SIMD-ified R-tree Query Processing and Optimization

Yeasir Rayhan and Walid G. Aref
Purdue University, West Lafayette, IN, USA
{yrayhan,aref}@purdue.edu

ABSTRACT

The introduction of Single Instruction Multiple Data (SIMD) instructions in mainstream CPUs has enabled modern database engines to leverage data parallelism by performing more computation with a single instruction, resulting in a reduced number of instructions required to execute a query as well as the elimination of conditional branches. Though SIMD in the context of traditional database engines has been studied extensively, it has been overlooked in the context of spatial databases. In this paper, we investigate how spatial database engines can benefit from SIMD vectorization in the context of an R-tree spatial index. We present vectorized versions of the spatial range select, and spatial join operations over a vectorized R-tree index. For each of the operations, we investigate two storage layouts for an R-tree node to leverage SIMD instructions. We design vectorized algorithms for each of the spatial operations given each of the two data layouts. We show that the introduction of SIMD can improve the latency of the spatial query operators up to 9×. We introduce several optimizations over the vectorized implementation of these query operators, and study their effectiveness in query performance and various hardware performance counters under different scenarios.

CCS CONCEPTS

• **Information systems** → **Query optimization**.

KEYWORDS

Single Instruction Multiple Data (SIMD), Spatial Query Processing, R-tree, Query Optimization

ACM Reference Format:
Yeasir Rayhan and Walid G. Aref. 2023. SIMD-ified R-tree Query Processing and Optimization. In *The 31st ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL '23), November 13–16, 2023, Hamburg, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3589132.3625610

1 INTRODUCTION

With the popularity and ubiquity of smart phones and location-based services, the amount of location-based data has grown tremendously in recent years. Processing location-data in timely fashion has become a big challenge. Parallelism is one way to deal with this problem. This can be achieved in the form of either *thread-level parallelism*, *instruction-level parallelism*, or *data-level parallelism*. In *thread-level parallelism*, multiple hardware threads work together in parallel to fully leverage the multi-core capabilities of modern CPU chips. In contrast, in *instruction-level parallelism*, a single core in a CPU chip executes multiple instructions possibly out-of-order in a single clock cycle. In *data-level parallelism*, a single core in a CPU chip applies a single instruction on multiple data units, i.e., integers, floats, doubles in a single clock cycle through Single Instruction Multiple Data (SIMD, for short) instructions.

Thread-level and instruction-level parallelism have been investigated extensively in spatial databases literature in the form of standalone query operators, query execution pipeline, and compilation of query plans [29]. However, this is not the case for data-level parallelism. Previous works [22, 32] in relational database management systems (RDBMS) on query operators, e.g., scan [17, 30], join [2, 4, 13], sorting [10, 23, 28], and query execution pipeline [11, 20, 25] suggest that database engines benefit from SIMD, mainly through *raw processing power* by working on multiple elements at once, *reduced instruction count* by imposing minimum pressure on the processor's decode and execution unit, and *conditional branch elimination* by relieving the processor from bad speculation and mispredicting branches. Hence, there exist parallel execution opportunities to utilize SIMD for improving query performance in spatial databases, which is the focus of this paper.

This is more so the case in modern CPUs that have evolved to equip each CPU core with its own SIMD execution unit be it in the same or different chip. These SIMD execution units are increasingly being equipped with wider SIMD registers (e.g., 512 bits), with more complex instruction sets, e.g., AVX512F, AVX512BW, AVX512CD, AVX512PF.[1] To benefit from SIMD capabilities and leverage per core data parallelism for processing queries in spatial databases, spatial data has to be laid out in a SIMD-friendly manner be it in main-memory or in disk to facilitate best use of SIMD instructions, and novel implementations of query processing algorithms. In this paper, we focus on main-memory two-dimensional R-Tree [8], and investigate how range select, and spatial join can benefit from SIMD vectorization. Furthermore, we study the effect of various storage layouts of R-Tree index nodes on the performance of these spatial operators.

With the advent of large-capacity main-memory chips, indexes can fit fully in main memory, disk I/O is no longer the bottleneck for the index operators. Rather, the bottleneck has shifted to the computational efficiency of the CPU, e.g., the number of instructions executed per clock cycle (IPC, for short), main-memory stalls, dominated by Last Level Cache misses (LLC misses, for short), Translation Lookaside Buffer misses (TLB misses, for short), and branch mispredictions. An LLC miss refers to the event of the processor attempting to access data that is not present in the last level cache and requires fetching data from memory. This miss incurs a

SIGSPATIAL '23, November 13–16, 2023, Hamburg, Germany

ACM ISBN 979-8-4007-0168-9/23/11.
https://doi.org/10.1145/3589132.3625610

[1] www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX_512

penalty of 89 ns for Intel Ice Lake processors[2]. TLBs are small caches that store the virtual-to-physical address mapping to speedup the translation of the virtual addresses requested by the processor to either access data or instructions. A single TLB miss can incur a miss penalty ranging from 7 to 30 clock cycles for Intel Ice Lake processors[2]. Analogously, processors incur significant penalty for a single branch misprediction, e.g., 17 clock cycles for Intel Ice Lake processors[2]. These misses not only impact query performance but also hinders the CPU from fully utilizing SIMD capabilities to the point that it can perform worse than its scalar counterpart. Thus, hardware-conscious data layout is a must to fully utilize SIMD capabilities of modern CPU architectures. We propose three different data layouts for the 2D R-Tree, and investigate their impact on the various hardware performance counters, e.g., LLC and TLB misses, branch mispredictions, and the number of instructions executed. We implement range select and spatial join over a main-memory R-tree, and present techniques to vectorize them using SIMD instructions with several optimizations. We study the performance trade-offs of these data layouts when performing these index operations based on the performance counters above.

We redesign the spatial select operator to better facilitate SIMD vectorization, and reduce LLC misses. Performing a select on an index typically results in cold LLC misses equal to the number of nodes accessed. Hardware prefetchers cannot hide these cold LLC misses, and it badly hurts index performance and the CPU's SIMD capabilities. Given that the R-Tree index nodes overlap each other, we introduce a queue to keep track of the nodes that need to be accessed as we perform a breadth-first search (BFS) over the index and perform software prefetching to bring the index node that will be accessed in cache in a timely manner. To reduce the overhead of the additional queue, we use SIMD instructions to insert multiple items into the queue with a single instruction. Also, we vectorize spatial join operations over a SIMD-based (or SIMDified) R-tree. We focus on the spatial nested-index join operator [5], where we start with the root of 2 R-Tree indexes, and traverse both trees simultaneously in top-down fashion until we reach the leaf nodes. If the MBRs of both indexes are unsorted, then this is the same as applying the range select operator, where the outer index node is the query rectangle. However, if the MBRs of the index nodes are sorted on one of the dimensions, then the performance of the nested index join can be improved through several optimizations. We introduce two optimizations for the nested index spatial join operation, where the index node MBRs are sorted on a pre-determined dimension.

We compare the vectorized implementation of these spatial operators against their scalar counterpart. The experiments show a speedup of up to 4×, and 9×, for select, and spatial join, respectively. We study the performance of the proposed optimizations, and investigate their best use scenarios.

The contributions of the paper can be summarized as follows.

- We present vectorized algorithms for 2D range select and indexed spatial join over a SIMD-ified R-tree.
- We investigate 3 data layouts for the R-Tree, and study the tradeoffs of these layouts for the index query operators.
- We compare our vectorized query operators against their scalar counterparts, and achieve a speedup of upto 9×.
- We introduce *5* optimizations for the vectorized query operators, and study their effectiveness under various conditions.

The rest of the paper proceeds as follows. Section 2 overviews the SIMD and prefetch instructions used in the paper, and introduces the proposed layouts for the R-tree nodes. Sections 3 and 4 present the vectorized spatial range select, and spatial join, respectively. Section 5 presents the experimental study. Section 6 discusses the related work, and Section 7 concludes the paper.

2 PRELIMINARIES

In this section, we overview SIMD and prefetching operations, and the various data layouts being investigated in this paper.

2.1 SIMD Instructions

CPU vendors provide SIMD capabilities via different instruction sets starting from SSE (operates on 4 32-bit data elements), AVX2 (8 32-bit data elements) to the recent AVX512 that operates simultaneously on 16 32-bit data elements. Coupled with the special-purpose CPU SIMD registers, theoretically these instructions, e.g., AVX512, can provide upto $W = 16\times$ speedup over the traditional scalar instructions. Below, we overview the SIMD instructions (AVX512) we use to implement the vectorized spatial query operators. Let $a - k$ be 32-bit data elements. Even though an AVX512 SIMD register can hold at most 16 32-bit data elements, for ease of illustration, we restrict the registers, i.e., source, target, and index vector to contain only 4, enclosed in []. Let || denote data elements located in memory along with a ↓ to point to a certain memory location.

1. Load Instructions:

1.1. Load: Vector load takes as input a memory location, say m, and loads contiguous elements starting at m into a target register, e.g.,

$$\underbrace{[f, g, h, i]}_{\text{Target vector}} \leftarrow \text{load}(\underbrace{|a, b, c, d, e, \overset{\downarrow}{\mathbf{f}}, \mathbf{g}, \mathbf{h}, \mathbf{i}, j, k|}_{\text{Memory}})$$

1.2. Gather: Takes as input an array, say A, with an index vector, $\overrightarrow{idx}$, and stores the A elements specified by the $\overrightarrow{idx}$ in order (as specified in the index vector) in a target register, e.g.,

$$\underbrace{[a, j, c, e]}_{\text{Target vector}} \leftarrow \text{gather}(\underbrace{[0, 9, 2, 4]}_{\text{Index Vector}}, \underbrace{|\mathbf{a}, b, \mathbf{c}, d, \mathbf{e}, f, g, h, i, \mathbf{j}, k|}_{\text{In-Memory Array}})$$

1.3. Expand Load: Takes as input a memory location, say m, along with a write mask k, and stores contiguous elements starting at m into a target register using Mask k, e.g.,

$$\underbrace{[\times, f, g, \times]}_{\text{Target vector}} \leftarrow \text{load}(\underbrace{\quad 0110 \quad}_{\text{Write Mask}}, \underbrace{|a, b, c, d, e, \overset{\downarrow}{\mathbf{f}}, \mathbf{g}, \mathbf{h}, \mathbf{i}, j, k|}_{\text{Memory}})$$

1.4. Broadcast: Takes as input a data element, say e, or a vector, say $\vec{v}$, and replicates e or part of the $\vec{v}$ across all lanes of a target register. There are many variants of Broadcast depending on what needs to be replicated, e.g., to duplicate $\vec{v}$'s two lower elements into all lanes of a target register, we perform:

$$\underbrace{[a, b, a, b]}_{\text{Target vector}} \leftarrow \text{broadcast}(\underbrace{[\mathbf{a}, \mathbf{b}, c, d]}_{\text{Source vector}})$$

[2] www.7-cpu.com/cpu/Ice_Lake.html

2. Store Instructions:
2.1. Store: Takes as input a vector, say $\vec{v}$, and a memory location, say m, and stores $\vec{v}$'s data elements in memory starting at m, e.g.,

$$\underbrace{|\times, \times, \times, \times, \overset{\downarrow}{\mathbf{f}}, \mathbf{g}, \mathbf{h}, \mathbf{i}, \times, \times|}_{\text{Memory}} \leftarrow \text{store}(\underbrace{[f, g, h, i]}_{\text{Source vector}})$$

2.2. Compress Store: Takes as input a vector, say $\vec{v}$, a write mask, say k, and a memory location, say m, and stores $\vec{v}$'s elements (indicated by k) into contiguous memory locations starting at m, e.g.,

$$\underbrace{|\times, \times, \times, \times, \overset{\downarrow}{\mathbf{g}}, \mathbf{h}, \times, \times, \times, \times|}_{\text{Memory}} \leftarrow \text{compress}(\underbrace{0110}_{\text{Write Mask}} , \underbrace{[f, g, h, i]}_{\text{Source vector}})$$

3. Permute: Takes as input a vector, say $\vec{v}$, an index vector, say $\overrightarrow{idx}$, and shuffles $\vec{v}$'s elements using $\overrightarrow{idx}$ into a target register, e.g.,

$$\underbrace{[d, c, c, b]}_{\text{Target vector}} \leftarrow \text{permute}(\underbrace{[3, 2, 2, 1]}_{\text{Index vector}} , \underbrace{[a, b, c, d]}_{\text{Source vector}})$$

4. Blend: Takes as input 2 vectors, say $\vec{v_1}, \vec{v_2}$, and combines $\vec{v_1}$'s and $\vec{v_2}$'s elements into a target register using an input mask k, e.g.,

$$\underbrace{[e, b, g, d]}_{\text{Target vector}} \leftarrow \text{blend}(\underbrace{1010}_{\text{Mask}} , \underbrace{[a, b, c, d]}_{\text{Source vector 1}} , \underbrace{[e, f, g, h]}_{\text{Source vector 2}})$$

5. Arithmetic Instructions:
5.1. Compare: Takes as input 2 vectors, say $\vec{v_1}, \vec{v_2}$, and compares $\vec{v_1}$ and $\vec{v_2}$ using a comparator, say op, store the result as a bitmask.

$$\underbrace{0111}_{\text{Target Mask}} \leftarrow \text{compare}(\geq, \underbrace{[a, b, c, d]}_{\text{Source vector 1}} , \underbrace{[d, a, b, c]}_{\text{Source vector 2}})$$

5.2. Masked Addition: Adds 2 vector registers, say $\vec{v_1}$ and $\vec{v_2}$, and stores the result in a target register using a write mask, k, e.g.,

$$\underbrace{[ae, bf, c, d]}_{\text{Target vector}} \leftarrow \text{masked add}(\underbrace{1100}_{\text{Write mask}} , \underbrace{[a, b, c, d]}_{\text{Source vector 1}} , \underbrace{[e, f, g, h]}_{\text{Source vector 2}})$$

2.2 Software Prefetch Instructions

CPUs, e.g., Intel's SSE extension of x86-64 Instruction Set Architecture provides prefetch intrinsics, e.g., `_mm_prefetch (char const* p, int hint)` that allow programmers or compilers to specify a virtual address that requires prefetching into cache, e.g., L1, L2 or L3, as specified by *hint*. The CPU can load a cacheline worth of data containing the specified addressed byte or, if busy, ignore it altogether. Seemingly very effective in hiding cache miss latency, software prefetch intrinsics can introduce computational overhead on the processor's computation unit, and stress the cache and memory bandwidth resulting in performance degradation when the processor is busy or when the data is already in cache.

2.3 Index Node Storage Layouts

An in-memory R-Tree index node contains upto a maximum fanout F of entries. Each entry is of the form $(key, ptr) \equiv (MBR, ptr)$. The key of each entry is the

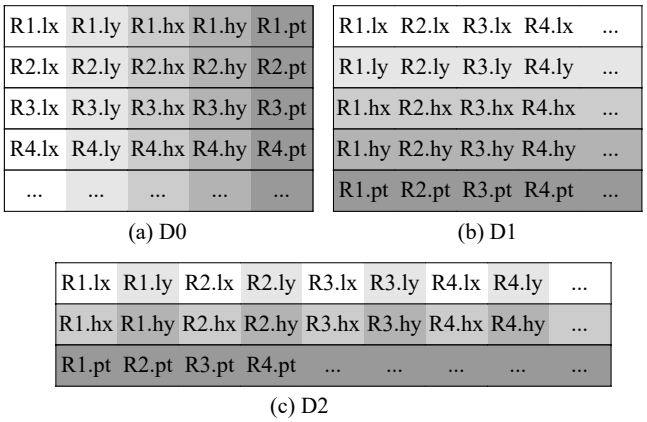


Figure 1: Different storage layouts of index nodes.

$MBR = (MBR.low_x, MBR.low_y, MBR.high_x, MBR.high_y)$ assuming 2D space. In addition, each index node contains the depth and the number of child MBRs or entries associated with the node. Based on the packing strategies of these entries we identify three index node layouts as follows. Refer to Figure 1 for illustration.

(1) **Node Layout D0:** The fields associated with each MBR entry, (MBR, ptr) are stored contiguously as proposed in the original R-Tree [8], and its variants, e.g., CR Tree [14], MR Tree [15] (cf. Figure 1a). One disadvantage of this index layout is its inability to efficiently facilitate SIMD instructions.
(2) **Node Layout D1:** We pack the $MBR.low_x$ of all the child MBRs in an array, followed by separate arrays for the $MBR.low_y, MBR.high_x, MBR.high_y$ of all the child MBRs and the addresses of all the child nodes $MBR.ptr$ (see Figure 1b). Packing the child MBR keys and the child MBR addresses allows applying SIMD instructions efficiently on the index nodes for the various query operators.
(3) **Node Layout D2:** The leftmost point of each MBR, $MBR.low : (MBR.low_x, MBR.low_y)$ is stored in an array followed by separate arrays for the rightmost point $MBR.high : (MBR.high_x, MBR.high_y)$ of all child MBRs and the addresses of all child nodes $MBR.ptr$ (see Figure 1c) .

3 SPATIAL SELECT

The scalar version of the spatial select algorithm follows a recursive approach, where the index is traversed depth-first starting from the root, and then following the child nodes that qualify the query predicate using the logical operators. The query predicate evaluation of the index nodes involve executing a compound selection condition with 4 separate selects, i.e., comparing the query predicate's high x, high y, low x and low y with the index node's low x, low y, high x and high y, respectively. When these select conditions are implemented using a logical operator, the assembly code generated by the compiler replaces the 4 select conditions with 4 conditional branches. If the selectivities of these select conditions are close to 0.5, it makes the processor's branch predictor unit's job of predicting accurate branches a lot harder. This may result in as many as 4 branch mispredictions impacting the query performance. In contrast, when the 4 select conditions are implemented using a

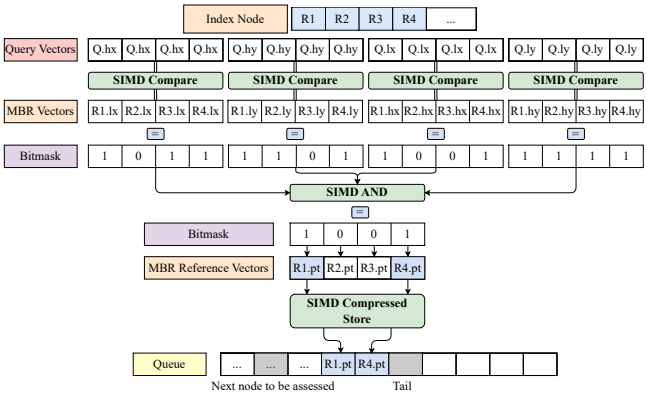


Figure 2: Spatial Select for D1.

bitwise operator, the compiler replaces them with a single conditional branch and evaluates all 4 of the select conditions [26]. Even though this requires executing a greater number of instructions, the branch misprediction penalty associated with this approach is expected to be smaller due to the possibility of only one branch misprediction. We implement both these variants for the spatial select to compare their performance.

In contrast, the vectorized version of the spatial select operator preforms a breadth-first traversal of the R-Tree, and maintains a queue Q to track the addresses of the internal nodes that need to be visited. The algorithm starts by visiting the root, and inserts into Q the child nodes that qualify the query predicate. Then, each qualified index node is dequeued, and is evaluated until the queue becomes empty. Refer to Figures 2 and 3 for the illustration of the algorithms for node layouts D1 and D2, respectively.

1. Query vector ($\overrightarrow{q}$) construction. Given a 2D query rectangle, i.e., key, the key parts, e.g., $q.low_x$ (for Node Layout D1) or $q.low$ (for Node Layout D2) are broadcast to construct the query vectors. The layout of the query vector needs to exactly match the layout of the index node that it operates on. This is necessary so that when an index node is loaded from memory into a SIMD register for select predicate evaluation, a SIMD comparison can be performed efficiently. This step is performed exactly once at the start of query execution. For Node Layout D1, it takes 4 `broadcast` instructions to load the 4 key parts, i.e., $q.low_x, q.low_y, q.high_x$ and $q.high_y$ in 4 separate vector registers, while for node layout D2 it takes 2 instructions to load the corresponding 2 key excerpts, i.e., $q.low$ and $q.high$, accompanied by 2 extra `masked load` instructions for the query vector to match the Node Layout D2.

```
// Query Rectangle
float q[4] = {q.low_x, q.low_y, q.high_x, q.high_y};
// Node Layout-D1: Broadcast q.low_x
__m512 qv_low_x = _mm512_set1_ps (q.low_x);
// Node Layout-D2: Extract q.low_x, q.low_y; then braodcast
__m128 t = _mm_mask_load_ps(t, 0x03, q);
__m512 qv_low = _mm512_broadcast_f32x2(t);
```

2. Child MBR vector ($\overrightarrow{mbr}$) construction. We apply the vector `load` instructions to load contiguously stored key excerpts of all the child node MBRs. For Node Layout D1, this requires executing 4 separate `load` instructions $\lceil \frac{n_c}{W} \rceil$ times to load the $MBR.low_x, MBR.low_y, MBR.high_x$ and $MBR.high_y$s' of the child MBRs, respectively. For the Node Layout D2, this requires executing 2 separate `load` instructions $\lceil \frac{2n_c}{W} \rceil$ times to load the contiguously stored $MBR.low$ and $MBR.high$s' of the child node MBRs. Here, n_c refers to the number of children of the index node.

```
// Node Layout-D1: Loads 1st 16 MBR.low_x of n;f=max-fanout
struct tnode1 n: lx[f],ly[f],hx[f],hy[f],ptr[f];
__m512 mbr_lx = _mm512_load_ps (n.lx);
// Node Layout-D2: Loads 1st 8 MBR.low of n;f=max-fanout
struct tnode2 n: lo[f*2],hi[f*2],ptr[f];
__m512 mbr_lo = _mm512_load_ps (n.lo);
```

3. Predicate evaluation. SIMD `comparison` instructions are executed on the constructed query and child MBR vectors to evaluate the predicates and generate a bitmask of the qualifying child nodes for further evaluation.

4. Queue insertion. We use a `masked compress store` instruction to store addresses of the qualified child nodes into Q. This leverages full SIMD capabilities by inserting into Q up to 8 addresses with a single instruction to make spatial select fully vectorized. This also improves cache locality as it requires the addresses of the child nodes to be loaded into SIMD registers only once, when they are in cache, ensuring full utilization of child addresses.

5. Prefetching. Unlike a B+ Tree, the overlap of the MBRs in the index nodes of an RTree may require to descend multiple index nodes at the same level when evaluating a select predicate. This feature along with the use of a queue exposes the need for prefetching to speedup spatial selects. To increase the likelihood that prefetching is beneficial, we maintain a parameter `pf_distance` to prefetch the index node that is `pf_distance` steps ahead in the queue. This scheme is effective when there are multiple nodes to evaluate at the same level as we traverse down the tree using a breadth-first traversal. This situation arises when the ratio of the nodes overlapping is relatively large in the R-tree and/or the queries are less selective. In such cases, the cold misses of the index nodes can be fully hidden irrespective of its being an internal or a leaf node. Based on the selectivity of the select operator, the number of instructions executed by spatial select is fairly small making it memory-bound, i.e., query execution time is spent on mostly the CPU's stalling on the LLC cache misses. Using the proposed prefetching scheme, these cache misses are reduced, and thus, improving query performance.

Avoiding recursion for SIMD-friendly tree traversal (O1). Avoid recursion to give a tree traversal algorithm the best chance to benefit from SIMD vectorization by introducing external data structures, e.g., queue to mimic recursion call stack, and use SIMD masked compress store instruction to store addresses of multiple qualified nodes or objects into the queue to better utilize cache locality and memory bandwidth.

Prefetching in tree-indexes that require traversing multiple nodes at the same level (O2). Use an external data structure, e.g., queue to facilitate looking up the addresses of the next-to-be-visited index nodes, and use software prefetch intrinsics to bring these nodes in cache ahead of time to hide the expensive cold cache miss latency.

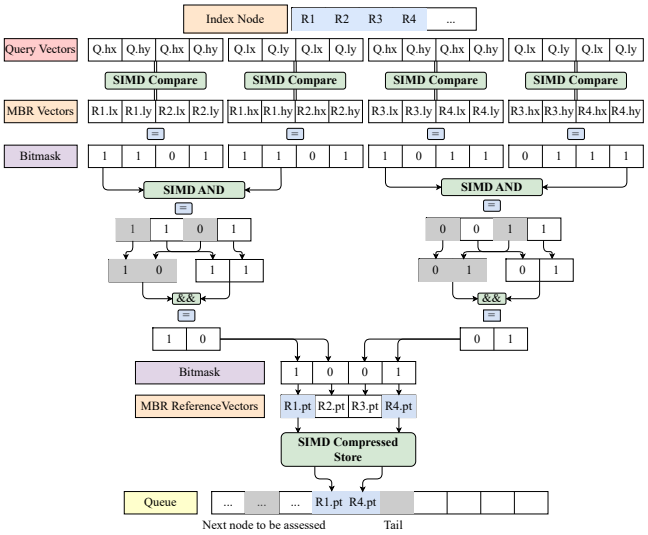


Figure 3: Spatial Select for D2.

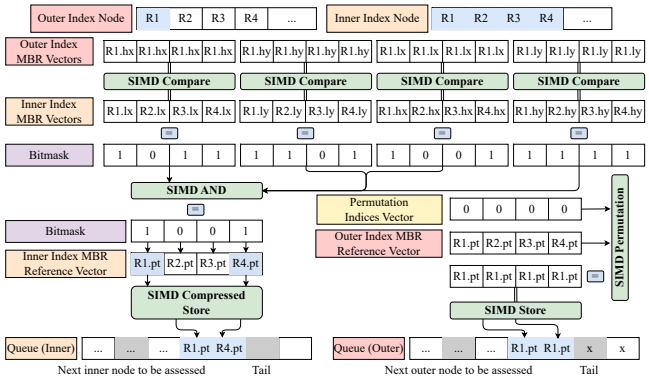


Figure 4: Spatial Join - *One* to *Many* Comparison for D1.

4 SPATIAL JOIN

A spatial join combines 2 spatial relations based on some spatial predicate, e.g., *intersects*. Many variants of spatial join algorithms exist. In this paper, we consider the *R-Tree Join* [5], where both input relations have R-tree indexes. The scalar version of this algorithm [5] traverses the two indexes simultaneously starting from both root nodes, and follows the child node pairs that intersect. For the vectorized implementation of the spatial join algorithm, we propose 2 approaches as discussed next.

4.1 Approach 1: One to Many Comparison

The vectorized implementation of this approach of our spatial join algorithm follows the same flow as the vectorized spatial select. The only difference is that we generate the outer index MBR vectors in place of the query MBR vectors. The key idea is to duplicate each MBR of an outer index child node across all the SIMD lanes and compare it with all the inner index child node MBRs, hence the term *one to many comparison*. Refer to Figures 4 and 5 that illustrates this approach for Node Layouts D1 and D2, respectively.

1. Inner index MBR vector ($\overrightarrow{mbr_{in}}$) construction. The MBR key excerpts of all the child node MBRs of the inner index node, i.e., $MBR.low_x$s' $MBR.low_y$s', $MBR.high_x$s' and $MBR.high_y$s' for Node Layout D1 or $MBR.low$s' and $MBR.high$s' for Node Layout D2 are loaded from memory using the vector `load` instruction.

2. Outer index MBR vector ($\overrightarrow{mbr_{out}}$) construction. Each child MBR of the outer index node is considered one at a time, and during each iteration the MBR key excerpt of the considered child MBR is `broadcast` to construct the outer index MBR vectors. For Node Layout D1, we construct separate outer index MBR vectors with duplicate $MBR.low_x$, $MBR.low_y$, $MBR.high_x$ and $MBR.high_y$ values in all the SIMD lanes. For Node Layout D2, we construct separate MBR vectors with duplicate $MBR.low$ and $MBR.high$ values.

3. Predicate evaluation. SIMD `comparison` instructions are applied to the constructed inner and outer index MBR vectors to produce a bitmask of the qualified inner index child nodes that require further processing along with the corresponding outer child node as a pair. Predicates are evaluated in 4 stages with 4 sets of comparisons, $[\overrightarrow{mbr}_{out,lx} \geq \overrightarrow{mbr}_{in,hx}]$, $[\overrightarrow{mbr}_{out,hx} \leq \overrightarrow{mbr}_{in,lx}]$, $[\overrightarrow{mbr}_{out,ly} \geq \overrightarrow{mbr}_{in,hy}]$ and $[\overrightarrow{mbr}_{out,hy} \leq \overrightarrow{mbr}^{ly}_{in}]$. Data Layout D2 takes 2 stages, $[\overrightarrow{mbr}_{out,low} \geq \overrightarrow{mbr}_{in,high}]$ and $[\overrightarrow{mbr}_{out,high} \leq \overrightarrow{mbr}_{in,low}]$. $\overrightarrow{mbr}_{out/in,k}$ is the MBR vector of the outer or inner index for key excerpt k based on the data layout.

Several optimizations apply for early pruning assuming that the nodes are sorted on an MBR key excerpt, e.g., $MBR.low_x$, $MBR.low_y$, $MBR.high_x$ or $MBR.high_y$. These strategies result in savings in terms of (i) the number of outer index child node MBRs considered, and (ii) the number of inner child node MBRs considered for both node layouts.

Let the index nodes be sorted on $MBR.low_x$ and the outer index child node MBRs are considered one by one in ascending order of the $MBR.low_x$. If predicate evaluation of $[\overrightarrow{mbr}_{out,lx} \geq \overrightarrow{mbr}_{in,hx}]$ produces a bitmask of all zeros (= $0x0000$), then all the child nodes of the outer index after the current one can be pruned because $mbr^{current}_{out,lx} \leq mbr^{next}_{out,lx}$. This reduces the number of instructions executed as it reduces the effective size of the outer index node.

Similarly, assuming that the inner index child MBRs are loaded in SIMD registers in ascending order of the $MBR.low_x$, early pruning can reduce the number of instructions executed. When $[\overrightarrow{mbr}_{out,hx} \leq \overrightarrow{mbr}_{in,lx}]$ generates a bitmask anything other than all ones ($\neq 0xFFFF$), the predicate evaluation of the next child MBR vectors of inner index can be skipped since $mbr^{current}_{in,lx} \leq mbr^{next}_{in,lx}$.

4. Queue Insertion. We use `masked compress store` instructions to store the addresses of the qualified inner node child MBR nodes contiguously into the queue of the inner index. For the considered child node of the outer index, we use `permute` instruction to duplicate the address of the outer index child across all the SIMD lanes and issue $\lceil \frac{\hat{n}_{in,c}}{W} \rceil$ vector `store` instructions to store the addresses in the respective queue of the outer index, where $\hat{n}_{in,c}$ states the number of qualified child nodes of the inner index.

It needs mentioning that the optimization strategies proposed in the predicate evaluation step for the vectorized implementation of the join algorithm can be applied to the scalar version as well.

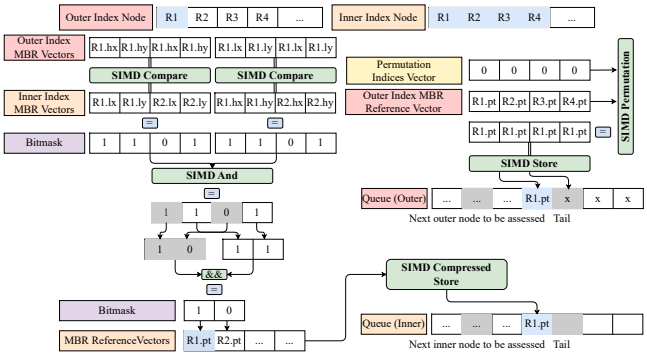


Figure 5: Spatial Join - *One* to *Many* Comparison for D2.

Slicing off parts of an outer index node (O3). Assume that the inner and outer index nodes are sorted on one of the MBR key excerpts, e.g., $MBR.low_x$. If for any child MBR of the outer index node, the join predicate on the sorted outer index key involving itself and all the child MBRs of the inner index node evaluates to no qualifying node-pairs, then the rest of the child MBRs can be skipped, i.e., part of the outer index node can simply be skipped.

Shrinking the MBR of an inner index node (O4). Assume that the index nodes are sorted on one of the MBR key excerpts, e.g., $MBR.low_x$. Given a *single* child MBR of the outer index node, if for any child MBR of the inner index node, the join predicate on the sorted inner index key involving both the child MBRs evaluate to disqualification, i.e., they do not intersect; then the rest of the child MBRs of the inner index node can be skipped to check for qualification against the given outer index child MBR, effectively reducing the size of the inner index node.

4.2 Approach 2: Many to Many Comparison

This approach of our vectorized implementation of the spatial join algorithm is specific to Node Layout D1. The only difference from the *One to Many* approach is in the predicate evaluation step, mainly, how the bitmask generation of $[\overrightarrow{mbr}_{out,hx} \leq \overrightarrow{mbr}_{in,lx}]$ is handled. The *one to many* approach considers each child node MBR of the outer index one at a time, and `broadcasts` it across all the lanes of a SIMD register to construct the outer index MBR vector. This requires executing $n_{out,c}$ `broadcast` and $n_{out,c}\lceil \frac{n_{in,c}}{W} \rceil$ SIMD `compare` instructions for evaluating $[\overrightarrow{mbr}_{out,hx} \leq \overrightarrow{mbr}_{in,lx}]$, where $n_{out/in,c}$ refers to the number of child MBR of the outer or inner index node. To reduce this large number of executions, we propose the following approach, where multiple (*many*) outer index child nodes are compared against a selected set (*many*) of inner index child nodes at once. Refer to Figure 6 for illustration.

1. Outer index $\mathbf{MBR_{hx}}$ vector ($\overrightarrow{mbr}_{out,hx}$) construction. The $MBR.high_x$ of all the child MBRs of the outer index are loaded into SIMD registers with the vector `load` instructions, 16 at a time. It takes $\lceil \frac{n_{out,c}}{W} \rceil$ vector `load` instructions to fully load all the $MBR.high_x$s' of outer index node's child MBRs. For each of these child MBR vectors, Steps 2-4 are carried in $\log_2 F + 1$ times. Here, F is the maximum fanout of the index.

2. Gather indices vector construction for inner index. We use `gather` instruction to load the $MBR.low_x$ of the desired child MBRs of the inner index. However, this requires constructing a `gather indices` vector specifying the indices of the $MBR.low_x$s' of the desired child MBRs. Initially, $n_{in,c}/2$ is duplicated across all SIMD lanes to generate the `gather indices` vector as we want to load the $MBR.low_x$ of the middle child MBR for the inner index, where $n_{in,c}$ refers to the number of child MBRs of the inner index node. For the following iterations, the `gather indices` vector is updated based on the bitmask generated from Step 3. This requires performing 2 SIMD masked `addition`.

3. Predicate evaluation. Once both the outer and inner index MBR vectors are constructed, we evaluate the $[\overrightarrow{mbr}_{out,hx} \geq \overrightarrow{mbr}_{in,lx}]$ predicate to generate a bitmask during each iteration.

4. Flip indices vector construction. A `flip indices` vector is required to track the eligible inner child MBRs for all the outer index child MBRs. For each child MBR in the outer index, the corresponding entry in the `flip indices` vector indicates the index of the inner child MBR, beyond which the other child MBRs can be ignored, as they do not qualify. Initially, F is duplicated across all the SIMD lanes to generate the `flip indices` vector, denoting all the inner index child MBRs qualify the predicate. For the following iterations, a `masked blend` instruction is performed to update the indices vector. After the completion of $\log_2 F + 1$ iterations for each outer index child MBR vector, the `flip indices` vector is stored in memory using the vector `store` instruction.

5. Extracting bitmask. Contrary to *Approach* 1, the predicate evaluation step of $[\overrightarrow{mbr}_{out,hx} \geq \overrightarrow{mbr}_{in,lx}]$ for *Approach* 2 produces bitmasks referring to the eligibility of inner index child MBRs for different sets of outer index child MBRs. Thus, it requires extracting the corresponding entry of an outer index child MBR from the `flip indices` vector and generating the bitmask. Once the predicate evaluation stage of $[\overrightarrow{mbr}_{out,hx} \leq \overrightarrow{mbr}_{in,lx}]$ is completed, the rest of the algorithm remains the same as in the *one to many* approach for evaluating the remaining 3 predicates.

Compared to the *one to many* approach that takes $n_{out,c}\lceil \frac{n_{in,c}}{W} \rceil$ SIMD comparison instructions for evaluating the $[\overrightarrow{mbr}_{out,hx} \leq \overrightarrow{mbr}_{in,lx}]$ predicate, the *many to many* approach takes at most $\lceil \frac{n_{out,c}}{W} \rceil (log_2 F + 1)$ comparison instructions for the same task. However, this comes with the additional cost of other SIMD instructions in the form of blend, masked add and gather operations. This further reduces the number of broadcast instructions in the sense that any outer index child node that does not qualify, i.e., the flip indices of the node remain undefined (Refer to the first lane of the final flip indices vector in Figure 6) can be ignored for the next stage of processing. Notice that the optimization technique *O4* proposed for *one to many* approach contrasts the optimization technique discussed above for the *many to many* approach. However, optimization *O3* is also applicable to this approach.

***Batched* shrinkage of an inner index node's MBR (O5)** To reduce the large number of SIMD broadcast and comparison instructions in O4, use gather instructions to selectively load inner index node's child MBRs and compare them against a batch (W) of outer index child MBRs instead of one.

Refer to Figure 6. Each node has exactly $n_c = 4$ child MBRs. The flip indices vector is set to all-undefined values as initially we consider all the child nodes to qualify and the outer index MBR vector contains all the MBRs from R1 to R4. The gather indices vector is set to all-2s ($n_{c,in}/2 = 2$) that generates the inner index MBR vector with all R3s' for the first iteration. Then, the outer index MBR is compared with the inner index MBR vector to generate the bitmask ($0b1001$) that is fed to the blend instruction pipeline along with the gather indices vector to generate the flip vector for the next iteration, i.e., $[\times, \times, \times, \times] \xrightarrow[0b1001]{[2,2,2,2]} [\times, 2, 2, \times]$. Similarly, the same bitmask is used to generate the gather indices for the next iteration, $[2, 2, 2, 2] \xrightarrow{0b1001} [3, 1, 1, 3]$. The same steps are repeated for the next iterations as illustrated in the Figure.

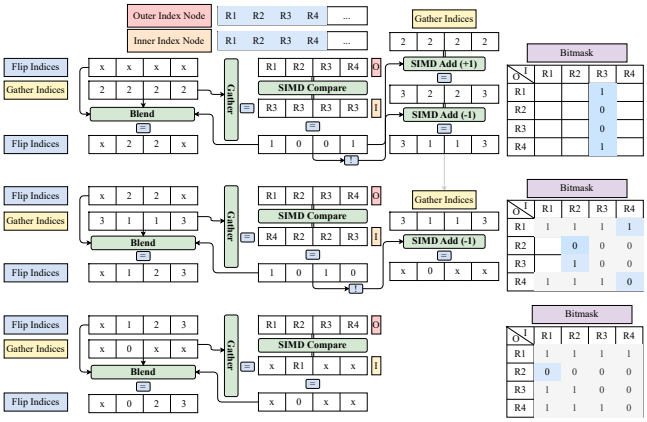


Figure 6: Spatial Join - *Many* to *Many* Comparison for D1. Each row is a different iteration.

5 EXPERIMENTAL EVALUATION

Experiments run on a server with Intel(R) Xeon(R) Gold 6330 CPU processors based on Intel IceLake microarchitecture and Linux OS. The L1-D, L1-I, L2, and LLC cache sizes are 2.6MB, 1.8MB, 70MB and 84MB, respectively. The DTLB cache contains 64 4KB pages. The machine supports 56 cores and 512-bit SIMD registers. The CPU clock frequency is 2.0 GHz. We compile with gcc 11.3.0 with flags `-funroll-loops`, and `-O3` enabled. We use Linux's perf events API to collect the hardware performance counters. All query operations are evaluated on an R-tree index with 10M 2D points synthetically generated that follow uniform distribution. We use 32-bit keys to present each dimension of the data points. The default maximum fanout of the index is 64 and the default selectivity of range select is 0.1%.

5.1 Spatial Select

5.1.1 Effect of SIMD. Figure 7 gives the query processing time and hardware performance counters: the number of retired (i.e., executed) instructions, L1-D cache misses, LLC misses, and branch mispredictions of the R-tree's scalar and vectorized range select operator with maximum fanout 64 and dataset size 10M. We examine the logical and bitwise scalar variants of range select. For the vectorized implementation, we examine 3 variants for each

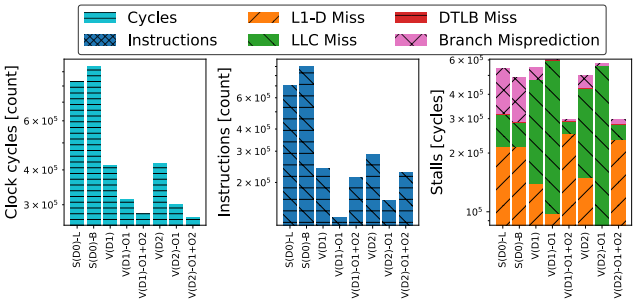


Figure 7: Spatial select: Effect of SIMD and optimizations. (Dataset size = 10M, Maximum fanout = 64, Selectivity = 0.1%)

data layout, D1, D2. Variants V(D1), V(D2) traverse the index recursively, Variants V(D1)-O1, V(D2)-O1 traverse the index via a queue, and Variants V(D1)-O1+O2, V(D2)-O1+O2 issue prefetch instructions with the queue. Data layout D2 with both optimizations O1 and O2 performs best, achieving a speedup of 2.97× over the best-performing scalar variant. It reduces the number of instructions 3.12×, LLC misses 2.18×, and branch mispredictions 18.30×. These factors boost the performance of the vectorized implementation. The worst performing vectorized variant, layout D2 with no optimizations, 1.91× outperforms the best-performing scalar variant. One consistent aspect for all vectorized variants with no prefetching is that they experience more LLC misses than the scalar variants, e.g., Layout D2 with no prefetching, V(D2)-O1 incurs 6.70× more LLC misses than the scalar implementation with bitwise operators.

Between the 2 scalar variants, the one with logical operators performs better. Even though the introduction of bitwise operators reduce the number of branch mispredictions by 1.10×, it comes at the cost of evaluating all the conditions of the select predicate, i.e., resulting in more instructions (1.27×). Thus, the benefit of the reduced number of branch mispredictions cannot mitigate the overhead due to the increased number of retired instructions.

5.1.2 Effect of optimizations. O1 reduces query latency by 1.32× and 1.40× for Data Layout D1 and D2, respectively. O1 avoids recursion and uses one instruction to enqueue addresses of at most 8 index nodes, thus reducing the number of retired instructions by up to 2× for both data layouts. Also, it reduces branch mispredictions by 14.80× and 5.71× for layouts D1 and D2, respectively, than the partially vectorized variant (V). But the introduction of the queue worsens cache performance as it results in 1.52× and 1.73× more LLC misses for both data layouts, respectively. This is expected as it requires an extra lookup to dequeue the address of the next qualifying index node. To mitigate the effect of bad cache performance, when O2 is applied on top of O1, it further improves the query performance 11.14% and 10.46% by reducing the LLC misses by 13.51× and 10.20× for layout D1 and D2, respectively. This reduced number of LLC misses comes at the cost of increased number of retired instructions, i.e., 1.68× for layout D1 and 1.43× for layout D2 in the form of software prefetch instructions. This prefetching scheme not only improves over the vectorized variants that suffer from heavy LLC misses, it outperforms the scalar versions as well in terms of LLC misses. Compared to the scalar (logical) select operator, prefetching-enabled vectorized operator exhibit 2.80× and 2.18× less LLC misses for storage layout D1 and D2, respectively.

5.1.3 Effect of node layouts. Between the 2 node layouts, D2 outperforms D1 by only 3.62%, with both optimizations, O1 and O2 enabled. After optimizing for LLC cache misses, L1-D misses become the bottleneck for range selects for the in-memory R-tree (c.f. Figure 8a). This is why D2 slightly outperforms D1 as it shows better L1-D cache performance despite the larger number of retired instructions. D2 has 1.06× less L1-D misses than D1. If we exclude prefetching and focus on prefetching-disabled vectorized variants, i.e., the partially vectorized implementation, D2 has better LLC performance with 1.18× less cache misses.

5.1.4 Effect of maximum fanout. As the maximum fanout of the R-tree increases, the performance of range select improves until it reaches a plateau at fanout 64, and then the performance starts to degrade. This is true for all scalar or vector implementations. The number of retired instructions, L1-D cache misses, and DTLB misses follow the same trend with the exception of LLC misses and branch mispredictions (Figure 9). Excluding range select variants (V-O1+O2) with software prefetching, all other variants with higher maximum fanout have lower LLC misses. As node size increases, the number of nodes probed by a range select reduces, resulting in less number of cold cache misses. In addition, larger node sizes enable the hardware prefetcher to fully kick in as the data addresses to be prefetched are more predictable, and prefetches to cache more child MBRs located contiguously in memory, ahead of their use.

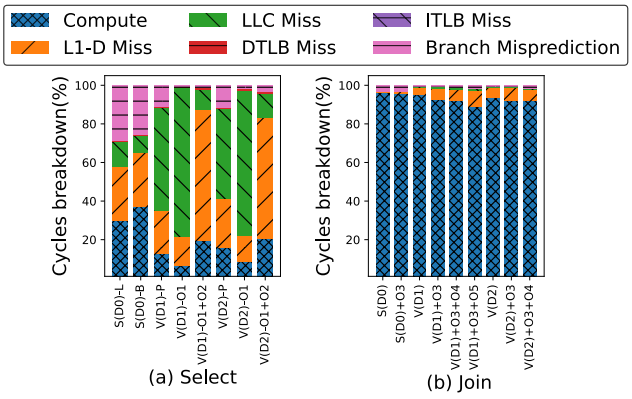


Figure 8: An approximation of the clock cycles breakdown (Dataset size = 10M, Max fanout = 64, Selectivity = 0.1%)

The fanout impacts the performance of the optimizations. For indexes with smaller fanout, as observed in Section 5.1.2, the incremental introduction of O1 and O2 improves query performance over the partially vectorized technique. However, when node size increases, the effect of both optimizations starts to diminish, e.g., from maximum fanout 1024 onwards, prefetching rather decreases query performance, and O1 outperforms O1+O2. From maximum fanout 512, the partial vectorized operator performs better. Even though prefetching still reduces the number of LLC misses for these indexes with larger fanouts, the increased number of retired instructions and L1-D cache misses hinder the benefits gained from it. Notice that prefetching increases the number of L1-D cache misses for indexes with larger fanout. The reason is that we use the hint `_MM_HINT_T0` when issuing prefetch requests to bring the node data that will be required in future into L1-D cache. With larger node sizes, this results in evicting active node data that are being worked on.

From fanout 512 onwards, the partially vectorized operator outperforms its vectorized counterparts with the optimizations by almost 1.20×. As node size increases, the probability that the addresses of the qualified entries get enqueued with the same instruction decreases, thus resembling a normal store instruction but with increased latency, hence degrading performance (Figure 10a).

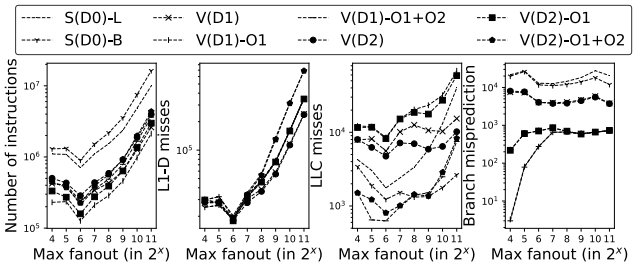


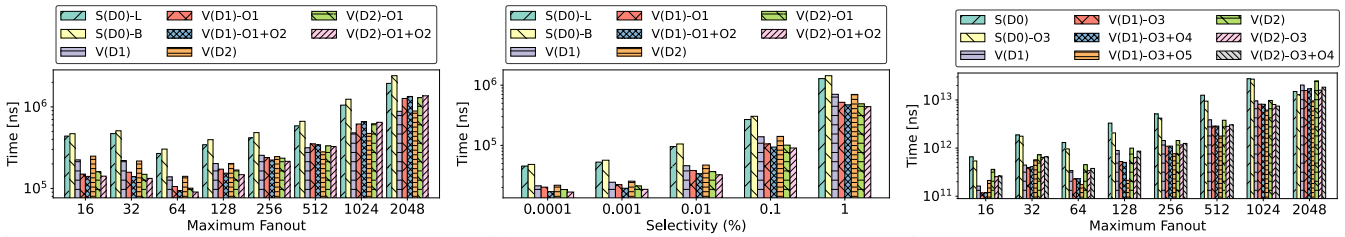
Figure 9: Spatial select: Effect of maximum fanout on h/w performance counters. (Dataset size = 10M, Selectivity = 0.1%)

5.1.5 Effect of selectivity. The observations made in Sections 5.1.1 to 5.1.4 remain valid for varying selectivity of the range select operation. Figure 10b compares the performance of different optimization techniques and data layouts under varying selectivity.

5.2 Spatial Join

5.2.1 Effect of SIMD. Figure 11 gives the performance of the scalar and vectorized implementations of R-tree spatial join in terms of query latency and hardware performance counters. The maximum index fanout is 64, and the data size is 10M points. We examine 2 variants of the scalar implementation, one with no optimizations (S-D0) and the other with O3, S-D0(O3) while having the index sorted on one on the MBR key excerpts. Similarly, we examine 7 variants of the vectorized implementation, 4 and 3 for Data Layouts D1 and D2, respectively. O4 and O5 are orthogonal to each other, hence only one can be applied with O3 for Data Layout D1. For Data Layout D2, it is not possible to apply O5. For O5 to take effect, the consecutive elements of an index node are to be sorted. Data Layout D2 packs both the $MBR.low_x$ and $MBR.high_x$ consecutively and the index node can be sorted on one of $MBR.high_x$ or $MBR.low_x$.

Figure 11 illustrates that all 6 SIMD variants of the join operator outperform the scalar variants. At worst, the vectorized implementation (V-D1) achieves 2.12× speedup over the best performing scalar variant. The speedup increases upto 5.53× for the best performing vectorized implementation, i.e., layout D1 with O5 and O3 (V-D1+O3+O5). A reduced number of executed instructions and branch mispredictions contribute to this speedup. Compared to S-D0(O3), variant V-D1(O3+O5) executes 7.55× less instructions and 14.50× less branch mispredictions. But the cache performance of these vectorized implementations is worse. The best case (V-D1+O3+O5) incurs 1.62× and 3.00× more L1-D, and LLC cache misses, respectively. Its DTLB cache performance is also 2× worse than S-D0(O3).



(a) Effect of maximum fanout in spatial select. (b) Effect of selectivity in spatial select. (c) Effect of maximum fanout in spatial join.

Figure 10: Comparing node layouts and optimizations for spatial select and join. (Default max fanout = 64, selectivity = 0.1%)

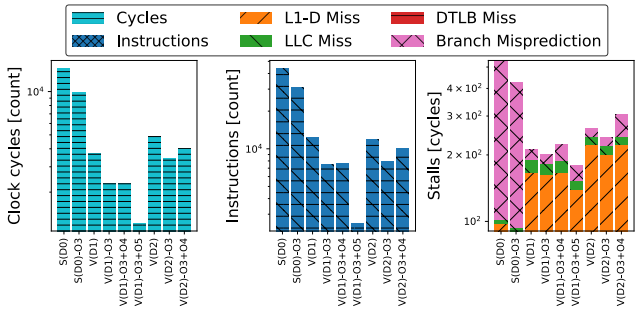


Figure 11: Spatial join: Effect of SIMD and optimizations. (Dataset size = 10M, Maximum fanout = 64)

5.2.2 Effect of optimizations. Out of the 3 optimizations for spatial join, O3 is the most effective. It reduces the effective size of the outer index node by pruning outer index child nodes, which positively impacts all hardware performance counters. The scalar version enhances query latency by 1.28×, while for vector layouts D0 and D1 the speedup is 1.50× and 1.49×, respectively. We can achieve an additional 1.00× and 1.31× improvement in terms of query latency by applying O4 and O5, respectively, on top of O3 for layout D1 by pruning multiple inner index child nodes.

Introducing O4 over O3 requires executing additional branches to check if the pruning condition is satisfied. This exposes the processor to further speculations resulting in more branch mispredictions, and hence requires executing more instructions, e.g., O4 incurs 1.85× more branch mispredictions for D1 and 2.84× for D2. However, these pruning strategies reduce the data footprint of the query, i.e., requiring less data to be fetched from memory. This can be observed in terms of an improved number of cache misses of O3 over no optimizations, and O4+O3, O5+O3 over O3. Hence, for O3+O4, there is a tradeoff between the number of instructions executed and the number of cache misses. For D1, the performance remains the same as O3, while for D2 it degrades (1.11×).

For D1, although O3+O5 has more branch mispredictions than O3, it executes 1.80× less instructions than O3. This is due to the *many to many* comparison strategy that requires less comparison instructions to execute and prunes the inner index node early. Hence, the performance gain from O3+O5 over O3+O4 is 1.32× in terms of query latency for layout D1. A better instruction count, cache performance and speculation attributes to this speedup.

5.2.3 Effect of node layouts. Contrary to range selects, in-memory spatial join is CPU-bound. Refer to Figure 8b. Generally, the data layout with less retired instruction counts outperforms the other. Thus, D1 outperforms D2 in all scenarios of spatial join.

5.2.4 Effect of maximum fanout. The common trend is that as the maximum fanout of the R-tree increases, the performance of spatial join degrades significantly irrespective of the utilization of SIMD or the optimization techniques, e.g., for the best performing vector algorithm, V-(D1)+O3+O5 on an R-tree of 10M points, a join is 54.90× slower for maximum fanout 2048 than for maximum fanout 64. The increase in the number of instructions executed, and L1-D, L1-I, LLC misses cause this degradation (Figure 10c).

The trends in Sections 5.2.1 and 5.2.2 for an R-tree with fanout 64 hold for the other fanouts, with a few exceptions. Node layout D1 with O3 (V-D1+O3), slightly outperforms the other D1 variants for smaller fanouts, i.e., 16 and 32. The clock cycles saved from minimizing cache misses cannot overpower the increase in number of instructions due to the additional pruning strategy. Similarly, V(D1)-O3+O4 outperforms V(D1)-O3+O5 for smaller fanouts on D1. To compensate for the reduction in comparison instructions, O5 has multiple costly gather, permute, blend instructions. Due to the logarithmic nature of this strategy, the savings only show when the fanout is sufficiently larger, i.e., starting from 64.

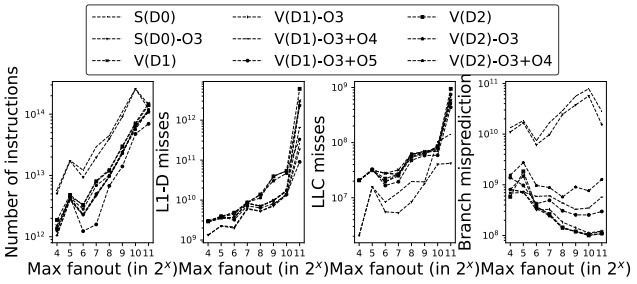


Figure 12: Spatial join: Effect of maximum fanout on h/w performance counters. (Dataset size = 10M)

6 RELATED WORK

Spatial Operators. A large body of work studies query processing in the context of spatial databases. [9, 27] study the Nearest-neighbor (NN) and k-nearest neighbor (kNN) queries in spatial databases following a depth-first and best-first approaches, respectively. Several spatial join algorithms exist, and differ based on

whether both [5], only one [3, 18] or none [1, 19, 21] of the input relations are indexed. While [5] traverses both indexes synchronously, [1] follows a plane-sweep approach sweeping through the query rectangles and data points that are sorted on one of the dimensions. [21] partitions the two spatial datasets into the same grid and extends over [1] to perform the join operation. In contrast to our work, none of the algorithms proposed in the spatial databases literature consider SIMD to vectorize the algorithms.

SIMD DB operators. An extensive list of vectorized operators exist in database literature to utilize SIMD capabilities of the hardware ranging from scan [17, 30], join [2, 4, 13] to compression [24], sorting [10, 23, 28], bloom filters [23]. Some of these operators exhibit linear access patterns, e.g., scan, sorting, while others, e.g., bloom filters, exhibit random access patterns. Our work falls in the category of random access vectorized operators.

SIMD and prefetching in tree indexes . There exists a branch of work with the same philosophy as of ours that design index node layouts for a limited number of index operations, i.e., tree traversal and search to benefit from SIMD vectorization, e.g., FAST [12], VAST [31], ART [16]. [7] studies prefetching in the context of SIMD to reduce cache misses and enhance the benefits gained from vectorization, while [6] studies prefetching in the context of tree indexes, i.e., B+ tree. In this paper, we study both SIMD and prefetching in the context of the R-tree. [26] proposes multiple partially vectorized search algorithms to traverse tree-like indexes, e.g., B+ tree, Quad tree using SIMD instructions. In contrast, our proposed algorithms are fully vectorized.

7 CONCLUSION

In this paper, we vectorize spatial range select and join operators, and investigate how spatial operators for an in-memory R-tree benefit from SIMD vectorization. The key findings can be summarized as follows.

- Vectorized range select operator outperforms the best performing scalar variant from 2× to 4×.
- Vectorized spatial join outperforms the best performing scalar variant from 4× to 9×.
- Vectorized select can benefit from avoiding recursion (O1) and prefetching (O2) by upto 1.63× and 1.84×, respectively.
- Vectorized join can benefit from slicing the outer index node (O3) by upto 1.63×.
- Shrinking the MBR of the inner index node *in batches* (O5) can speedup the vectorized join by upto 2.09× (O4).
- Data Layout D1 is favorable for CPU-bound operators, e.g., join, compared to Data Layout D2 that is favorable for memory-bound operators.
- The vectorized R-tree with smaller maximum fanout performs better than the one with larger fanout.

8 ACKNOWLEDGEMENTS

Walid G. Aref acknowledges the support of the National Science Foundation under Grant Number IIS-1910216.

REFERENCES

[1] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. 1998. Scalable Sweeping-Based Spatial Join. In *VLDB*. Morgan Kaufmann, 570–581.

[2] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. IEEE Computer Society, 362–373.

[3] Ludger Becker, Klaus H. Hinrichs, and Ulrich Finke. 1993. A New Algorithm for Computing Joins with Grid Files. In *ICDE*. IEEE Computer Society, 190–197.

[4] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*. ACM, 37–48.

[5] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD*. ACM Press, 237–246.

[6] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2001. Improving Index Performance through Prefetching. In *SIGMOD*. ACM, 235–246.

[7] Zhuhe Fang, Beilei Zheng, and Chuliang Weng. 2019. Interleaved Multi-Vectorizing. *PVLDB* 13, 3 (2019), 226–238.

[8] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. ACM Press, 47–57.

[9] Gísli R. Hjaltason and Hanan Samet. 1999. Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.* 24, 2 (1999), 265–318.

[10] Hiroshi Inoue and Kenjiro Taura. 2015. SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures. *PVLDB* 8, 11 (2015), 1274–1285.

[11] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *PVLDB* 11, 13 (2018), 2209–2222.

[12] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*. ACM, 339–350.

[13] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB* 2, 2 (2009), 1378–1389.

[14] Kihong Kim, Sang Kyun Cha, and Keunjoo Kwon. 2001. Optimizing Multidimensional Index Trees for Main Memory Access. In *SIGMOD*. ACM, 139–150.

[15] Kyung-Chang Kim and Suk-Woo Yun. 2004. MR-Tree: A Cache-Conscious Main Memory Spatial Index Structure for Mobile GIS. In *W2GIS*, Vol. 3428. Springer, 167–180.

[16] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. IEEE Computer Society, 38–49.

[17] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *SIGMOD*. ACM, 289–300.

[18] Ming-Ling Lo and Chinya V. Ravishankar. 1994. Spatial Joins Using Seeded Trees. In *SIGMOD*. ACM Press, 209–220.

[19] Ming-Ling Lo and Chinya V. Ravishankar. 1996. Spatial Hash-Joins. In *SIGMOD*. ACM Press, 247–258.

[20] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *PVLDB* 11, 1 (2017), 1–13.

[21] Jignesh M. Patel and David J. DeWitt. 1996. Partition Based Spatial-Merge Join. In *SIGMOD*. ACM Press, 259–270.

[22] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*. ACM, 1493–1508.

[23] Orestis Polychroniou and Kenneth A. Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*. ACM, 755–766.

[24] Orestis Polychroniou and Kenneth A. Ross. 2015. Efficient Lightweight Compression Alongside Fast Scans. In *DaMoN*. ACM, 9:1–9:6.

[25] Orestis Polychroniou and Kenneth A. Ross. 2019. Towards Practical Vectorized Analytical Query Engines. In *DaMoN*. ACM, 10:1–10:7.

[26] Kenneth A. Ross. 2004. Selection conditions in main memory. *ACM Trans. Database Syst.* 29 (2004), 132–161.

[27] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. 1995. Nearest Neighbor Queries. In *SIGMOD*. ACM Press, 71–79.

[28] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*. ACM, 351–362.

[29] Ruby Y. Tahboub and Tiark Rompf. 2020. Architecting a Query Compiler for Spatial Workloads. In *SIGMOD*. ACM, 2103–2118.

[30] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB* 2, 1 (2009), 385–394.

[31] Takeshi Yamamuro, Makoto Onizuka, Toshio Hitaka, and Masashi Yamamuro. 2012. VAST-Tree: a vector-advanced and compressed structure for massive data tree traversal. In *EDBT*. ACM, 396–407.

[32] Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *SIGMOD*. ACM, 145–156.