A Scalable and Generic Approach to Range Joins

Maximilian Reif
Technical University of Munich
reif@in.tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

ABSTRACT

Analytical database systems provide great insights into large datasets and are an excellent tool for data exploration and analysis. A central pillar of query processing is the efficient evaluation of equi-joins, typically with linear-time algorithms (e.g. hash joins). However, for many use-cases with location and temporal data, non-equi joins, like range joins, occur in queries. Without optimizations, this typically results in nested loop evaluation with quadratic complexity.

This leads to unacceptable query execution times. Different mitigations have been proposed in the past, like partitioning or sorting the data. While these allow for handling certain classes of queries, they tend to be restricted in the kind of queries they can support. And, perhaps even more importantly, they do not play nice with additional equality predicates that typically occur within a query and that have to be considered, too.

In this work, we present a kd-tree-based, multi-dimension range join that supports a very wide range of queries, and that can exploit additional equality constraints. This approach allows us to handle large classes of queries very efficiently, with negligible memory overhead, and it is suitable as a general-purpose solution for range queries in database systems. The join algorithm is fully parallel, both during the build and the probe phase, and scales to large problem instances and high core counts.

We demonstrate the feasibility of this approach by integrating it into our database system Umbra and performing extensive experiments with both large real world data sets and with synthetic benchmarks used for sensitivity analysis. In our experiments, it outperforms hand-tuned Spark code and all other database systems that we have tested.

PVLDB Reference Format:
Maximilian Reif and Thomas Neumann. A Scalable and Generic Approach to Range Joins. PVLDB, 15(11): 3018 - 3030, 2022.
doi:10.14778/3551793.3551849

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://gitlab.db.in.tum.de/max.reif/rangejoin-reproducibility.

1 INTRODUCTION

Over the last years, we observed two major trends in data processing: The amount of data collected is vastly growing, and data analysis techniques are becoming more and more refined. Database systems provide an excellent base for managing these very large datasets and provide highly tuned implementations to rapidly answer analytical questions. One very typical and well-understood challenge are joins on large amounts of data based on equivalence predicates. However, for many datasets (especially with temporal or sensor data) queries arise that contain joins on range predicates, so-called range joins.

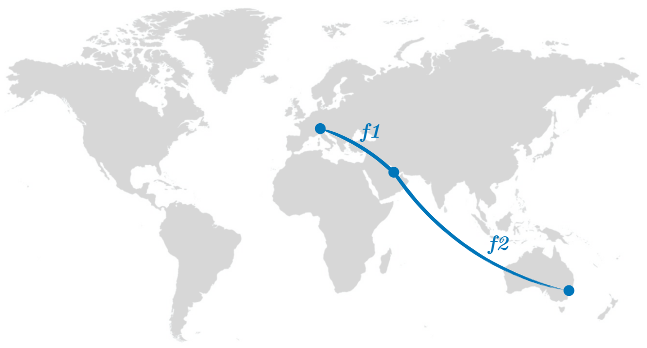


Figure 1: Flight routing with stop-over

A straightforward example is a flight routing search: Given a large database of flight connections, we would like to find affordable flights from Munich to Sydney. Since no direct flights are available, we want to find connections with a stopover, as shown in Figure 1. A major constraint is that we are only interested in connections with a transit duration between 45 minutes and three hours. A query answering this question could look like this:

```
select *,
from flights f1, flights f2
where
f1.orig = 'MUC' and f2.dest = 'SYD' and
f1.dest = f2.orig and
f2.takeoff between
   f1.landing + '45 minutes' and
   f1.landing + '3 hours'
order by f1.price + f2.price limit 10
```

In this case, the join has two join conditions. The equivalence predicate *f1.dest = f2.orig* and the range predicate *f2.takeoff between f1.landing + '45 minutes' and f1.landing + '3 hours'*. Thus, the join could be considered an equi-join with a range-residual or a range-join with an additional equivalence-predicate. Other examples for range joins are: The matching of vehicle sensor data to vehicle rides (defined by a time frame) or the mapping of IP addresses to subnets [37]. Moreover, there are applications, which require the evaluation of multiple range predicates, so-called multi-dimensional range joins. Examples are: Finding return trips in taxi-ride datasets (Section 6.3.3) or combining bird sightings and weather reports [23] based on location and time data. Additional equivalence predicates, as in the flight example, are also very common and should be incorporated into a range join algorithm.

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551849

Most database systems, including DuckDB [30], Postgres [35], Hyper [16], Oracle, and Microsoft SQL Server, execute the flight query using an equi-join implementation. This leads to large execution times since the equivalence predicate is not very selective (e.g. 1 % selectivity for the flight example), and the residual between condition has to be evaluated with $\mathcal{O}(n^2)$ complexity for all flights within an equivalence group. If a dataset is large, the result cannot be computed in an acceptable time frame.

This observation led to the development of different range joins, which handle the non-equality condition efficiently, but which tend to handle the equality part poorly, and which are restricted in the types of queries they support:

Sort based approaches, as implemented by Oracle, Vertica and MonetDB [28, 32, 37, 38], sort the input on the non-equality attribute, and can be used for optimizing range joins on a single dimension. Oracle only supports band joins (fixed size range) but cannot handle additional equivalence predicates. MonetDB's approach supports variable sized ranges, but no additional equality predicats. Vertica's optimization works for variable sized ranges and supports additional equivalence predicates but is susceptible to overlapping ranges. A single range tuple, overlapping all other ranges leads to quadratic runtime. Additionally, all three optimizations (of Vertica, Oracle and MonetDB) are limited to one-dimensional range / band join conditions.

Partitioning based approaches, as implemented by Databricks and others [10, 11, 23, 34], partition the input such that join partners can only occur in (hopefully) few partitions, which improves the runtime significantly. However, they are not suitable for the general case with variable range sizes. Since the size of the range condition can be tuple-dependent, we cannot determine an optimal bin-size automatically. Databricks requires the user to choose a suitable bin size [10], which is problematic.

We tackle these deficits by introducing a kd-tree-based range join algorithm. It supports a wide range of queries, including non-constant bound sizes, supports multi-dimensional ranges, and integrates equality predicates into the lookup process. Through careful engineering, the implementation is fully multi-threaded, during both the build and the probe phase, and offers excellent performance for a wide range of use cases. It is a very versatile general-purpose solution, suited for the integration into a database system that is not special-cased for a particular problem. Nevertheless, we can show in Section 6 that an implementation of our generic approach still outperforms other systems with more specialized algorithms, including some hand-written solutions.

We provide a practical solution for efficient range joins inside an RDBMS, including corner cases like null values, duplicates, strings with collations, outer joins, mark joins for query unnesting [27], additional equivalence predicates, etc. The join algorithm is fully integrated into our database system Umbra [26] and is selected by the query optimizer when appropriate for the query. Our main contributions are:

- A generic kd-tree-based, multidimensional range join algorithm with optimizations for equivalence predicates.
- A parallelization scheme for all aspects of the join with zero single-threaded scans.
- A detailed performance evaluation, leveraging a full implementation of our approach and two related approaches into the database system Umbra, dealing with the corner cases of a real system.

Section 2 gives a short overview of range joins in general. Section 3 presents the kd-tree-based range join algorithm. Section 4 shows how the algorithm is parallelized. Section 5 highlights implementation details for integrating the operator into existing systems and how the query optimizer can choose when to execute a range join. Section 6 evaluates the algorithm and its implementation by systematically showing the performance and how it correlates with differences in cardinalities, selectivities, etc. We also compare the proposed range join to two different algorithms (implemented into Umbra) for an in-system comparison. Afterwards, the performance is compared to commercial and scientific database systems and implementations using artificial benchmarks and real-world examples. Section 7 gives an overview of related work, and Section 8 summarizes the results.

2 BACKGROUND

Range joins are conjunctive join expressions of the form $P \bowtie_{p_1 \wedge \ldots \wedge p_n} R$, where a predicate p_i has the form $P.x_i$ `between` R_{min_i} `and` R_{max_i}. Note that, at least conceptually, an equi-join predicate is a special case of a range query where $R_{min_i} = R_{max_i}$. The goal of this work is to compute the result of such a range join as efficiently as possible.

A band join is a more restricted form of a range join where the range is certain and has a fixed size, for example, $P.x_i$ `between` $R.y_i - 10$ `and` $R.y_i + 11$. We mention this because some other approaches only support band joins and not generic range joins. In the general case of range joins, the range can differ for each tuple. To clarify the explanations in this paper, we call the relation P "points" and the relation R "ranges". A range join outputs a tuple $p \circ r$ for all points $p \in P$ within each range $r \in R$. As in the flight example, a range join can also be a self join, where P and R resemble the same relation (e.g. *flights f1* and *flights f2*).

The approach presented in this paper builds an ad-hoc kd-tree index structure on the "points" relation (build side) and performs lookups for all "ranges" (probe side). Both relations (on the build and probe side) do not necessarily have to be base table relations. Both could be intermediate results of other relational algebra operations. Therefore, the size of both relations is not known a priori and can only be estimated until the algebra trees below are evaluated. The algebra tree of the build side is evaluated first, and all build-side-tuples have to be materialized. The query optimizer will try to choose the smaller relation (if they differ in size) for the build side due to the better worst-case complexity and the lower memory footprint.

Kd-trees [3, 5] are an attractive multidimensional index structure for range joins due to their linear memory consumption and worst-case lookup complexity of $\mathcal{O}(n^{1-\frac{1}{k}} + F)$ [5] where n is the number of tuples stored in the tree, k the number of dimensions and F the output size. Other data structures like range trees [4] would also be possible options, but have prohibitive space requirements if the number of range predicates is high. Kd-trees offer good performance with linear space. The construction of a balanced kd-tree can be achieved in $\mathcal{O}(nlog(n))$ using a linear-time median selection

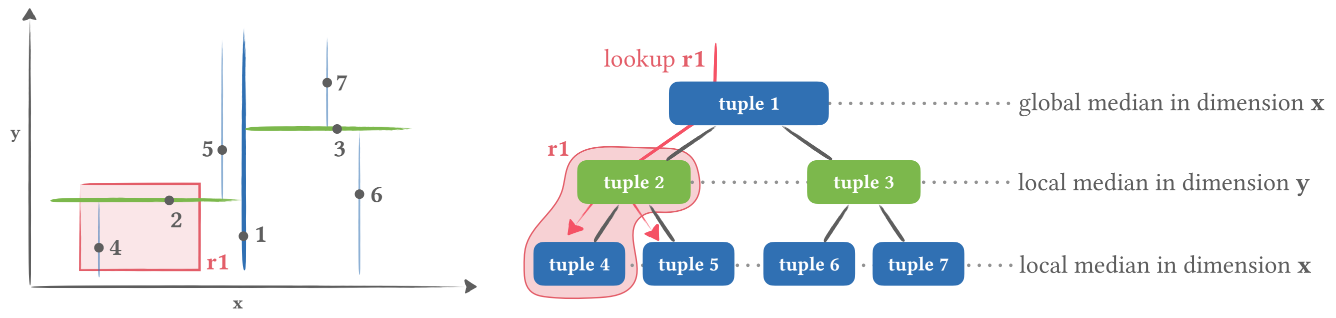

y
x
7
5
3
6
2
1
4
r1
lookup r1
tuple 1
global median in dimension x
r1
tuple 2
tuple 3
local median in dimension y
tuple 4
tuple 5
tuple 6
tuple 7
local median in dimension x

Figure 2: Kd-tree example with two dimension and 7 tuples (points). Dimension 1 is shown in blue, dimension 2 is green. Left: planar projection of the points. Kd-tree nodes are represented by the separation lines. Right: corresponding kd-tree.

algorithm. Bringing this into practice is more involved, though, as there are many challenges like the need to parallelize everything, unknown data distributions, duplicates in the input, and NULL values.

3 KD-TREE RANGE JOIN

Efficient multidimensional range joins require an index structure that allows efficient range queries. Kd-trees, as shown in Figure 2, are a multidimensional data structure to store points and allow efficient range lookups. Using a kd-tree as a join index provides several challenges. The representation in memory should have low overhead, and the tree should be balanced to allow fast range lookups. Additional equivalence predicates cannot efficiently be handled using the kd-tree itself and require special treatment. Therefore, we handle equality predicates using a hashtable. The following subsections explain how these challenges are addressed. At the end of this section, we show the single-threaded version of the algorithm in pseudocode. Details on the parallelization follow in Section 4.

3.1 KD-Tree as an Index for Points

Figure 2 shows how a kd-tree is used as a multidimensional index, for an example with seven tuples representing two-dimensional points. The left side depicts a planar projection of the points; the right side shows the corresponding kd-tree. Each node in the kd-tree represents one tuple and acts as a one-dimensional separator for tuples of its subtrees. All tuples smaller than the node are contained in the left subtree, all tuples larger than the node are contained in the right subtree. In Figure 2, the center, vertical line through tuple 1 represents the root in the kd-tree and splits 2, 4, 5 from 3, 6, 7. The horizontal lines represent kd-tree nodes on the second level and split 5 from 4 and 7 from 6. For k dimensions each node on level l separates its subtrees in dimension $d = l \mod k$. In the 2d case in Figure 2 the root node splits all points in the x dimension, nodes on level 2 split their children in the y dimension, and nodes on level 3 again in the x dimension. To achieve a balanced kd-tree, every node uses the median as a separator. For an even number of tuples we choose the tuple at $\lfloor 0.5 \cdot (start + end + 1) \rfloor$ as the median.

A range query on the kd-tree returns all points within a range (rectangle r1 in the example). The kd-tree reduces the number of intersection checks since subtrees, that do not intersect the range, can be skipped. In Figure 2, only the nodes/tuples on the red lookup path have to be checked to find the result - tuples 2 and 4.

3.1.1 *Memory Layout.*

To improve cache efficiency, the kd-tree is stored compactly in a contiguous array in preorder as shown in Figure 3. Each node only consists of a single pointer to its corresponding tuple. The root node is the first element in the array. Child pointers or separators are not necessary. The storage location of the subtrees can implicitly be calculated. For a tree in memory between *begin* and *end*, the left subtree starts at position *begin* + 1, the right subtree starts at $\lfloor 0.5 \cdot (start + end + 1) \rfloor$. This implicit child addressing is only possible if the tree is perfectly balanced. A tradeoff is whether the tree should only contain pointers to the tuples, compared to storing the tuples' values in-place. Regarding performance, in-place tuple storage would be beneficial for tuples of small size, due to improved caching. For large tuples, the pointer indirection pays off, since tuples have to be swapped frequently during the build-phase. We thus preferred the pointer version, it also simplifies the code due to the fixed entry size. To some degree, cache misses caused by the pointer indirection can be reduced by active prefetch-hinting, as shown in Section 5.2.

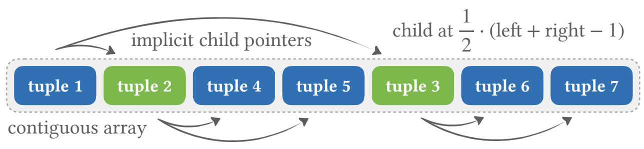


Figure 3: Kd-tree as a contiguous array in memory. Subtrees can be accessed implicitly, pointers are not necessary.

3.1.2 *Optimal Balance.*

There is a tradeoff between building optimally balanced kd-trees and almost balanced kd-trees: While an almost balanced kd-tree would be faster to construct, it has two significant disadvantages. (1) The lookup time increases since an unbalanced tree typically has more levels. And (2), the child node location cannot be calculated implicitly, leading to increased node sizes since the separator (between left and right subtree) would have to be stored. Disadvantage (1) is relevant because typically, the query optimizer tries to choose the larger relation for the probe side, so an improvement in lookup time can justify a slightly more expensive build phase. But disadvantage (2) is even more important to minimize the space consumption. Therefore our approach uses perfectly balanced kd-trees.

3.1.3 *Hash partitions for equivalence predicates.*

Our approach incorporates equality predicates into the algorithm. Considering them simply as zero-width range conditions would be inefficient. If the join condition contains equivalence predicates, tuples are grouped based on the equivalence attributes using a hashtable and one kd-tree is built for each equivalence group. The hashtable itself does not contain the tuples, but it is used as an index to quickly access the relevant kd-tree, as shown in Figure 4. This approach has two advantages: Performance-wise, hashtables have been proven to be the favorable index structure for equi-conditions [21], and the individual kd-trees are smaller and faster to construct.

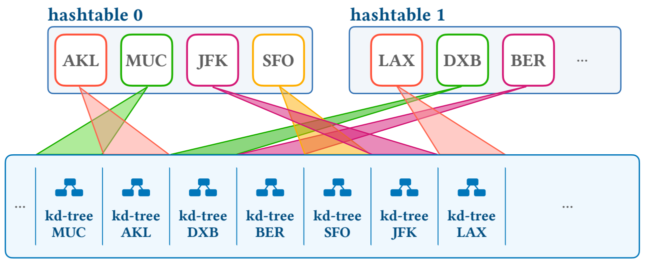


Figure 4: Hashtables for equivalence groups. Example: Flights are partitioned by destination airport. For each destination a (one-dimensional) kd-tree is built.

In contrast to a typical hash join, we expect many duplicate values for the equivalence column(s). If this assumption would not hold, a regular hash-join would be the better option and should be introduced by the query optimizer. Therefore, the requirements of a range join scenario with additional equivalence predicates are similar to the requirements of a typical group-by operator with potentially many tuples per group. Our approach to the equivalence groups is conceptually similar to the preaggregation approach [13] used for the group-by operator of Umbra. The approach involves multiple smaller hashtables to improve parallelization. For this reason, Figure 4 depicts multiple hashtables.

3.2 Lookup

To find all join partners for each range tuple, a hashtable lookup is performed for the equivalence predicate. If a match is found, the entry for the equivalence group contains the start and end pointers (see Figure 4) to the kd-tree for all tuples within the equivalence group. Afterwards, a kd-tree lookup is executed. The kd-tree is traversed by checking if the root element intersects the range. If yes, a match is found, and both child subtrees are traversed. If no, and the range's upper bound is smaller than the root, the left subtree is traversed; otherwise, the right subtree is traversed. Using a manual stack, this is implemented without expensive recursive function calls.

3.3 The Algorithm

The following pseudocode shows how the range join algorithm works conceptually (without parallelization), for a database system adopting the produce / consume model [25]. Conceptually, the join is implemented in three different functions: consumeBuild, buildIndex and consumeProbe, which are typical for join operators adopting the produce consume model. However, their implementation is specific to the proposed range join. Code nested inside "*if equiPredicatePresent:*" is only executed for range joins with equivalence predicates.

3.3.1 ConsumeBuild.

Conceptually, the *consumeBuild* function is called for every tuple on the build side. The tuple is materialized, and if equivalence predicates are present, the tuple count for the equivalence group is incremented. The count is stored in the hashtable ht.

```
def consumeBuild(tuple):
  materialize(this.storage, tuple)
  if equiPredicatePresent:
    if this.ht.contains(tuple.equiColumn):
      this.ht[tuple.equiColumn].count += 1
    else:
      this.ht[tuple.equiColumn] = {count: 1}
```

3.3.2 BuildIndex.

The *buildIndex* function is called after all tuples from the build side have been consumed. The storage array for the kd-tree(s) is allocated. If equivalence predicates exist, a slice of the kd-tree array is assigned to every equivalence group. The slices are then filled with pointers to the tuples of each equivalence group. Afterwards, a kd-tree is built for every group. In the case without equivalence predicates, one single tree is built.

```
def buildIndex():
  this.arrayStart = allocateArray(tupleCount)
  this.arrayEnd = arrayStart + tupleCount
  if equiPredicatePresent:
    ptr = this.arrayStart
    for eqGroup in ht:
      eqGroup.start = eqGroup.end = ptr
      ptr += eqGroup.count
    for tuple in storage:
      ht[tuple.equiColum].end++ = &tuple
    for eqGroup in ht:
      buildKdTree(eqGroup.start, eqGroup.end)
  else:
    ptr = this.arrayStart
    for tuple in storage:
      *(ptr++) = &tuple
    buildKdTree(this.arrayStart, this.arrayEnd)
```

3.3.3 ConsumeProbe.

Conceptually, the *consumeProbe* function is called for every tuple on the probe side. If equivalence predicates exist, and a match can be found in the hashtable, the corresponding kd-tree is selected. Finally, the kd-tree lookup is performed, and every match is passed to the next operator in the pipeline via *consume(match)*.

```
def consumeProbe(tuple):
  treeStart = this.arrayStart
  treeEnd = this.arrayEnd
  if equiPredicatePresent:
    if this.ht.contains(tuple.equiColumn):
      treeStart = this.ht[tuple.equiColumn].start
      treeEnd = this.ht[tuple.equiColumn].end
    else:
      return
  for match in this.kdLookup(treeStart, treeEnd):
    this.parent.consume(match)
```

4 PARALLELIZATION MODEL

One key aspect of the kd-tree range join is that it can be highly parallelized. The parallelization of the probe side is trivial since the read-only lookups can be performed concurrently. However, the build side provides some challenges, and single-threaded work has to be limited to an absolute minimum. Our solution does not require any single-threaded scans on the whole input.

This section explains the parallelization model in multiple steps. First, we show how we parallelize the build phase of the hashtable efficiently. Second, we describe the parallel construction of the kd-tree(s). Third, if a single or very few root nodes exist, the median selection is parallelized. Step three is only employed if no equivalence predicate or very few equivalence groups exist.

4.1 Parallel Hashtable Construction with Thread-Local Preaggregation:

As mentioned before, we expect the range join to contain only few equivalence groups and many tuples per equivalence group. Therefore, separating tuples into groups provides similar particularities as a group-by operator. In such a group-by-case, a thread-local preaggregation [22] can provide significant performance gains compared to a single global hashtable [13]. Our approach uses this concept, but implements some aspects differently.

First, each tuple on the build side is materialized into chunked memory blocks. A hash value is calculated for all equivalence-key-values. To obtain the tuple count per equivalence group locally and globally, a count aggregate is computed per equivalence group using a (conceptually single) local hashtable. To allow parallel merging of the thread-local hashtables, the aggregation is split into multiple hash tables using the hash prefix. Tables with different hash prefixes can then be merged in parallel, as shown in Figure 5. The number of hashtables should be a power of two (to simplify the prefix calculation) and should exceed the number of CPU cores to improve the load distribution. For our implementaion, we use the upper 9 bits of the hash prefix to select one of 512 hashtables which allows full parallelization for most systems. (Too many hashtables would cause more memory overhead for small problem instances with only a few tuples.) Each hashtable is implemented as a dynamically growing robin hood hashtable for data locality. A local hashtable entry stores the equivalence key and the current local tuple count for the corresponding equivalence group. If an equivalence key occurs for the first time, a new hashtable entry is inserted. If a key reoccurs, the tuple counter of the hashtable entry is incremented.

After all tuples are materialized, the thread-local hashtables are merged into 512 global, growing robin hood hashtables. As stated before, this can be executed multi-threaded without locking. After merging, the global hashtables contain the global tuple count for all equivalence groups, while the local hashtables contain the

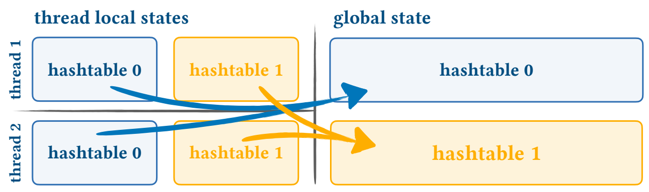


Figure 5: Parallel merge of thread local hashtables.

thread-local tuple count for each thread state. The main difference compared to the preaggregation [22] for a group-by is the usage of dynamically growing hashtables instead of fixed-size hashtables that would evict entries into output streams in the case of hash collisions. This behavior is essential to retain the local tuple count.

The following steps are specific to the proposed range-join with equality predicates: After the hashtable merge, a contiguous array is allocated - large enough to store tuple pointers for all tuples, and each equivalence group is assigned to a slice of that array as shown in Figure 4. Each thread then reiterates its tuples, reserves a sub memory slice in the tuple-pointer array for each equivalence group, and stores pointers to its tuples in the reserved area. Only the memory slice assignment in the tuple pointer array must be synchronized, which can be achieved by a single atomic value per equivalence group. As a result, the beginning of each slice is offset by the prefix-sum of the preceding equivalence groups' sizes, and the tuple pointers can be written without further synchronization.

Finally, the 512 global hashtables contain entries to all equivalence groups, and each entry points to a slice of the tuple pointer array. The next step is to construct a kd-tree for every slice of the tuple pointer array.

4.2 Parallel KD-Tree Construction:

The parallelization of the kd-tree construction requires special care to avoid single-threaded scans on the whole input. Therefore, we employ a three-stage parallelization scheme. It consists of intra-node, inter-node, and subtree parallelization and differs slightly for the two cases, with and without equality predicates.

If equivalence predicates exist, a kd-tree is built for every equivalence group. If multiple groups are available, individual kd-trees can be built in parallel. At first, single nodes are scheduled to different threads (inter-node parallelization). Eventually, after a certain number of subtrees are available, each thread continues with the single-threaded construction of whole subtrees (subtree parallelization). For actual implementations, this threshold should be larger than the core count to improve the work distribution among threads if, for example, one equivalence group contains fewer tuples than others. However, the threshold should not be too large to avoid scheduling overhead. As rough guidance: Our implementation sets this threshold to 1000 - but the algorithm is not sensitive to that exact number.

If no equivalence predicates exist, the build phase of one large kd-tree for the whole build side has to be parallelized (see Figure 6). Since the kd-tree is constructed top-down, subtrees can be efficiently parallelized on lower levels, but the median selection for the root node would be single-threaded on the whole input.

Building a node in the kd-tree consists of two steps. (1) Choosing the median in the dimension of the current tree level and (2) partitioning all tuples below by that median. Single nodes construction can be parallelized by executing a parallel median selection and partitioning (intra-node parallelization). After building the first few tree levels using parallel median selection, the algorithm switches to inter-node parallelization (as described above) as soon as enough subtrees exist to utilize a significant portion of the CPU. The inter-node parallelization scales much better with the number of CPU cores than the parallel median selection and partitioning. We use

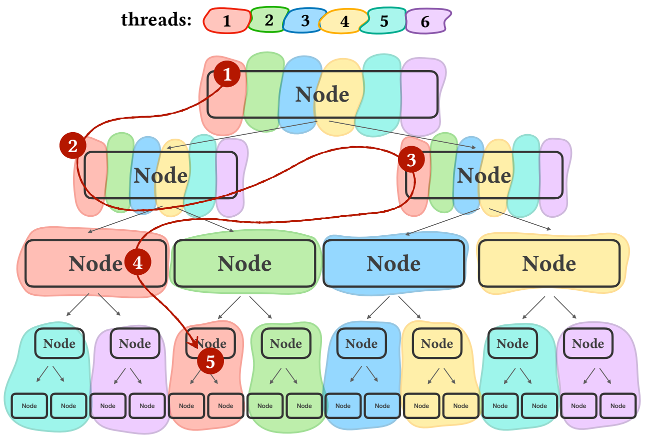


Figure 6: Parallel kd-tree building for a single tree adopting intra-node, inter-node, and subtree parallelization

an 8 node limit for our experiments - we switch to the inter-node parallelization as soon as eight nodes can be built simultaneously. However, this setting can be CPU specific and should automatically be measured during the DBMS installation process.

Parallel kd-tree construction is also essential to ray-tracing in computer graphics. The idea of changing the parallelization scheme for lower tree levels is widely spread [8, 33, 41]. For the parallelization of individual nodes, the algorithms differ. As stated, we chose to implement a parallel median selection and parallel partitioning.

Figure 6 shows all three stages of parallelization: The upper nodes are built using intra-node parallelism ① - ③, then we switch to inter-node parallelization ④ and finally, we employ subtree parallelization ⑤. Each colorful bubble in Figure 6 represents a set of tuples (morsel) that have to be processed. These morsels are assigned to threads for processing using our morsel-driven scheduler. The described parallelization model is independent of morsel-driven parallelism. Other parallelization techniques (e.g., spawning threads) are also possible, although they might increase overhead. The red arrow in Figure 6 shows the order in which morsels are scheduled to thread 1.

4.3 Median Selection

The median selection is the most expensive operation during the kd-tree construction. Therefore, it is appealing to implement a fast selection algorithm with linear complexity. Although the median selection can be implemented with guaranteed linear-time worst-case complexity, e.g. using the Median-of-Medians algorithm [9], it is considerably slower for practical applications than e.g. Quickselect for random or sorted inputs, as shown in the following example with 10 million integers using a single thread (g++, -O3, AMD RYZEN 9-5950X):

Algorithm (Median of 10 M Integers)	Random	Sorted
Median of Medians $\mathcal{O}(n)$	465.89 ms	153.197 ms
Quickselect $\mathcal{O}(n^2)$	109.02 ms	13.34 ms

Quickselect has an average-case complexity of $\mathcal{O}(n)$ [9]. But *average* depends on the data. Duplicates, for example, frequently occur in databases and could lead (depending on the partitioning scheme) to the worst-case complexity of $\mathcal{O}(n^2)$. To avoid quadratic runtimes in case of duplicates, we execute a three-way partitioning scheme that splits the input into three partitions: smaller than the pivot, equal to the pivot, and larger than the pivot.

This way, the median selection does not lead to $\mathcal{O}(n^2)$ complexity in the presence of duplicates. However, to guarantee the $\mathcal{O}(n)$ worst-case performance, we implemented Introselect [24]. Introselect starts with a duplicate aware Quickselect phase and falls back to the worst-case optimal Median-of-Medians algorithm if the linear-time constraint cannot be met using Quickselect.

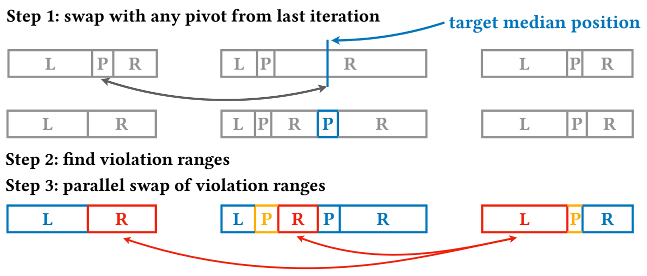
One favorable side effect of applying the Introselect algorithm is that all tuples are partitioned (by the median) after execution. This is essential for the kd-tree construction.

4.4 Parallel Median Selection

For the parallel creation of single kd-tree nodes, the median selection and partitioning have to be parallelized. The parallel median selection itself is a well-known problem [14, 15]. The partitioning side effect, however, is often sacrificed in parallel implementations.

We implemented the following approach for the parallel median selection: First, the whole tuple array is split into multiple morsels, each representing a slice of the tuple array. For three morsels, the median-of-three is evaluated, and a global pivot is chosen as the median of these three medians. Then the morsels are partitioned by the global pivot using a three-way partitioning scheme. Each morsel now has three partitions: Smaller than the pivot (left), equal to the pivot (middle), and larger than the pivot (right). Afterwards, the global count of all elements left and right of the pivot is calculated, and a new global pivot is chosen for the next iteration. If the median is found, all morsels are locally partitioned. However, globally, the data is mixed (as shown in Figure 7). In order to achieve global partitioning, we perform a parallel merge phase that exchanges local ranges of tuples until all data is partitioned globally.

Figure 7 shows how this parallel merge phase can be implemented for an example with three morsels: First of all, one of the pivot tuples is swapped to the global median position. For each morsel, we identify three sub-slices - left of the pivot (L), pivot (P), and right of the pivot (R). Afterwards, all slices violating the global partitioning are identified and appended to a slice-swap-list. Special care is required if one of the slices contains the global median element. Afterwards, the violating sub-slices can be exchanged in parallel by scheduling sub-slices with a similar number of tuples to worker threads. Now the tuples can be swapped concurrently without further synchronization. The overhead for these operations is amortized by choosing reasonably large morsels.


Step 1: swap with any pivot from last iteration
target median position
L
P
R
Step 2: find violation ranges
Step 3: parallel swap of violation ranges
Figure 7: Parallel merge of partitions

5 IMPLEMENTATION IN A COMPILING DATABASE SYSTEM

We integrated the proposed range join algorithm into our compiling database system Umbra [26]. Such an implementation has to deal with corner cases, which (in our opinion) can be very insightful since many theoretically interesting algorithms can lose their benefits once implemented in a real system. This section focuses on the integration into the query optimizer, the code generation and active prefetching to reduce cache misses.

5.1 Query Optimizer

When does it make sense for the query optimizer to introduce a range join operator? Whenever Umbra receives a SQL query, the query is translated to relational algebra. Our query optimizer applies optimizations on the relational algebra level in multiple passes. During the last "physical" pass, the query optimizer chooses how each join is implemented. The optimizer introduces the kd-tree range join if range conditions are present. Finally, Umbra compiles the optimized query plan to machine code and executes it.

Since the compile time of a range join is considerably higher than a hash or blockwise-nested-loop join, the query optimizer only introduces range joins if the upfront compilation cost can be amortized later by a faster execution. We use cost estimations: The build side P is estimated with $c_0 \cdot |P| \cdot (1 + log_2(|P| \cdot sel_{eq})) + c_1$ and the probe side R with $(c_2 \cdot |R| \cdot (1 + log_2(|P| \cdot sel_{eq})) + c_3)$. The constants c_i are system dependent and should be masured during the installation process. sel_{eq} denotes the selectivity estimate for equality predicates. If the cost estimation remains below the cost estimation for the hash join, umbra will introduce the kd-range-join operator. Figure 8 depicts the optimizer behaviour for an example with an equality predicate selectivity of 0.1 and 0.01.

The canonical approach to a join with a predicate "P.x between R.min and R.max" uses P as the relation for the build side and R on the probe side. If pure band join conditions (constant range size) occur, the between expression can be inverted. This is very appealing since we prefer to build the index structure for the smaller relation. Imagine a join of P1 and P2 on "P1.x between P2.x - 3 and P2.x + 5". If the cardinality of P1 is larger than the cardinality of P2, it would be suboptimal to canonically build the kd-tree for relation P1. In such cases the optimizer inverts the between condition to P2.x between P1.x - 5 and P1.x + 3. We currently support the addition and subtraction of constants since the inversion of such expressions can be achieved in linear time without recursion. A more sophisticated

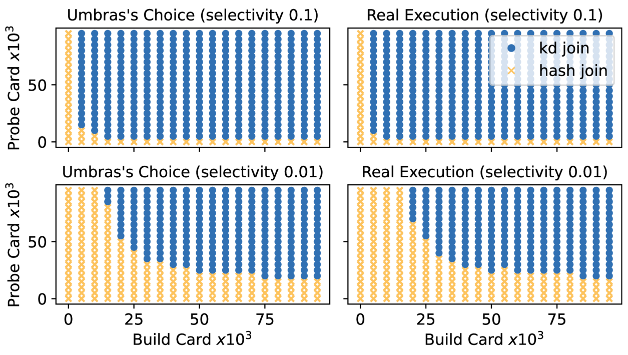


Figure 8: Query optimizer, hash join vs. kd-tree range join.

approach could be considered for large datasets since the total query runtime is dominated by the execution time.

In the case of multiple between conditions, we analyze all between conditions and check if they could be inverted. Since range conditions cannot be inverted on a syntactic level, their build side is fixed. For both variants - P1 as build side or P2 as build side, we count the number of applicable between conditions and choose the variant with more between conditions. We use the smaller relation if both sides are equally suited as the build side. Finally, we invert all between conditions, to match the chosen build side (if possible).

5.2 Code Generation

Whenever the user runs a query, Umbra translates the query into executable machine code and runs the code to evaluate the query result. Umbra adopts the produce-consume model and the pipelining concept for data-centric code generation [25]. To reduce query compile time, we only generate query-specific code and use precompiled (C++) *runtime functions* for common, generic tasks. Runtime functions can be called from generated code and vice versa.

Therefore, most of the logic for the tree construction is implemented in C++ and called from generated code. However, since the tree building is data-dependent and the data types are unknown at system compile time, at least the compare function has to be generated for a given query. This leads to a problem: The compare function is called many times per tuple and introduces a significant overhead for the function call. To avoid the frequent function calls, we decided to generate the whole partition function for the median selection at query compile time. Our implementation uses a branch-free, three-way version of the Lomuto [9] partitioning scheme. The same argument holds for the kd-tree lookup. The tree traversal for a single range involves multiple comparisons. Therefore the whole tree lookup code is generated for every query.

5.3 Prefetching

As described in Section 3, the kd-tree stores tuple pointers. This indirection causes cache misses during execution. This is especially unsatisfactory during the Lomuto partitioning phase, where many accesses occur in a known order. In the partitioning loop, the tuples are accessed sequentially. Iterating over the tuple pointers introduces a double indirection, and the CPU cannot automatically prefetch the tuples into the cache. This short program in C++ (iterating over all tuples) illustrates the issue:

```
Tuple pivot;
for(Tuple** iter = begin; iter < end; ++iter) {
  Tuple* tuplePtr = *iter;
  // Load tuple pointer (hurtful cache miss here)
  int comparisonResult = compare(pivot, *tuple);
}
```

Since the tuples are spread in memory, this can be an issue, especially on machines with multiple NUMA nodes [20]. These cache misses dominate the execution performance of our range join implementation. The effect can be reduced by manually introducing prefetch-hint instructions. Using this small optimization can reduce cache misses on the build side, leading up to 13 % speedups compared to the non-prefetch implementation, as shown in the evaluation.

6 EVALUATION

The evaluation section is split into three subsections. Subsection 6.1 presents artificial benchmarks to demonstrate the broad spectrum where the kd-tree join can be applied beneficially. We discuss internal tradeoffs, advantages, and limitations of the kd-range-join. Subsection 6.2 compares different algorithms within Umbra. Subsection 6.3 compares the performance to other systems using the same benchmarks and two real-world examples.

All experiments are conducted on an AMD RYZEN 9-5950X 16 core CPU (32 execution contexts) with 64 GB DDR4-3200 memory and an M.2 Samsung 980 1TB SSD, using Ubuntu 21.10.

6.1 Artificial Benchmarks and Tradeoffs

As a running example, we choose an artificial benchmark with a single range join on two tables - named points and ranges. The base example has k dimensions and an additional equivalence predicate. We use the following schema with non-nullable integers:

$\texttt{points}(x_0 \ldots x_{k-1},\ x_{eq})$
$\texttt{ranges}(r_0\texttt{min} \ldots r_{k-1}\texttt{min},\ r_0\texttt{max} \ldots r_{k-1}\texttt{max},\ r_{eq})$

Tuples for the *points* table are generated randomly using a uniform distribution. The value range for each dimension is $[0, \sqrt[k]{n}[$. So the points are randomly distributed on a k-dimensional grid. Duplicates occur - as they could in a real-world application. Tuples of the range table have a lower (min) and upper (max) bound for each dimension. The range width is used to adjust the join selectivity. The lower bound of each range is generated randomly using a uniform distribution. All generated ranges have the same length; however, our implementation does not use this fact for optimizations.
We use the following query to test multidimensional range joins:

```
select count(*) from points, ranges
where x_eq = r_eq and
x0 between r0_min and r0_max and ...
x<k-1> between r<k-1>_min and r<k-1>_max;
```

The base case for all measurements (unless stated otherwise) uses the following parameters: 1 million points; 1 million ranges; range size (max - min) = 1 (two points per range); 10 % selectivity for the equality predicate; k = 2 dimensions.

In the following experiments, we change these settings individually to demonstrate their impact on the runtime performance. For each configuration, the query is repeatedly executed 10 times with 3 warmup runs. We report the median of the total runtime (compilation + execution).

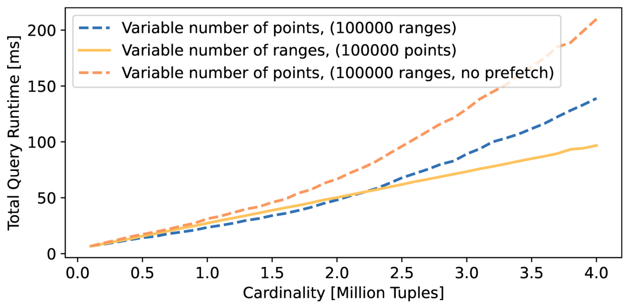


Figure 9: Impact of cardinality

6.1.1 Impact of cardinality. To observe the impact of cardinality, we adjusted the cardinalities for the points and ranges relation separately. Figure 9 shows the runtime in milliseconds for the benchmark query on the y axis. The yellow line depicts the runtime, depending on the size of the ranges relation. The number of points remains fixed at 100,000 while the number of ranges varies between 100,000 and 4 million. The execution time grows linearly with the number of range tuples. This is expected as the number of lookups (on static index structures) is simply increased.

Increasing the cardinality of the points relation leads to larger kd-trees that have to be constructed, increasing the runtime spent during both build and probe phase. The blue graph shows superlinear growth due to the superlinear complexity of the kd-tree construction and lookup. This is expected and emphasizes that it is beneficial to invert band-join conditions in the optimization phase (Section 5.1) to build the kd-tree for the smaller relation. However, before the superlinear complexity dominates, adding tuples to the build side is faster due to the active prefetching optimization. The orange plot shows the same test, while prefetching is turned off.

6.1.2 Impact of selectivity. To measure how the join behaves depending on the selectivity of the range condition, we measured the query execution times for different range sizes. For the running example with two dimensions, we vary the range size between 0 and 20 in both dimensions. A range size of 0 in both dimensions will statistically intersect 1 point (simulated equi join), a range size of 1 intersects 4 tuples, etc., up to a range size of 19 intersecting 400 points. Figure 10 shows the result for our measurements. The x-axis represents the average number of points intersected by each two-dimensional range. Expectedly, the runtime increases as the output size of the join grows quadratically with the range size. The extreme case, with all ranges intersecting all points, is the cross-product after all. However, the performance degradation adapts gradually. This means, that the range join is also suitable for a coarser range join condition, with additional (more selective) residual join predicates. A real-world example for this scenario is presented in Section 6.3.3.

6.1.3 Impact of equivalence predicates. One major goal was to incorporate equality predicates into the range join, since they typically occur in addition to range predicates. Therefore, we measured the query execution times with varying selectivities of the equivalence predicate, as shown in Figure 11. For a selectivity of 1 (all tuples have the same value for the equivalence column), the equivalence condition is always true and only a single kd-tree is built.

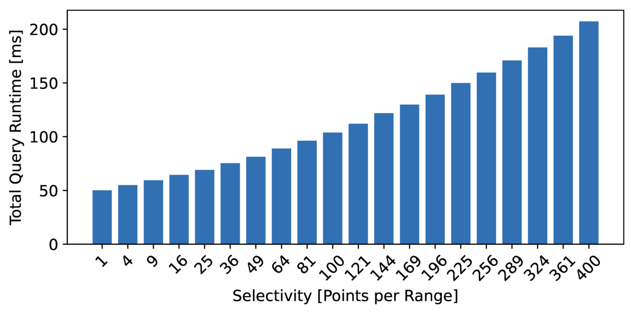


Figure 10: Impact of selectivity

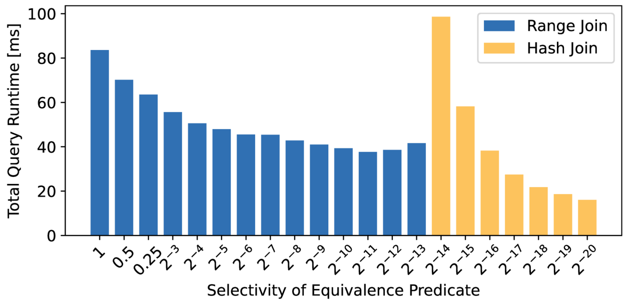


Figure 11: Impact of equivalence predicates

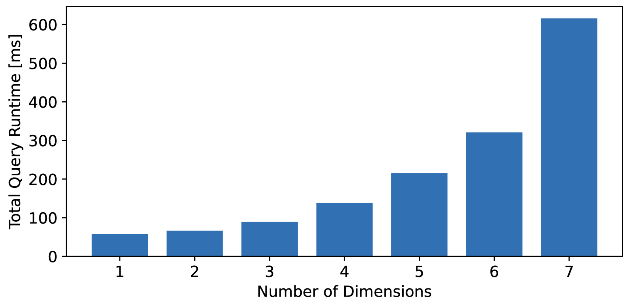


Figure 13: Impact of dimensions

A selectivity of 0.5 means that each tuple belongs to one of 2 equivalence groups. A selectivity of 2^{-20} represents approximately a 1:1 relation. Figure 11 shows that, for the range join, the runtime decreases with decreasing selectivity values. Without the equality predicate optimization of Section 3.1.3, the runtime would remain almost identical to the selectivity = 1 case.

For very low selectivities (e.g. 2^{-20}), most join candidates are eliminated by the equality condition. At this point, the regular hash join (treating the range condition as a filter) becomes the faster option, and will be chosen by the query optimizer. In Figure 11 the optimizer chooses the hash join for all selectivities $\leq 10^{-14}$. The outlier with selectivity $\leq 10^{-14}$ is caused by an imperfect cost estimation by the optimizer - a hash join is used, although a range join would still be the faster option. However, this also depends on the selectivity of the range conditions, which are hard to estimate and thus not incorporated into the cost estimation (Section 5.1).

6.1.4 Impact of parallelization. As Section 4 shows, the range join algorithm is fully parallelizable for both the build side and the probe side of the range join. Figure 12 shows the performance gain for the example benchmark depending on the number of threads. In this case, we change the running example to a single equivalence group (eq-selectivity = 1), which is harder to parallelize. This way, the parallel hash-table construction and all three stages of kd-tree parallelism are employed. This is the most difficult case.

The blue line in Figure 12 shows the parallel speedup using the running example with 1M tuples. The performance drop for > 6 threads is caused by Umbra's adaptive query compiler. For long-running queries, the adaptive query compiler compiles the query initially using our flying-start backend [17] and switches to an optimized LLVM version during execution. The compilation of an optimized version is only triggered if the runtime benefit is large enough to justify the additional compilation overhead. For more than six threads, the execution using the flying start backend is so fast that the optimized recompilation is not triggered in the example. However, the resulting runtime is still faster in all cases compared to only using the LLVM compiler due to LLVM's increased compile time. The yellow plot shows the speedup for a more extensive example with 10 million tuples for both relations. In this case, the adaptive compiler always switches to the LLVM version during execution, and the speedup rises to 15x.

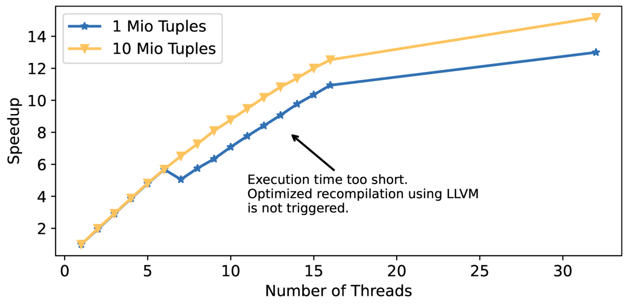


Figure 12: Impact of parallelization

The whole kd join execution time is dominated by cache misses. For the 10 M tuple example, 54 % of the execution time is caused by two instructions. Both are load instructions for tuples in the kd-tree. These cache misses occurs due to the pointer indirection. Hyper-threading typically reduces the impact of cache misses. This is also the case in our example. Figure 12 shows another 21 % speedup by using hyper-threading (32 threads).

6.1.5 Impact of dimensions. As we support the general case of k-dimensional range joins, we also tested the performance impact related to the number of dimensions. Due to the curse of dimensionality, we expect a significant reduction in performance for high-dimensional range predicates. Figure 13 depicts the impact of an increase in dimensionality for our running example. The number of points remains constant for the experiment - they are uniformly distributed on an n-dimensional grid. The range size is set to zero. Therefore, every tuple of the range relation statistically has one join partner. This change keeps the output size of the join constant. The lookup speeds decrease due to the $\mathcal{O}(n^{1-\frac{1}{k}})$ lookup complexity. Currently, the optimizer uses as many between predicates as possible for the kd-join. Despite the increasing lookup complexity, this is still favorable if the between predicates are statistically independent and reduce the result size. If between predicates are statistically dependent or very unselective, utilizing more range conditions can be harmful. Nevertheless, queries with 6 or more-dimensional range conditions are probably quite rare.

6.1.6 Impact of prefetching. Section 4 highlights the advantage of explicit prefetching to reduce the number of cache misses during the Lomuto partitioning phase of our quickselect implementation. To measure its impact, we compared the speedup for varying prefetch distances to a version without prefetching:

Distance	1	2	3	7	5	6	7	8
Speedup	1%	8%	12%	12%	13%	13%	13%	13%

The prefetch distance is measured in tuples. For example, a prefetch distance of 5 means that the fifth next tuple is prefetched in the Lomuto partitioning loop. We measure a speedup of ca. 13 % compared to the non-prefetch version. In the example, the exact prefetch distance is not extremely relevant. A default prefetch distance of 7 tuples has improved the performance in all datasets we tested. Any prefetching distance greater than three seems to produce good results.

Bottom line. The previous sensitivity analysis shows that the proposed range join works for a large variety of applications. The join offers very good performance for multiple dimensions, a wide range of selectivities, and scales very well on a multi-core system. For almost all cases, the execution time is well in the sub-second range for two relations with a million tuples and robust to a broad range of parameters.

6.2 Comparison to Related Approaches Within Umbra

This section compares the proposed kd-tree range join to two other (parallel) implementations of range joins within Umbra. The first implementation is a sort-probe-based range join which sorts the points relation during the build phase and performs a binary search for each range to find matching tuples (similar to MonetDB). Secondly, we implemented the approach of Vertica, which builds an index on the ranges relation [37]. Thus, it is an optimization for a "few ranges, many points" case. The ranges are sorted based on their lower bounds and a running maximum is calculated for the upper bounds to speed up the lookup process [37]. In the case of multiple between conditions, both algorithms select one range condition for the range join evaluation and treat other conditions as filters. Figure 14 compares the runtime of the three implementations for five different scenarios: ① 1 dimension, no overlap; ② 2 dimensions, no overlap; ③ 1 dimension, overlap; ④ 2 dimensions, overlap; ⑤ 4 dimensions, overlap. Each experiment was repeated for three different cardinality configurations: ⓐ 1 M points, 10 k ranges; ⓑ 10 k points, 1 M ranges; ⓒ 1 M points, 1 M ranges.

The experiments only focus on the range predicate evaluation and do not include equality predicates (as the equality predicate optimization could be applied to all three algorithms). For the overlap

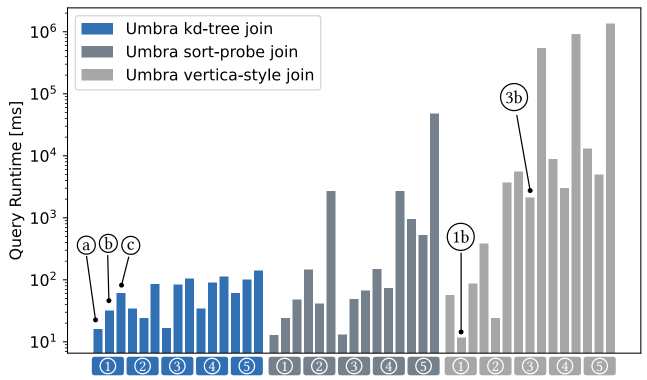


Figure 14: In-system comparison. Different range joins algorithms in Umbra. Only the kd-tree join can handle all testcases efficiently.

experiments, we added a single range to the dataset, that contains all points. The plot in Figure 14 depicts the total query runtime for all experiments and cardinality configurations from left to right. We can observe multiple properties of the different algorithms:

- In general, the performance of the kd-tree-based range join only varies within one order of magnitude for all experiments. So the DBMS user can generally expect (even for 4d queries with overlapping data) a result in a reasonable time frame.
- In the simplest of all cases ① (1d, no overlap), the kd-tree range join performs similarly to both approaches with a slight advantage for the sort-probe algorithm. Thus, the performance benefit of more specialized implementations is rather low.
- Only for the "1d, no-overlap, many points, few ranges" case (1b), Vertica's approach has a distinct advantage compared to the other two approaches. However, as soon as a single overlap range occurs e.g., in (3b), the runtime increases by two orders of magnitude.
- Overlap cases cannot efficiently be handled by Vertica's approach. If overlaps occurs ③-⑤, the Vertica algorithm performs worse by multiple orders of magnitude.
- For multiple dimensions ②, ④, ⑤, both Vertica-style and sort-probe perform significantly worse because both algorithms only optimize for a single between predicate.

We conclude that it is beneficial for a database system to adopt a kd-tree-based range join algorithm that allows fast execution of a wide variety of queries. Narrower approaches do not offer significant gains while they suffer from problematic performance deficits for certain queries.

6.3 Comparison to Other Systems

This subsection compares the kd-tree range join implementation in Umbra to other systems. First, we compare Umbra to other available databases with and without range join optimizations using our artificial benchmark and a flight routing example. Afterwards, we use a taxi-ride example to compare Umbra's execution time to a partition-based implementation using Spark.

All tests were conducted on the same machine, and all database systems are configured using their default settings. The thread count for parallel execution was set to 32 (if possible).

6.3.1 Artificial Benchmark. To evaluate how Umbra, leveraging our range join implementation, compares to other database systems, we executed the benchmark from the previous subsection on commercial and research systems. We tested Tableau HyPer v.0.0.14751 (python package, default settings), DuckDB v.0.3.4 (python package, configuration: "threads": 32), MonetDB v.11.31.7 (docker, configuration: "nthreads": 32), DBMS-X and Postgres v.14.2 (docker, "max_parallel_workers_per_gather": 32). No table-indexes were used. Except Umbra, DBMS-X and MonetDB have built-in range join optimizations. The other systems do not. In the case of MonetDB, the range join optimization is not triggered due to the additional equality predicates - an evaluation that compares a sort-probe algorithm (as implemented by MonetDB) can be found in Section 6.2. Since systems without active range join optimizations will perform asymptotically worse, arbitrary large speedups could be

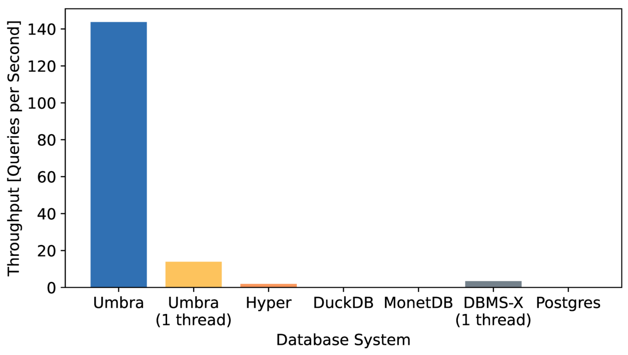


Figure 15: System comparison - artificial benchmark

demonstrated by increasing the cardinalities. However, the performance differences are very significant even for a smaller dataset with 100,000 tuples per relation. Figure 15 reports the throughput in queries per second for all database systems. All results have been checked for correctness.

Umbra's throughput exceeds all other database systems by a factor of 30. The positive effect of the one-dimensional range join optimization implemented by DBMS-X can be clearly seen in this comparison. Since DBMS-X only uses a single thread for the range join evaluation, we also included the single-threaded runtime measurement for Umbra for a fair comparison. Nevertheless, Umbra is still faster than the other systems using a single thread.

6.3.2 Flight Routing. In this experiment, we compare the performance of the different systems for a flight routing search similar to the introduction. Given such a task, we evaluate how the different systems compare. We collected a flight dataset, containing 127426 scheduled flights for four months of an European airline. As an example, we would like to count all stopover flights using the following query:

```
select count(*)
from flights f1, flights f2
where f1.orig != f2.orig and f1.dest = f2.orig
   and f2.takeoff between
       f1.landing + interval '45 minutes' and
       f1.landing + interval '3 hours';
```

Figure 16 shows the throughput for the flight example on a logarithmic scale. The flight routing search involves a one-dimensional

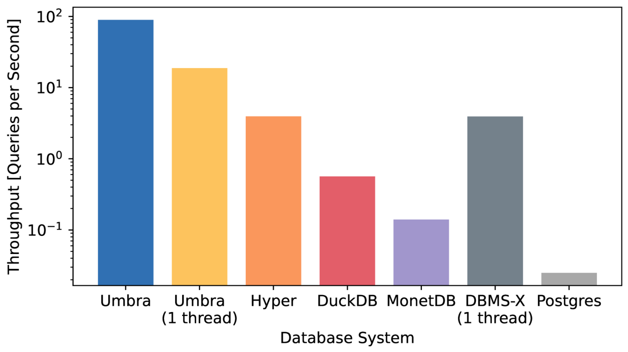


Figure 16: System comparison - flights

band join with an additional equivalence condition. In this scenario, DBMS-X's range join optimization for hash joins fully applies and pays off considering its single-threaded execution. But still, the largest throughput - by a factor of more than 10 - is achieved using the kd-tree range join implementation in Umbra.

6.3.3 New York Taxi Rides. Now we compare the proposed range join implementation to a hand-written program using Apache Spark (version 3.2.0, scala version 2.12.15) [40]. This benchmark uses the well-known New York taxi rides dataset [36]. The main idea is to find return trips of taxi rides. A return trip consists of two rides with close pickup and dropoff coordinates, e.g., less than 90 m radius (r) and a time difference between 0 and 8 hours. A location is considered close if it intersects the circle around another location. Since the great-circle distance between two locations is expensive to compute, we can easily calculate a bounding rectangle and use it as a range join predicate. The bounding rectangle acts as a coarse join condition and the result is refined using additional filter predicates. The SQL query to find return trips for $r = 90m$ looks like this:

```
select count(*) from rides r0, rides r1
where
-- 5d range join
r0.plat between
   r1.dlat - latOffset(90) and
   r1.dlat + latOffset(90) and
r0.plon between
   r1.dlon - lonOffset(90, r1.dlat) and
   r1.dlon + lonOffset(90, r1.dlat) and
r0.dlat between
   r1.plat - latOffset(90) and
   r1.plat + latOffset(90) and
r0.dlon between
   r1.plon - lonOffset(90, r1.plat) and
   r1.plon + lonOffset(90, r1.plat) and
r1.pt between r0.dt and r0.dt + '8 hours' and

-- Filter result for matches in circle
r0.dt < r1.pt and r0.dt + '8 hours' > r1.pt and
dist(r0.plat, r0.plon, r1.dlat, r1.dlon) < 90 and
dist(r1.plat, r1.plon, r0.dlat, r0.dlon) < 90;
```

The query can be evaluated using a five-dimensional range join. *plat* is an abbreviation for *pickup latitude*, *dlat* for *dropoff latitude*, *pt* for *pickup time*, etc.. *latOffset*, *lonOffset* and *distance* are scalar user-defined functions using pure SQL. Both *offset* functions are used to compute the bounding box. Strictly speaking, the query can only be translated to a range join and not to a band join, since the longitude offset depends on the latitude.

We compare the Umbra execution of the SQL query to a Spark implementation for this return-trip problem. The Spark program implements a partition-based band join. The reduction to a band join is possible by assuming all locations to have a latitude below 41.16°. Therefore, the bounding rectangle size is fixed and not data-dependent. Every coordinate value is assigned to a bucket (id) with a size of $2 \cdot$ *band width*. For latitudes, the band with is *latOffset(90 m)*. For longitudes, we assume a band width of *lonOffset(90 m, 41.16°)*. Afterwards, an equi-join is performed on the bucket ids. For every dimension, two buckets have to be checked since tuples from the build and the probe side could be close, but still end up in different buckets. We measured the following runtimes for "Yellow Taxi Trip Records, January 2016":

Radius	Number of Return Trips	Umbra	Spark
90 m	4470311	3.64 s	8.63 s
150 m	19297047	6.74 s	10.48 s

The Spark code is available in our reproducibility package. It is optimized to this specific example and difficult to write, compared to the SQL query. Nevertheless, Umbra's execution is faster and does not require manual tuning to optimize the join algorithm.

7 RELATED WORK

Since band- and range joins do occur in real-life analytical workloads, several optimizations and algorithms have been proposed and implemented to avoid quadratic evaluation in such cases. We split the related work into three groups: Partition-based approaches, sort-based approaches, and other related approaches.

7.1 Partition Based Approaches

DeWitt et al. published a partitioned band join algorithm in 1991 [11] and compared it to optimizations for sort-merge band joins as they could be found in systems back then. The proposed partitioning band join algorithm assumes low main memory sizes. Partitioning remains the core concept of several solutions for implementing band joins as in [23, 34] and can be very beneficial in the case of band-joins where the range size is fixed. For range joins with variable range sizes, it can be impossible to determine a suitable bin size. A commercial system implementing partition-based range joins is Databricks [10]. Choosing a suitable bin size is delegated to the user by requiring query hints. Our approach does not require manual tuning and supports multiple range conditions.

Work by Li et. al. [23] proposes recursive partitioning of data in the case of multidimensional, distributed band joins. The concept presented there is orthogonal to the local range join implementation on the distributed nodes and could therefore be combined with our approach in a distributed scenario.

7.2 Sort Based Approaches

On the other hand, Oracle and Vertica adopt sort-based approaches. Oracle offers a band-join optimization for their built-in sort-merge join [28] which was patented in 2018 [32]. Oracle's optimization is limited to one-dimensional band joins. Additional equivalence predicates seem to force a hash join, even if the equivalence predicate is very unselective. Our approach covers many more circumstances, like multiple dimensions, true range joins, and additional equivalence predicates.

MonetDB implements a sort-probe based approach [38] which is very similar to the sort-probe join evaluated in Section 6.2. As with Oracle, additional equality conditions force a hash join, even if they are unselective. Moreover, there is no support for multiple dimensions.

Vertica takes a different approach to range joins by using the "ranges" relation as an index. According to their blog post, Vertica sorts the ranges [37] based on their lower bound and calculates a running maximum for the upper bound. However, Vertica's approach relies on the assumption of only having ranges with little to no overlap. A single tuple with an unselective range (overlapping other ranges) dominates the running maximum calculation and prevents an efficient query execution. Our algorithm is tolerant to such overlaps since the index structure is built on the points relation as shown in Section 6.2. For multidimensional range joins Vertica chooses the first between condition in the SQL query for the optimization, other range conditions are treated as filters on the result. We use a multidimensional index structure that can still improve runtime when adding more dimensions if they are selective.

7.3 Other Approaches and Overlap Joins

In the past kd-trees or variations of it were used in database systems as multidimensional base table index structures [18, 19, 29, 31]. In contrast, we use them as an ad-hoc index structure for joins on relations without a table index.

Band joins can also be evaluated for continuous data leveraging dynamic interval trees [1]. Since our approach only uses an index structure on the build side of the join, the probe side can be used for a data stream if the build side remains fixed. Umbra supports continuous views as shown in [39], and the implementation also employs the new kd-tree range join operator if beneficial.

A more general case of range joins are interval joins, where both relations contain intervals, and the join condition correlates to the intersection, gap distance, etc., of the two intervals. Examples for this are *band-joins on interval data* [6] or *in-memory interval joins* [7]. However, as Dignös et. al. [12] show, overlap conditions can be rewritten into a disjunction of two range conditions. Although, they [12] mainly focus on sort-merge joins and range joins leveraging table index structures like B-trees [2], the ideas are also applicable to our range join algorithm, which is very well suited for data exploration and data analysis. Adopting this technique, our range join implementation can also be used to support overlap joins efficiently.

8 CONCLUSION

Whenever databases are used for real-life data analytics, dealing with sensor, location, or temporal data, queries that contain joins with range predicates are quite common. Range joins are a case of non-equi joins that can still be very selective. In order to achieve acceptable performance, a database system should not resort to quadratic algorithms in such cases. Band joins are a subproblem with fixed size ranges. Different range and band join algorithms have been proposed in the literature to allow efficient evaluation of such join results. However, most approaches focus on particular cases rather than a generic solution. We present a generic multi dimensional range join algorithm using kd-trees, that outperforms all systems we tested and even beats hand-tuned implementations using the Spark data-processing engine. We show that an efficient implementation in a database system (in our case Umbra) is possible and can deal with corner cases like duplicates, null values, and different join methods. We show that a query optimizer can beneficially introduce range joins during physical optimization. However, we can also show that a refined selectivity estimation for range join predicates would be beneficial and could be future work.

REFERENCES

[1] Pankaj K Agarwal, Junyi Xie, Jun Yang, and Hai Yu. 2005. Monitoring continuous band-join queries over dynamic data. In *International Symposium on Algorithms and Computation*. Springer, 349–359.

[2] Rudolf Bayer and Edward McCreight. 2002. Organization and maintenance of large ordered indexes. In *Software pioneers*. Springer, 245–262.

[3] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[4] Jon Louis Bentley. 1978. *Decomposable searching problems*. Technical Report. Carnegie-Mellon Univ Pittsburgh PA Dept of Computer Science.

[5] Jon Louis Bentley. 1979. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering* 4 (1979), 333–340.

[6] Panagiotis Bouros, Konstantinos Lampropoulos, Dimitrios Tsitsigkos, Nikos Mamoulis, and Manolis Terrovitis. 2020. Band Joins for Interval Data.. In *EDBT*. 443–446.

[7] Panagiotis Bouros, Nikos Mamoulis, Dimitrios Tsitsigkos, and Manolis Terrovitis. 2021. In-memory interval joins. *The VLDB Journal* 30, 4 (2021), 667–691.

[8] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L Bocchino Jr, Sarita V Adve, and John C Hart. 2010. Parallel SAH kD tree construction.. In *High performance graphics*. Citeseer, 77–86.

[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[10] Databricks. 2022. *Range Join Optimization*. Retrieved 2022-07-05 from https://docs.databricks.com/delta/join-performance/range-join.html

[11] David J DeWitt, Jeffrey F Naughton, and Donovan A Schneider. 1991. *An evaluation of non-equijoin algorithms*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[12] Anton Dignös, Michael H Böhlen, Johann Gamper, Christian S Jensen, and Peter Moser. 2021. Leveraging range joins for the computation of overlap joins. *The VLDB Journal* (2021), 1–25.

[13] Philipp Fent and Thomas Neumann. 2021. A practical approach to groupjoin and nested aggregates. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2383–2396.

[14] Leonor Frias and Jordi Petit. 2008. Parallel partition revisited. In *International Workshop on Experimental and Efficient Algorithms*. Springer, 142–153.

[15] Leonor Frias and Jordi Petit. 2009. Combining digital access and parallel partition for quicksort and quickselect. In *2009 ICSE Workshop on Multicore Software Engineering*. IEEE, 33–40.

[16] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.

[17] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *The VLDB Journal* 30, 5 (2021), 883–905.

[18] Masaru Kitsuregawa, Lilian Harada, and Mikio Takagi. 1989. Join Strategies on KB-Tree Indexed Relations.. In *ICDE*. 85–93.

[19] Masaru Kitsuregawa, Mikio Takagi, and Lilian Harada. 1994. Algorithms and performance evaluation of join processing on KD-tree indexed relations. *Systems and computers in Japan* 25, 3 (1994), 78–90.

[20] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors. *Queue* 11, 7 (2013), 40–51.

[21] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2013. Massively parallel NUMA-aware hash joins. In *In Memory Data Management and Analysis*. Springer, 3–14.

[22] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 743–754.

[23] Rundong Li, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Near-Optimal Distributed Band-Joins through Recursive Partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2375–2390.

[24] David R Musser. 1997. Introspective sorting and selection algorithms. *Software: Practice and Experience* 27, 8 (1997), 983–993.

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

[26] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*.

[27] Thomas Neumann and Alfons Kemper. 2015. Unnesting arbitrary queries. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015).

[28] Oracle. [n.d.]. *Oracle Joins*. Retrieved 2022-07-05 from https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/joins.html

[29] Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. 2003. Bkd-tree: A dynamic scalable kd-tree. In *International Symposium on Spatial and Temporal Databases*. Springer, 46–65.

[30] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[31] John T Robinson. 1981. The KDB-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 10–18.

[32] Lei Sheng, Rafi Ahmed, Andrew Witkowski, and Sankar Subramanian. 2018. Sort-Merge Band Join Optimization. Retrieved 2022-07-05 from https://www.freepatentsonline.com/y2018/0101573.html

[33] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. 2007. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum*, Vol. 26. Wiley Online Library, 395–404.

[34] Valery Soloviev. 1993. A truncating hash algorithm for processing band-join queries. In *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE, 419–427.

[35] Michael Stonebraker and Lawrence A Rowe. 1986. The design of Postgres. *ACM Sigmod Record* 15, 2 (1986), 340–355.

[36] New York City Taxi and Limousine Commission (TLC). [n.d.]. *TLC Trip Record Data*. Retrieved 2022-07-05 from https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[37] Vertica. 2015. *What Is a Range Join and Why Is It So Fast?* Retrieved 2022-07-05 from https://www.vertica.com/blog/what-is-a-range-join-and-why-is-it-so-fastba-p223413/

[38] Meng Wan, Chao Wu, Jing Wang, Yulei Qiu, Liping Xin, Sjoerd Mullender, Hannes Mühleisen, Bart Scheers, Ying Zhang, Niels Nes, et al. 2016. Column store for GWAC: a high-cadence, high-density, large-scale astronomical light curve pipeline and distributed shared-nothing database. *Publications of the Astronomical Society of the Pacific* 128, 969 (2016), 114501.

[39] Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. 2020. Meet me halfway: split maintenance of continuous views. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2620–2633.

[40] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[41] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)* 27, 5 (2008), 1–11.