Efficiently Indexing Large Data on GPUs with Fast Interconnects

Josef Schmeißer
j.m.schmeisser@gmail.com
Unaffiliated
Germany

Clemens Lutz*
clutz@nvidia.com
NVIDIA
Santa Clara, CA, USA

Volker Markl
volker.markl@tu-berlin.de
BIFOLD, DFKI GmbH, TU Berlin
Berlin, Germany

ABSTRACT

Modern GPUs have long been capable of processing queries at a high throughput. However, until recently, GPUs faced slow data transfers from CPU main memory, and thus did not reach high processing rates for large, out-of-core data. To cope, database management systems (DBMSs) restrict their data access path to bulk data transfers orchestrated by the CPU, i.e., table scans. When queries expose selectivity, a full table scan wastes bandwidth, leaving performance on the table. With the arrival of fast interconnects, this design choice must be reconsidered. GPUs can directly access data at up to 7× higher bandwidth, whereby bytes are loaded on-demand.

We investigate four classic and recent index structures (binary search, B+tree, Harmonia, and RadixSpline), which we access via a fast interconnect. We show that indexing data can reduce transfer volume. However, when embedded into an index-nested loop join, we find that all indexes fail to outperform a hash join in the most interesting case: a highly selective query on large data (over 100 GiB). Therefore, we propose *windowed partitioning*, an index lookup optimization that generalizes to any index. As a result, index-nested loop joins run up to 3–10× faster than a hash join. Overall, we show that out-of-core indexes are a feasible design choice to exploit selectivity when using a fast interconnect.

1 INTRODUCTION

High-throughput query processing is enticing for cloud data warehouses, which handle exabytes of data every day [2, 5]. Previous work has shown that GPU-enabled DBMSs are able to improve query execution speed [8, 14, 17, 18, 37, 48]. However, high speedups have only been attainable for small data sets that fit into the on-board GPU memory [44]. In contrast, large data are stored *out-of-core*, e.g., in CPU memory [17]. Thus, data inherently traverse an interconnect to the GPU, causing a *data transfer bottleneck* [29].

Fast access to large out-of-core data can be achieved in two ways:

Reducing the transfer volume. Large-scale analytical queries are often highly selective. This is reflected widely used benchmarks [6, 39]. Recent DBMSs take advantage of selectivity through secondary index structures to reduce storage accesses [42, 49]. Index structures such as B-trees [3, 4, 43, 50] and others [19, 24, 27, 28] have been investigated on GPUs to accelerate point and range lookups. However, existing transfer optimizations focus on table scans [7, 32, 38, 40, 45, 48]. In contrast to these techniques, an index reduces the transfer volume by exploiting predicate

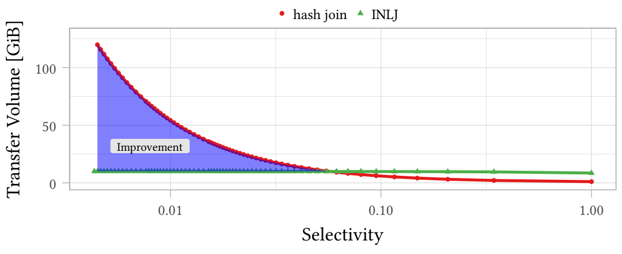

Figure 1: For selective predicates, a hash join transfers more data than necessary across the interconnect. In contrast, index joins reduce the data transfer volume.

selectivity to skip entire rows. However, skipping rows results in an irregular access pattern, which current PCI-e interconnects cannot handle efficiently [29]. This becomes even more challenging if indexes are stored in CPU memory. In that case, the index traversal incurs transfers, albeit with a higher locality [41].

Increasing the transfer rate presents an orthogonal path to higher query performance. Recent fast interconnects such as NVLink [33, 34, 36] and Infinity Fabric [1] are emerging as a new technology to transfer data between the CPU and GPU. As this next hardware generation provides the GPU with high bandwidth to CPU memory, GPUs are able to scan tables on a level playing field with CPUs. However, this does not lead to a speedup over CPUs in scan-intensive queries, as CPU memory bandwidth becomes the limiting factor [29]. Although fast interconnects improve random access bandwidth, fined-grained accesses still limit the performance, e.g., of out-of-core hash tables [30]. Thus, fast interconnects only lay the groundwork for processing large data volumes using GPUs.

In this paper, we investigate how GPUs can leverage index structures to transfer less data than a table scan, as shown in Fig. 1. Fast interconnects support data-dependent memory accesses, a crucial feature for index lookups to CPU memory. This enables us to evaluate state-of-the-art GPU index structures using an index-nested loop join (INLJ). We find that an INLJ does not outperform a hash join, even when join selectivity is low. Thus, we propose to (1) partition lookup keys to reduce search path divergence, and (2) reestablish the tuple stream by partitioning inside of windows. We show the generality of our approach by applying it to all of our index structures.

Our main contributions are:

(1) We analyze the performance limitations of three state-of-the-art GPU indexes and a binary search when scaling an INLJ up to 120 GiB over a fast interconnect. To the best of our knowledge, we are the first to apply GPU indexes to out-of-core data.

*Work partially conducted at TU Berlin.

Table 1: Overview of interconnect receive bandwidth.

GPU	Interconnect	Bandwidth
various	PCI-e 4.0	32 GB/s
various	PCI-e 5.0	64 GB/s
AMD MI250X	Infinity Fabric 3	72 GB/s
NVIDIA V100	NVLink 2.0	75 GB/s
NVIDIA GH200	NVLink C2C	450 GB/s

(2) With detailed hardware performance counter measurements, we show how partitioning the search keys improves the index traversal path and thereby avoids GPU TLB misses.
(3) We propose a partitioning window approach that retains the performance of partitioned lookups, without materialization.
(4) Overall, we find that for selective joins, index structures can scale the INLJ to large datasets with a high throughput. Thus, INLJs achieve speedups of up to 3–10× over a hash join.

The remainder of this work is structured as follows. In Section 2, we provide a brief background on fast interconnects and summarize related work on GPU index structures. We then analyze why indexes do not live up to expectations for out-of-core lookups in Section 3. Subsequently, in Section 4, we experimentally evaluate how partitioning mitigates those issues. In Section 5, we contribute our partitioning window approach. Finally, we discuss our insights in Section 6 and give our concludes in Section 7.

2 BACKGROUND AND RELATED WORK

We summarize the background on fast interconnects. Further, we cover related work on index structures and partitioned joins.

2.1 Fast Interconnects

In this work, we assume that data are stored in CPU memory due to their large size. To access these *base relations*, the DBMS transfers them to the GPU across an interconnect. With fast interconnects, GPUs receive data from the CPU at rates up to 450 GB/s, which we summarize in Table 1. On GH200 platforms with large CPU memory, the receive rate by itself (i.e., no bidirectional transfers) exceeds the CPU memory bandwidth [35]. Thus, fast interconnects can eliminate the data transfer bottleneck.

The GPU is able to dereference pointers to CPU memory. Pointers can lead to *data-dependent memory accesses*, e.g., while traversing linked index structures such as B+trees. That means access locations are determined by the data's content [16], and the access pattern is subject to the data order and distribution. To fulfill the memory request, the GPU does not transfer a memory page, as with heterogeneous memory management via PCI-e [20]. Instead, the GPU fetches a cacheline across the interconnect [30]. Thus, DBMSs can take advantage of fine-grained accesses to CPU memory.

Prior research shows that DBMSs can enable new functionality based on these new features [21, 29–31, 40]. We build on a detailed hardware analysis [30] to investigate index structures.

2.2 Index Structures

Recent works propose new learned index structures, and optimize existing indexes for GPUs. We give a brief summary.

The RadixSpline [25] is a learned index on a sorted array. It works by defining *spline points* in the data, which are captured together with their location. An offset array points to radix partitions of the spline points, based on the most significant bits of the keys. A lookup first locates the two spline points closest to the lookup key, and interpolates these to start a binary search in the data. In contrast to prior work, we evaluate the RadixSpline on the GPU.

GPU-optimized index structures are tuned for the high parallelism and SIMT architecture of GPUs [3, 4, 24, 27, 43, 47, 50], and ray tracing cores [19]. We focus on Harmonia [47], a B+tree with a high lookup throughput. Harmonia introduces several optimizations, the main one being its cooperative sub-warp tree traversal. A *warp* is the unit of execution on GPUs and consists of 32 threads on NVIDIA GPUs [9]. Harmonia parallelizes key comparisons with a warp, recognizing that lookup paths are less likely to diverge near the root node. As some comparisons are unnecessary, Harmonia divides the warp into *sub-warps* to parallelize over lookup keys as well. In contrast to previous research, we investigate lookups conducted by the GPU to large base relations in CPU memory.

2.3 Join Input Partitioning

Partitioning data ahead of hash joins has been previously studied on GPUs [10, 15, 30]. However, with some exceptions, partitioned joins are detrimental to overall query performance [13]. On top, partitioning both inputs consumes additional memory equal to the input size. By comparison, hash joins and INLJs integrate well into query execution because these joins pipeline the probe-side input. I.e., the tuple stream is processed on-the-fly. Best-effort partitioning [12] extends these semantics to tuple batches by partitioning probe-side batches on-the-fly.

Our partitioning window approach fundamentally differs from partitioned joins, as it does not materialize either input. Instead, it partitions the probe-side input on-the-fly. In contrast to best-effort partitioning, our approach is based on an INLJ and does not partition the index. As the index is typically declared on the larger join input, the reuse factor [12] would be low. Inspired by *windows* in stream processing [11], we only modify the tuple stream with the aim to optimize index traversals.

3 ANALYSIS OF OUT-OF-CORE INDEX STRUCTURES

As a basis for our work, we analyze out-of-core index accesses using NVLink 2.0. We describe how the GPU performs out-of-core index accesses in our index structure implementations. Then we introduce our experiment setup, and proceed to experimentally evaluate the indexes in an INLJ.

3.1 Out-of-Core Index Accesses

In this work, we cover four GPU index structures: a RadixSpline, Harmonia, a standard B+tree, and a binary search. A common drawback of these index structures is their tendency to perform more memory accesses per lookup key than a hash join, i.e., $O(log(n))$ vs. $O(1)$ expected accesses. On the surface, these accesses are data-dependent and we might assume they incur an irregular pattern. We detail why indexes do not result in $O(log(n))$ remote accesses in out-of-core scenarios.

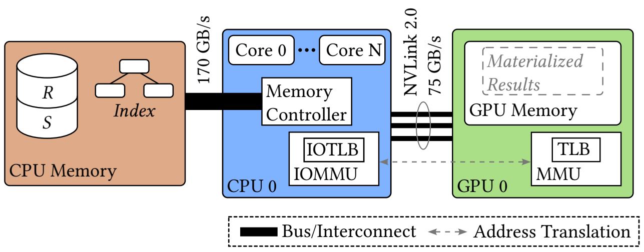

Figure 2: Overview of our hardware setup and data domains.

During a hash table lookup, the first access incurs a single random access as defined by the hash function. For reasonably large hash tables, this first access is almost always guaranteed to generate a cache miss. However, hash table implementations try to avoid further cache misses on hash conflicts and for multi-map semantics. For example, a linear probing scheme only accesses contiguous addresses. This pattern has a high data locality as the next entry are located either in the same or an adjacent cacheline. With bucket chaining the situation is quite similar. Here, the entry contains a pointer to the first bucket, which incurs another random access. However, buckets do not necessarily form a linked-list of individual items. Instead, multiple items can be gathered into blocks to increase data locality. However, hash tables perform best when stored in GPU memory, precisely how DBMSs utilize them in hash joins.

The picture is different for index structures such as B+trees, on which INLJs are based. After the first few key lookups, the upper-most tree levels are assumed to be cached and do not incur memory accesses. Within each tree node, a binary search locates the lower bound index of a queried key. This potentially leads to multiple accesses per node in the lower tree levels, as large nodes span multiple cachelines. As binary search divides the search space in each iteration, random accesses patterns can occur not only when traversing nodes, but also within each node. Using smaller nodes has also been suggested [22], but has the disadvantage that fewer keys fit into each node. As a result, the tree grows in height, in turn leading to more tree levels being traversed. However, in GPU-optimized indexes such as Harmonia, sub-warps traverse the index. Thereby traversals amortize some cache misses over multiple keys, although the traversal paths ultimately diverge.

Overall, caching and sub-warp traversals reduce the remote memory accesses incurred by out-of-core index structures.

3.2 Experiment Setup

We give an overview of our experiment setup, shown in Fig. 2. First, we introduce the experiment environment and methodology. We describe the workloads and join selectivity. Finally, we specify the data and a hash join baseline.

Environment. Our benchmark machine is equipped with two IBM POWER9 CPUs at 3.8 GHz with 16 cores each and a total of 256 GiB memory. The system is furthermore equipped with 2 NVIDIA Tesla V100-SMX2 GPUs of which we only use one in our benchmarks. The machine is set up to use 1 GiB huge pages. We found that using huge pages of this size improves the repetition accuracy of our experiments compared to 2 MiB, although performance is approximately equal.

Workload. Our workload is inspired by queries such as TPC-H Q4 and Q12, which have a large input to a single join with a

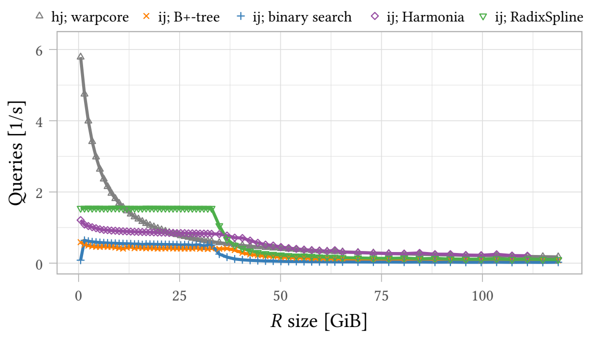

Figure 3: Query throughput on a single join with different index structures in an INLJ and a hash join as reference.

low join selectivity. We perform a query with an equality join, implemented as an INLJ. Our INLJ is a text book implementation that calls an index structure in the inner loop. We consider four index structures, namely: a B+tree, Harmonia, a RadixSpline, and a binary search. The B+tree is configured with 4 KiB nodes, Harmonia with 32 keys per node. All index structures and base relations are stored in CPU memory, and are directly accessed over the interconnect. The query materializes its results into GPU memory[1].

Methodology. We report throughput as queries per second, as this metric is oblivious of the join type. The throughput measures the entire query run, which includes windowed partitioning and result materialization. We assume that the larger relation is indexed, hence probe with the smaller relation. To reflect real-world use, we assume the index already exists when the query is run.

Join Selectivity ranges between 0.4–100%. The join result size remains constant as *S* and the match rate are fixed. However, selectivity decreases due to scaling the size of *R*.

Data. Our test dataset consists of two relations *R* and *S*. *R* contains unique, sorted keys. *R* is sorted to accommodate the binary search and RadixSpline, but not strictly necessary for the B+tree and Harmonia. *S* represents foreign keys into *R*, and is drawn from *R* following a uniform random distribution. Each relation contains a single 8 byte integer attribute to maximize the tree height of indexes. Throughout this work we keep *S* fixed at 2^{26} tuples (512 MiB) while scaling the size of *R*. Unless mentioned otherwise, *R* ranges between 2^{26}–$2^{33.9}$ tuples (0.5–120 GiB). However, size limit of *R* is reduced for the B+tree and Harmonia due to memory capacity constraints.

Baseline. We add a hash join baseline for comparability with existing GPU joins. We use the recent WarpCore library [23] for its `MultiValueHashTable` [26]. In our runs, we configure it with a 50% load factor and a block size of 512 keys. The hash table is kept in GPU memory. We flip the input relations to build on the smaller relation and reduce the hash table size. To reflect real-world use, the query builds the hash table on-the-fly, which we include in the throughput measurement.

3.3 Experiments

In the following, we demonstrate that fast interconnects form a basis for out-of-core INLJs by measuring the query throughput

[1]Large results could be spilled to CPU memory.

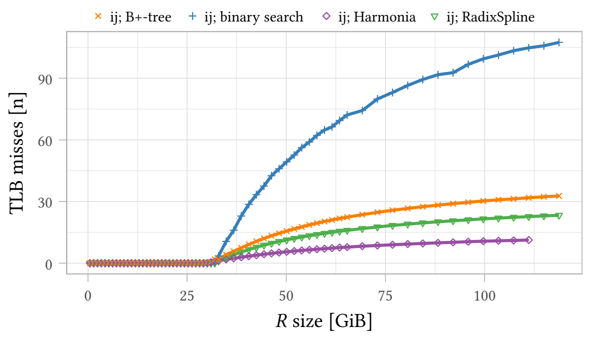

Figure 4: Index structure traversals incur address translation requests on the GPU.

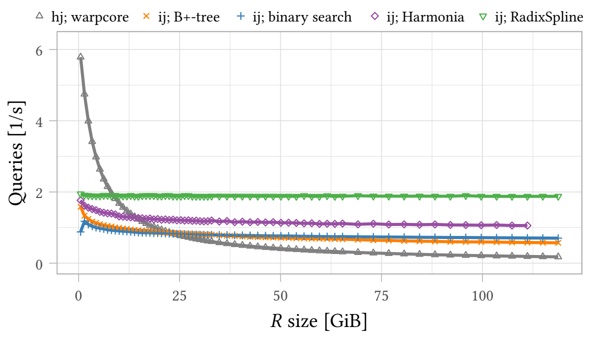

Figure 5: Throughput when partitioning lookup keys.

of a INLJs. However, the INLJs incurs TLB misses that reduce performance, which we record in a second graph.

3.3.1 Query Throughput. We analyze the effect on query throughput when scaling the indexed relation in Fig. 3. We focus on the join to emphasize INLJ throughput over the interconnect. The workload is defined as in Section 3.2.

Approach. The GPU implementation of INLJ dispatches a thread for each tuple of the probe side relation. Our probe side relation does not include any filter predicates to avoid warp divergence effects, i.e., all threads within a warp are always fully occupied. However, the selective join introduces filter divergence [18], a special case of warp divergence, in the index lookup. In case of Harmonia, threads are dynamically rescheduled into sub-warps, whereby each sub-warp is responsible for the lookup of a single tuple at once. The sub-warp progresses unto the next tuple, until each tuple in the initial warp has been processed.

Observations. The INLJ does not outperform the hash join, even at the low selectivities incurred by a large R relation. Ideally, we would expect the INLJ to exhibit a flat horizontal line, as the selectivity decreases with growing R. The result is a constant transfer volume. Instead, the INLJ experiences a sudden drop in throughout when R grows beyond 32 GiB. In contrast, hash join throughput does not drop suddenly. Instead, the queries per second of the hash join decreases smoothly with the growing transfer volume per query incurred by the table scan. In summary, the INLJ does not perform as expected.

3.3.2 TLB Misses. We analyze address translation requests to determine why the INLJ experiences a performance drop. Our GPU model has a *translation look-aside buffer* (TLB) range of 32 GiB [30], which correlates with our observation. In Fig. 4, we establish causality by measuring the number of translation requests per lookup. The experiment setup is identical to Section 3.3.1.

Approach. The TLB caches virtual to physical address mappings. Modern GPUs have multiple TLB levels, but we simplify the discussion to the last-level TLB. When a TLB miss occurs, the GPU issues an address translation request across the interconnect to the CPU. The CPU's I/O memory management unit sends the translation back to the GPU. The POWER9 CPU exposes hardware performance counters with which we measure the GPU's translation requests. We refer to Lutz et al. [30] for details.

Observations. For small relations, there are near zero translation requests. However, at the 32 GiB mark, the translation request rate of all INLJs spikes upwards. At 111 GiB of data, binary search requests 105 translations per key. In contrast, Harmonia experiences only 11.3 requests. As fulfilling translation requests incurs a high latency on the order of 3 µs [30], we identify TLB misses to be the main cause of the INLJ throughput drop.

4 PARTIALLY SORTED INDEX ACCESSES

Having identified that GPU TLB misses cause a performance degradation, we tackle the underplaying issue of random memory accesses. Our aim is to reduce TLB misses by partially sorting the lookup keys before executing the INLJ.

We give an intuition of how index traversals cause TLB misses, walk through a basic solution approach, and evaluate our findings.

4.1 Understanding TLB Misses in Index Structure Traversals

In our previous experiment, traversing the index results in TLB thrashing. Take binary search as an example. Given a random sequence of lookup keys searching uniformly distributed data, each traversal likely takes $log_2(N)$ steps. As threads processes keys in parallel, memory accesses evict TLB entries loaded by other threads in the shared TLB. By the time a thread is ready to process the next key, its previous traversal path is no longer cached in the TLB. In effect, nearly every memory access incurs a TLB miss.

Reordering the lookup keys to achieve better data locality can prevent TLB thrashing. In Harmonia, sorting keys improves lookup throughput by facilitating coalesced memory accesses and reducing warp divergence during the traversal [47]. The authors find that fully sorting the keys is not necessary, because the most significant bits of a key decide the leaf node and thus the traversal path. We determine that these effects are not significant for large data sets (Yan et al. [47] evaluate on 1.5 GiB of data). However, Harmonia's approach inspires a way to mitigate TLB misses.

4.2 Improving Data Locality

In our approach, we reduce the number of TLB misses by partitioning the lookup keys. Partitioning improves the spacial data locality of lookups, as each partition contains keys located close in memory. During processing, neighboring threads should traverse the index for keys within the GPU TLB's 32 GiB range. Hence, we optimize the memory access pattern to hit the TLB, thereby improving throughput.

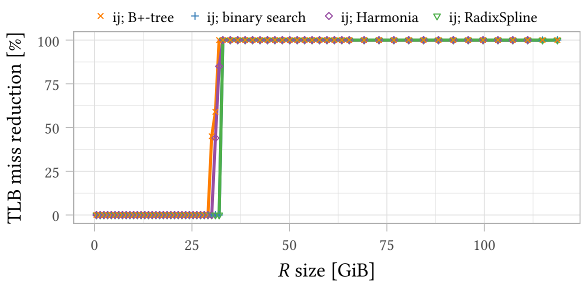

Figure 6: Partitioning the keys in the query reduces address translation requests per index lookup.

We have to consider two aspects in order to determine the bits on which to partition. First, the bits which are relevant for mapping keys to memory pages. Second, bits that define the index traversal path of the lookup. As the data set is smaller than the processor's address space, the most significant bits of keys are identical. Hence, these bits do not affect the output of a comparator. Simultaneously, the least significant bits fall into the same memory page. Thus, we choose bits starting at the bit splitting the root node, down to the bit above the page size.

4.3 Experiments

Similar to Section 3, we perform two experiments. We repeat the INLJ throughput analysis, but include the partitioning operator as part of the INLJ. Subsequently, we evaluate whether partitioning leads to a reduction in TLB misses.

4.3.1 Query throughput. In Fig. 5, we scale the size of R to investigate the query throughput of INLJs. Otherwise, the experiment setup is identical to Section 3.3.1.

Approach. Different to our previous experiment, the INLJ operator partitions the lookup keys before executing the join. In our implementation, we radix partition the lookup keys using the linear allocator-based software write-combining algorithm [46], due to its high performance in GPU memory. We set it to 2048 partitions, ignoring the 4 least significant bits of the key. We do not change the hash join baseline, as its table scan is not subject to frequent TLB misses.

Observations. In contrast to the results in Fig. 3, the sudden drop in performance is now remedied. The INLJ achieves a more consistent throughput for all index structures, also in the range beyond 32 GiB. However, even below that limit, query throughput is higher. Unsurprisingly, the binary search and tree-based indexes follow a subtle downwards trend due to their logarithmic lookup complexity. At 111 GiB, the INLJs achieve 0.6, 0.7, 1, and 1.9 Q/s respectively for the B+tree, the binary search, Harmonia, and the RadixSpline. This contrasts to 0.2 Q/s for the hash join. In summary, partitioning speeds up the INLJ by up to 10× over the hash join.

4.3.2 TLB Misses. To confirm that the performance increase results from less TLB misses, we compare the TLB behavior of Section 3.3.2 and our partitioning approach. Our results are depicted in Fig. 6.

Approach. Like before, we record the number of address translation requests per key. In the plot, we show the percentage of translation requests that have been eliminated relative to Section 3.3.2.

Observations. The improvement at the TLB range boundary is nearly 100%. This indicates that almost no address requests occur. A close inspection shows that binary search still experiences about 0.1 translation requests per lookup. However, the other indexes have almost zero requests per key. The tree-based index structures see the improvement a data point before the others, presumably due to their larger persistent state. Overall, partitioning eliminates the INLJ throughput drop for large data sets by reducing TLB misses.

5 PARTITIONING WINDOW

The approach in Section 4 materializes the lookup keys, which is undesirable as outlined in Section 2.3. As a solution, we contribute our partitioning window approach.

5.1 Our Approach

We propose to restrict partitioning to a window in order to restore the pipeline while maintaining a high TLB hit rate.

Our approach. We divide the stream on-the-fly into disjoint, fixed-size batches, i.e., *tumbling windows* [11]. When a window is "closed", we partition the tuples contained in the window. After that, the window is passed on to the INLJ, which computes the join for the window's contents and continues the stream. In principle, any partitioning operator and INLJ variants can be used. We suggest applying a radix partition, and the INLJ described in Section 3.

Closing the window occurs either when the window reaches its capacity, or no more tuples are available on the probe-side of the join. Although this approach could be augmented with stream processing semantics, we focus on batch processing and rely on an outer loop, e.g., a scan operator, to end the input stream.

We apply two GPU optimizations: concurrent kernel execution and window size tuning.

Concurrent kernel execution. The pipeline consists of multiple GPU kernels, which directly access CPU memory. However, not all kernels transfer data. If kernels were to run consecutively, the interconnect would be underutilized. Therefore, we achieve transfer-compute overlap by permitting the GPU to execute two CUDA streams simultaneously with concurrent kernel execution, as described by Lutz et al. [30].

Window size tuning is important to avoid TLB misses. A small window (i.e., a vector) takes advantage of hardware caches to store tuples. Conversely, a large window (e.g., 100 MiB) amortizes TLB misses over more tuples. Given that GPU TLB misses cost an order-of-magnitude more than GPU memory accesses [30], one might lean towards a large window size. However, our following experiment suggests that small windows suffice.

5.2 Experiments

We experimentally establish the effectiveness of our partitioned window approach.

5.2.1 Window Size. In Fig. 7, we evaluate how the window size affects query throughput. We reuse the setup outlined in Section 3.2.

Approach. In this experiment, we keep relation S as-is with 2^{26} tuples, but fix R at 100 GiB. We vary the window size from 2^{18} to 2^{26} tuples (2–512 MiB), and record the query throughput for each index structure.

Observations. The throughput of all index structures remains within 2×, indicating that the GPU TLB does not cause a performance drop. However, index structures react differently to

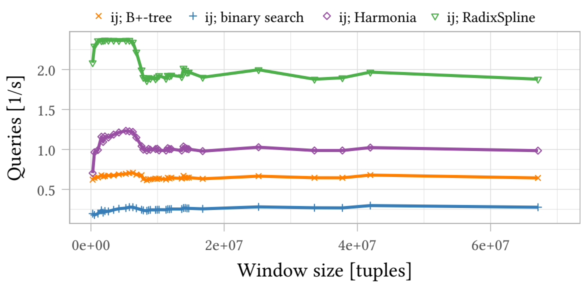

Figure 7: Impact of the window size on query throughput.

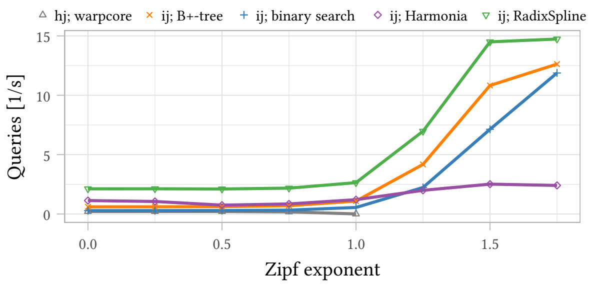

Figure 8: Query throughput of skewed lookup keys.

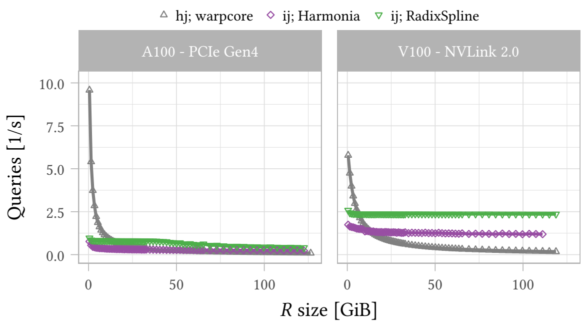

Figure 9: Comparison of PCI-e 4.0 and NVLink 2.0.

window size scaling. In fact, small window sizes in range 4–52 MiB yield the highest throughput for the RadixSpline. Harmonia also prefers small windows. In contrast, binary search and the B+tree show only minor performance variation.

5.2.2 Skewed Lookup Keys. We present skew handling in Fig. 8.

Approach. To determine the effect of skew, we Zipf-distribute the lookup keys in the exponent range 0–1.75. The relation sizes are the same as Section 5.2.1. We set the window size to 32 MiB.

Observations. Throughput increases with Zipf exponents higher than 1.0. With exponent 1.0, we calculate a 69% chance of hitting the L1 cache. Our approach is able to handle this high skew. However, the hash join degrades to a long probe chain. After 10 hours, we terminated the measurement run.

5.2.3 Hardware Comparison. In Fig. 9 we study our partitioned windowing on a different hardware setup. We show the two fastest INLJ variants, RadixSpline and Harmonia.

Approach. We compare an NVIDIA A100 with PCI-e 4.0 to our previous GPU setup. We set the window size to 32 MiB. The relation sizes are as in Section 5.2.1.

Observations. The hash join achieves 1.7× higher throughput on the A100, as it is a faster GPU. Therefore, the crossover point of INLJ vs. hash join on the A100 is at 13.9 GiB (3.6%), compared to 6.2 GiB (8.0%) on the V100. We presume that this is because fast interconnects have higher random access throughput than PCI-e [29], which makes them better-suited for index lookups. However, we leave a detailed analysis of out-of-core indexing using PCI-e to future work. Nonetheless, in principle, our approach is portable to other GPU and interconnect architectures.

In summary, our insight is that our partitioned window approach successfully achieves a high GPU TLB hit rate together with pipelineability on reasonably small windows.

6 DISCUSSION

We summarize and discuss the findings of our investigation.

Fast interconnects enable indexing large data sets in CPU memory. By using a fast interconnect, the GPU requests CPU memory at a cacheline granularity. We exploit this feature to index data up to 120 GiB, allowing the GPU to select an index scan instead of a full table scan. In our experiments, the index reduces the transfer volume by up to 12×.

Partitioning lookup keys reduces TLB misses. For large data, we find that TLB misses incur a throughput drop up to 16.7×. Our new partitioning window approach counteracts this hardware limitation. Simultaneously, our approach avoids materializing the lookup keys. As a result, throughput remains consistently high when scaling the data volume. Our evaluation reveals that our approach can be applied to multiple index types.

Index structures improve the throughput of selective joins. In our experiments, we demonstrate that an out-of-core INLJ can outperform a hash join below 8.0% selectivity. We recommend choosing a RadixSpline, as it has 1.1–1.8× higher throughput than the second-best index, Harmonia. However, Harmonia is a good alternative if the index must support inserts and updates.

7 CONCLUSION

The transfer volume directly impacts the time taken to access data from the GPU. In this paper, we have shown that index structures reduce the transfer volume incurred by selective joins on the GPU. We have contributed a partitioned window approach that increases the locality of lookup keys, and thereby efficiently performed index lookups across a fast interconnect. In our investigation, we have demonstrated promising results with speedups up to 10×.

ACKNOWLEDGMENTS

We thank Daniel Jünger (NVIDIA) for his advice on WarpCore.

REFERENCES

[1] AMD 2021. *AMD CDNA 2 architecture.* AMD. Retrieved Oct 8, 2024 from https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf

[2] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022.* ACM, 2205–2217. https://doi.org/10.1145/3514221.3526045

[3] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martin Farach-Colton, and John D. Owens. 2019. Engineering a high-performance GPU B-Tree. In *SIGPLAN*. ACM, 145–157. https://doi.org/10.1145/3293883.3295706

[4] Muhammad A. Awad, Serban D. Porumbescu, and John D. Owens. 2022. A GPU Multiversion B-Tree. In *PACT*. ACM, 481–493. https://doi.org/10.1145/3559009.3569681

[5] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 2326–2339. https://doi.org/10.1145/3514221.3526054

[6] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *TPCTC (Lecture Notes in Computer Science)*, Vol. 8391. Springer, 61–76. https://doi.org/10.1007/978-3-319-04936-6_5

[7] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust query processing in co-processor-accelerated databases. In *SIGMOD*. ACM, 1891–1906. https://doi.org/10.1145/2882903.2882936

[8] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (2023), 441–454. https://doi.org/10.14778/3632093.3632107

[9] Erik Lindholm and John Nickolls and Stuart F. Oberman and John Montrym. 2008. Nvidia Tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (2008), 39–55. https://doi.org/10.1109/MM.2008.31

[10] Johns Paul and Bingsheng He and Shengliang Lu and Chiew Tong Lau. 2019. Revisiting hash join on graphics processors: A decade later. In *ICDEW*. IEEE, Washington, DC, USA, 294–299. https://doi.org/10.1109/ICDEW.2019.00008

[11] Juliane Verwiebe and Philipp M. Grulich and Jonas Traub and Volker Markl. 2023. Survey of window types for aggregation in stream processing systems. *VLDB J.* 32, 5 (2023), 985–1011. https://doi.org/10.1007/S00778-022-00778-6

[12] Marcin Zukowski and Sándor Héman and Peter A. Boncz. 2006. Architecture-conscious hashing. In *DaMoN*. ACM, 6–es. https://doi.org/10.1145/1140402.1140410

[13] Maximilian Bandle and Jana Giceva and Thomas Neumann. 2021. To partition, or not to partition, that is the join question in a real system. In *SIGMOD*. ACM, 168–180. https://doi.org/10.1145/3448016.3452831

[14] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556.

[15] Sioulas, Panagiotis and Chrysogelos, Periklis and Karpathiotakis, Manos and Appuswamy, Raja and Ailamaki, Anastasia. 2019. Hardware-conscious hash-joins on GPUs. In *ICDE*. IEEE, Washington, DC, USA, 698–709. https://doi.org/10.1109/ICDE.2019.00068

[16] Yinan Li and Ippokratis Pandis and René Müller and Vijayshankar Raman and Guy M. Lohman. 2013. NUMA-aware algorithms: The case of data shuffling. In *CIDR*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper121.pdf

[17] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *SIGMOD Conference*. ACM, 1603–1618.

[18] Henning Funke and Jens Teubner. 2020. Data-Parallel Query Processing on Non-Uniform Data. *Proc. VLDB Endow.* 13, 6 (2020), 884–897. https://doi.org/10.14778/3380750.3380758

[19] Justus Henneberg and Felix Schuhknecht. 2023. RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. *Proc. VLDB Endow.* 16, 13 (2023), 4268–4281.

[20] John Hubbard, Gonzalo Brito, Chirayu Garg, Nikolay Sakharnykh, and Fred Oh. 2023. *Simplifying GPU application development with heterogeneous memory management*. NVIDIA. Retrieved Oct 3, 2024 from https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management

[21] Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. 2023. RMG Sort: Radix-Partitioning-Based Multi-GPU Sorting. In *Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 06.-10, März 2023, Dresden, Germany, Proceedings (LNI)*, Birgitta König-Ries, Stefanie Scherzinger, Wolfgang Lehner, and Gottfried Vossen (Eds.), Vol. P-331. Gesellschaft für Informatik e.V., 305–328. https://doi.org/10.18420/BTW2023-15

[22] Rize Jin and Tae-Sun Chung. 2010. Node Compression Techniques Based on Cache-Sensitive B+-Tree. In *9th IEEE/ACIS ICIS 2010*. 133–138. https://doi.org/10.1109/ICIS.2010.9

[23] Daniel Jünger, Robin Kobus, André Müller, Christian Hundt, Kai Xu, Weiguo Liu, and Bertil Schmidt. 2020. WarpCore: A library for fast hash tables on GPUs. In *HiPC*. IEEE, 11–20. https://doi.org/10.1109/HIPC50609.2020.00015

[24] 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. https://doi.org/10.1145/1807167.1807206

[25] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *aiDM@SIGMOD*. ACM, 5:1–5:5.

[26] Robin Kobus, André Müller, Daniel Jünger, Christian Hundt, and Bertil Schmidt. 2021. MetaCache-GPU: Ultra-fast metagenomic classification. In *ICPP*. ACM, 25:1–25:11. https://doi.org/10.1145/3472456.3472460

[27] Martin Koppehel, Tobias Groth, Sven Groppe, and Thilo Pionteck. 2021. CuART - A CUDA-based, Scalable Radix-Tree Lookup and Update Engine. In *ICPP*. ACM, 12:1–12:10. https://doi.org/10.1145/3472456.3472511

[28] Jiesong Liu, Feng Zhang, Lv Lu, Chang Qi, Xiaoguang Guo, Dong Deng, Guoliang Li, Huanchen Zhang, Jidong Zhai, Hechen Zhang, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2024. G-Learned Index: Enabling Efficient Learned Index on GPU. *IEEE Trans. Parallel Distributed Syst.* 35, 6 (2024), 795–812. https://doi.org/10.1109/TPDS.2024.3381214

[29] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *SIGMOD Conference*. ACM, 1633–1649.

[30] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *SIGMOD Conference*. ACM, 1017–1032.

[31] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. 2022. Evaluating Multi-GPU Sorting with Modern Interconnects. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1795–1809. https://doi.org/10.1145/3514221.3517842

[32] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. HPCache: Memory-Efficient OLAP Through Proportional Caching. In *DaMoN*. ACM, 7:1–7:9. https://doi.org/10.1145/3533737.3535100

[33] NVIDIA 2017. *NVIDIA Tesla V100 GPU architecture*. NVIDIA. Retrieved Oct 8, 2024 from https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf WP-08608-001_v1.1.

[34] NVIDIA 2024. *NVIDIA Blackwell architecture technical brief*. NVIDIA. Retrieved Oct 8, 2024 from https://nvdam.widen.net/s/q8f9llv72p/nvidia-blackwell-architecture-technical-brief Version 1.1.

[35] NVIDIA 2024. *NVIDIA GH200 Grace Hopper Superchip*. NVIDIA. Retrieved Oct 8, 2024 from https://resources.nvidia.com/en-us-grace-cpu/grace-hopper-superchip Version JUL24.

[36] NVIDIA 2024. *NVIDIA GH200 Grace Hopper Superchip architecture*. NVIDIA. Retrieved Oct 8, 2024 from https://nvdam.widen.net/s/qjzrmfdn2j/nvidia-grace-hopper-superchip-architecture-whitepaper-v1.0 Version 1.21.

[37] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving Execution Efficiency of Just-in-time Compilation based Query Processing on GPUs. *Proc. VLDB Endow.* 14, 2 (2020), 202–214. https://doi.org/10.14778/3425879.3425890

[38] Holger Pirk, Stefan Manegold, and Martin L. Kersten. 2014. Waste not... Efficient co-processing of relational data. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 508–519. https://doi.org/10.1109/ICDE.2014.6816677

[39] Meikel Pöss, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why you should run TPC-DS: A workload analysis. In *PVLDB*. ACM, 1138–1149.

[40] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. GPU-Accelerated Data Management Under the Test of Time. In *CIDR*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p18-raza-cidr20.pdf

[41] Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2022. B^2-Tree: Page-Based String Indexing in Concurrent Environments. *Datenbank-Spektrum* 22, 1 (2022), 11–22.

[42] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. 2024. Two birds with one stone: Designing a hybrid cloud storage engine for HTAP. *PVLDB* 17, 11 (2024), 3290–3303. https://doi.org/10.14778/3681954.3682001

[43] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A Hybrid B+-tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *SIGMOD*. ACM, 1523–1538. https://doi.org/10.1145/2882903.2882918

[44] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *SIGMOD*. ACM, 1617–1632. https://doi.org/10.1145/3318464.3380595

[45] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based Lightweight Integer Compression in GPU. In *SIGMOD*. ACM, 1390–1403. https://doi.org/10.1145/3514221.3526132

[46] Elias Stehle and Hans-Arno Jacobsen. 2017. A memory bandwidth-efficient hybrid radix sort on GPUs. In *SIGMOD*. ACM, 417–432. https://doi.org/10.1145/3035918.3064043

[47] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. 2019. Harmonia: a high throughput B+tree for GPUs. In *PPoPP*. ACM, 133–144.

[48] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS. *PVLDB* 15, 11 (2022), 2491–2503. https://doi.org/10.14778/3551793.3551809

[49] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-time OLAP database system at Alibaba Cloud. *PVLDB* 12, 12 (2019), 2059–2070. https://doi.org/10.14778/3352063.3352124

[50] Weihua Zhang, Zhaofeng Yan, Yuzhe Lin, Chuanlei Zhao, and Lu Peng. 2020. A High Throughput B+tree for SIMD Architectures. *IEEE Trans. Parallel Distributed Syst.* 31, 3 (2020), 707–720. https://doi.org/10.1109/TPDS.2019.2942918