The VLDB Journal (2022) 31:1215–1236
https://doi.org/10.1007/s00778-022-00744-2

SPECIAL ISSUE PAPER

To share or not to share vector registers?

Johannes Pietrzyk[1] · Alexander Krause[1] · Dirk Habich[1] · Wolfgang Lehner[1]

Received: 7 May 2021 / Revised: 26 December 2021 / Accepted: 19 March 2022 / Published online: 28 April 2022
© The Author(s) 2022

Abstract
Query execution techniques in database systems constantly adapt to novel hardware features to achieve high query performance, in particular for analytical queries. In recent years, vectorization based on the *Single Instruction Multiple Data* parallel paradigm has been established as a state-of-the-art approach to increase single-query performance. However, since concurrent analytical queries running in parallel often access the same columns and perform a same set of vectorized operations, data accesses and computations among different queries may be executed redundantly. Various techniques have already been proposed to avoid such redundancy, ranging from concurrent scans via the construction of materialized views to applying multiple query optimization techniques. Continuing this line of research, we investigate the opportunity of sharing vector registers for concurrently running queries in analytical scenarios in this paper. In particular, our novel sharing approach relies on processing data elements of different queries together within a single vector register. As we are going to show, sharing vector registers to optimize the execution of concurrent analytical queries can be very beneficial in single-threaded as well as multi-thread environments. Therefore, we demonstrate the feasibility and applicability of such a novel work sharing strategy and thus open up a wide spectrum of future research opportunities.

Keywords Database systems · Query execution · SIMD · Vectorization · Work sharing

1 Introduction

Single Instruction Multiple Data (*SIMD*) is a well-known parallelism class in Flynn's taxonomy [15] and characterized by the fact that multiple processing units perform the *same operation* on *multiple data elements*. Thus, the traditional usage of SIMD enables data-level parallelism for single physical query operators, but no *parallelism* of different query execution tasks. Such SIMD capabilities are existing in today's mainstream processors using specific SIMD instruction set extensions. While Intel provides Streaming SIMD Extension (SSE) as well as Advanced Vector Extension (AVX), ARM offers NEON and Scalable Vector Extension (SVE) [47]. The core principle of these extensions—usually referred to as *vectorization*—is thus to execute a single instruction on a *vector* of data elements.

Vectorization has become a core technique to improve query processing performance especially in state-of-the-art in-memory column store engines [1,2,39]. However, current approaches mainly focus on vectorizing isolated query processing tasks, like selection [14,25,38,51], join [4–6,23,48], sorting [10,36,45], partitioning [35], and (de-)compression [3,11,27,53]. Yet, they mainly target the optimization of single query performance [2,19,39], while relying on a hardware-conscious way of using SIMD intrinsics [35,39, 55]. Thus, the provided data-level parallelism is fully utilized for a highly vectorized query-at-a-time execution model.

Moreover, a current hardware trend can be seen in the increasing complexity of the SIMD instruction sets and the growth of vector sizes. For instance, while Intel's AVX2 operates on 256-bit vector registers, Intel's newest extension, AVX-512, uses vector registers with 512-bit. Even more

This article is an extension of earlier published work [32] and partly funded by the German Research Foundation (DFG) within the RTG 1907 (RoSI) as well as by NEC Corporation.

✉ Johannes Pietrzyk
johannes.pietrzyk@tu-dresden.de

Alexander Krause
alexander.krause@tu-dresden.de

Dirk Habich
dirk.habich@tu-dresden.de

Wolfgang Lehner
wolfgang.lehner@tu-dresden.de

[1] Database Systems Group, Technische Universität Dresden, Dresden, Germany

flexible is ARM's newest SIMD extension SVE aiming at a configurable vector register width of up to 2048 bits [47]. The obvious idea behind wider vector registers is to store and process more data elements within a single instruction and thus increase the system performance. However, considering the details of query operator execution, achieving higher performance for single query execution (*query-at-time*) by just increasing the vector size is a challenging task for a variety of reasons: For example, by calculating a hash-based join or aggregation on an increasing number of elements in parallel (within a single vector register), the probability of key and hash collisions within the vector register increases and thus implies negative effects regarding the overall performance [33].

Our core contribution Instead of using vectorization to further optimize traditional query-at-a-time processing in analytical scenarios, we aim at using vector registers as work-sharing resource to optimize the execution of concurrently running queries. In general, sharing resources in query execution has a long tradition and a variety of approaches has been proposed [9,16,20,30,42]. Interestingly and to the best our knowledge, there is no work considering a vector register as the unit of work-sharing. To foster this line of research, we introduce and systematically evaluate an initial approach adapting known work-sharing concepts for vector registers: our vector sharing idea is to process data elements of different queries together within a single vector register leading to novel *vectorized multi-query* operators. This allows to fully exploit the wide vector registers on the one hand and reduce the probability of collisions for individual operators at the other hand.

Detailed contributions and outline To explore our idea of using large vector registers as the unit of work-sharing, we start with a simple analytical query template consisting of a `filter` and `aggregation` operator. We use a fully vectorized column-oriented implementation according to the query-at-a-time processing approach as a starting point. Based on this representation, we outline our novel *vectorized multi-query* operator implementations allowing to evaluate several queries at once as well as adjustments to different column-, operator- and query-sharing scenarios. As a final contribution, we exploit this flexibility to conduct a systematic experimental evaluation in single-threaded as well as multi-threaded environments to better understand vector registers as work-sharing resource. Thus, the remainder of this paper is structured as follows:

1. We start with an introduction to vectorization as well as an overview of vectorization and work-sharing approaches in database systems in Sect. 2.
2. Section 3 introduces our analytical query template used throughout the paper as a running example for different implementation styles. We will also discuss the use within a traditional *Single Instruction Single Query (SISQ)* implementation as well as within our novel *Single Instruction Multiple Query (SIMQ)* approach to simultaneously process different queries within single vector registers.
3. We present results of our systematic evaluation identifying situations to use vector registers as a work-sharing resource, i.e., when *to share or not to share*. As we are going to explain in Sect. 4, sharing vector registers across concurrently running queries in a single-threaded as well as a multi-threaded environment seems to be very beneficial in many cases to increase the query throughput.

Finally, we conclude the paper with a broader discussion in Sect. 5 and a short conclusion in Sect. 6.

2 Background and related work

This section provides an introduction to key features of vectorization as general background information. Following that, we present an overview of the application of vectorization in database systems to increase query processing performance. Then, we finally review available work-sharing approaches in database systems to cover all necessary areas for our paper.

2.1 Vectorization in general

Vectorization allows to process a fixed number of values with a single instruction and in a single register, the so-called *vector register*. Hence, vectorization is based on the *Single Instruction Multiple Data (SIMD)* parallel paradigm [21]. To increase the single-thread performance as priority goal, mainstream CPUs are usually equipped with instruction sets to enable a vectorized processing [21]. An SIMD instruction set offers two extensions to the basic instruction set: (i) vector registers, which are larger than traditional scalar registers being typically 32 or 64 bits wide, and (ii) vector instructions working on these vector registers.

A vector register contains multiple scalar data elements of the same data type, but a variety of data types are supported: long bit vector; 64-bit, 32-bit, 16-bit, and 8-bit integers; and double-precision floating-point numbers [21]. The data type that a vector register holds is restricted only to what vector instructions expect as inputs and outputs. The fundamental vector instructions are basic instructions that operate on multiple data elements. These include arithmetic operations (addition, subtraction, etc.), Boolean operators (e.g., AND, OR, XOR), logical and arithmetic shifts, and data type conversion. These instructions are characterized by the feature that they do not have dependencies between the data elements of the same vector register. Nevertheless, some

SIMD instructions sets have been enhanced by more complex operations like advanced arithmetic instructions operating horizontally. Horizontal operations combine multiple data elements from the same vector. For example, if we have a vector comprising a set of integers, and we want to compute the sum of those integers, then we may use a horizontal add operation that sums up all of the data elements in a vector.

In the past years, hardware vendors have regularly introduced new extensions operating on wider vector registers. For instance, Intel's Advanced Vector Extensions (AVX) operates on 256-bit and Intel's newest version AVX-512 uses even 512-bit vector registers. The wider the vector registers, the more data elements can be stored and processed in one vector, which promises significant speedups. For example, in a 512-bit vector register, we are able to store and process 64 8-bit or 16 32-bit integer values simultaneously providing a high data-level parallelism. Besides wider vector registers, Intel has also introduced more complex vector instructions as well. For example, a new instruction feature set in AVX-512 is called Conflict Detection (AVX-512CD) and the key features are: (i) the generation of conflict free subsets, i.e. subsets that contain no duplicate elements, and (ii) the count of leading zero bits of the elements in a vector.

From a programming perspective, it is common to use C/C++ language extensions with compiler SIMD intrinsics to achieve highly efficient vector code. A compiler SIMD intrinsic is a function call that maps to a specific vector instruction or a small sequence of instructions. These also require additional data types for parameters and/or the return values, such as types that map to vector registers. With that, the programmer explicitly directs the compiler to use certain vector instructions with intrinsics, but relies on the compiler for optimizations such as register allocation and instruction scheduling. An alternative way to compiler intrinsics is auto-vectorization of scalar code [21]. However, this is not as flexible and powerful as explicitly manipulating SIMD variables with SIMD intrinsics.

2.2 Vectorization in database systems

Nowadays, vectorization is a state-of-the-art optimization technique in column-store database systems and typically applied to isolated database operators [3,35,55] to mainly to reduce the query latency. Many vectorized implementations for joins [4–6,23,48] and sorting [10,36,45] have been proposed. Moreover, linear access operators such as scans [14, 25,38,51] and compression techniques [3,11,28,34,38,49,53] are well investigated; mainly because they represent an easy target for vectorization. In this context, Damme et al. [11] systematically evaluated the impact of different Intel SIMD instruction set extensions with vector sizes of 128, 256, and 512 bits on the behavior of compression algorithms. The main observation and motivation for their work is that speedups are lower for larger vector sizes because the algorithms quickly become memory-bound when the computations are accelerated through wider vector registers, and thus processing more data elements at once does not necessarily pay off. Nonlinear access operators, such as hash-tables and partitioning, have also been investigated and comprehensively evaluated [33,35,37]. Interestingly these exhibit a similar behavior of lower performance increase with growing vector register sizes.

In addition to isolated operators, vectorization and prefetching have been combined to minimize cache misses on the query level [13,31]. Even two analytical query engines have been recently introduced in which vectorization is integrated within the fundamental design of the systems: VIP [39] and MorphStore [12,19]. VIP, on the one hand, is built bottom-up from pre-compiled data-parallel sub-operators and fully implemented in AVX-512 [39]. On the other hand, the key feature of MorphStore is its novel compression-enabled and highly-vectorized processing model [19]. Aside from that, Lang et al. [26] presented an approach to vectorize entire query pipelines using vector refill algorithms.

Nevertheless, all available applications of vectorization in database systems focus on improving the single query performance for a query-at-a-time execution. Moreover, the vectorization is normally done in a *hardware-conscious way* via hand-written code using SIMD intrinsics [26,35,39,55]. To enable a hardware-oblivious vectorization approach without sacrificing the performance, we introduced the *Template Vector Library (TVL)* for in-memory column-stores [50]. This *TVL* offers hardware-oblivious but column-store specific primitives, which are similar to SIMD intrinsics. Thus, the state-of-the-art vectorized programming approach does not change, but the explicit vectorization can be done in a hardware-independent way. Furthermore, the *TVL* is also responsible for mapping the provided hardware-oblivious primitives to different SIMD extensions. For this mapping, the *TVL* includes a plug-in concept and each plug-in has to provide a hardware-conscious implementation for all primitives. In the base case, a TVL primitive can be directly mapped to a SIMD intrinsic. However, if the necessary SIMD intrinsic is not available, an efficient hardware-conscious workaround can be implemented. This implementation is independent of any query operator and must be done only once for a specific SIMD extension.

2.3 Work sharing in database systems

In addition to the efficient execution of individual queries, the optimization of concurrent queries is also highly interesting for database systems to increase the overall query throughput. Thus, work-sharing across concurrently executed queries has been an active research field for many decades, whereby work-sharing is defined as any operation reducing the total

amount of work in a system by eliminating redundant computation or data accesses [22]. The techniques proposed so far for analytical queries (OLAP) include: (i) cooperative scans, (ii) multi-query optimization, (iii) materialized views, and (iv) simultaneous pipelining.

The main idea of *cooperative scans* is to share data scans across queries that are executed at the same time. This cooperative scan technique has been investigated for disk [24,56,58] as well as for main-memory oriented database systems [40,41]. However, these approaches are limited to scans of single relations. In contrast, CJoin presented an approach to extend the idea of *cooperative scans* to join operations, especially to multi-query star-joins [9].

Another approach is denoted as *multi-query optimization (MQO)* [46]. MQO involves (1) the detection of common sub-expressions in concurrently executed queries, (2) the evaluation of these sub-expressions only once to avoid redundant work, and (3) reusing the results to answer the queries. These sub-expressions are usually determined on-the-fly for a set of concurrently executed queries [54]. In the same vein, Materialized Views (MV), another work-sharing technique, persistently store frequently used intermediate results for reuse in a variety of different queries [43].

Additionally, QPipe [20], CJoin [9], and MQJoin [29] use pipelines for work-sharing. In QPipe [20], each query operator is evaluating several queries at once, resulting in an operator-centric paradigm for work-sharing. CJoin and MQJoin mainly focus on sharing work in joins. Other related works in that domain are SharedDB [16,17], BatchDB [30], and OLTPShare [42]. While SharedDB and BatchDB consider mixed workloads of analytical (OLAP) and transactional (OLTP) queries, OLTPShare focuses on work-sharing in pure OLTP scenarios. In SharedDB, for example, incoming queries are batched and compiled into one single query execution plan. During the execution of this plan, further incoming queries are queued. However, none of these approaches considered vector registers as a work-sharing resource.

3 Sharing vector registers

The primary goal of this paper is to present a first approach alongside a systematic and comprehensive evaluation of whether, when, and how vector registers are suitable for work sharing. To achieve that, we initially restrict ourselves to workloads of concurrent analytical queries. For the sake of simplicity and demonstration of our overall approach, we assume that every query follows the same template. This query template is inspired by query 1.1 from the Star-Schema Benchmark (SSB) [44]. As depicted in Fig. 1, our query template filters a column (*:A*) for a predicate (*:Pred*) and aggregates all corresponding valid tuples from column (*:B*). According to a vectorized query processing in column-store engines, we assume that (*:A*) and (*:B*) contain sequences of fixed-sized integer values in the range of 8 to 64 bit and have the same sequence length [1,2,39]. Moreover, a concrete query instance of our template requires three inputs, namely (i) a value for the filter predicate, (ii) an aggregate function, and (iii) a pair of input columns (*:A*) and (*:B*).

Using these tuning knobs, we are able to define individual workloads with different degrees of data- (by adjusting the input columns and the predicate) and computation sharing (by adjusting the aggregate function) opportunities. That means, concurrent queries in a specific workload can have any combination of aggregate function, columns, predicates, or nothing in common. A relatively simple example for such an individual workload is highlighted in Fig. 1, where four queries share the filter- and the aggregate function but they apply different predicates on (*:A*) and access disjoint columns (*:A*) and (*:B*)—denoted by A_0,...,A_3 and B_0,...,B_3, respectively. A typical columnar query execution plan for our query template consists of two physical operators, a `filter` and an `aggregation`, which are executed subsequently. These operators are executed according to either an operator-at-time [7,19] or a vector/block-at-a-time processing model [8,31]. In the case of our query template, the main difference is the size of the intermediate results between the `filter` and `aggregation`-operator. Combined with an assumed positional addressing scheme, the characteristics of both models are more or less equivalent and, thus, we do not longer differentiate between the execution models.

Generally, the necessary operators for our queries are easy to vectorize because both operators have a linear access behavior on the corresponding columns and introduce no data dependencies between consecutive values. On a very abstract level, the vectorized `filter` operator sequentially reads multiple values from column (*:A*), applies the filter predicate, and outputs a mask where a 1-bit indicates that the corresponding value satisfies the predicate. The subsequent vectorized `aggregation` operator also sequentially reads multiple values from column (*:B*), applies an aggregation function on these values, according to the 1-bits from the mask of the previous filter operator. Finally, the aggregated values are passed as a result. Since both columns for each query have the same physical order, our two query operators can iterate evenly over both columns.

To illustrate a concrete use-case and without loss of generality, we now assume that our columns contain 64-bit integer values and vector registers have the size of 256-bit, which results in four data elements being processed in parallel. Since we materialize at least one byte, the vector mask resulting from the `filter` is an 8-bit value where the lower four bits are of interest, i.e., one bit per data element in our vector register. To separate the concerns of data flow control and actual computations, we split up our physical query opera-

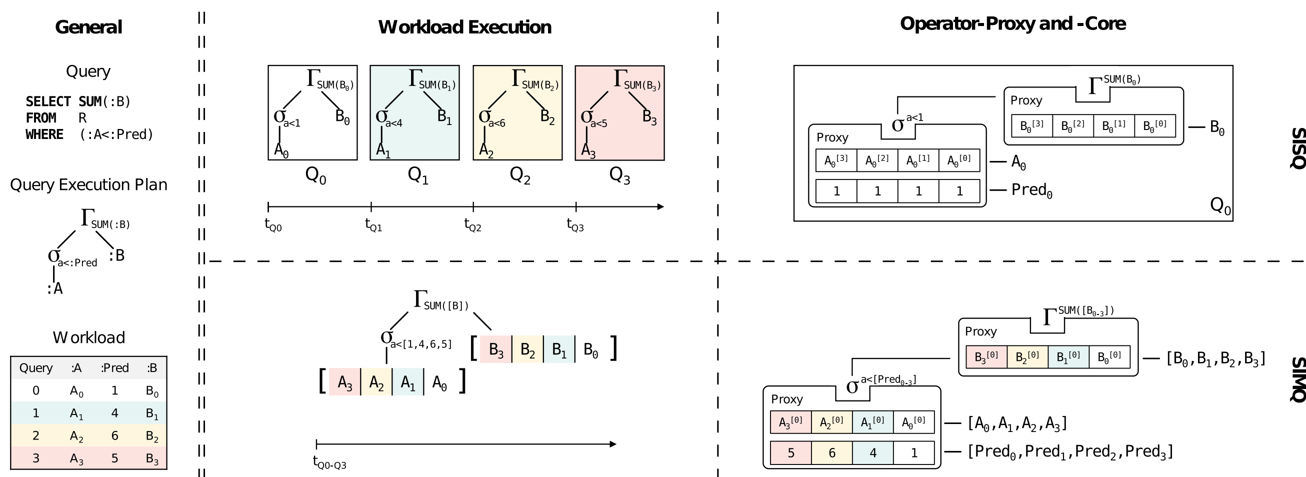

Fig. 1 Illustration of our use case with 4 different queries including the execution according to *SISQ* and *SIMQ*

tors into a *proxy* and a *core*: The *proxy* iterates over the given input column, retrieves the corresponding vector registers, and forwards them to the *core*. Initially, the filter *proxy* has to load the query-specific predicates into an additional vector register. The *core* subsequently applies the computations on the vector registers and returns the result. The *proxy* materializes the result into the corresponding output buffer for the next operator. For our two physical query operators, the operator cores are rather simple, as they only execute a single vector instruction as follows:

(i) `Filter-Core` This core calls an intrinsic comparing two vector registers a and p, producing a vector mask m (`_mm256_cmplt_epi64_mask`). A 1-bit at m_i indicates $a_i < p_i$.

(ii) `Aggregation-Core` This core performs a conditional operation, consequently producing a vector register r. The operation expects a vector register *src*, containing the previous aggregated values, two input vector registers b and c as well as a vector mask m. In general, the values from b and c are handled according to the respective aggregation function if the corresponding bit within m is set to 1. Otherwise, the corresponding value from *src* will be used for the result. As the aggregation is a reduction operator, reducing all valid elements into a single result, we set $c := src$. For this paper, we implemented three different aggregation functions.

`SUM`: The function calls `_mm256_mask_add_epi64`. If a 1-bit is at *m[i]*, $r[i] := b[i] + src[i]$, otherwise $r[i] := src[i]$.
`MIN`: The function calls `_mm256_mask_min_epi64`. If a 1-bit is at *m[i]*, $r[i] := min(b[i], src[i])$, otherwise $r[i] := src[i]$.
`MAX`: The function calls `_mm256_mask_max_epi64`. If a 1-bit is at *m[i]*, $r[i] := max(b[i], src[i])$, otherwise $r[i] := src[i]$.

While our operator cores work utterly unaware of how their input is generated, we can implement two different proxies to realize two different query execution variants: (i) *SISQ* and (ii) *SIMQ*. The *SISQ* variant corresponds to the traditional full vectorization of a single query as done, e.g., in [7,8,19,31,39]. Following this traditional processing style, multiple queries in our workload are executed individually (sequentially in a single-thread case, c.f. Fig. 1 Workload Execution-SISQ), whereby each query profits from full vectorization. Regarding a multi-thread scenario, the queries are executed on different cores in parallel, following the inter-query parallelism paradigm. In contrast to this model, our *SIMQ* variant uses vector registers as shared resources by processing multiple queries at once, as shown in the lower half of Fig. 1. This implies that our operator-proxies handle columns from different queries and build appropriate vector registers for work sharing. The general idea is adapted from QPipe [20]; however, QPipe does not consider vector registers, but parallelizes over several cores.

3.1 SISQ: Single Instruction Single Query

To realize the traditional full vectorized execution of our simple query template, the *SISQ* proxy for the `filter`-operator receives a pointer to the column (*:A*) as well as a single predicate value (*:Pred*) for the comparison. In the first step, the predicate value is broadcasted into a vector register using `_mm256_set1_epi64x`. Secondly, the proxy iterates over column (*:A*). A single proxy-iteration consists of (i) a vector transfer operation (`_mm256_load_si256`) loading 256-bit of data from (*:A*) into a vector register and (ii) invoking the `filter-core` with the predicate and data vector registers. That means multiple data values of a single column are compared with a single predicate value at once. As already described before, the output of the `filter-core` is an 8-bit

value. Thus, only the lower four bits correspond to the input of the `filter-core`. In general, it would be possible to materialize this value right away, but this would lead to doubling the write operations and memory footprint of the result in total. Consequently, we call the `filter-core` twice—one after another—and pack two results $mask_{it1}$ and $mask_{it2}$ from the `filter-core` into a single 1-byte value: $mask_{it1} \mid = (mask_{it2} << 4)$. The resulting bitmask is then materialized into an output buffer by the `filter` proxy. After each iteration, the column pointer is increased by $2 \cdot ec$, where ec denotes the element count (vector register size, i.e., 256, divided by value size, i.e., 64), and the pointer to the output buffer is incremented as well.

The *SISQ* proxy related to the `aggregation` receives a pointer to the result buffer of the preceding `filter` operator and a pointer to column (*:B*) on which the aggregation should be executed. The data are prepared similarly to the `filter` operator using a stepwidth of two. As two bitmasks are encoded in one byte, the input-proxy has to split them up accordingly: $mask_{it1} = mask \,\&\, 0\text{x}F$ (and $mask_{it2} = mask \,\&\, 0\text{x}0F0$). The *src* register, containing the current aggregation result, is initially set to zero. The loaded data together with the according bitmask are handed over to the `aggregation-core`. Thus, the `aggregation` is a reduction operator, and the result of this core is used as the input for the following iterations.

To execute a workload consisting of multiple concurrent queries, each query is individually executed using *SISQ*. In a single-threaded environment, the execution obviously results in a sequential execution order enforcing no explicit work-sharing in the form of data or computation sharing; only caching may contribute implicitly to workload optimization. In a multi-threaded environment, several queries are executed in parallel.

3.2 SIMQ: Single Instruction Multiple Queries

In contrast to the *SISQ* proxy approach (taking care of the in- and output for operators with a single query scope), the *SIMQ* proxy for an operator is associated with a workload scope consisting of multiple queries. As illustrated in Fig. 2, the workload scope may have multiple expressions in terms of column- and query-sharing opportunities for a *vectorized multi-query* (*VMQ*) operator such as our `filter` or `aggregation`. In Fig. 2, we again assume a 256-bit vector register with 64-bit column values as a foundation, such that we can hold four values within a single vector register. These four values may come from one, two, or four different columns and may be associated with one, two, or four different queries, respectively.

3.2.1 Design space

From a memory sharing perspective, we have to distinguish two different cases, namely (i) base data access and (ii) intermediate data access, while the latter case is equal to the first case if the considered (sub)queries, which produced the intermediate, are structurally and functionally equal. The following describes from a conceptual point of view what different kinds of sharing potentials exist, depending on whether base- or intermediate data can be shared across queries.

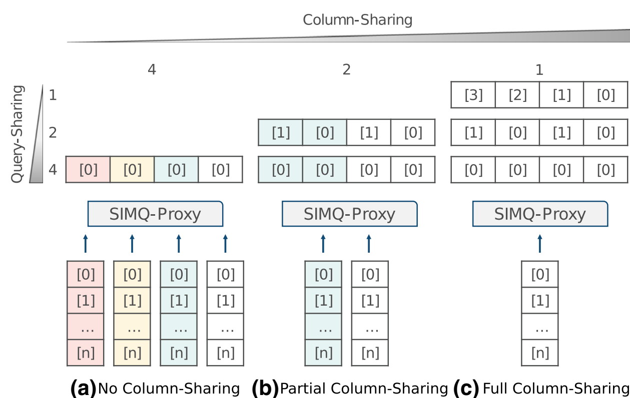

Fig. 2 *SIMQ:* column- and query-sharing opportunities for base data

(i) Sharing access to base data From a base data column perspective, we can distinguish three column-sharing modes for the multi-query operators:

No column-sharing Each value within a vector register originates from a different column, implying that every value is associated with a different query. As shown in the left *SIMQ* proxy of Fig. 2a, each element of a vector register is used to process a single value from a different column and, thus, the *SIMQ* proxy loads only one value from each column in every iteration. In this case, four query operators are executed in parallel using a vector register as a shared resource, whereby the columns are each processed sequentially.

Full column-sharing In contrast to the previous mode, a *VMQ* operator processes only one column for multiple concurrent queries, whereby the query count can be 1, 2, or 4 (right *SIMQ* proxy in Fig. 2c). On the one hand, if the query count equals 1, we end up with the *SISQ* variant described above by loading four values of the column at once. On the other hand, if the query count equals 4, then the *SIMQ* proxy only loads one column value per iteration but processes this single value for four different queries at once. If the query count is 2, then 2 column values are always loaded and processed for two queries resulting in vectorized parallel processing of multiple queries. In general, the advantage of this full column-sharing is that a loaded column value is processed by multiple queries simultaneously, which reduces the overall memory footprint and improves cache friendliness.

Partial column-sharing The two previous sharing modes are corner cases, and partial column-sharing covers everything in-between as shown in Fig. 2b.

Afterwards, the prepared vector registers are pushed into the associated *core*. The result of a core can be a vector register or a vector mask. Considering the `filter` core of our query template, the core produces a mask, where a 1-bit indicates that the corresponding element in the vector register, hereinafter referred to as lane, satisfied the predicate. This mask is materialized and can be used for the following query execution stages. It must be noted here that the order of the lane-to-query mapping is preserved within the resulting bitmask. In the context of our query template, bitmasks are highly relevant and thus we conducted several benchmarks investigating how intermediate data from incongruent preceding stages should be handled. While it is technically possible to split up bitmasks and store smaller portions in different locations, it significantly harmed the overall execution time. This can be explained through the generally higher costs of store operations compared to load operations on the one hand. On the other hand, splitting bitmasks incurs the maintenance of intermediate buffers, which heavily relies on branching. Thus, we concluded that bitmasks should be materialized as one unit and split up in the loading phase of the next *multi-query* operator, according to its sharing mode. Assuming the example of full column and query sharing, the first block of four bits of the bitmask is associated with the first element from the `filter` input column. Within these four bits, the first bit corresponds to the first query, the second bit to the second query, and so forth.

(ii) Sharing access to intermediate data When accessing intermediate data, not only the column perspective is relevant for the association of the lanes within a *VMQ* operator. Furthermore, the order of the previous operator-stage has to be taken into account. As described before, the output of a *VMQ* operator implicitly encodes the column- and query sharing mode of the according proxy. Without loss of generality, we assume that a *VMQ* operator accesses a single column (full column-share). When considering the preceding distribution of active operators, we can distinguish three different sharing modes of our query template:

Full predecessor-sharing The number of participating queries in the current stage equals the number of the previous stage. In contrast to the full query- and column share mode from Fig. 2, the proxy has to load four consecutive values from a single source since every value is specifically linked to a query within the *VMQ* operator. Consequently, the proxy behaves as if only a single query is executed.

Minimal predecessor-sharing As shown in the left upper part of Fig. 3, only a single query is assigned to a *VMQ* operator while the previous stage executed four queries. When operating on base data, this would be comparable to *SISQ* execution. However, the resulting order of the preceding stage requires a step width equal to four since only every fourth value is associated with the active query.

Partial predecessor-sharing While the two previous sharing modes are corner cases, partial predecessor-sharing covers everything in between.

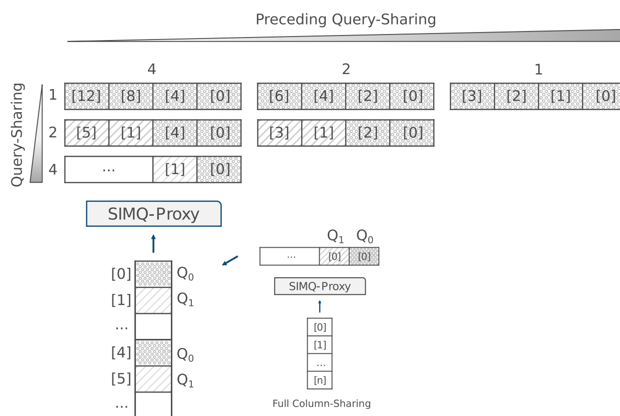

Fig. 3 *SIMQ:* Query-sharing opportunities for intermediate data

3.2.2 Design space implementation

A highly flexible *SIMQ* proxy is crucial for efficiently exploring the described design space. This proxy has to build the appropriate vector registers and masks for the operator cores depending on the number of accessed columns, active queries, and the preceding number of queries. Moreover, this *SIMQ* proxy has to transfer the different query predicates in a vector register for the `filter operator` in an initial step. Since our general design uses data columns and bitmasks as in- and output of an operator-stage and both types have severe differences regarding the addressing and utilization, we have to implement the proxies separately.

(i) Column-proxies The *SIMQ* proxy for base data columns can be implemented fourfold, leveraging different SIMD primitives: `SIMQ-GATHER`, `SIMQ-BUFFER`, `SIMQ-SET` and `SIMQ-BROADCAST`. In the following, we explain each of them in more detail.

`SIMQ-GATHER:` The most straightforward way of building a vector register from arbitrary memory locations is to use a random-access vector load instruction such as `_mm256_i64gather_epi64`. Intel, for example, introduced this feature with AVX2. As depicted in Fig. 4a, the vector gather takes a base-pointer for the random access, a vector register representing the positions that should be transferred by using a relative offset to the starting address, and a scale factor indicating the

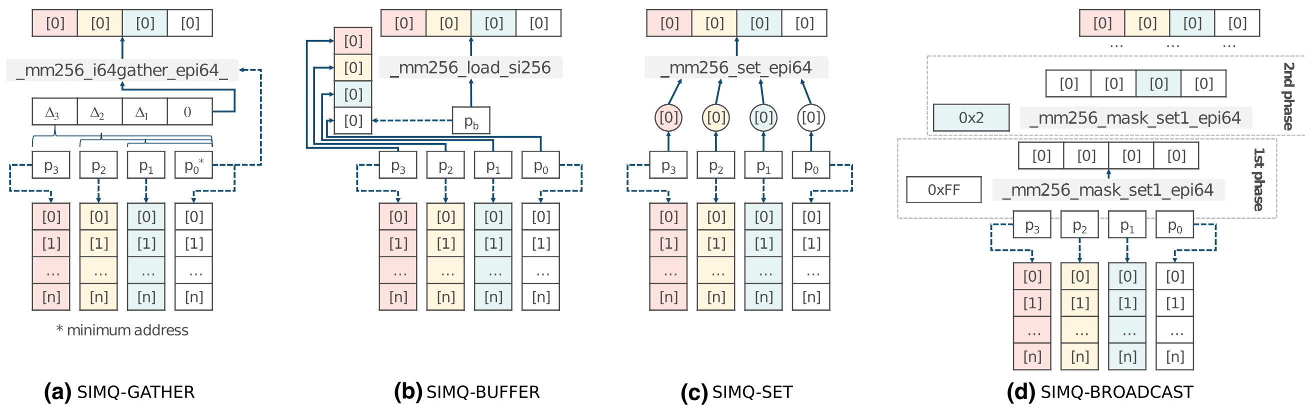

Fig. 4 Different *SIMQ*-proxies building AVX2 vector registers in no column-sharing mode

vector stride, i.e., distance between elements. To receive the required parameters for the `SIMQ-GATHER` proxy, the minimum address of all input columns is determined, and the relative offsets of all columns are calculated and initially transferred into a vector register. The overall execution flow is similar to the *SISQ* proxy with an iteration step size $2 \cdot \frac{ec}{qc}$, where ec denotes the element count of a vector register and qc denotes the number of queries. However, the most crucial drawback is that we linearly scan the columns with an instruction for random access.

`SIMQ-BUFFER`: To overcome the drawback of `SIMQ-GATHER`, an alternative approach for building a vector out of different memory locations is to use an intermediate buffer and a vector load instruction afterwards (see Fig. 4b). To do so, no preprocessing steps are needed. During every iteration within an operator, the data pointers of the different columns are de-referenced and written into the buffer in a scalar way. The buffer is then loaded into a vector register, and the pointers are incremented appropriately. The drawback of this solution is that we need an additional buffer mechanism to build the vector registers.

`SIMQ-SET`: An alternative yet similar approach, which avoids an additional buffer and a vector load instruction, can be realized by setting the values directly. As shown in Fig. 4c, for every iteration, the columns are dereferenced and stored in local variables. These variables are then transferred into a vector register using a vector set instruction such as `_mm256_set_epi64`.

`SIMQ-BROADCAST`: A fourth way is shown in Fig. 4d. This so-called `SIMQ-BROADCAST` is based on conditional broadcasting using `_mm256_mask_set1_epi64`, which was introduced by Intel with AVX512. This instruction needs a source vector register, an 8-bit mask, and a value. The mask is used to indicate at which positions the specific value should be set within the resulting vector register. For the remaining parts, the corresponding values are taken over from the source vector register. Thereby, the conditional broadcasting has to be executed repeatedly until the input for the `core` can be retrieved. To illustrate the procedure, we call a specific conditional broadcast as the n-th phase. In the first phase, all bits are set to 1 in the mask (`0xFF`), resulting in a vector register only containing the value from the first column. In the second phase, the mask has the value $0x2$ and the vector register from the first phase is used as the source register. The conditional broadcast sets the value from the second column at the second position in the new resulting register. All remaining elements are the same as after the first phase. Consequently, to build up a vector register from four columns, the broadcast consists of four phases for 256-bit vector registers and 64-bit values. That means, the effort differs depending on the underlying configuration: vector length, data value size, and the number of different columns.

(ii) Bitmask-proxies While proxies that operate on columns handle addressable values, namely with a size of at least one byte, bitmask-proxies may have to reorganize a specific subset of such values. A possible scenario is depicted in Fig. 5 where four queries produced a mask in a previous stage, and only two queries are executed in the current stage. Given a 256-bit AVX2 vector register with 64-bit wide elements, every query occupies two lanes in the current stage, while in the previous stage, only one lane was associated with a query. Consequently, the bit-order coming from the preceding stage cannot be used as-is for the current one without reordering. As described before, the filter result is packed together by concatenating two results into one byte. As highlighted in Fig. 5, the relevant bits (z, y, z, w) have a stride of four. This stride means that the bits associated with the first query are at positions 0 and 4, the bits for the second query reside at positions 1 and 5 within the first byte of the input bitmask. The core expects a bitmask with a bit order equal to the order

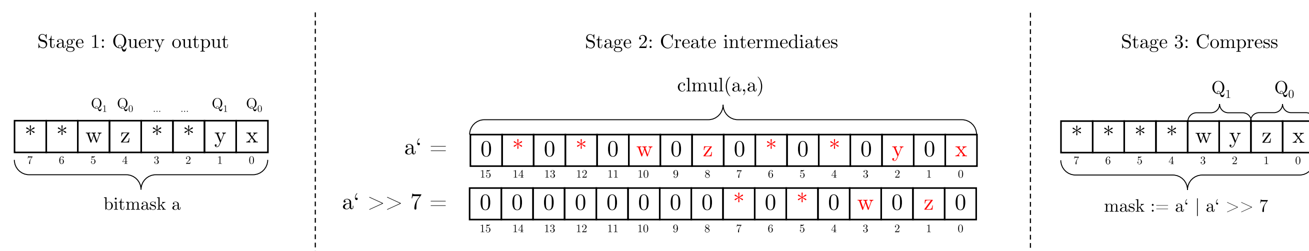

Fig. 5 Different *SIMQ*-proxies building AVX2 vector masks with partial predecessor-sharing

within the vector register. Thus, the bit from position 4 must be relocated to position 1, and the bits at positions 1 and 5 have to be relocated to positions 2 and 3, respectively. This relocation can be achieved by a series of bitwise shift and mask operations followed by logical *OR*ing the updated values. As this approach is rather cumbersome and error-prone, we exploit the effect of a carryless multiplication of the value of interest a with itself. This multiplication results in spreading the bits from a by interleaving it with 0-bits. The new value a' then has to be logical *OR*ed with itself shifted by seven to the right. The resulting bitmask contains the bits in the correct order and can be used by the core. If the discrepancy of the number of active queries compared to previous active queries increases, we have to apply the carryless multiplication multiple times based on the previous call to spread the bits of interest accordingly.

3.3 Summary

To sum up, we outlined an approach to use vector registers as a work-sharing resource for the optimization of concurrent analytical queries. In particular, our idea is to process data elements of different queries together within a single vector register leading to *VMQ* operators. As already mentioned, the sharing potential mainly depends on the vector size and the data value size. The combination of 64-bit data values and a vector size of 256-bit allows up to 4 concurrent queries on classic Intel hardware. In the case of 512-bit wide vectors and a small 16-bit data size, this number may even be increased to 32 parallel queries. In contrast to single query processing, we therefore are able to exploit wider vector registers, e.g., a maximum vector size of 2048-bit with ARM SVE.

4 Evaluation

To evaluate whether and when vector registers are suitable as a work-sharing resource, we implemented[1] the presented *SISQ* and *SIMQ* approaches for all available SIMD extensions relying on the TVL[2] as a vector-hardware abstraction [50]. Consequently, we had to implement our variants only once, using TVL-provided hardware-oblivious vector primitives. The actual mappings to vector intrinsics happened at compile-time. While TVL supports all primitives which are needed for our approach, some do not have a tangible SIMD intrinsic counterpart provided by the hardware. The missing intrinsics are compensated by using scalar workarounds. However, these workarounds may introduce overhead, making it hard to compare the overall behavior of our approach on different hardware. To mitigate this challenge, we only evaluated methods that are fully supported by the target system. That is, for all Intel processors, which provide the vector lengths of 128-bit (SSE), 256-bit (AVX2), and 512-bit (AVX-512) and for ARM featuring 128-bit (NEON) wide vector registers. We ran our experiments on three different platforms whose specifications are shown in Table 1. We chose these platforms because of their differences in the maximum core frequency, the architecture characteristics, and the available caches. For all experiments, we ensured that they happened entirely in-memory, we repeated every experiment 30 times, and averaged the results. To minimize the influence of unpredictable cache effects, every base data column in our evaluation had a size of 128 MiB. Thus, the data processed by a single query did not fit into the cache as a whole. We implemented two different processing styles namely operator-at-a-time and vector/block-at-a-time for *SISQ* and *SIMQ* (cf. Sect. 3). For the materialization of intermediate results, we used bitmasks, where a 1-bit indicates a valid tuple during query execution. Thus, the number of memory accesses and the number of executed instructions is independent of the selectivity induced by the predicates. Consequently, we generated random base data columns without any specific distribution which was agnostic to query execution in this particular scenario. Since *SIMQ* aims on workload optimization, we use the query throughput as evaluation measure—*number of queries that are executed per second (qps)*—and report absolute values as well as relative improvements of our *SIMQ* approach compared to the state-of-the-art *SISQ* execution to show the possible benefit

[1] https://github.com/db-tu-dresden/SharedVector.

[2] https://github.com/MorphStore/TVLLib.

Table 1 Environment specifications used to compare *SISQ* and *SIMQ*

Manufacturer		Intel		Huawei
Architecture		x86	MIC	ARMv8.2
Processor	Name	Xeon Gold 6126	Xeon Phi 7250	Kunpeng 920-6426
	Frequency	2.6–3.7 GHz	1.4–1.6 GHz	0.2–2.6 GHz
	CPU-Count (Threads)	12 (24)	68 (272)	128 (128)
	SIMD-Extensions	SSE, AVX(2), AVX512 (F, CD, BW, DQ)	SSE, AVX(2), AVX512 (F, CD, ER, PF)	Neon
	Cache (L1/L2/L3)	32 KiB/1 MiB/19 MiB	32 KiB/1 MiB/-	8 MiB/64 MiB/256 MiB
RAM		92 GB DDR4-2666	204 GB DDR4-2400	500 GB DDR4-2933
OS		CentOS 7.7.1908	Ubuntu 18.04.1	Ubuntu 20.04.2
Compiler	Name (Version)	clang 9.0.0-2		gcc 9.3.0
	Flags	-O3 -flto -fno-slp-vectorize -march=native		-O3 -flto -fno-tree-vectorize -march=armv8-a+crypto

of *SIMQ*. We identified six dimensions to compare *SISQ* and *SIMQ*, and hence our tuning knobs are: (i) the vector register size, namely 128-bit, 256-bit, and 512-bit, (ii) the number of accessed column pairs within a specific workload, (iii) the number of queries within a workload, (iv) the bit-size of single entries with a column, (v) the degree of operator variance for the second stage of our query template and (vi) the batch size.

4.1 Single-threaded evaluation

We begin our evaluation with purely single-threaded experiments, because SIMD is a hardware-driven approach to improve single-thread performance. Consequently, all experiments ran on a single core and, thus, in a fully controlled environment.

Since we expect the behavior of our *SIMQ* processing model to depend on the frequency of the CPU cores and the available caches, we carried out our systematic evaluation on three different platforms. As depicted in Table 1, we chose two Intel platforms, where the Xeon Gold 6126 provides a high base- and maximum core frequency alongside a three-leveled cache hierarchy and the Xeon Phi with low base- and maximum core frequency and no L3 cache, respectively. As a third architecture, we selected a Kunpeng 920-6426, which is an ARMv8.2 architecture. The base core frequency of the Kunpeng is lower than both Intel platforms, while the maximum core frequency is higher than the one of the KNL and as high as the base frequency of the Xeon Gold. Since the Kunpeng only supports ARM's SIMD extension called NEON with 128-bit vector registers, we could not measure the effects of bigger vector registers for this platform. In addition, ARM Neon does not support vectorized random access operations. Consequently, we did not implement `SIMQ-Gather` for this platform, either. However, we are expecting ARM to prove a suitable platform for evaluating the general applicability of our approach. The achieved throughput results of our running example scenario are depicted in Fig. 6. Every dimension of our systematic evaluation is shown in one row, while we organized the results for every platform column-wise.

(i) Influence of SIMD-register size To evaluate the impact of the vector register size, we varied only this dimension and kept the other dimensions constant. All queries executed a `filter` followed by an `aggregation-sum` operator. The queries accessed the same column pair for *full column-sharing*, and the number of queries was equal to the number of elements that fit into a vector register. The value size was fixed to 64-bit. This means two queries were executed when using 128-bit vector registers, four queries for 256-bit and eight queries for a 512-bit vector register, respectively. For *SISQ*, the queries were executed sequentially.

As shown in Fig. 6a1–a3, very good improvements for the query throughput (*qps*) for all *SIMQ* proxy approaches are achieved. Interestingly, the *qps* on the Xeon Gold 6126 using *SISQ* is only marginally affected by the SIMD-register size due to the high core frequency and the fact that the execution of our query is memory-bound. Overall around 50 queries were processed per second by the state-of-the-art vectorized implementation. This holds true for any adjustment to the overall setting we made (cf. Fig. 6a1–f1). *SIMQ* reduces the number of fetched cache lines of the specific workload by the number of concurrently executed queries. The wider the vector registers, the more queries can be executed simultaneously, and therefore the achieved query throughput improved in the range of 1.98x up to 7.18x, compared to *SISQ* reaching up to 365 queries per second with 512-bit wide vector registers. Surprisingly, even the random access `SIMQ-GATHER` variant, which can be considered to be rather expensive—compared to linear

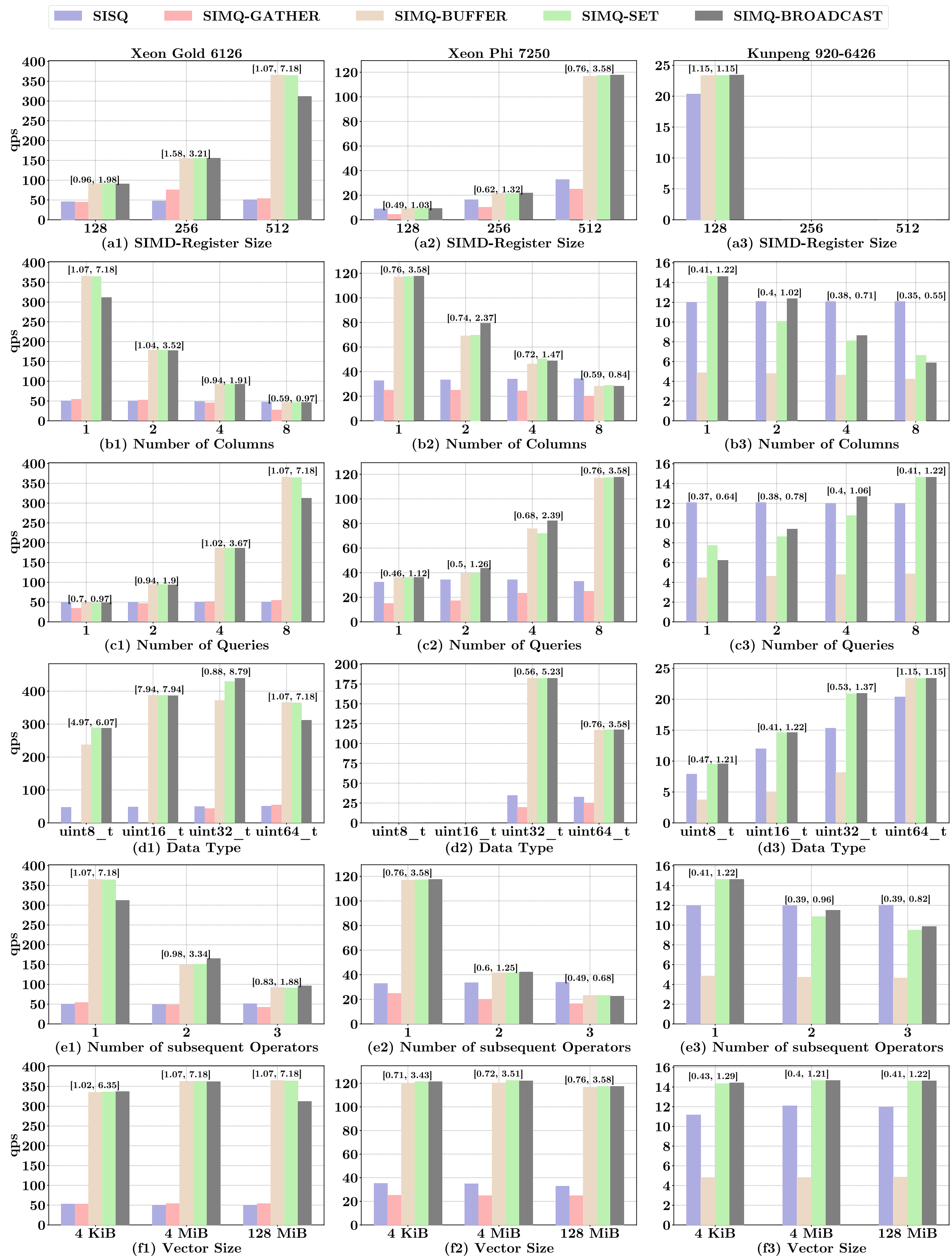

Fig. 6 Evaluation results of our systematic comparison of *SISQ* and *SIMQ* using one thread. We report the number of queries that are executed per second (qps) as bars and the relative improvement of *SIMQ* compared to *SISQ* as a range displayed as the text above the bars

memory access in *SISQ*—was never slower than the *SISQ* baseline except when only two queries are processed, on the Xeon Gold 6126. When processing two queries with a vector size of 128-bit, all linear access *SIMQ* proxies achieved a similar improvement of 1.98x compared to *SISQ*. For wider vector registers `SIMQ-BUFFER` and `SIMQ-SET` perform slightly better compared to their linear access counterpart `SIMQ-BROADCAST`. For 256-bit vector registers, *SIMQ* reached an improvement of around 3.21x. When using 512-bit wide vector registers, `SIMQ-BUFFER` and `SIMQ-SET` were 7.18x times faster, `SIMQ-BROADCAST` achieved improvements of 6.13x. `SIMQ-GATHER` reached a peak improvement compared to the sequential execution when running on 256-bit wide vector registers, which may be a result of reduced core frequency when using 512-bit wide vector registers [18,52]. This impacts the speed of address calculation in vector registers which may effect the memory transfer rate.

Overall, the lower core frequency of the Xeon Phi and the Kunpeng result in fewer improvements compared to *SISQ*. As highlighted in Fig. 6a2, no throughput improvements could be achieved on the Xeon Phi for 128-bit sized vector registers executing two queries in parallel. When executing four queries, using 256-bit vector registers, *SIMQ* achieves an improvement of up to 1.32x. Using 512-bit vector registers as a shared resource leads to an overall workload execution time improvement of 3.58x. While the improvements of SIMQ fall short compared to the execution on the Xeon Gold 6126, it is worth mentioning that the query throughput of *SISQ* scales linearly with the vector width, ranging from 9 *qps* with 128-bit vectors up to 33 *qps* for 512-bit vectors. When using vector registers to share memory access and computations across multiple queries, we can report a query throughput of 22 *qps* using 256-bit wide vector registers and 118 *qps* for 512-bit vector registers, respectively. Running our experiments on the Kunpeng with 128-bit vector registers (cf. Fig. 6a3), *SIMQ* speeds up the workload execution by a factor of 1.15x. This is better than the corresponding results on the Xeon Phi, which is most probably a result of the higher CPU frequency.

Since vector sizes of 512-bit (AVX-512) deliver the overall best improvements, our further evaluation concentrates on that, except for the Kunpeng, where only 128-bit vector registers are available.

(ii) Influence of column-sharing mode As a next step, we evaluated the influence of column-sharing modes by varying the number of accessed unique column pairs. We always executed eight queries per workload. Consequently, we used 64-bit sized data values on Intel platforms and 16-bit sized data values on the ARM platform. The results are shown in the Fig. 6b1–b3. As expected, the number of accessed columns influences the workload performance. While we achieve the best throughput for one column pair for all eight queries (full column-sharing), we achieve the worst for eight unique column pairs for eight queries (no column-sharing) ending up in a slowdown on all platforms. That means, the more different columns were accessed, the less physical memory accesses are omitted when using *SIMQ* resulting in reduced query throughput.

Running on the Xeon Gold, `SIMQ-GATHER` reached similar performance as `SISQ` with an improvement of 1.07x for one column per operator and 1.04x for two column pairs respectively; it ended up slowing down the throughput by 1.06x when accessing four and 1.69x when accessing eight different column pairs (see Fig. 6b1). Notably, the improvement of linear access *SIMQ* proxies decreased with more various locations from 3.52x (2 column pairs) to 1.91x (4 column pairs). While accessing eight different column pairs from 8 queries is indisputably the worst-case scenario from a work-sharing perspective, surprisingly, all measured *SIMQ* variants except for `SIMQ-GATHER` could keep up with the workload execution using *SISQ* resulting in negligible slowdowns. Moreover, we see a similar behavior for the Xeon Phi, yet ending up in less query throughput improvement for every combination (cf. Fig. 6b2). As shown in Fig. 6b3, various data locations harm the performance of our *SIMQ* approach quite drastically, ending up in 1.19x lower qps compared to *SISQ*.

Instead of evaluating eight queries, we can also reduce the number of queries by increasing the data-level parallelism per query at the same time.

(iii) Influence of query count To evaluate this query-sharing impact, we measured a workload consisting of a varying number of queries, which all operate on the same pair of columns consisting of 64-bit values. On the Xeon Gold, we can observe that the overhead of `SIMQ-GATHER` deteriorates the performance for one and two queries, while all other *SIMQ* variants can keep up with the *SISQ* execution model in all cases (see Fig. 6c1). On the one hand, if the number of concurrent queries grows, the *SIMQ* processing pays off rather quickly, resulting in an improvement of around 3.67x for four queries and up to a factor of 7.18x for executing eight concurrent queries. On the other hand, if the number of concurrent queries equals one, then *SISQ* and *SIMQ* perform mostly equally, underlining the generally low overhead of SIMQ.

The results for the Xeon Phi are shown in Fig. 6c2. As already observed in the previous results, the overall throughput improvement achieved is lower compared to the Xeon Gold. However, we can see similar overall behavior, where for at least four queries, our *SIMQ* approach pays off. Additionally, while the overall improvements on the Kunpeng are even lower, compared to the Intel platforms, *SIMQ* needs at least four queries to be executed within a workload to benefit from work-sharing (cf. Fig. 6c3).

To sum up, the achievable improvement grows, when more queries are executed in parallel. However, the maximum

number of concurrent queries is limited to 8 when working with 64-bit values with a vector size of 512-bit and 16-bit values for 128-bit vector registers, respectively. Consequently, we investigated the impact of smaller value sizes as a next evaluation step.

(iv) Influence of value size Figure 6d1–d3 shows the variation of the underlying value size in the spectrum of 8-, 16-, 32- and 64-bit. The number of executed queries depends on the value sizes in a way that the maximum amount of queries was executed. Consequently, when using a 512-bit wide vector register, 8 queries were executed for 64-bit sized values and up to 64 queries for 8-bit sized values, respectively. On the one hand, this has a direct effect on the memory which is processed during workload execution since every query operates on two 128 MiB sized columns. Thus, the overall memory footprint of a *SISQ* workload doubles, when the size of the data type is halved. On the other hand, the value size can be expected to have a significant impact on *SIMQ*. While there is no change in terms of memory access, the number of executed SIMD-transfer operations also doubles when the value size is halved. All queries accessed the same pair of columns.

Since the Xeon Phi does not support the AVX512-BW feature set, which is needed to operate on byte- and half-word sized values within a 512-bit vector register, we did not conduct any experiments for these datatypes on that machine. The Xeon Gold can execute all *SIMQ*-proxies for 32- and 64-bit sized values but does not support `SIMQ-GATHER` with value sizes smaller than that. Thus, `SIMQ-GATHER` was not executed for these value sizes. The fact that the number of executed queries correlates with the value size has an almost linear impact on the workload execution time of the *SISQ* variant, leading to a two-times longer execution time when the size of the processed values are halved on the Xeon Gold and Xeon Phi. This underlines our claim that our queries are memory-bound running on the Xeons using AVX-512 since the overall workload memory footprint that grows with smaller value sizes does not significantly affect the average number of finished queries per second. `SIMQ-BROADCAST` and `SIMQ-SET` are the best performing *SIMQ* proxies for all value sizes except for 64-bit since they need only a single instruction to build up the vector register when accessing one column for the maximum number of queries.

As shown in Fig. 6d2, d3, the value size significantly impacts the performance of *SIMQ* compared to *SISQ*. This leads to an throughput improvement of up to 5.23x for 32-bit values on the Xeon Phi and 1.37x on the Kunpeng. Interestingly, while *SIMQ* reached the most significant qps improvements over *SISQ* when operating with 32-bit values, the overall qps for *SISQ* is quite significantly affected by the value size on the Kunpeng, where bigger values lead to higher query throughputs. Another particularity of *SIMQ* running on the Kunpeng is the `SIMQ-BUFFER`. While the different proxies establish relatively robust and similar improvements on Intel, writing data into an intermediate buffer and subsequently transferring the data into vector registers does not pay off in any situation on ARM, except when processing 64-bit values.

So far, every query within the workload executed a `filter` core followed by an `aggregation-sum` core. However, we assert that our *SIMQ* approach is also beneficial when queries are only similar in structure, but the internal functionality differs. Since our approach aims to use vector registers as a shared resource and vector instructions for sharing instructions, different operations imply disjoint and sequential execution. In general, it would be possible to mitigate this challenge by fusing operators, but this would be beyond the scope of this paper. To verify our claim, we implemented two additional aggregation operators and executed similar queries with different aggregation operators.

(v) Influence of operator variance Figure 6e1–e3 shows the results of our experiment investigating the impact of operator variety. Following our systematic approach, we fixed the vector size to 512-bit for Intel and 128-bit for ARM, respectively. Furthermore, all operators consume the same column pair containing 64-bit (16-bit on ARM) sized values. Moreover, the workload consists of eight queries. The leftmost part of Fig. 6e1–e3 presents our previous introduced workload, where all queries execute a *filter* followed by an *aggregation-sum*. If the number of subsequent operators equals two, four queries within the workload are changed to execute a *minimum aggregation* instead of a *sum*. When processing this workload using the traditional *SISQ* approach, the differences in the overall performance are negligible. However, when using our *SIMQ* approach, the overall query throughput gains over the sequential counterpart on the Xeon Gold decreases by 50% (cf. Fig. 6e-1). The gap between an expected slowdown of approximately 33% and the actual slowdown can be explained on the one hand through the additional costs introduced by the carryless multiplication, see Fig. 5. On the other hand, it can be explained through the increased number of cache misses, leading to a less efficient cache usage in the second stage of the query execution. While in the first stage of the workload execution, the `filter` is executed with full query-share mode, in the second stage, two `aggregation` operators are executed sequentially, contributing to only four queries each, i.e., partial query-sharing mode, transferring the same data into cache twice. However, our *SIMQ* approach outperformed a sequential execution of the queries by a factor of up to 3.3x.

As highlighted in Fig. 6e2, when running on the Xeon Phi, *SIMQ* achieves an improvement of up to 1.25x compared to *SISQ*. On the Kunpeng, the small vectors lead to a slowdown of 1.04x when the query-sharing is reduced within the second stage (cf. Fig. 6e3).

For three subsequent operators, the workload consists of four queries executing an *aggregation-sum*, and two queries execute an *aggregation-min* and an *aggregation-max*, respectively. Consequently, the possible query-sharing potential and memory transfer savings of the second query-stage decreases further and our experiments reflect this. Compared to the *SISQ* execution, we observed a maximum throughput improvement of 1.88x on the Xeon Gold while slowing down the workload execution by 1.21x on the Xeon Phi and 1.22x on the Kunpeng, respectively. The observed behavior shows that our approach is superior to traditional vectorized processing even if the functional sharing potential is only 50%.

(vi) Influence of batch size The general drawback of the operator-at-a-time model is the full materialization of intermediate results. To mitigate the materialization costs, the vector-at-a-time processing model has been proposed, i.e., working on batches or blocks of data that fit into the processor caches [57]. This makes the materialization cheaper because operators are working on cached results of previous operators [57]. The batch sizes should be large enough to limit the function call overhead and small enough to fit in the CPU caches. As shown in [57], this has a positive effect on the performance of analytical queries compared to the classical operator-at-a-time processing. Since our *SIMQ* approach aims at optimizing memory accesses for query workloads, we finally investigated the impact of a vector-at-a-time processing model on the execution time of our workload. Therefore, we divided an input column into batches and forwarded the batches to the appropriate operator. The operator materialized its output, which is then consumed by subsequent operators. Consequently, we wrapped our query execution into a loop, dividing the columns into batches, which either fit into the first level cache or the last level cache, respectively.

To evaluate the impact of the batch size, we varied only this dimension and kept the other dimensions constant. Eight queries executed a *filter*, followed by an *aggregation-sum* operator using 512-bit (128-bit on ARM) vector registers. The queries accessed the same column pair for *full column-sharing* and the value size was fixed to 64-bit. The results of our experiments are depicted in Fig. 6f1–f3. Surprisingly, the number of processed values within one batch does virtually not affect the execution time of *SISQ* processing. Thus, we could show that our proposed *SIMQ* approach can also be successfully applied to the vector-at-a-time processing model. Generally, we observed similar results as for the operator-at-a-time model leading to the same conclusions.

Lessons learned Analyzing these results, we clearly showed that our *SIMQ* concept could increase the workload performance substantially in many situations. As we use vector registers as a multi-scalar register, the degree of query parallelism correlates with the number of available vector lanes, which equals the quotient of the vector size in bits and the processed value size in bits. The higher the query parallelism, the better the improvements (cf. Fig. 6a, c), while modern CPU architectures are optimized for 32-bit and 64-bit wide values, respectively (cf. Fig. 6d).

Since our approach does not omit actual calculations but reuses cache resident data for multiple queries, the more queries access the same data in parallel, the better the achievable improvements (cf. Fig. 6b). Interestingly, we found that—regardless of the sharing potential or variety of memory accesses—vectorized random access via a *gather* cannot compete with other methods using temporal buffers or registers, even though the actual memory access pattern follows a linear pattern. We argue that the hardware implementation of the *gather* instruction is just too expensive compared to the other investigated methods to pay off. We also showed that our approach performs best when the shared operator consumes base data or the intermediate data is not processed prior to the execution, e.g., the number of concurrent operators mismatch the number of the previous one (cf. Fig. 6a, e).

To better understand the possible use-cases, we use two metrics to assess our novel approach of sharing vector registers, namely (i) query latency and (ii) query throughput. Since our approach aims at reducing physical memory access, we take the two corner-cases for data-sharing potential into consideration (cf. Fig. 6b). Since our approach executes multiple queries vector-concurrently, the result of a single query is completed when all queries are processed, which increases the latency of a single query compared with a state-of-the-art vectorized query-at-a-time processing model. Consequently, considering the latency of a single query on the Xeon Gold 6126, we can report that our approach adds 0.35x to the latency of a single state-of-the-art vectorized implementation when all queries share the same data access, up to 7.34x when no physical data access is omitted, respectively. On the Xeon Phi 7250, results are even worse, adding 1.33x up to 8.65x to single query latency, and 5.59x up to 12.86x on the Kunpeng 920-6426, respectively. Even though the latency-increase on the Xeon Phi 7250 is surprisingly low for the optimal case, considering that with around 35% higher latency, the whole workload is finished compared to a single query, we argue that our approach is not a good fit for latency optimization.

However, as described above, using vector registers as a shared resource offers excellent potential to improve the query throughput. This advancement is achieved due to better cache utilization or physical I/O reduction through *SIMQ* since, in contrast to *SISQ*, every value from a shared column must be moved into the cache only once for the whole workload execution. Moreover, even for the corner-case of no possible data sharing, our approach introduces an arguably small overhead of 3.1% on the Xeon Gold 6126 (19% on the Xeon Phi 7250, 81.8% on the Kunpeng 920-6426).

As multi-threaded execution can be considered as a common approach to increase the overall query throughput, we

conducted respective experiments as a next step to better understand how our approach can be used in such an environment.

4.2 Multi-threaded evaluation

Besides single-thread performance, modern database systems try to exploit the available hardware parallelism as much as possible. Therefore, the thread-level parallelism and simultaneous multi-threading (SMT) capabilities of the employed processors should be used to their maximum potential. With SMT, a single processor core is able to execute multiple threads concurrently. Hence, we distinguish between physical and logical cores. Table 1 shows the available threads per single processor core in brackets. Clearly, the Kunpeng processor does not feature any SMT capabilities, while the cores of Xeon Gold or Phi can execute 2 or 4 threads concurrently.

Multi-threaded execution features several important use cases. First, parallel query execution can increase the system performance by a factor equal or close to the number of employed threads. This execution model can be used twofold: (i) one query is assigned to one thread and multiple queries run in parallel (inter-query parallelism) and (ii) one query uses multiple threads to accelerate data-independent operations (intra-query parallelism). For this paper, we leverage multi-threading for inter-query parallelism.

With the rise of cloud computing, modern database systems face a varying availability of compute resources or a strict limitation to a fixed number of processor cores. Resource availability can change depending on the current system load, i.e., over- or under-provisioning due to heavy- or small load; or simply through throttling the hardware to save on money. We see this as a second important use case, where we investigate the effect of SIMQ on the overall query throughput given a specific budget of computing resources, compared to traditional SISQ.

In general, in this section, on the one hand, we present our results comparing our *SIMQ* approach with inter-query parallel vectorized processing. On the other hand, we demonstrate that our *SIMQ* approach can also benefit when used in a parallel manner in a multi-threaded architecture.

4.2.1 Parallel SISQ vs. SIMQ

For the experiments in Fig. 7, we set the number of employed threads for SISQ equal to the number of executed queries in the SIMQ case. That is, if SIMQ packs 8 queries into one 512-bit wide register, SISQ is executed with 8 parallel threads. For the Kunpeng, threads were pinned to succeeding physical cores. Since the Xeon Phi is a MIC (Many Integrated Core), which is, like the Xeon Gold, designed to provide heavy parallelism through SMT, we packed the execution threads densely on a physical core and all of its corresponding logical cores. All investigated platforms also exhibit a Non-Uniform-Memory-Access (NUMA) behavior due to being multiprocessor systems. To avoid unpredictable NUMA-effects during runtime, we did only allow the usage of threads and memory of a single socket (i.e., one processor). In the case of parallel SISQ, we measured the start and end-time per thread and calculated the overall runtime by accumulating the time needed for executing the query per thread. For SIMQ, we can simply measure the runtime of one operator execution. Consequently, we only used a single thread to process the specific workload. Our improvement metrics do not only consider the query throughput of the system, but also the overall resource efficiency. That is, if SIMQ can complete 8 queries using 1 thread in the same time as SISQ with 8 threads, it exhibits a relative improvement of 8x, since we only use $\frac{1}{8}$ of the resources of SISQ. We organized the experiments according to the description in Sect. 4.1.

(i) Influence of SIMD-register size Figure 7a1–a3 depicts the relative improvement of SIMQ compared to SISQ, when differently sized vector registers are used for processing 64-bit values of a single column. As already described in Sect. 4.1, we can only report results for 128-bit wide registers for the Kunpeng processor, since it does not provide any larger registers. SIMQ can process 2, 4 and 8 queries in one register for 128, 256 and 512-bit wide registers, respectively. At first sight, we observe that both Intel platforms generally display the same scaling behavior with increasing register size. However, we want to point out that the vector register size correlates with the number of queries within our workload. Consequently, the number of threads grows linear with the SIMD-register size for *SISQ*. Comparing the qps on the Xeon Gold and the Kunpeng for 128-bit wide registers, executing two queries on two threads concurrently with a sequential execution (cf. Fig. 6a1), we can report an improvement of almost 2. The same observation holds for 256-bit and 512-bit wide registers on the Xeon Gold, respectively. As mentioned in the previous section, our queries are memory-bound on the Xeon Gold 6126; wider vector registers do not improve the query throughput. Thus, the qps raise is an immediate consequence of inter-query parallelism.

However, our workload execution using a single thread heavily benefits from higher data-level parallelism on the Xeon Phi by nearly doubling the qps when the size of the used vector register is increased by 2x. Consequently, we expected a corresponding improvement for the inter-query parallel processing. Interestingly, a second thread improves the execution of two queries using SSE by only 1.35x, 1.9x with four threads using AVX2, and 3.39x with eight threads and AVX512, respectively. We argue that the absence of an L3 cache, shared among all cores and can therefore be used to avoid redundant memory transfers, leads to this suboptimal improvement.

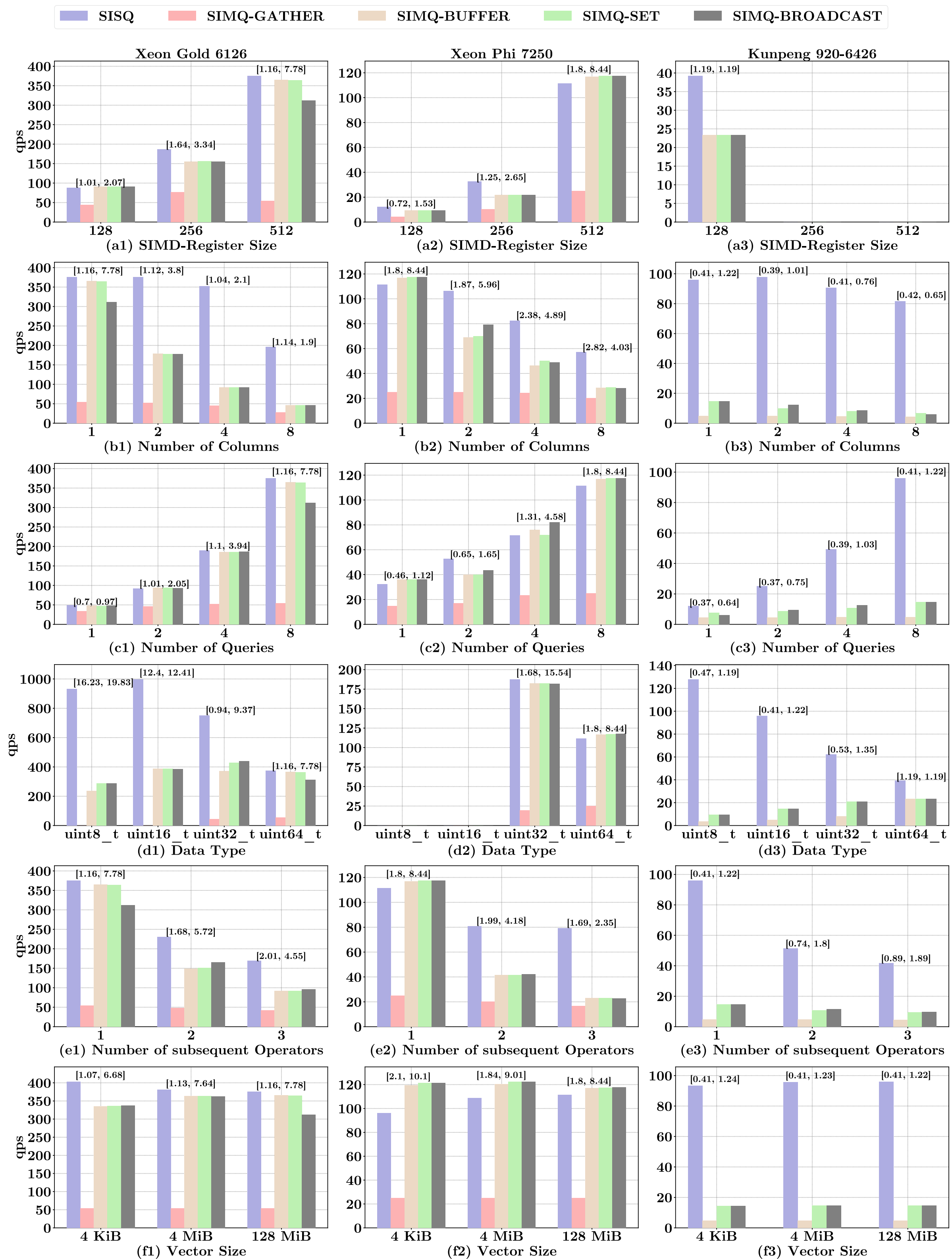

Fig. 7 Evaluation results of our systematic comparison of *SISQ* (1 thread per query) and *SIMQ* (1 thread total). We report the number of queries that are executed per second by the system as bars and the relative improvement of reached qps per thread of *SIMQ* compared to *SISQ* as a range displayed as the text above the bars

Comparing the average qps of a multi-threaded *SISQ* execution with our *SIMQ* approach, we see that for almost all cases, inter-query parallelism outperforms our approach slightly. However, considering the deployed resources, *SIMQ* achieves a higher efficiency on all investigated platforms up to 8.44x.

(ii) Influence of column-sharing mode We executed the experiments of Fig. 7b1–b3 with 64-bit data elements within AVX512 registers and thus eight concurrent queries on the Intel processors and 16-bit data elements for eight queries on the Kunpeng. As expected, a lower degree of shared columns, i.e., a higher column count, leads to an overall decrease in the performance. While this is expected for *SIMQ*—the more columns we have to load into one register, the fewer data can be shared—it also applies for *SISQ* although to a lesser extent. This stems from the decrease in implicitly sharable data access through the shared caches. Here, the scenario of 1 shared column among all threads is expectedly the best case for SIMQ, with an improvement of 7.78x and 8.44x on the Xeon Gold and Phi, respectively. Contrary to the Intel hardware, SIMQ can only outperform SISQ when accessing less than four columns on the Kunpeng processor.

(iii) Influence of query count This set of experiments was executed on AVX512 registers with 64-bit data elements on Intel processors and with 16-bit elements for Kunpeng; all are using one column. Figure 7c1 and c2 basically follows the trend from Fig. 7a1 and a2. We observe that increasing the query count of SIMQ inside one AVX512 register behaves almost identical to using the maximum query count for SSE (128-bit) or AVX2 registers (256-bit). An actual relative improvement of SIMQ over SISQ on the Kunpeng processor can be observed for at least 4 concurrent queries. Below that threshold, the data massage inside the SIMQ proxy creates too much overhead.

(iv) Influence of value size The value size directly influences the number of concurrent queries inside a register. We ran these experiments on 64-, 32-, 16- and 8-bit wide data elements using AVX512 registers for Intel processors, while varying the thread count for SISQ between 8, 16, 32 and 64 queries. For the Kunpeng processor, this results in 2, 4, 8, 16 threads, due to narrower register size of ARM Neon. All queries share one column, i.e., full column-sharing. Notably, the Intel Xeon Phi does neither support vectorized processing of 16- or 8-bit elements with 512-bit vector registers, and thus we did not run the experiments for these configurations. On the Xeon Gold, we see a tremendous increase of qps for *SISQ* when operating on smaller values. However, considering the qps per core, we can report a relative improvement of *SIMQ* of up to almost 20x for 8-bit values. It must be noted that for 8-bit values, 64 queries are executed on 64 threads and 32 threads for 16-bit values, respectively. Since the Xeon Gold has only 12 physical (and 24 logical) cores, every thread executes two to three queries. Consequently, the query throughput depends on the thread scheduling to a certain extent. We argue that this is the main reason for the lower improvements for 8-bit and 16-bit values. On the Xeon Phi, the *SISQ* query throughput is almost increased by 2x when processing 32-bit values with 16 threads compared to 64-bit values with eight threads. Surprisingly, *SIMQ* can keep up with this improvement leading to a 2x higher relative improvement (15.54x) compared to 64-bit values. On Kunpeng, *SIMQ* cannot compete with inter-query parallelism from a pure query throughput perspective. Especially for smaller values, we see a significant descent of the query throughput, probably due to the increased overhead of reorganizing the data. However, when it comes to resource utilization, *SIMQ* is slightly better compared to *SISQ* with a maximum improvement of 1.35x with 32-bit values.

(v) Influence of operator variance As for the previous setups, the experiments in Fig. 7e1, e2 use AVX512 with 64-bit data and thus 8 concurrent queries, while Kunpeng in Fig. 7e3 uses 16-bit data; all three employ full column-sharing. We expected the query throughput for *SISQ* to be unaffected by operator variability. Surprisingly, the experiments show that different operators decrease the performance significantly. On the one hand, this probably results from the additional memory which contains the instruction code, primarily when two logical cores execute different code paths since the L1i-cache has to be flushed whenever a context switch occurs. On the other hand, the vectorized min/max vector instructions have a higher latency than the vectorized sum instruction. Notably, this benefits the relative improvement per core of our *SIMQ* approach with up to 4.55x better efficiency on the Xeon Gold, 2.35x on the Xeon Phi, and 1.89x on ARM for three subsequent operators, respectively.

(vi) Influence of batch size Varying the batch sizes for our workload execution does not strongly influence the average query throughput. Interestingly, the impact on *SISQ* is contrary to *SIMQ* for the Xeon Gold and Xeon Phi. On the Xeon Gold, smaller batches of data lead to higher query throughputs for *SISQ* while it harms *SIMQ*. Unlike for the Xeon Gold, *SISQ* benefits from bigger batches on the Xeon Phi, while a batch-at-a-time processing style has no significant impact on the query throughput of our *SIMQ* approach.

Lessons learned Inter-query parallelism can be very beneficial in a scale-up system for the overall query throughput. This seems to be particularly the case when the executed queries are memory-bound, and the underlying memory bus system does not work to capacity serving a single thread. If the workload offers shareable memory access, inter-query parallelism may benefit more by omitting costly memory transfers into the cache if the cache is shared across multiple cores. However, when taking the number of used processing resources into account, *SIMQ* improves the overall efficiency while keeping up with the multi-threaded state-of-the-art processing model in some best-case scenarios.

4.2.2 Parallelized throughput on a thread budget

The budgeted execution experiment limits the number of parallel threads to 8, while we used the same thread-to-core assignment as in the previous experiment. First, we spawned all threads and synchronized the start of their workload executions. Once all threads were ready to go, they were triggered to infinitely execute our query patterns, but with continuously varying filter predicates to simulate a more natural workload scenario. We measured the overall throughput by incrementing the per-thread query count by 1 after each complete iteration for SISQ and simply by 8 for SIMQ, since we pack 8 queries into one register. The experiments were then executed for 60 seconds each. Figure 8 shows the relative query throughput of SIMQ compared to SISQ on all processors.

To compare the different experiments, we divide the total executed queries of *SISQ* by the accumulated CPU time of all *SISQ* threads in seconds (qps_{SISQ}). The same applies to the queries executed using *SIMQ* (qps_{SIMQ}). The relative improvement is then calculated as the quotient of qps_{SIMQ} and qps_{SISQ}.

(i) Influence of SIMD-register size As already described in the previous sections, in our first experiment, we varied only the size of the used vector register. All queries accessed a single pair of columns, consisting of 64-bit values. The *SIMQ* variants executed two queries per thread in parallel when using 128-bit vector registers and four queries for 256-bit, and eight queries for 512-bit. Consequently, using eight threads in total, the *SIMQ* approach executes up to 64 queries concurrently compared to only eight queries when using the traditional *SISQ* approach.

In contrast to the previous experiments (cf. Fig. 7a1), the query throughput remains nearly unaffected by wider vector registers when processing one query after another, going up from 344.15 queries per second to 373.42 qps for 256-bit vector registers and 375.4 qps for 512-bit vector registers, respectively. Contrary, when using *SIMQ*, doubling the vector size leads to a almost two times higher query throughput, starting with 478.08 qps when operating on 128-bit vector registers and 886.11 qps or 1715.98 qps for either AVX2 or AVX512. Consequently, *SIMQ* achieved an up to 4.57x higher throughput than its *SISQ* counterpart on the Xeon Gold which is shown in Fig. 8a1. On the Xeon Phi traditional vector processing benefits from wider vector registers. Using 128-bit vector registers, we observed an average query throughput for our *SISQ* implementation of 28.92 qps. When the vector register size is doubled, the throughput increases to 62.13 qps, ending up with 108.57 qps for 512-bit vector registers. As expected, the relative improvement of *SIMQ* for wider vector-registers on the Xeon Phi is similar to the Xeon Gold, yet lower in general. At the maximum, the query throughput is increased by a factor of 2.57x, resulting in a query throughput of 278.55 qps using `SIMQ-BUFFER`. Running on the Kunpeng, *SISQ* processing finished on average 147.34 queries per second while *SIMQ* carried out up to 193.72 qps resulting in a maximum improvement of 1.33x.

(ii) Influence of column-sharing mode When comparing Fig. 8b1–b3 to our single-thread experiments (cf. Fig. 6b1–b3), we can see a similar picture. However, the achieved improvements of *SIMQ* are slightly less than their single threaded counterpart. Surprisingly, on the Xeon Phi the random access *SIMQ*-proxy, i.e., `SIMQ-GATHER`, improved the overall query throughput by up to 1.55x, executing 107.04 qps compared to 69.14 qps for *SISQ*, as long as there were some data accesses to share. While on the Kunpeng, we observed at least the same performance for one and two accessed columns in the single thread scenario. Accessing more different columns, however, leads to a decreased query throughput. As already seen in the previous experiments, the `SIMQ-BUFFER` does not pay off in any situation on ARM, except when processing 64-bit values.

(iii) Influence of query count In this experiment, all queries used AVX512 registers with 64-bit data elements and 16-bit elements on the Kunpeng, respectively. All queries operate on one column pair. Figure 8c1, c2 again follows the trend from Fig. 8a1, a2. The more queries were executed in parallel using *SIMQ*, the better the improvements. Notably, the introduced overhead for preparing the vector registers to be used via *SIMQ* processing does not vanish on the Xeon Phi. This leads to an improvement for one query using *SIMQ* of up to 1.7x and 1.77x for two queries, resulting in 148.3 qps and 153.44 qps, respectively. The results of our experiments on the Kunpeng, showed in Fig. 8c3 again show that on ARM using 128-bit vector registers, *SIMQ* needs at least four queries to pay off.

(iv) Influence of value size The results of our experiments investigating the impact of the processed value size are depicted in Fig. 8d1–d3. As already described, the biggest available vector register was used. Every query accessed on column pair and the number of queries executed by our *SIMQ* approach depends on the investigated value size. When 64-bit values are processed, eight queries were executed in parallel per thread, four queries for 32-bit values, respectively. While the Xeon Phi only fully supports these two value sizes, the Xeon Gold and the Kunpeng also support vector processing on byte and word-sized values. As shown in Fig. 8d1, d2, the relative improvements on the tested follow the same trend as from our single-thread evaluation (cf. Fig. 6d1, d2). *SISQ* achieved the highest qps when operating with 64-bit values while *SIMQ* reached its peak performance on Intel machines with 32-bit values and with 64-bit values on the Kunpeng, respectively.

(v) Influence of operator variance The impact of operator sharing on the Xeon Gold, depicted in Fig. 8e1 is similar to the single-thread scenario, shown in Fig. 6e1, yet again,

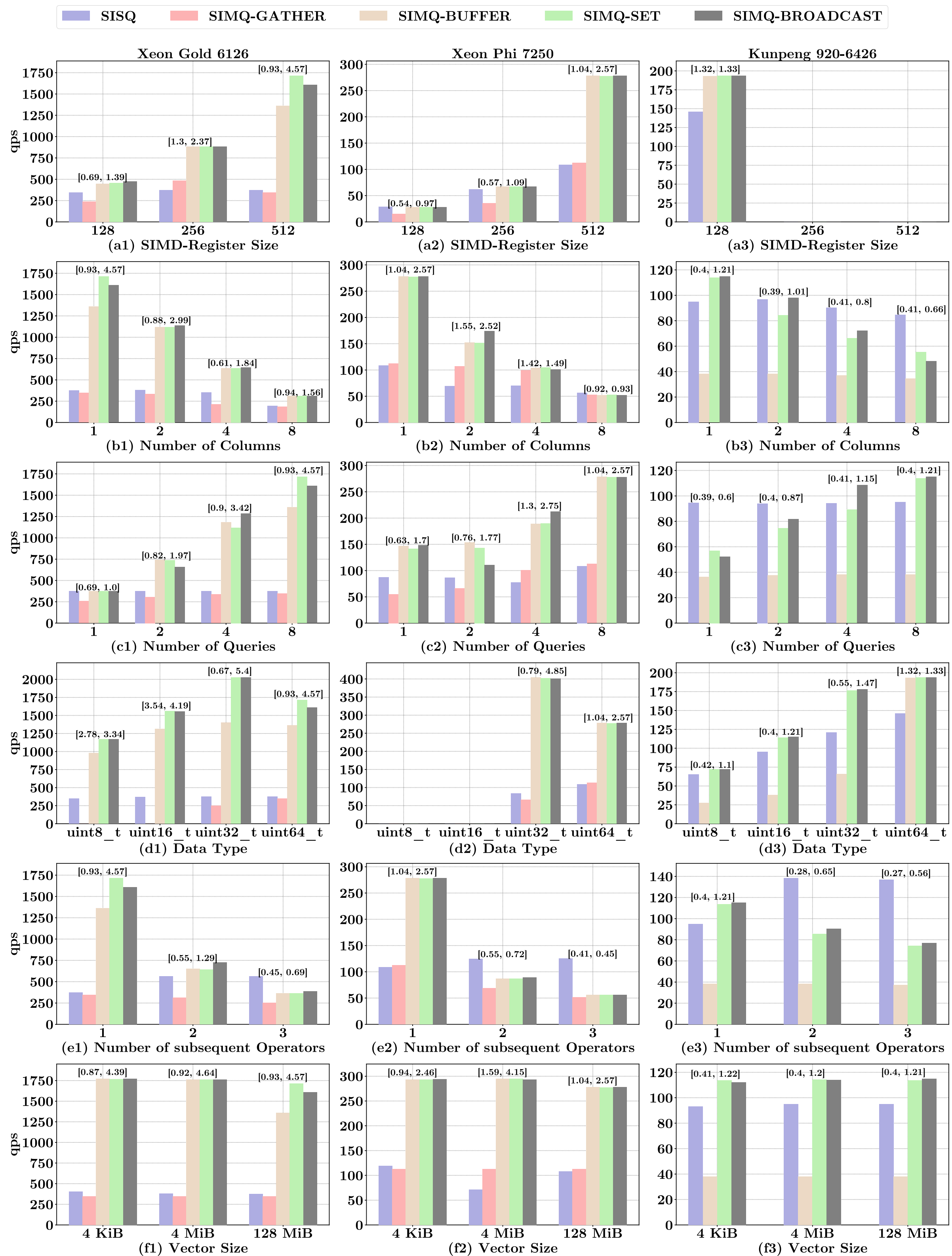

Fig. 8 Evaluation results of our systematic comparison of *SISQ* and *SIMQ* using a thread budget of 8 threads

the improvements of *SIMQ* over *SISQ* are smaller compared to their single-thread counterpart. On the Xeon Phi, this trend is continuing. However, while *SIMQ* in the single-thread case is beneficial for up to two subsequent operators, when executed in an SMT environment, the queries have to be structural and functional congruent to be efficiently parallelized via *SIMQ*. The same applies to the Kunpeng, wherein the best-case scenario, an improvement of up to 1.21x, which translates to 115.02 qps compared to 90.27 qps using *SISQ*, could be reached. In the measured worst-case scenario, *SIMQ* reduces the query throughput by up to 1.79x (76.81 qps) for `SIMQ-BROADCAST` and 3.7x (37.14 qps) when using the generally sub-optimal performing `SIMQ-BUFFER` on ARM.

(vi) Influence of batch size As already discussed in the previous evaluation sections, when executing our queries using a vector/block-at-a-time processing model, the used batch size does only slightly affect the overall behavior on the Xeon Gold and Kunpeng.

Lessons Learned: As expected, *SIMQ* can also leverage inter-query parallelism. Consequently, the query throughput can be significantly improved compared to a state-of-the-art vectorized processing. As already shown in the previous experiments, the different *SIMQ*-proxies perform pretty similarly, except for the *gather*. However, it seems to be the case that every architecture has a single proxy that performs best, regardless of the particular situation. On the Xeon Gold, we see that `SIMQ-SET` performs best, while it is `SIMQ-BROADCAST` on the Xeon Phi and Kunpeng, respectively.

5 Discussion

Existing work-sharing solutions typically aim to decrease I/O costs by increasing the number of computations on data in fast memory, e.g., cache, or even wholly omitting redundant memory accesses. The first class is typically carried out through highly specialized database operators like a Cooperative scan. The latter is implemented by reusing common intermediate results of several queries through plan adjustments and intermediate materialization.

Our approach, however, uses state-of-the-art SIMD extensions to trade hardware-provided data-level parallelism for query parallelism on a conceptual level. Furthermore, we designed our prototype as a proxy which may replace the data accessing part of an arbitrary existing vectorized database operator. Therefore, we argue that our approach could be compatible with other work-sharing techniques to improve those even further.

However, to better grasp our approach, we compare it with two prominent representatives of the work-sharing classes described above, namely (i) Cooperative Scans and (ii) MQO, respectively.

Cooperative scans are specialized Scan-Operators that repeatedly iterate over a chunk of data—typically base-data—and execute multiple scan operations on the loaded data for every query which registered itself in a dedicated queue. Cooperative Scans enable diverse predicate comparators and predicate values on the same predicate attribute.

Our approach, however, fundamentally changes the way of transferring and organizing data into vector registers. Therefore, a higher degree of freedom in data access can be achieved than with Cooperative Scans since it is possible to load from different base data locations and even intermediates. However, as already mentioned above, our approach depends on database operators using the exact vector instructions.

MQO denotes a class of workload optimization techniques, where similar (sub)queries within a workload are identified, and the results are calculated once, materialized, and reused by other queries. While this can be very beneficial for reducing a given workload's compute and memory costs, a crucial requirement for MQO is sharable (sub)queries where one subquery produces the same result or at least a superset of the results required by the other (sub)queries. This necessity makes determining suitable candidates for materializing intermediate results particularly challenging.

Our approach does not omit any computations but executes one database operator of multiple queries at a time using SIMD data-level parallelism. The results of every executed operator are zipped together, producing a superset of those operators. While this reduces the complexity of finding suitable sharing candidates, the result size grows linearly by the number of parallel executed operators. By creating those zipped supersets of multiple results, two challenges arise. On the one hand, it may be necessary to partially unzip the combined results if the succeeding operator does not exhibit the same sharing potential as the prior operators.

This led to a remarkable decline in the overall speedup of our approach, as depicted in Figs. (6, 7, 8)e. On the other hand, since our approach can combine and concurrently execute operators that access different data of various sizes, the number of concurrent operators may decline over time. It remains to be examined how those shifts of concurrency can be addressed efficiently from an algorithmic point of view. However, we argue that both challenges underline the necessity of a runtime adaptive proxy design, which enables workload- and data-dependent adjustments for using vector registers as a shared resource across multiple database operators.

Our experiments showed that our novel approach of using vector registers as a shared resource could substantially increase the overall query throughput of a workload. However, we argue that the decision about which query-operators can be shared and how they should be shared is not trivial and opens an interesting field for further research.

6 Conclusion

In this paper, we have presented a novel approach facilitating vector registers as a work-sharing resource. We implemented and systematically compared a state-of-the-art fully vectorized single-query (*SISQ*) execution with our novel proposed multi-query vectorized (*SIMQ*) execution approach for workloads consisting of concurrent queries with a similar structure. Our key finding is that *SIMQ* achieves high speedups compared to the state-of-the-art *SISQ* execution in a single-threaded environment. Furthermore, we observed a reasonable *SIMQ* performance without significant drawbacks compared to *SISQ* in worst-case scenarios such as (i) no data-sharing between concurrent queries or (ii) only a single query within a workload. In addition, we could show that *SIMQ* can be efficiently used to reduce the overall computation resources needed to execute a workload while keeping up in terms of execution time in a multi-thread environment. Also, we demonstrated that using a multi-threaded multi-query vectorized approach in an environment with strict limitations to the available compute resources can significantly increase the average query throughput. Thus, we deduce that *SIMQ* can be very beneficial in single-threaded as well as multi-threaded environments. Finally, we showed that there is no single best way of implementing *SIMQ* and the advantage depends on the size of used vector registers as well as workload and query-specific parameters. Thus, we conclude that vector registers offer great potential as a work-sharing resource and this potential increases with growing vector sizes which open up a broad spectrum of future research opportunities.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abadi, D., Boncz, P.A., Harizopoulos, S., Idreos, S., Madden, S.: The design and implementation of modern column-oriented database systems. Found. Trends Databases **5**(3), 197–280 (2013)
2. Abadi, D.J., Boncz, P.A., Harizopoulos, S.: Column oriented database systems. PVLDB **2**(2), 1664–1665 (2009)
3. Abadi, D.J., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: SIGMOD (2006), pp. 671–682
4. Balkesen, C., Alonso, G., Teubner, J., Özsu, M.T.: Multi-core, main-memory joins: sort vs. hash revisited. PVLDB **7**(1), 85–96 (2013)
5. Balkesen, C., Teubner, J., Alonso, G., Özsu, M.T.: Main-memory hash joins on modern processor architectures. IEEE Trans. Knowl. Data Eng. **27**(7), 1754–1766 (2015)
6. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core CPUs. In: SIGMOD (2011), pp. 37–48
7. Boncz, P.A., Kersten, M.L., Manegold, S.: Breaking the memory wall in MonetDB. Commun. ACM **51**(12), 77–85 (2008)
8. Boncz, P.A., Zukowski, M., Nes, N.: MonetDB/x100: hyper-pipelining query execution. In: CIDR, pp. 225–237 (2005)
9. Candea, G., Polyzotis, N., Vingralek, R.: A scalable, predictable join operator for highly concurrent data warehouses. PVLDB **2**(1), 277–288 (2009)
10. Chhugani, J., Nguyen, A.D., Lee, V.W., Macy, W., Hagog, M., Chen, Y., Baransi, A., Kumar, S., Dubey, P.: Efficient implementation of sorting on multi-core SIMD CPU architecture. PVLDB **1**(2), 1313–1324 (2008)
11. Damme, P., Ungethüm, A., Hildebrandt, J., Habich, D., Lehner, W.: From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. ACM Trans. Database Syst. **44**(3), 9:1–9:46 (2019)
12. Damme, P., Ungethüm, A., Pietrzyk, J., Krause, A., Habich, D., Lehner, W.: Morphstore: analytical query engine with a holistic compression-enabled processing model. CoRR arXiv:2004.09350 (2020)
13. Fang, Z., Zheng, B., Weng, C.: Interleaved multi-vectorizing. Proc. VLDB Endow. **13**(3), 226–238 (2019)
14. Feng, Z., Lo, E., Kao, B., Xu, W.: Byteslice: pushing the envelop of main memory data processing with a new storage layout. In: SIGMOD (2015), pp. 31–46
15. Flynn, M.J.: Some computer organizations and their effectiveness. IEEE Trans. Comput. **21**(9), 948–960 (1972)
16. Giannikis, G., Alonso, G., Kossmann, D.: Shareddb: killing one thousand queries with one stone. PVLDB **5**(6), 526–537 (2012)
17. Giannikis, G., Makreshanski, D., Alonso, G., Kossmann, D.: Workload optimization using shareddb. In: SIGMOD (2013), pp. 1045–1048
18. Gottschlag, M., Brantsch, P., Bellosa, F.: Automatic core specialization for AVX-512 applications. In: SYSTOR 2020: the 13th ACM international systems and storage conference, Haifa, Israel, October 13–15, 2020. ACM, pp. 25–35 (2020)
19. Habich, D., Damme, P., Ungethüm, A., Pietrzyk, J., Krause, A., Hildebrandt, J., Lehner, W.: Morphstore-in-memory query processing based on morphing compressed intermediates LIVE. In: SIGMOD, pp. 1917–1920 (2019)
20. Harizopoulos, S., Shkapenyuk, V., Ailamaki, A.: A simultaneously pipelined relational query engine. In: SIGMOD, Qpipe, pp. 383–394 (2005)
21. Hughes, C.J.: Single-Instruction Multiple-Data Execution. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, San Rafael (2015)

22. Johnson, R., Hardavellas, N., Pandis, I., Mancheril, N., Harizopoulos, S., Sabirli, K., Ailamaki, A., Falsafi, B.: To share or not to share? In: VLDB, pp. 351–362 (2007)
23. Kim, C., Sedlar, E., Chhugani, J., Kaldewey, T., Nguyen, A.D., Blas, A.D., Lee, V.W., Satish, N., Dubey, P.: Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. PVLDB **2**(2), 1378–1389 (2009)
24. Lang, C.A., Bhattacharjee, B., Malkemus, T., Wong, K.: Increasing buffer-locality for multiple index based scans through intelligent placement and index scan speed control. In: VLDB, pp. 1298–1309 (2007)
25. Lang, H., Mühlbauer, T., Funke, F., Boncz, P.A., Neumann, T., Kemper, A.: Data blocks: hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: Özcan, F., Koutrika, G., Madden, S. (eds.) SIGMOD, pp. 311–326 (2016)
26. Lang, H., Passing, L., Kipf, A., Boncz, P.A., Neumann, T., Kemper, A.: Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. VLDB J. **29**(2–3), 757–774 (2020)
27. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. Softw. Pract. Exp. **45**(1), 1–29 (2015)
28. Lemire, D., Boytsov, L., Kurz, N.: SIMD compression and the intersection of sorted integers. Softw. Pract. Exp. **46**(6), 723–749 (2016)
29. Makreshanski, D., Giannikis, G., Alonso, G., Kossmann, D.: Mqjoin: efficient shared execution of main-memory joins. PVLDB **9**(6), 480–491 (2016)
30. Makreshanski, D., Giceva, J., Barthels, C., Alonso, G.: Batchdb: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In: SIGMOD, pp. 37–50 (2017)
31. Menon, P., Pavlo, A., Mowry, T.C.: Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. PVLDB **11**(1), 1–13 (2017)
32. Pietrzyk, J., Habich, D., Lehner, W.: To share or not to share vector registers? In; 16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020, pp. 12:1–12:10 (2020)
33. Pietrzyk, J., Ungethüm, A., Habich, D., Lehner, W.: Fighting the duplicates in hashing: conflict detection-aware vectorization of linear probing. In: BTW, pp. 35–53 (2019)
34. Plaisance, J., Kurz, N., Lemire, D.: Vectorized vbyte decoding. CoRR arXiv:1503.07387 (2015)
35. Polychroniou, O., Raghavan, A., Ross, K.A.: Rethinking SIMD vectorization for in-memory databases. In: SIGMOD, pp. 1493–1508 (2015)
36. Polychroniou, O., Ross, K.A.: A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In: SIGMOD, pp. 755–766 (2014)
37. Polychroniou, O., Ross, K.A.: Vectorized bloom filters for advanced SIMD processors. In: DaMoN@SIGMOD, pp. 6:1–6:6 (2014)
38. Polychroniou, O., Ross, K.A.: Efficient lightweight compression alongside fast scans. In: DaMoN@SIGMOD, pp. 9:1–9:6 (2015)
39. Polychroniou, O., Ross, K.A.: Towards practical vectorized analytical query engines. In: DaMoN@SIGMOD, pp. 10:1–10:7 (2019)
40. Qiao, L., Raman, V., Reiss, F., Haas, P.J., Lohman, G.M.: Main-memory scan sharing for multi-core CPUs. PVLDB **1**(1), 610–621 (2008)
41. Raman, V., Swart, G., Qiao, L., Reiss, F., Dialani, V., Kossmann, D., Narang, I., Sidle, R.: Constant-time query processing. In: ICDE, pp. 60–69 (2008)
42. Rehrmann, R., Binnig, C., Böhm, A., Kim, K., Lehner, W., Rizk, A.: Oltpshare: the case for sharing in OLTP workloads. PVLDB **11**(12), 1769–1780 (2018)
43. Roussopoulos, N.: View indexing in relational databases. ACM Trans. Database Syst. **7**(2), 258–290 (1982)
44. Sanchez, J.: A review of star schema benchmark. CoRR arXiv:1606.00295 (2016)
45. Satish, N., Kim, C., Chhugani, J., Nguyen, A.D., Lee, V.W., Kim, D., Dubey, P.: Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In: SIGMOD, pp. 351–362 (2010)
46. Sellis, T.K.: Multiple-query optimization. ACM Trans. Database Syst. **13**(1), 23–52 (1988)
47. Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Prémillieu, N., Reid, A., Rico, A., Walker, P.: The ARM scalable vector extension. IEEE Micro **37**(2), 26–39 (2017)
48. Teubner, J., Müller, R.: How soccer players would do stream joins. In: Sellis, T.K., Miller, R.J., Kementsietsidis, A., Velegrakis, Y. (eds.) SIGMOD, pp. 625–636 (2011)
49. Ungethüm, A., Pietrzyk, J., Damme, P., Habich, D., Lehner, W.: Conflict detection-based run-length encoding—AVX-512 CD instruction set in action. In: ICDE Workshops, pp. 96–101 (2018)
50. Ungethüm, A., Pietrzyk, J., Damme, P., Krause, A., Habich, D., Lehner, W., Focht, E.: Hardware-oblivious SIMD parallelism for in-memory column-stores. In: CIDR (2020)
51. Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., Schaffner, J.: Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. PVLDB **2**(1), 385–394 (2009)
52. Zarubin, M., Damme, P., Krause, A., Habich, D., Lehner, W.: SIMD-MIMD cocktail in a hybrid memory glass: shaken, not stirred. In: SYSTOR '21: The 14th ACM International Systems and Storage Conference, Haifa, Israel, June 14–16, 2021. ACM, pp. 17:1–17:12 (2021)
53. Zhao, W.X., Zhang, X., Lemire, D., Shan, D., Nie, J., Yan, H., Wen, J.: A general simd-based approach to accelerating compression algorithms. ACM Trans. Inf. Syst. 33(3), 15:1–15:28 (2015)
54. Zhou, J., Larson, P., Freytag, J.C., Lehner, W.: Efficient exploitation of similar subexpressions for query processing. In: SIGMOD, pp. 533–544 (2007)
55. Zhou, J., Ross, K.A.: Implementing database operations using SIMD instructions. In: SIGMOD, pp. 145–156 (2002)
56. Zukowski, M., Boncz, P.A.: From x100 to vectorwise: opportunities, challenges and things most researchers do not think about. In: SIGMOD, pp. 861–862 (2012)
57. Zukowski, M., Boncz, P.A., Nes, N., Héman, S.: MonetDB/x100—a DBMS in the CPU cache. IEEE Data Eng. Bull. **28**(2), 17–22 (2005)
58. Zukowski, M., Héman, S., Nes, N., Boncz, P.A.: Cooperative scans: dynamic bandwidth sharing in a DBMS. In: VLDB, pp. 723–734 (2007)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.