data types, it also allows to skip overflow checking altogether (as we know it cannot occur). Please note that SIMD instruction sets, unlike scalar CPU instruction sets, do not provide support for overflow flags, generically, so overflow prevention is crucial for SIMD.

However, for (sum) aggregates, we cannot so easily avoid overflow checking. If we would know a limit on the amount of tuples that will be aggregated, we could obtain bounds on the outcome of the aggregate function by multiplying this limit with the Max and Min (if negative) statistic on the aggregated expression. Often, a query pipeline can derive a particular limit on the amount tuples that pass through it. For instance, in HyPer, execution is morsel-driven, so the outer-loop will only be taken as many times as the morsel size. Also, the full table (partition) size is also such a bound, however in case of joins and its worst case of Cartesian product, multiple such bounds would need to be multiplied to get a reliable limit.

Alternatively, the system might impose a maximum-tuple-bound at a particular extreme value that will never occur in practice (e.g. one US trillion or 2^{40} tuples passing through a single thread – it would take hours and the system could in fact raise a runtime error if it would pass in reality). If we take that approach to Q1, we can see that under the logic of bounding the max-tuples-per-thread to 2^{40}:

- `sum(l_discount)`, `sum(l_quantity)`, `sum(l_extendedprice)` will fit a 64-bit integer.
- `sum(l_extendedprice*(1-l_discount)*(1+l_tax))`, and `sum(l_extendedprice*(1-l_discount))` will need a 128-bit integer.

Of course, if the decimal type of the summed expression has a lot of precision and the actual values stem from a wide range, then 128-bit integers might still not be enough in terms of this limit. Even though systems often do not support any decimal larger than what fits in a 128-bit integer, queries with these characteristics should not simply fail, because, in practice in many cases the result will fit (after all, we are computing a worst-case bound). For this cases, a system trying to apply overflow prevention in aggregates does need to have a back-up solution that in fact performs the (slower) overflow checking, using CPU overflow flags or comparisons. One can argue whether operating on 128-bit integers is efficient, anyway. These data types are supported by C/C++ compilers (`__int128_t`), however modern CPUs do not support them natively, so every simple calculation must be mapped to multiple 64-bit calculations that get combined. Moreover, there is no SIMD support for such 128-bit calculations at all.

In all, systems that support overflow prevention as an optimization for aggregate calculations are still faced with significant cost for these primitives. This is one of the reasons that HyPer could claim such a performance advantage over VectorWise in Q1 in [9]. The "in-register" aggregation that

[3]The astute reader will remark that SIMD instruction sets provide no byte-granularity multiplication, however in case of multiplication by a constant (as here) where we can guarantee no overflow, multiplication instructions on wider SIMD widths can be used to do the job.

we will introduce in the sequel reduces memory access (by aggregating in a register instead) but has as additional advantage that overflow prevention can be applied much more aggressively. In fact, it allows to do almost all sum() calculation on the 64-bit granularity which, in turn, *is* SIMD-friendly.

3. OUR Q1 IMPLEMENTATION FLAVORS

The implemented flavors we created in standalone micro-benchmark code reach from X100-inspired (using vectorized execution, similar to what VectorWise would do) over HyPer-inspired (data-centric, tuple-at-a-time, similar to what HyPer would do) to hand-written implementations in AVX-512 [4], as well as an implementation in Weld [11] [5]. Our flavors include variations in data types used, aggregation storage layout (*NSM/DSM* denotes how the aggregates are stored: row-wise resp. columnar), overflow checking/prevention as well as variations of the aggregation algorithm. Table 1 gives a quick overview over the Q1 flavors implemented and benchmarked.

3.1 In-register aggregation

In the context of Q1, with few columns, the simple implementation with a thread-private aggregate table is used. As described earlier, finding the aggregate slot (computing the group-ID) can be done very fast using the perfect identity hash optimization (otherwise, a hash-table could be used). When SIMD-izing such aggregation, special care is needed to avoid anomalies like lost updates in the aggregation table. This could happen, because when multiple aggregates are updated within a SIMD register, it is not guaranteed that these (partial) aggregates belong to different groups. Thus, a naive *scatter* to the aggregation table might lead to lost updates. This issue can be addressed in multiple ways: (1) the system might prevent conflicts, (2) conflicts are detected and resolved (in register) before the aggregation table is updated, or (3) the layout of the aggregation table is modified in a way to allow for conflict-free updates.

As opposed to array-based aggregation we propose in-register aggregation which prevents conflicts, and reduces the amount of load/stores (scatter/gathers). It is based on the idea that it is favorable to virtually reorder all active vectors, via modification of the selection vector, such that all tuples belonging to the same group appear consecutively. Afterwards the (virtual) order can be exploited to aggregate more efficiently.

Figure 2 depicts an example. The algorithms consists of two phases: First, a – *partial shuffle* – is performed which uses the *group_id*s and the selection vector to build a new permutation of the selection vector in which all the groups are clustered. The algorithm consists of two steps: First, we store the per-group consecutive positions. We use two arrays. The first array maps the group-ID to a position in the second array (indirection to minimize the active working set). The second array stores the consecutive positions in slots. Whenever a new group appears, we allocate vector-size many slots in the second array (edge case: all tuples in a

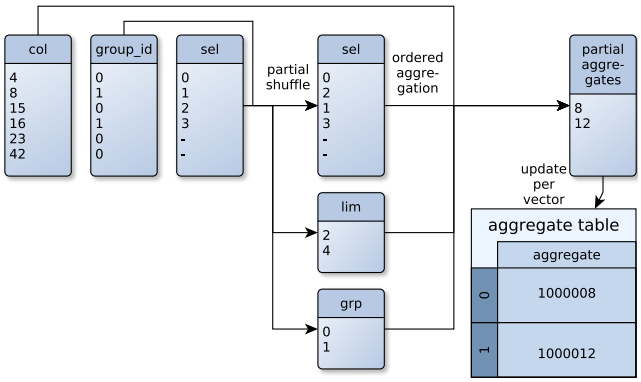

Figure 2: In-register aggregation with one column (*col*), index into aggregate table (*group_id*) and selection vector (*sel*)

stride belong to one group). Note that this allocation of slots can fail in case of too many groups. Afterwards we build the new selection vector by iterating over the groups and copying the positions. In addition, this step also computes the group boundaries (*lim*) and memorizes in which order the groups appear (*grp*) which will be needed in the next phase.

Using a lot of care, we were able to create an initial fast scalar version of this shuffle that operates at just 6 cycles/-tuple on Sandy Bridge.

Additionally it is possible to SIMD-ize *partial shuffle* using AVX-512. In our implementation we exploit the conflict detection instruction (`vpconflictd`) which, in the 32-bit version, takes a 16 (32-bit) integers as its input and produces 16 (32-bit) masks. For one element i it checks whether the elements from 0th to $(i-1)$th (previous elements in the SIMD lane) are equal the element i. If this is the case for an element k, then the k-th bit of mask i will be 1, otherwise it will be 0. If done on the SIMD register full of group-IDs, it is possible to determine how many tuples end up in the same before the before a given tuple (number of 1 bits in mask i.e. population count [6]). Together with a per-group offset, this produces unique indexes which are used to scatter out the positions from the selection vector. The per-group offset is the current location where new positions into a group will be written and is essentially an array of 16-bit indexes indexed by the group-ID. Afterwards the per-group offsets have to be updated by scattering the (unique) indexes back. Note that in case of conflicts the highest SIMD lane [7] wins, which - here - is the highest index. After the previous steps have been done for the whole selection vector, the new selection vector will be built. This, per-group, copies the stored positions into the resulting selection vector. On Knights Landing our SIMD-ized *partial shuffle* was able to operate at 9 cycles/tuple whereas the initial scalar version runs at 13 cycles/tuple.

The second phase calculates the aggregates using ordered aggregation (Algorithm 1). This aggregation algorithm exploits the fact that groups now appear in order. Hence, it just aggregates the column values until the group boundary is detected, then it updates the final aggregates in the table

[4]Source code can be found under `https://bitbucket.org/adms17/expl_comp_strat`

[5]Our implementation uses the open source version of Weld (`https://www.github.com/weld-project/weld`, accessed on July 11 2017). We had to extend Weld with 128-bit integer arithmetic for this.

[6]Sadly, in AVX-512 there is no instruction that calculates the population count in parallel. Hence we implemented a fast population count using a in-register lookup table with 32-bit granularity, which in our specific case leads to 4 lookups done via `vpermd`.

Flavor name	X100	HyPer	Data types	Overflow	Layout	Aggregation	Comments
X100 Full NSM Standard	✓	-	Full	Prevent	NSM	Standard	
X100 Full DSM Standard	✓	-	Full	Prevent	DSM	Standard	
X100 Full NSM Standard Fused	✓	-	Full	Prevent	NSM	Standard & fused	
X100 Full NSM In-Reg	✓	-	Full	Prevent	NSM	In-register	
X100 Compact NSM Standard	✓	-	Compact	Prevent	NSM	Standard	
X100 Compact DSM Standard	✓	-	Compact	Prevent	DSM	Standard	
X100 Compact NSM Standard Fused	✓	-	Compact	Prevent	NSM	Standard & fused	
X100 Compact NSM In-Reg	✓	-	Compact	Prevent	NSM	In-register	
X100 Compact NSM In-Reg AVX-512	✓	-	Compact	Prevent	NSM	In-register	Optimized for AVX-512
HyPer Full	-	✓	Full	Detect (flag)	NSM	Standard	
HyPer Full OverflowBranch	-	✓	Full	Detect (branch)	NSM	Standard	
HyPer Full NoOverflow	-	✓	Full	Prevent	NSM	Standard	
HyPer Compact	-	✓	Compact	Detect (flag)	NSM	Standard	
HyPer Compact OverflowBranch	-	✓	Compact	Detect (branch)	NSM	Standard	
HyPer Compact NoOverflow	-	✓	Compact	Prevent	NSM	Standard	
Weld	-	-	Full	Prevent	NSM	Standard	
Handwritten AVX-512	-	-	Full	Prevent	NSM	Standard (SIMD)	Hand-written in AVX-512
Handwritten AVX-512 Only64BitAggr	-	-	Full	Prevent	NSM	Standard (SIMD)	Hand-written in AVX-512 all aggregates in 64-bit arithmetic

Table 1: Q1 flavors

using the partial aggregate. The depicted *partial aggregates* array does not exist because the ordered aggregation directly updates the aggregate in the aggregate table. Note that ordered aggregation can relatively easily be implemented using SIMD.

Algorithm 1 Ordered aggregation

```
1: procedure AGGR_SUM(sel[], col[], lim[], grp[], nrGroups)
2:     g ← 0
3:     k ← 0
4:     while g < nrGroups do
5:         sum ← 0
6:         while k < lim[g] do
7:             sum ← sum + col[sel[k]]
8:             k ← k + 1
9:         grpID ← grp[g]
10:        table[grpID].sum ← table[grpID].sum + sum
11:        g ← g + 1
```

This method avoids read/write conflicts which would otherwise occur on a per-tuple basis. Further, the type of the partial aggregate can be restricted because we know that the partial sum is computed maximally for the whole vector. The vector-size, say 1024, is a much tighter bound than 2^{40}. This means all partial aggregates in Q1 fit in a 64-bit register. In case of Q1 this removes almost all expensive 128-bit arithmetic from the hot-path. The 128-bit additions still have to be done, but only once per group, per vector.

We finally note that "in-register" aggregation is an adaptive way of handling aggregates. The partial shuffle is very fast, but can fail, if there are too many distinct group values in the vector (more than 64). In that case, normal vectorized aggregation primitives are used, and we use exponential back-off to re-try with in-register aggregation much later. Such *micro-adaptive* behavior is easy to integrate in vectorized query processors [13].

3.2 Q1 in AVX-512

The hand-written AVX-512 implementation utilizes overflow prevention and pushes 16 tuples-at-a-time through its pipeline. Expressions are evaluated using AVX-512 instructions. The `l_shipdate` selection is realized using bit-masks created with AVX-512 comparisons and will later be used to mask out tuples not satisfying the predicate.

The group-IDs are computed from `l_linestatus` and `l_returnflag`. Both columns are 8-bit wide in this case (Full, not Compact types). In order to later exploit the 32-bit gather/scatter, we cast them to 16-bit, generate the group-ID and then cast them to 32-bit to generate the final index in the aggregate table.

A side effect of using SIMD is that write conflicts can occur even within a single thread. In order to achieve conflict-free and parallel updates to the aggregate table, we allocate eight consecutive slots for each aggregate, whereas eight is the degree of data-parallelism that can be achieved in AVX-512 with 64-bit values. The slots are used to maintain partial aggregates that belong to the same group and each slot is associated with exactly one SIMD lane. Thus *gathers* and *scatters* from/to the i-th SIMD lane may only address the i-th slot.

4. EVALUATION

For most experiments we used a dual socket machine with two Intel Xeon E5-2650 v2 (Sandy Bridge) and 256 GiB main memory running Fedora 24 with Linux kernel version 4.7.9 and GCC 6.3.1. In order to reduce interference with NUMA effects the test process was bound to processor 0 (on NUMA node 0) whereas all memory allocations were bound to NUMA node 0.

4.1 Standard vs. in-register aggregation

To find out in which situations the in-register aggregation excels, we compare it against the standard array-based aggregation on the - above mentioned - Sandy Bridge machine. The setup of the experiment is as follows: We used a vectorized execution pipeline. For each column we load a number of vectors and then aggregate each column into a sum of each column's values for each group. Each vector consists of up to 1024 values and the size of the input is 10^8 tuples. We generate a uniformly distributed *group-ID* (key) column. These group-IDs are spread across the aggregate table in row-wise layout with a spreading factor of 1024. All columns including the group-ID are 64-bit integers. The aggregates are 128-bit integers and it is assumed that partial aggregates fit into a 64-bit integers.

Figure 3 compares standard aggregation with in-register aggregation in the block-at-a-time processing model i.e. for n aggregates n aggregation primitives have to be called. It can be seen that for a low number of aggregates both aggregation strategies perform almost equal while with increasing number of aggregates in the query, the standard aggregation shows a worse performance mainly due to (1) with more aggregates the cost of the partial shuffle is better amortized (2)

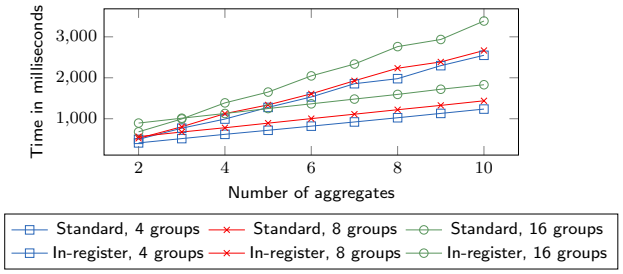

Figure 3: Standard vs. in-register aggregation, 128 bit aggregates

Standard aggregation touching the same cache lines containing the aggregates in the aggregate table many times and (3) load/store conflicts when only a few entries (groups) in the aggregate table are thrashed.

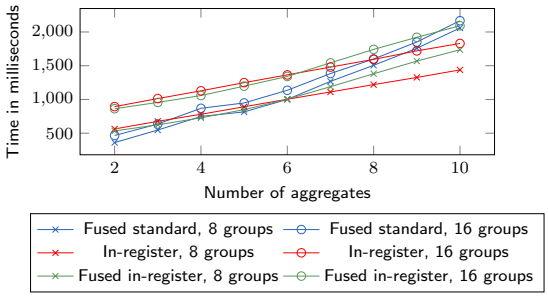

Figure 4: Fused Standard vs. in-register aggregation, 128-bit aggregates

Problem (2) can be mitigated by fusing the aggregation primitives together through merging them into one loop that updates multiple aggregates (essentially "Loop Fusion", note that this typically requires JIT compilation, see [14], as the combination of aggregates is only known at query time). In Figure 4 the fused standard aggregation is compared to the fused and non-fused in-register aggregation. In our case Loop Fusion improves the standard aggregation's performance in comparison to in-register aggregation but problems (1) and (3) (in case of few distinct group values) still hold, which is the reason why in-register aggregation out-performs the standard aggregation for more than 6 (8 groups) resp. 8 (16 groups) aggregates. Interestingly fusing the in-register ordered aggregation primitives provides hardly any benefit (≤ 6 aggregates) and can even be detrimental (> 6 aggregates).

4.2 Q1 flavors

Based on the promising results of Section 4.1 we implemented multiple versions of Q1 in order to compare their response times, single-threaded. Starting from 3 base implementation we derived different flavors: A vectorized X100-alike implementation which processes a block-at-a-time and prevents overflows, a HyPer-alike which processes a tuple-at-a-time and checks for overflows and a hand-written AVX-512 version which processes a block of 16 tuples at a time and prevents overflow. Flavors include different methods for overflow detection and prevention, different aggregation techniques, varying aggregate table layout, as well as, different data representations.

We tested these flavors on two machines. One is the - above mentioned - Sandy Bridge machine. Figure 5 visualizes the response times each flavor achieved. Refer to Table 1 for a description of each flavor. It can be seen that the vectorized approach together with in-register aggregation, compact data types and overflow prevention (*X100 Compact NSM In-Reg*) is able to outperform the other approaches. As visualized this approach also beats the HyPer-alike implementations with and *without* overflow detection. Generally-speaking, fusing the aggregate calculation into one primitive improves the response time because the aggregates which are accessed concurrently are often in the same cache-line. Further, standard aggregation can be beaten by in-register aggregation in Q1. NSM appears to be the better choice, as aggregates are closer together in memory as compared to DSM and compact data types tend to speed up vectorized processing whereas - in Q1 - they slow down HyPer-alike implementations.

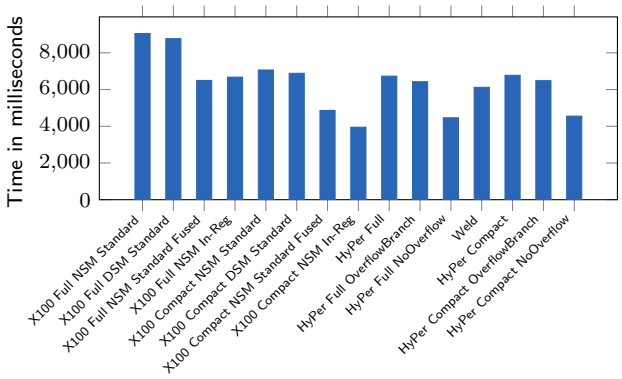

Figure 5: Different Q1 implementations on Sandy Bridge using scale factor 100.

Additionally we evaluated our Q1 flavors on an Intel Xeon Phi 7210 (Knights Landing) with 110 GB of main memory running Ubuntu 16.04 LTS using Linux Kernel 3.10.0 and GCC 5.3.1. The main memory is split into different NUMA regions: Four regions à 24 GB represent the normal (DRAM) main memory whereas the other four NUMA regions represent the accessible High-Bandwidth Memory (HBM). We limited the scale factor to 75 because scale factor 100 would exceed the local (DRAM) main memory capacity of a single NUMA node and would have caused interference with High Bandwidth Memory and/or cross-node NUMA traffic.

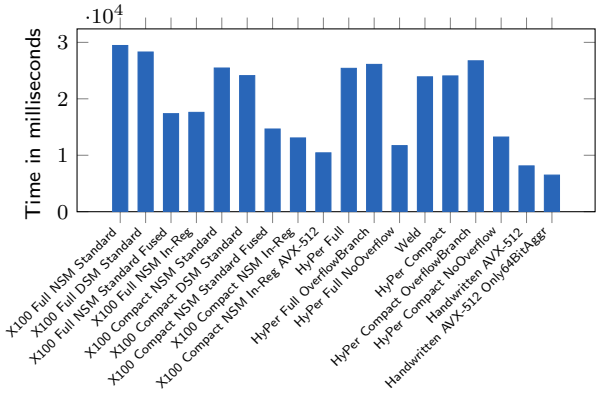

Figure 6: Different Q1 implementations on Knights Landing using scale factor 75.

Figure 6 plots each flavor's response time. In general it shows a similar picture as with the Sandy Bridge machine

with one exception being the hand-written AVX-512 implementation(s) which are the fastest of the flavors tested. Also it can be seen that other implementations can be optimized using AVX-512 i.e. wider SIMD and more complex operations are available as instructions. Additionally it can be said that the HyPer implementation performs very slowly, which is caused by overflow detection through GCC's builtins, whereas the implementation without overflow detection performs better than the average.

5. CONCLUSIONS

We argued for redesigning database systems to allow using more compact data types during query evaluation then naturally provided by the schema. This allows to maximize SIMD data parallelism and leads to more efficient processing. Further, we presented in-register aggregation, an efficient aggregation technique that can further exploit the benefits of compact data types as partial aggregates may use smaller types themselves and can be vectorized easily. We compared multiple implementations of TPC-H Q1 against each other (HyPer-inspired, X100-inspired and a hand-written Q1) on different hardware. Our proposed combination of vectorized execution, compact data types, overflow prevention and in-register was able to outperform flavors of HyPer-alikes and X100-alikes but not the hand-optimized version of Q1 directly implemented in AVX-512. Through the positive results we have shown that it is possible to improve upon the current state-of-art which confirms it is still worth to explore the design space further. In future work, we plan to further explore even more compact data representations (compressed execution) and explore methods to fully take advantage of compact data representations during query evaluation.

6. REFERENCES

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013.

[2] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.

[3] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 54–65, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[4] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, volume 5, pages 225–237, 2005.

[5] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[6] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 543–554, New York, NY, USA, 2010. ACM.

[7] Intel. Intel 64 and ia-32 architectures software developer's manual, September 2016. [Accessed on June 28, 2017].

[8] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*, pages 311–326. ACM, 2016.

[9] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[10] Oracle. Oracle announces breakthrough processor and systems design with sparc m7. `https://www.oracle.com/corporate/pressrelease/sparc-m7-102615.html`, October 2015. [Accessed on June 12, 2017].

[11] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[12] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1493–1508, New York, NY, USA, 2015. ACM.

[13] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1231–1242, New York, NY, USA, 2013. ACM.

[14] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 33–40, New York, NY, USA, 2011. ACM.

[15] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[16] G. Swart and S. Chavan. Hardware-software co-design for data management. VLDB, 2016.

[17] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.

[18] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.

APPENDIX

A. STANDARD VS. IN-REGISTER AGGREGATION

Figure 9 and Figure 8 extend our micro-benchmarks from Section 4.1 on a larger amount of groups.

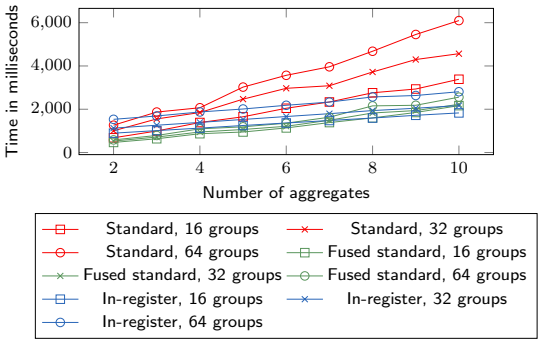

Figure 7: Standard vs. in-register aggregation, 128-bit aggregates, on Sandy Bridge

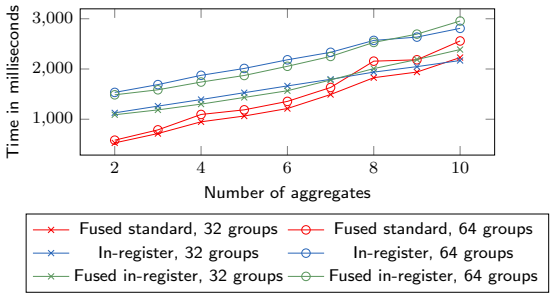

Figure 8: Fused Standard vs. fused in-register aggregation, 128-bit aggregates, on Sandy Bridge

Figure 9 extends the micro-benchmark with 64-bit integers as aggregates.

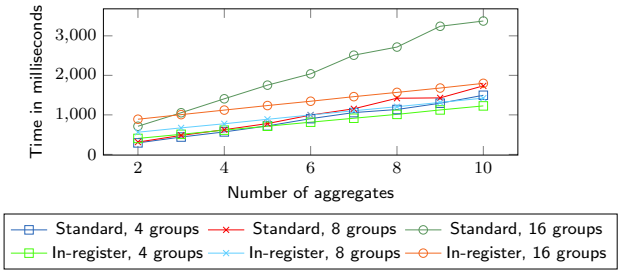

Figure 9: Standard aggregation vs. in-register, 64 bit aggregates, on Sandy Bridge