SonicJoin: Fast, Robust and Worst-case Optimal

Ahmad Khazaie
a.khazaie19@imperial.ac.uk
Imperial College London
United Kingdom

Holger Pirk
pirk@imperial.ac.uk
Imperial College London
United Kingdom

ABSTRACT

The establishment of the AGM bound on the size of intermediate results of natural join queries has led to the development of several so-called worst-case join algorithms. These algorithms provably produce intermediate results that are (asymptotically) no larger than the final result of the join. The most notable ones are the *Recursive Join*, its successor, the *Generic Join* and the *Leapfrog-Trie-Join*. While algorithmically efficient, however, all of these algorithms require the availability of index structures that allow tuple lookups using the prefix of a key. Key-prefix-lookups in relational database systems are commonly supported by tree-based index structures since hash-based indices only support full-key lookups. In this paper, we study a wide variety of main-memory-oriented index structures that support key-prefix-lookups with a specific focus on supporting the Generic Join. Based on that study, we develop a novel, best-of-breed index structure called Sonic that combines the fast build and point lookup properties of hashtables with the prefix-lookups capabilities of trees and tries. To evaluate the performance of a variety of indices for worst-case optimal joins in a modern code-generating DBMS, we leveraged flexible, compile-time metaprogramming features to build a framework that creates highly efficient code, interweaving (at a microarchitectural level) a generic join implementation with any appropriate index structure. We demonstrate experimentally that in that framework, Sonic outperforms the fastest existing approaches by up to 2.5 times when supporting the *Generic Join* algorithm.

1 INTRODUCTION

Joins are among the most expensive relational operators, both in terms of CPU cost as well as memory consumption. In particular, the efficient processing of multi-way joins (i.e., joins of multiple relations) is challenging because worst-case costs grow exponentially with the number of relations [38].

Until recently, the state of the art for multi-way joins was to treat them as a sequence of binary (i.e., two-way) joins. An immense body of research [13, 17, 23, 24] covers algorithms, data structures and optimization techniques for binary joins. Virtually all of these algorithms follow the same pattern: in a preparatory phase, some supporting index structure (usually a hash table, sorted array or tree-index) is built on one or both of the tables. In the process phase, the index structures are used to accelerate the actual result calculation (e.g., hash-probing or merging of sorted runs).

While these strategies have linear effort in the best case, in the case of poor join order selection, the number of output tuples can grow exponentially. While little can be done to avoid these costs for worst-case queries (e.g., when a cartesian product is requested), the choice of a poor join order can cause even queries with empty result sets to have exponential costs.

However, it is possible, to design multi-way join algorithms that are resilient to poor join orders: Atserias, Grohe and Marx [11] were the first to theoretically prove the existence of join algorithms that asymptotically perform work in the order of the size of the final join result – a bound we call the AGM-bound for the authors' initials. Since the size of the final result is independent of the chosen join order, such algorithms are optimal in their worst-case behaviour. We refer to such algorithms as worst-case optimal join algorithms. Ngo, Porat, Re and Ruta proposed a join algorithm that is worst-case optimal [38] followed by a more general worst-case optimal join algorithm [39] (in this paper, we refer to the first algorithm as Recursive Join while we call the second one the Generic Join algorithm).

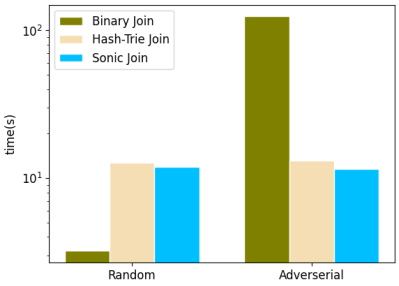

Figure 1: Binary Join vs Sonic Join vs Hash-Trie Join

The index structures to support the Generic Join are significantly more expensive to build than those supporting a binary hash-join: to assess the magnitude of the performance difference, we implemented the Generic Join algorithm using the BTree implementation from Abseil library [7] as a supporting index structure. As a baseline, we implemented a sequence of (fully inlined) binary hash-joins (based on Abseil's hash-set) and a Hash-trie Join proposed by others [22]. We compare the three implementations for a triangle counting query and vary the underlying data distribution from uniform random to maximally adversarial. Figure 1 shows the running time of each algorithm. Overall, we find that the Generic Join outperforms the hash-join and Hash-trie Join for the adversarial input because it is more work-efficient. However, we find the opposite for the random input: the binary join is faster than either of worst-case optimal join algorithm as there are no exploding intermediate results and it can take advantage of a fast-to-build (hash) index.

Similar to most binary join algorithms, the Generic Join algorithm requires a supporting index structure, specifically, a multi-level index on each table ordered by the same 'global' attribute order (called the *total order* in [39]). While the supporting index structures can, in principle, be persisted and reused, worst-case optimal join algorithms (like their binary counterparts) depend on query-specific indices. This complicates index reuse as it leads to the classic problems of secondary indexing: deciding which ones to persist, maintaining them under update, selecting which ones to use during query evaluation, etc. Consequently, the fast building of supporting indices is crucial to join performance.

© 2023 Copyright held by the owner/author(s). Published in Proceedings of the 26th International Conference on Extending Database Technology (EDBT), 28th March-31st March, 2023, ISBN 978-3-89318-092-9 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

We address this problem by developing an index structure that is fast to construct while efficiently supporting the operations necessary to support Worst-Case Optimal Join algorithms. Specifically, our contributions are:

- Based on a careful analysis of the microarchitectural behavior of the Generic Join Algorithm, we develop *Sonic*, a hardware-conscious index structure designed to support algorithms that, like most worst-case optimal join algorithms, rely on early elimination of result candidates
- Having developed a highly efficient index structure, we develop an equally efficient implementation of the Generic Join algorithm. Contrary to existing work [4, 22], our approach does not rely on just-in-time code generation. Instead, we developed a compile-time framework that takes advantage of the metaprogramming features of C++. This approach exploits the underlying c++-compiler to support any index structure that exposes the required primitive operations. This approach is highly efficient: it outperforms the well-known EmptyHeaded system and Hash-Trie Join that rely on just-in time code generation.
- Using the join framework, we perform an extensive study of state-of-the-art index structures with respect to their fitness to support worst-case optimal join algorithms. We demonstrate that *Sonic* outperforms all other indexing methods when supporting the Generic Join, outperforming competitors by a factor exceeding 2× in relational as well as graph-processing benchmarks.

The remainder of this paper is organized as follows: In Section 2, we introduce the necessary background on worst-case optimal joins. In Section 3, we develop the Sonic index structure, present our implementation of the Generic Join in Section 4 and evaluate both in Section 5. We discuss related work in Section 6 and ideas for future work in Section 7. Finally, we conclude in Section 8.

2 BACKGROUND

In this section, we provide the necessary background on worst-case optimal joins.

2.1 The AGM Bound

Atserias, Grohe and Marx [12] aimed to obtain the largest possible query result size through formal analysis of any query. They construct a *hypergraph* $H(V, E)$ for a query, the set of vertices, V, denoting the attributes in the query and the *hyperedges* E denoting its relations. They proved that a tight bound on the size of the result for any (natural) join query is tighter than the (obvious) exponential bound. To illustrate that, consider a triangle join query, $Q(a, b, c) = R(a, b) \bowtie S(b, c) \bowtie T(c, a)$. The hypergraph $H(V, E)$ for Q is constructed using $V = \{a, b, c\}$ and $E = \{(a, b), (b, c), (c, a)\}$.

An edge cover for a hypergraph $H(V, E)$ is a subset $C \subseteq E$ of hyperedges such that for each vertex $v \in V$ there is at least one edge $e \in C$ such that $v \in e$. By assigning a weight $u_i = 1$ if $e_i \in E$ and $u_i = 0$ if $e_i \notin E$, and expressing the cover requirements as inequalities, the edge cover can be formulated as an integer programming problem:

$$a : u_R + \quad u_T \geq 1$$

$$b : u_R + u_S \quad \geq 1$$

$$c : \quad u_S + u_T \geq 1$$

By modifying the linear problem to allow edge weights to be non-integer between 0 and 1, a *fractional edge cover* for the hypergraph can be obtained. Grohe and Marx proved that using the fractional edge cover, a tight upper bound for the size of relational join queries can be achieved [12, 25].

For the example above, if $|R| = |S| = |T| = n$, a straightforward bound is n^3, i.e. the cartesian product of the three relations Atserias, Grohe and Marx [12] showed that when Q is the set of triples of constants (a,b,c), $|Q| \leq |R|^{u_R} \times |S|^{u_S} \times |T|^{u_T}$ and if we set $R = S = T$ the AGM bound is minimized when $u = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$, which provides $|Q| \leq n^{\frac{3}{2}}$ as the result [42].

2.2 Worst-case Optimality

A join algorithm is called worst-case optimal if its running time over a database instance I is bounded by the query's worst-case output size (AGM bound). Formally, where u_j is the weight of hyperedge e_j in the fractional edge cover, N_j is the size of the relation R_j, and $Q = \bowtie_{j \in \{1,..,l\}} R_j$. To achieve the worst-case output size, the tightest AGM bound for the hypergraph should be determined by minimizing the right-hand side of the inequality above. Consequently, the following linear optimization problem needs be solved:

$$min \sum_{j=1}^{l} \log(N_j) \times u_j \quad | \quad \sum_{j:v \in j} u_j \geq 1 \quad \forall v \in V$$
$$u_j \geq 0 \quad \forall j \in E$$

Where V and v_j are the set of attributes in the query Q and the set of attributes in the relation R_j, respectively.

The cost of an optimal join algorithm would be $O(f(|V|, |E|) \times (\prod_{j \in E} N_j^{u_j} + \sum_{j \in E} N_j))$ where $f(|V|, |E|)$ is the preparation cost (extracting the order of attributes), the term $\sum_{j \in E} N_j$ reflects the cost of reading the inputs and the term $\prod_{j \in E} N_j^{u_j}$ reflects the query size bound. Any algorithm (asymptotically) satisfying this bound in its complexity is called worst-case optimal. Having established the bounds of the join query, let us discuss the join algorithm.

2.3 The Generic Join Algorithm

The Generic Join algorithm as proposed by Ngo et al. can be distinguished in a preparation phase and a join phase that we will explain in this section.

2.3.1 Preparation. Most worst-case optimal join algorithms rely on a specific order of attributes which can be achieved by different approaches [3, 4, 22, 39]. In the context of the Generic Join algorithm, to support the prefix-lookup operation, the attribute order in every index must follow the *total order*: an ordering of the queried relation's attributes that makes the execution time of of the

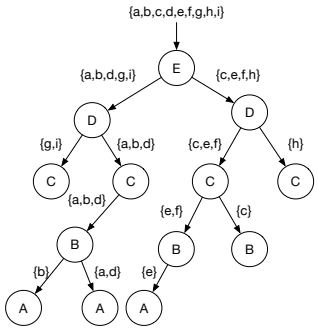

Figure 2: Query Plan Tree

lookup queries become linear. Formally, for a given query $Q = \bowtie_{e \in E} R_e$ and the total order $\gamma = A_1, A_2, ..., A_n$, the order of attributes in a relation $R_e({A_1}^e, {A_2}^e, ..., {A_k}^e)$, should be aligned with γ such that for any $i \leq j$ the attribute ${A_i}^e$ precedes ${A_j}^e$. So, given a tuple $u_{A_1,...,A_i}$, a prefix-lookup on R_e can enumerate all the tuples $\pi_{A_{i+1},...,A_j}(R_e[u_{A_1,...,A_i}])$ in the relation R_e.

Ngo et al.[39] proposed to construct a so-called "Query Plan Tree (QP-Tree)" and derive the order of attributes in the following manner: nodes in the QP-tree represent the hyperedges of join query graph with each node representing a subset of query attributes, called the node's *universe*. The universe of the QP-tree's root is the set of all query attributes and the attributes of the root are the same attributes as the corresponding hyperedge/relation. The universe for the right child of the root contains the intersection of the root's universe and its attributes. The universe of the left child of the root contains the difference of the root's universe and its attributes. Both the left child and the right child of the root are labeled by the next hyperedge given an arbitrary order.

Figure 2 illustrates an example of a query plan tree for $Q = R_A(a, b, d, e) \bowtie R_B(a, d, f, c) \bowtie R_C(g, c, h, i) \bowtie R_D(a, b, d, h) \bowtie R_E(f, c, e, h)$. In this example $V =<a, b, d, f, e, g, c, h, i>$ and the universe (u) of the right child and left child of the root are $u_{rc} =<c, e, f, h>$ and $u_{lc} =<a, b, d, g, i>$, respectively. The total order extracted for the query is $\gamma =<g, i, b, a, d, e, f, c, h>$, but as there is no suffix $f = A_{k+1}, ..., A_n$ such that for some $0 \leq k < n$, f exists in γ, this order is not *compatible* with the query. By permutating the attributes of the relations they can be queried according to the total order. In practice, this means that the optimal order is query specific and indices cannot be pre-built. Consequently, build-time is a crucial performance factor.

2.3.2 Generic Join. Having discussed the preparation, let us explain the procedure of computing the results for a given query of a representative worst-case optimal join algorithm.

The algorithm proposed by Ngo et al. (shown in Alg. 1) scans the smallest relation which contains the first attribute in the total order. It executes a prefix-lookup operation on the other relations to find all tuples that have a common prefix for the attributes and produces a candidate sub-tuple of the final result (line 3). By comparing the size of the scanned relation with the AGM bound of its right sub-problem, the smallest relation is selected to join with the sub-tuple (line 9). Note that the order of the attributes in the index is aligned with their order in the total order (see Section 2.3.1), looking for a sub-tuple in other relations is a prefix-lookup on the join condition(s) (line 12).

The Generic Join algorithm adheres to the AGM bound as its execution time is $|E||V| \prod_{j \in E} {N_j}^{u_j}$. The build time for the index structure is $|V|^2 \sum_{j \in E} N_j$ and satisfies the following conditions:

- Lookup time for $u_{A_1,...,A_i} \in \pi_{A_1,...,A_i}(R_e)$ is $O(i)$
- Time to count for $|\pi_{A_{i+1},...,A_j}(R_e[u_{A_1,...,A_i}])|$ is $O(i)$
- When the output is non-empty, returns all the tuples in the set $\pi_{A_{i+1},...,A_j}(R_e[u_{A_1,...,A_i}])$ in linear time.

Having established the background for worst-case optimal join, we present a novel index structure to support the Generic Join in the following section.

3 SONIC INDEX

In this section, we present the design of the Sonic index structure. We start by discussing the requirements and options to support the Generic Join. Based on those, we describe the structure of the Sonic index followed by a description of the supported operations. Before concluding this section, we briefly discuss tuning parameters and memory requirements.

Algorithm 1: Generic Join Algorithm

```
   Input: Hypergraph H = (V, E)
   Input: Relations R_e, e ∈ E
   Input: Fractional cover x = (x_e), e ∈ E
1  Function RecursiveJoin(H, R_e, x_e):
2      if |V| = 1 or V ⊆ e_i, ∀e_i ∈ E then
3          return ∩_{e∈E} R_e
4      Q ← ∅                          // Q is the set of tuples to be returned
5      Pick f ∈ E such that f is a suffix of γ
6      f' ← V \ f
7      E_1 ← e ∈ E | e ∩ f' ≠ ∅
8      E_2 ← e ∈ E | e ∩ f ≠ ∅
9      for every t ∈ RecursiveJoin(H_1 = (f', E_1 \ f), (R_e)_{e∈E_1}, (x_e)_{e∈E_1}) do
10         if x_f < 1 and |R_f| ≥ ∏_{e∈E_2\f} |R_e[t]| · x_f/(1−x_e) then
11             for each t' ∈ RecursiveJoin(H_2 =
                   (f, (E_2 \ f), (R_e)_{e∈E_2\f}, (x_e/(1−x_e))_{e∈E_2\f}) do
12                 Q ← Q ∪ prefixLookup(R_f[t], t')
13         else
14             for every t' ∈ R_f do
15                 if prefixCount(R_e[t], t'_f) for every e ∈ E_2 then
16                     Q ← Q ∪ {t ∪ t'}
17     return Q
```

3.1 Requirements for the Generic Join

The Generic Join algorithm requires an index that supports *insert*, *point lookup*, *prefix lookup*, and *count prefix*. While tree-based indices support the required operations, they involve more computational effort and pointer chasing to insert the key than hash tables. Hash-based indices are cheaper to build but do not support the prefix-lookup operation.

A straightforward way to extend the principle of hash tables to support these operations is to build a hash table of hash tables as proposed in [19]. In such a hierarchical hash table, each row in the first hash table stores a pointer to another hash table in the next level and so forth so on. This approach has multiple drawbacks. First, by increasing the level of indirections the point-lookup costs increase. Second, as each key in level *k* has a pointer to one hash table in level *k+1*, the total number of hash tables grows exponentially with the number of levels which leads to memory fragmentation. Third, each hash table requires a non-trivial amount of memory. If memory is not large enough to fit all the hash tables, a hash table of hash tables does not scale. Fourth, if a hash table overflows it would require expensive re-hashing/re-building, potentially at multiple levels.

A structure that hierarchically stores each data tuple at each level, could avoid such extra costs. Such a structure could support prefix-lookup queries by returning all the tuples in the levels after the prefix match. We developed just such an index structure: *Sonic*. Sonic does not suffer from indirection overhead, memory fragmentation, memory overhead, or expensive re-hashing.

3.2 The Sonic Data-Structure

As discussed in Section 1, the objective of the Sonic index is to strike a balance between fast building and fast prefix lookup. To that end, Sonic merges design elements from hash tables, trees, and tries and augments them with supplemental components to achieve high performance. Figure 3 illustrates the general structure of Sonic for a table with four columns. For each column except the last, Sonic contains a substructure, called *Level*, that forms a hierarchy (laid out from left to right in the figure). Levels in the Sonic index follow the general structure of a hash table

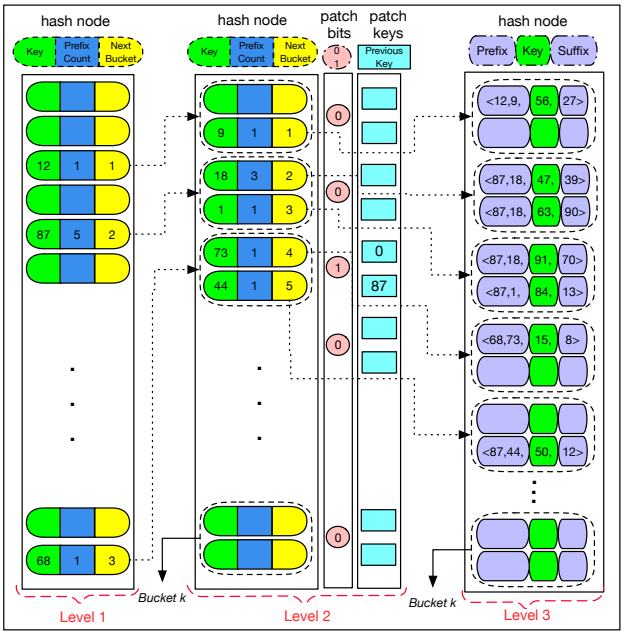

Figure 3: Sonic Hash Table (Bucket Size = 2)

in that they contain keys (in green), and payloads (in yellow). In addition, they contain the number of tuples that have a common prefix, called *prefix counter* (in blue). The payload can be either a pointer (more specifically an offset) to a bucket in the next level, where the next attribute is stored; or a tuple. In the last *level* of the index, the tuple (purple section) is stored alongside the key to avoid adding an extra layer (for the last attribute) in the structure. Middle levels, i.e., any level except the first and the last, store *patch bit* per bucket (the red circle) and a *patch key* per slot (the blue box) to eliminate false positives.

3.3 Patching to reduce false positives

Sonic is designed to support worst-case optimal join algorithms at the performance required for in-memory data processing. This requires avoiding performance hazards such as memory fragmentation, pointer chasing, and complex branching. While Sonic can serve as an index structure to support the Generic Join, it may produce false positives as entries at *inner levels* (i.e. levels between the first and last) cannot be distinguished by their prefix. While this does not affect correctness (false results are filtered out), it requires a (potentially costly) traversal of all levels of the index, when false positives occur at an inner level.

All *inner levels* have two auxiliary fields that allow the elimination of these positives: first, one bit per bucket which we call the *patch bit*. The patch bit is set to 1 if the bucket contains attributes of tuples with different keys at the immediately preceding level (we refer to the bucket as *'patched'*). Patch bits of one level are only stored in that level, i.e., not propagated to the next one. If the patch bit is set, the key in the preceding level is replicated in the second auxiliary field: the *patch key* column. While replicating only the immediately preceding key does not guarantee to eliminate false positives (they could occur in a level before the preceding), it makes them unlikely (in our experiments we found that only 10% of the buckets in the second-level are patched). As the size of tuples increases, the probability of a false positive being propagated to the last level decreases further (we have not observed this case in the workloads). The remaining false positives are eliminated in the last level using the stored payload to verify the results.

Algorithm 2: insert

```
Input: Tuple t = < a_1, a_1, ..., a_k >
1  Function insert(t):
       /* acquire lock if parallel                                   */
2      if t[level] not in keys then
3          entry_slot ← get_next_available_slot()
4          keys[entry_slot] ← t[level]
5          prefix_count[entry_slot] ← 1
6          if level ≠ last_level then
7              next_level_bucket[entry_slot] ←
               get_bucket_number(next_level_insert(t))
8      else
9          entry_slot ← index of t[level]
10         if level ≠ last_level then
11             prefix_count[entry_slot]+ = 1
12             next_bucket ← next_level_bucket[entry_slot]
               next_level_entry ← next_level.insert(t, next_bucket)
13             next_level_bucket[entry_slot] ←
               next_level_entry.bucket_number
14     if is_patch_bucket then
15         patch_key[entry_slot] ← t[level − 1]
       /* release lock if parallel                                   */
16     return entry_slot
```

To avoid non-aligned memory access, patch bits and keys are stored in memory regions separate from the key-value pairs (indicated by the rectangular boxes in Figure 3). The patch-bit vector is designed for a minimal footprint to keep it cache-resident. The patch keys are rarely accessed in inner levels (only if a bucket is patched) and are, thus, negligible in terms of memory traffic.

Sonic achieves the required performance for worst-case optimal join algorithms by combining a highly predictable, single-allocation, approximate data structure (the unpatched levels) with a 'disambiguating' mechanism (the patch structure) to guarantee correctness. Such a data structure is unprecedented and highly novel. The design is based on the insight that disambiguating can be expensive if it is rare.

3.4 Sonic Operations

The build of the Sonic index is comprised of a scan of the input and the insertion of every tuple (one at a time). The probe phase of the join consists of a series of lookup queries. In this section, we describe the operations supported by Sonic to meet the Generic Join requirements.

3.4.1 Insert. A new tuple t $(a_1, ..., a_k)$ is inserted into a Sonic index by hierarchically storing one attribute per level. Similar to a hash-trie, values are hashed to determine their position in the level. However, instead of hashes the values are stored to allow the index to match the exact prefix at each level without requiring further traverse of the table. Based on the initial hash, Sonic performs linear probing to find a slot. Additionally, a counter for the key is set which holds the number of tuples that match the prefix.

At the first level, inserting t is virtually identical to insertion in a hash-table with the key a_1: If the key is already present at that level, the insert immediately progresses to the next level after incrementing the prefix counter (Alg. 2: 11-12). If the key is not yet present, the prefix counter is set to 1 and a new, empty bucket is allocated at the second level, and a pointer to it is stored as the *Next Bucket* (Alg. 2: 3-5). To accelerate the reservation of new buckets sonic keeps a pointer to the next empty bucket during the build phase.

At the second level, a_2 constitutes the key. Its position is determined by summing up the *Next Bucket* value with a hash of a_2

Algorithm 3: prefixLookup

```
Input: Tuple t = < a_1, a_1, ..., a_l >
Input: integer bucket_number = k
1  Function prefixLookup(t, k):
2      if level = last_level then
3          for every Tuple t ∈ Bucket k do
4              results ⋃ {t}
5          return results
6      if length(t) > level then
7          entry_slot ← index of t[level]
8          next_bucket ← next_level_bucket[entry_slot])
9          return next_level.prefixLookup(t, next_bucket)
10     else
11         bucket ← bucket of t[level]
12         if patch_bit[bucket] == 1 then
13             for every key ∈ bucket do
14                 key_index ← index of key
15                 if patch_key[key_index] == t[get < level − 1 >] then
16                     next_bucket ← next_level_bucket[key_index])
17                     prefix ← {t, key}
18                     next_results ←
                       next_level.prefixLookup(prefix, next_bucket)
19                     results ⋃ next_results
20         else
21             for every key ∈ bucket do
22                 key_index ← index of key
23                 next_bucket ← next_level_bucket[key_index])
24                 prefix ← {t, key}
25                 next_results ←
                   next_level.prefixLookup(prefix, next_bucket)
26                 results ⋃ next_results
```

modulo the size of the bucket (different combination schemes are possible). If the slot is empty, a_2 is inserted and the prefix counter set to 1. The *next bucket* value set to the first empty bucket in the third level.

In the example shown in Figure 3, *<12,9,56,27>* is inserted by hashing *12* into the first level and the prefix counter set to 1. As the level 2 is empty, *9* is placed in the second slot of the first bucket of the second level with the prefix counter set to 1. Finally, the tuple is stored in the third level and no prefix counter is required at this level. When inserting *<87,1,84,13>*, the *87* is already present in the first column, so the prefix counter is incremented by 1 and *1* is inserted in bucket 2 in the next level.

If all the bucket slots are full, the probing continues until a free slot is found – a_2 has 'spilled into' a different bucket. In this case, to disambiguate the values, the bucket is marked as *patched* and a_1 stored as the *patch key* and the values that are already present in the bucket are patched (Alg. 2: 15). In Figure 3, inserting *<87,44,50,12>* illustrates the overflowing and patching logic: since *87* exists in the first level the process updates the prefix counter to 3 and follows the bucket index to the next level. When probing bucket 2 of level 2, it is found to be full. Hence the next bucket is probed and an empty slot is found. This means that bucket 3 at level 2 now contains keys with prefixes *68* and *87*. Thus, it needs patching: the patch bit for bucket 3 is set and the patch key for the newly inserted key and *73* is set to *87* and 0, respectively.

3.4.2 Parallel insert. Sonic uses locks to ensure the integrity of the hash table (multiple identical keys only occur once in every level). To reduce the locking overhead we implemented key-range locking per level and found empirically that locking at a granularity of *8192* provides robust and close-to-optimal performance (never more than 30% worse than optimal).

3.4.3 Prefix lookup. If the query tuple is a prefix, the result is either the number of data tuples or the actual data tuples that match the hierarchical lookups. Assume, e.g., the query tuple *t'*(a_1,a_2,...a_l) where $l < k$ and k is the length of data tuples in the index. The search traverses the first l-1 levels, looking up for a_1 to a_{l-1}. At level l, if the search is successful, count prefix operations are answered immediately using the *prefix count* value.

The *next bucket* field leads us to the next level where we can find all data tuples that match the prefix (Alg. 3: 7-9). For each entry in the bucket at level $l + 1$ the search moves to the next level and continues the process at level $l + 2$. The same steps are taken for all match elements from level $l + 2$ towards the last level $k - 1$ (Alg. 3: 11-26) and all data tuples are retrieved from the *leaf nodes* (Alg. 3: 5). Point lookup is a special case of prefix lookup that the prefix has the same length as the data tuples. Likewise, the count prefix returns only the number of the tuples with a common prefix. In Figure 3 to lookup *<68, 73, 15, 8>*, the hash of *68* is computed to determine its position in the first level. As the key is found, the *next bucket* number, i.e. 3, is used to lookup *73* in the second level of the index. This bucket is patched; therefore, the patch key should be checked and in case of success, the search process is continued in bucket 4 in the last level. Then, the remaining elements of the tuple are compared with the tuple stored in the last level the operation returns a successful result.

3.5 Space Overhead

Before concluding this section, let us briefly discuss Sonic's memory overhead. For a tuple *t*(a_1,...,a_k) in which element at position i, e_i, has size DTS_i and an overallocation factor OF, Sonic allocates:

$$OF \times \left(\sum_{i=1}^{i=k-1} DTS_i + ((k-2) \times 8B) + \sum_{i=2}^{i=k-2} DTS_i + \sum_{i=1}^{i=k} DTS_i + 1b \right)$$

At each level, the element, e_i *i = 1, ... k-2* is stored as key. For the first and inner levels, the prefix counter and the *next bucket* pointer are stored alongside the patch structure and the tuples are stored in the last level. For 1000 tuples, 4 integers each, the Sonic index requires at least 24KB of memory.

4 SONIC JOIN

Naturally, a high-performance index structure is only one part of building a high-performance join implementation. A second, equally important, component is a highly efficient implementation of the algorithm itself. The current trend towards code-generating/just-in-time compiling database engines [35, 43] demonstrates that index structures need to be tightly interwoven with the processing engine. EmptyHeaded [4], e.g., follows that approach: it generates code for a custom index structure. However, such an integrated design makes it virtually impossible to evaluate different index structures on a 'level playing field', i.e., all other factors being equal. It also limits applicability as, for example, large value domains do not dictionary encode well. To address this problem, we followed a different approach: implementing the critical section of the algorithm exclusively using C++ templates and allowing the compiler to specialize the code. In this section, we provide details to illustrate how performance is achieved by a hardware-conscious implementation.

4.1 Worst-case Optimal Joins in C++ templates

To facilitate the evaluation of different index structures under the assumption that their code would be JIT-generated by a modern database engine, we developed an implementation of the Generic Join algorithm that can emulate the behavior of JIT-compiling DBMSs. Our implementation uses C++ templates to generate, inline and optimize code as well as data structures at compile time using the compiler's own optimizer. The objective

Listing 1: Sonic Join API

```
1  constexpr Capacity 1024
2  constexpr BucketSize 8
3
4  using TableSchema = tuple<int, int>;
5  using TableAttributeIDs = AttributeIndex<0, 1>;
6  using TotalOrderSchema = tuple<int, int>;
7  using AttributeIndexingOrder = AttributeIndex<0, 1>;
8
9  using Index = Sonic<Capacity, BucketSize, int, int>;
10 using IndexAdapter = SonicIndexAdapter<TableSchema,
       TotalOrderSchema>;
11
12 auto tableA =
13     Relation<IndexAdapter, TableSchema, TableAttributeIDs,
           AttributeIndexingOrder>(inputData);
14 auto tableB =
15     Relation<IndexAdapter, TableSchema, TableAttributeIDs,
           AttributeIndexingOrder>(inputData);
16
17 join(tableA, tableB);
```

Listing 2: Index Join Adapter

```
1  template <size_t... Offsets>
2  auto lookupByPrefixWithIndices(
3    TotalOrderSchema const& t,
4    make_index_sequence<PrefixLength>> const){
5      index.prefixLookup
6        ((get<get<PrefixIndices>(Offsets...)>(t))...);
7  }
8
9  template <size_t... ViablePrefix>
10 auto lookupByPrefixAndScatterIntoTuple(
11   TotalOrderSchema const& t,
12   size_t const prefixLength,
13   index_sequence<ViablePrefix...> const&){
14     ((result = ((ViablePrefix+1 == prefixLength) ?
15       lookupByPrefixWithLength<ViablePrefix+1>(t)
16       : result)), ...);
17 }
```

of designing such a framework is to assess the performance of a code-generating DBMS that is achieved without the prohibitive effort of implementing the various data structures in such a DBMS. Briefly, this framework allows 'plugging in' any C++ index as long as it provides the required operations and generates highly optimized code (inlined, vectorized, etc.). To the best of our knowledge, there is no similar framework available for public use and our contribution in building such a framework that could be useful for other researchers.

In our implementation, the entire query plan is expressed as a C++ template expression and instantiated by the template pre-processor. Through the use of modern C++ template meta-programming features such as parameter packs, non-type template parameters and SFINAE, even recursive data structures (such as Sonic) and recursive algorithms (like the Generic Join) can be expressed in a concise manner.

To gain an impression of the API, consider Listings 1 and 2: listing 1 shows an interface of a self-join in our engine. Lines 1 and 2 define the capacity and the bucket size in Sonic. The schemas for the input table and the *final tuple*, (contains the values of the total order), are represented in lines 4 and 6 (in this case, all tables have the same schema). In line 5 an integer id is assigned to each column using a non-type template parameter, which is used in the join operation to join tuples based on the common column ids. Line 7 represents the index of each column in the total order. These two non-type template parameters avoid run-time iterations for constructing the final tuples (see List. 2).

An index structure is defined in line 9 with the defined capacity and size of buckets – while we use Sonic here, any index is possible as long as an adapter is defined to support the lookup operations. Types of the tuple elements, i.e. types of keys in the index, are provided in a pack of template parameters. To instantiate the index structure, these parameters (known at compile time) will be unpacked and used at the associated level of the index. To construct the index, a tuple has to be decomposed and each element needs to be inserted into the appropriate level. Using the SFINAE feature of C++, the compiler can instantiate the right data structure for each level based on the type of the element.

The following line (10) defines the index adapter that extracts an index-compatible prefix (using the non-type template parameters (9 and 10)) from a final tuple schema, which has the total order schema. After performing the prefix operation by the index, the result is placed in the final tuple and the result of the join is computed. This mapping process is done at compile-time and does not impose any run-time costs on the system. In Listing 1, lines 13 and 15 construct two indices and the order of the attributes is determined by their position in the final tuple. In case the schema of the input table is not consistent with the total order schema, the attributes are reordered by the index adapter.

Listing 2 illustrates the details of the prefix extraction and the replacement of the results from index prefix lookup. For the prefix-lookup function (13) on a given final tuple, the prefix is extracted. To do so, a pack of non-type parameters for the given prefix length is generated by the compiler from a recursive function (line 13). By unpacking these parameters (line 6), the column ids of the prefix attributes are calculated. Subsequently, the prefix lookup is performed (4) and the preliminary results are used for the next steps to join with the other tables. The same process is repeated to extract the prefix from the new tuples and perform the prefix lookup on the next relations. Since this process is implemented using template meta-programming, the compiler generates the code for extracting the prefixes at compile time thus there is no performance drop caused by the adapter.

Once the indices are created the join function is called, the last line in Listing 1 causes the instantiation of the actual join. Similar to the datalog notation, the query is 'implicitly' defined through the tables AttributeID parameter (attributes with the same ID are joined). Afterwards, the compiler's low-level optimizer generates the data structure, eliminates constants and common subexpressions and inlines function calls. The result is an implementation that, while time-consuming to compile, allows easy experimentation with different queries, index structures and tuple types.

This framework can be used to perform a variety of experiments to select the best index with the highest performance without implementing all the alternatives in a DBMS. All that is needed is to define the schema of the tables and the schema of final tuples, then call the join function. Since the join operation is fully templated, the compiler generates the most optimized code and reduces time and human effort.

4.2 Parameters

Like most index structures, to accommodate different workloads tuning parameters of Sonic need to be set to achieve the best performance. Like classic hash tables, Sonic relies on overallocation to avoid long probe chains, which are particularly harmful as they lead to overhead for checking patches. We found that overallocation by higher factors improves performance at the cost of a higher memory footprint. To support parameter exploration and facilitate experimentation, Sonic is implemented in highly templatized C++ and accepts parameters, i.e. Key type, hash function, bucket size, and capacity, at compile-time.

4.3 Determining the total attribute order

The last component of a worst-case optimal join implementation is the determination of total attribute order. While we believe that determining the total order in a C++ template program at

compile time would be possible but the cumbersome effort does not worth the time. Because this order is required only once at the beginning of the program to instantiate the index and all the calculations, e.g. building the QP-tree, traversing the nodes, and extracting the order, should be done using sophisticated metaprogramming. Consequently, we implemented the total order algorithm in a Python script that faithfully implements the total order calculation algorithm proposed by Ngo et al. [39] and emits a file containing the total order as a C++ template instantiation per table (basically line 2 in Listing 1). While we are aware that alternative total order calculation algorithms have been proposed [34], we consider the discussion and evaluation of these outside the scope of this paper.

5 EVALUATION

We are interested in Sonic's performance compared to competitors with similar feature sets: we compare point-lookup runtime to established hash-maps and prefix-lookup time versus trees and tries. A purpose of this study is to evaluate the worst-case optimal join algorithm; we analyze the performance of the implemented Generic Join algorithm in different ways, i.e. using different indices, varying key sizes and different types of queries. We compare the achieved results with binary join and an existing JIT-code generation system, Umbra [22]. In this section, we provide the results of experiments for both the Sonic index and the Generic Join algorithm.

5.1 Experimental Setup

All experiments were conducted on a system with two Intel-Xeon Silver 4114 CPU (10 cores running at 2.2 GHz 32 KB L1, 256 KB L2 and 25600 KB L3 cache). This system has 32GB DDR4 RAM all experiments have been made on a single thread. We used Google Benchmark v1.5.0 as a microbenchmark library to measure the performance of each index as an isolated component. Also, it is used for macro benchmarking and evaluating the performance of the join algorithms. We used clang++ 11 compiler and all the experiments have been executed on Linux Ubuntu 18.04 OS.

5.2 Data

For the microbenchmarks, the input is a sequence of uniform random numbers generated by Zipfian distribution. Unless otherwise stated, the size of each input table is 256M tuples with the number of columns varying from 2 to 8.Consequently, the size of the input data ranges from 4 GB to 16 GB and it is assured that it does not fit into the cache.

5.3 Workload

We compared the execution time for index operations in Sonic with different members from each group. For point or prefix-lookup experiments, half of the query tuples are selected from tuples which have not been inserted into the index. This allows a good estimate of index performance in the real world and makes sure that all levels of the index are traversed during the search. For prefix lookups, the prefix size is considered half the size of keys in table. We evaluated the join algorithm for real-world graph-like data using Wikipedia vote network, the Epinions trusts network, Facebook and Twitter social networks [32] as well as Join Order Benchmark [30] queries to evaluate the algorithm on Internet Movie Database (IMDB) dataset.

5.4 Baselines

To establish baselines, we evaluated Sonic against indices that are highly tuned and fast in comparison to other competitors: Abseil Hash Set [7], Adaptive Radix Tree (ART) [31], TLX-BTree [15], Hierarchical Abseil Hash Map [7], HAT-Trie (HTrie) [1], Hash-Trie [22], Tessil Robin Hood Fast Hash Map [45], SuRF [48] for build and point lookup. We use the same hash function (Murmur[2]) for all hash-based indices to provide an accurate comparison.

Since not all indices support prefix-lookup, we compared Sonic's with TLX-BTree, ART, Hierarchical Abseil Hash Map (a hierarchical hash table which is a hash table of Abseil Hash Map tables), and Tessil HAT-Trie for prefix-lookup and count-prefix operations. Unfortunately, the current implementation of SuRF does not support prefix lookup and it only provides approximate count-prefix. So we exclude it from these operations experiments.

5.5 Build

The first step in any indexed join algorithm is building the index. Naturally, the objective is minimal build time. We evaluated the performance of building an index for 256M randomly generated tuples while varying the number of columns from 2 to 8. Figure 4 illustrates that for a two-column table, Sonic has the lowest build time as avoids any extra traversing of the levels but its performance drops by increasing the number of columns because of traversing more middle layers. Generally, trees and tries are expensive indices to build; expectedly, HAT-Trie and BTree have high build times. Hierarchical Hash Map build time grows dramatically when the number of columns increases and it needs to be expanded exponentially to allocate a separate hash table for a different prefix. Abseil Hash Set, Fast Map, and SuRF are very robust against an increased number of columns and have a minimum rise among the indices. Hash-Trie postpones building the intermediate tables for single or unaccessed tuples. So, the initial build time is robust in this experiment for a different number of columns. However, in practice the build time is higher than other competitors due to extra access to the tuples and redistributing them into the middle layers.

5.6 Point lookup

Figure 5 illustrates the point-lookup performance of different indices for ten thousand random lookups in a table. Expectedly, hash-based indices are fast for point lookup: Tessil Hash Map and Abseil Hash set perform better than others. Sonic performs well for the two-column table lookup but takes more time to find larger tuples as it needs to traverse more layers. Similarly, Hierarchical Hash Map is fast for two-column table (because it does a single lookup in a hash table) but its performance drops when the tuple size increases. BTree and Tessil HAT-Trie have high point-lookup time due to pointer chasing and their hierarchical structure. In HAT-Trie, the large number of key comparisons is responsible for the low performance. SuRF has robust point-lookup time but it is not faster than hash indices. Hash-Trie performs well in comparison to tree-based indices but as it needs to build the intermediate tables once the tuple is accessed, its lookup time is lower than other hash based indices such as Sonic.

5.7 Prefix Lookup and Count Prefix

Prefix lookup and count prefix are essential to determine the list/number of all match tuples with the given prefix in *linear time*. Consequently, we compared the indices that support these

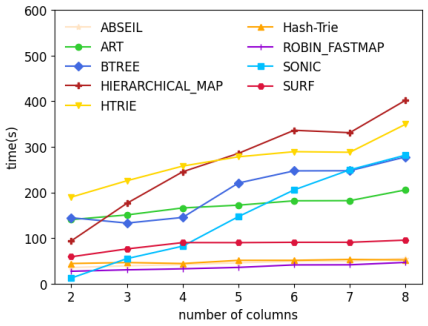

Figure 4: Build Performance

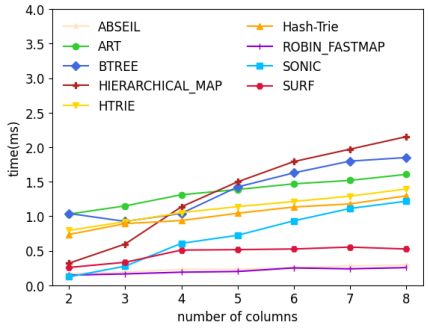

Figure 5: Point Lookup Performance

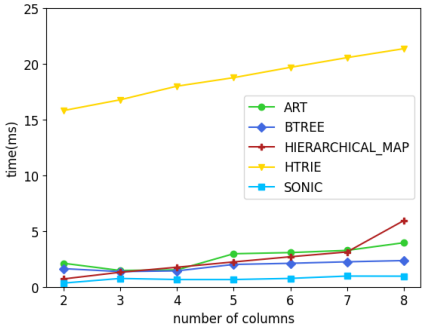

Figure 6: Prefix Lookup Performance

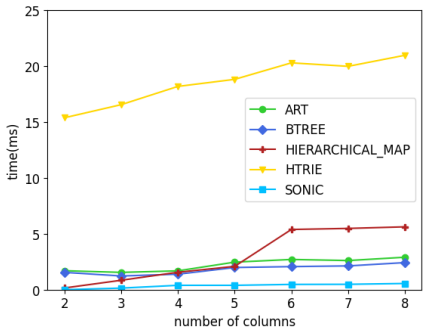

Figure 7: Count Prefix Performance

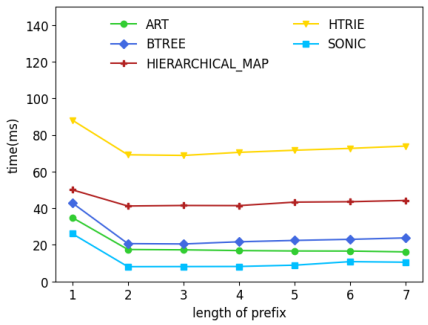

Figure 8: Performance for different Length of Prefix

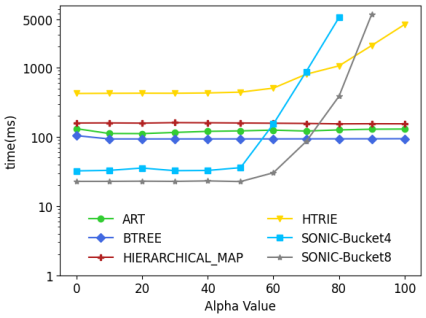

Figure 9: Prefix Lookup Performance for different alpha value (Zipfian distribution)

operation The results of performing ten thousand prefix-lookups in Figure 6 demonstrate that Sonic is the fastest among all competitors. The performance of the Hierarchical Hash Map drops as the size of the tuple increases and the chains of hash tables that need to be searched become longer. ART and BTree have a similar performance and both are faster than HAT-Trie, because of they require fewer key comparisons and they have lower cache miss rate.

Figure 7 demonstrates the results of ten thousand prefix-count queries in which Sonic is the fastest among all competitors. Hierarchical Hash Map is faster than BTree and HAT-Trie in the two-column table, as it does not need to traverse any long chains of hash tables, but as the tuple size increases its performance drops. Similar to the prefix lookup, ART and BTree have a close performance, although ART becomes slower when the tuple size increases.

5.8 Length of the prefix

In prefix-lookup operation, a longer prefix leads to fewer results. Figure 8 illustrates the performance of different indices for different prefix lengths. The Sonic index performs better when the length of the prefix is longer, as the hash lookup is faster when the key is determined. The performance of the tree and trie-based indices are largely unaffected by the length of the prefix. Since the data is almost uniformly distributed, the performance of all indices do not change significantly by increasing the length of the prefix. However, changing the data distribution could impact the lookup time.

5.9 Skew

Figure 9 shows the performance of different indices for skewed data in ten thousand prefix-lookup operations on an eight-column table. The prefix length is set to four and the alpha parameter in the Zipfian distribution has been increased from 0 to 1. The performance of the Sonic index and HAT-Trie drop for highly skewed data due to long chains of key comparisons in the leaves. Sonic with a larger bucket size performs better in prefix-lookup operation but it may impact the build time. Figure 18 illustrates the memory usage by index and as mentioned before (see Section 3.5), Sonic memory footprint is a constant factor of the data size.

5.10 Sonic Bucket Size

The size of the buckets in Sonic can be tuned for the memory allocation and performance trade-off. Large bucket size leads to a higher overallocation factor but reduces the operation time. Figure 17 shows the operation time (build, point-lookup, and prefix-lookup) on uniformly distributed data for different bucket sizes. By increasing the bucket size, build time grows but the point-lookup time and prefix-lookup time decrease due to a lower bucket overflow rate.

5.11 Sonic Parallel Build

Sonic performs concurrent inserts by acquiring locks at the granularity of multiple buckets which affects index performance. Figure 16 illustrates the build time for a varying number of threads and the lack of NUMA optimization is apparent. An interesting future work could be extending the current Sonic implementation to a NUMA-optimized version.

5.12 Variable-length keys

Sonic supports variable-length keys but the key comparisons reduce its performance (dictionary encoding techniques could solve this problem). Figure 13 illustrates the performance of Sonic,

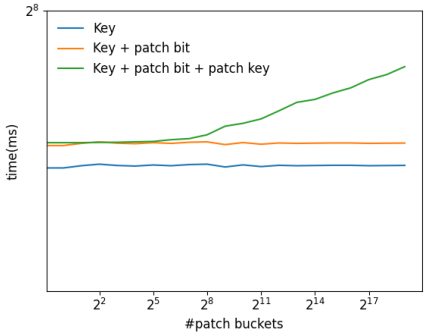

Figure 10: Lookup time for different No. patch buckets

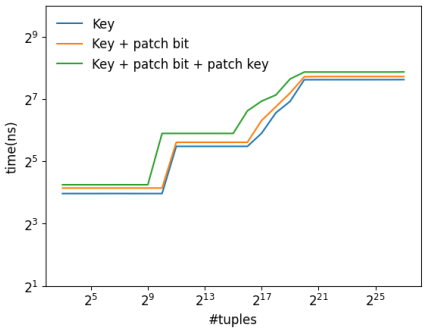

Figure 11: Lookup time for different index size

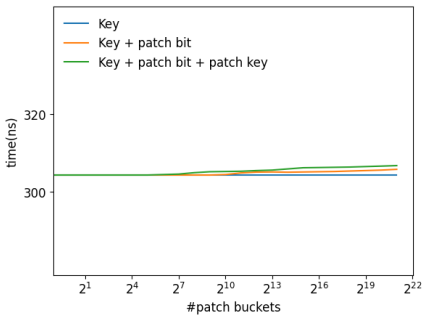

Figure 12: Insert time for different No. patch buckets

which has the worst performance compare to other indices. However, in practice most systems implement joins on strings using dictionary-encoded strings, in which case Sonic would perform as it does on an integer key.

5.13 Cache Footprint

In order to verify that the Sonic index and specifically the patch structure is cache efficient, we measured the lookup time of one million random point-lookup operations in a table that contains *16M* tuples for three different situations: only checking the hash key in the column, checking the hash key plus checking patch bit, and checking the hash key, patch bit and patch key. In the first two experiments, since no extra operation is required even if the bucket is patched, the lookup time is constant. While in the third case, we artificially set the patch bits of the buckets to 1 for an increasing number of buckets, which means the index needs to perform extra comparisons for the patch keys; thus, the lookup time increases (Figure 10).

In Figure 11 we vary the size of the index and measure the point-lookup time for random tuples in the same setup. As can be seen, as long as the index fits in the L1 cache the lookup time per tuple remains constant. When the size of the index increases, the number of cache misses increases and the lookup time per tuple rises. Since the patch structure requires more memory space, the green line (lookup time for a hash key, patch bit, and patch key) increasing point is lower than the other two cases. To conclude, these two figures demonstrate that the Sonic index patch structure is cache efficient when the index fits into the cache and when the size of the index is larger than the L3 cache, which supports the hardware-conscious design of the Sonic index. Based on the results represented in Figure 12, the computational cost of the patch structure is negligible and the disambiguating mechanism does not impose significant overhead.

5.14 Join Performance

One of our key objectives is to evaluate the performance of the established Generic Join algorithm when supported by different index structures. To this end, we evaluate the performance of the Generic Join in cycle counting problems and in join queries against EmptyHeaded [4] and Hash-Trie worst-case optimal join [22] as references. As an additional baseline, we implemented each join as a sequence of fully pipelined binary joins (we do not evaluate materializing join algorithms, e.g. radix-joins, due to their poor cache locality). The Generic Join has found application in graph processing frameworks such as EmptyHeaded or a relational DBMS such as Umbra. Consequently, we evaluated the performance of Generic Join using different indices for cycle counting in a graph. Figure 14 illustrates the performance of Generic Join with different indices for cycles in 3, 4, and 5 tables corresponding to finding triangles, rectangles, and pentagons in a graph. Each table in this experiment has *16M* rows and two columns. Sonic has been configured for the best performance and it is the fastest index in all three experiments. Hash-Trie Join's performance is very close to the Sonic index but due to extra access to tuples for build and not being able to eliminate singleton tuples without accessing the actual tuple it is slightly slower. BTree and HAT-Trie have very close performances. Hierarchical hash map, in this specific problem, performs well as the chain of hash maps is as short. EmptyHeaded runs out of memory in joining 5 tables and did not finish the experiment. This is surprising insofar as graph processing was the key workload for which EmptyHeaded was designed. However, it validates the performance of our implementation of the Generic Join using C++ templates (as well as the optimization capabilities of modern compilers). Additionally, we compared its results with a snapshot of Umbra (9be9093cc) and observed only a slight difference, which supports our purpose for the framework.

5.15 Generic Join vs Hash-Trie Join (Umbra)

As mentioned in Section 1, Hash-Trie Join, as implemented in Umbra [22], can be improved and there are circumstances that assumptions in the Hash-Trie Join approach do not lead to the optimal performance. Hash-Trie join is a specialized version of the worst-case optimal join algorithm that evaluates the query by assuming that each weight in the fractional edge cover equals 1. Under that assumption, the size of the *anchor* relation (i.e. the relation which is scanned to filter out the values without a match in other relations) is smaller than the size of the join of all other relations avoids the cost of the computations to estimate the size of that sub-problem in the Generic Join. Umbra's Hash-Trie index implements singleton pruning and lazy expansion to improve the build time. However, in cases, such as when tables are sorted differently, the removed Hash-Trie layers in the singleton pruning phase can be useful in the join processing to avoid redundant iterations. As the data becomes more skewed, Hash-Trie Join performance decreases because it requires building middle layers at run-time and traversing the Hash-Trie twice, and re-distribute the tuples. Figure 15 illustrates such a case for joining relations $R1(a, b, d, e)$, $R2(a, c, d, f)$, $R3(a, b, c)$, $R4(b, d, f)$, and $R5(c, e, f)$ and data distribution is such that cover the above description. In this case, both Sonic and Hash-Trie (Umbra) are performing better than the binary join but Sonic outperforms Hash-Trie by a factor of 2 because Umbra is not in fact worst-case optimal.

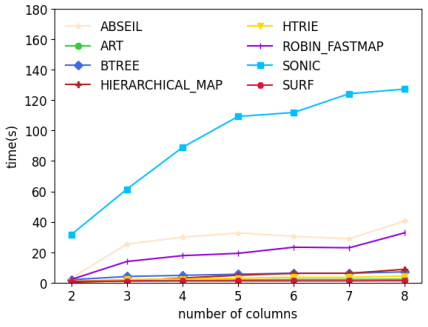

Figure 13: Build Performance for Variable-length Keys

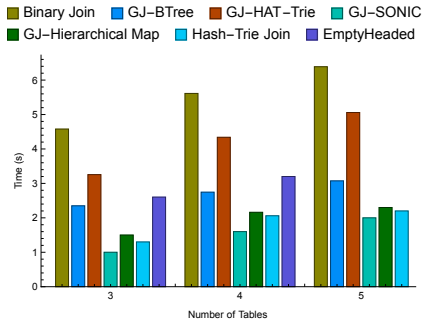

Figure 14: Cycle Counting Performance for Synthetic data

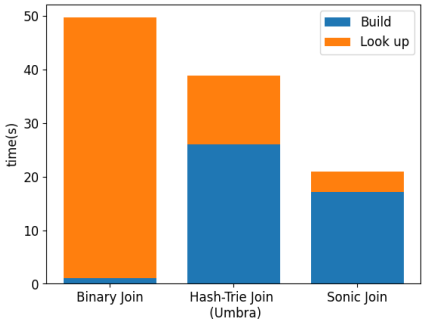

Figure 15: Sonic vs Hash-Trie Join Performance

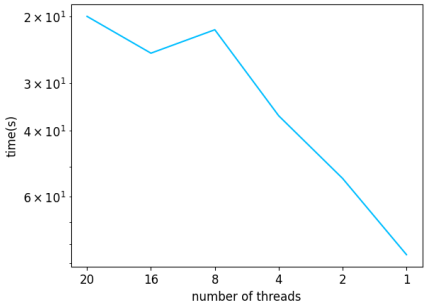

Figure 16: Sonic Parallel Build Performance

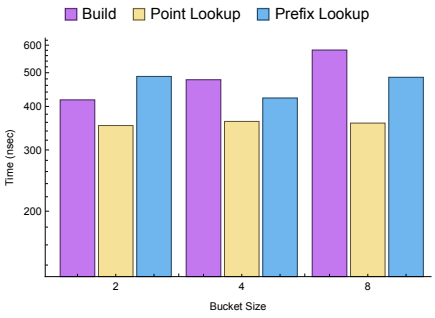

Figure 17: Sonic Bucket Size Performance

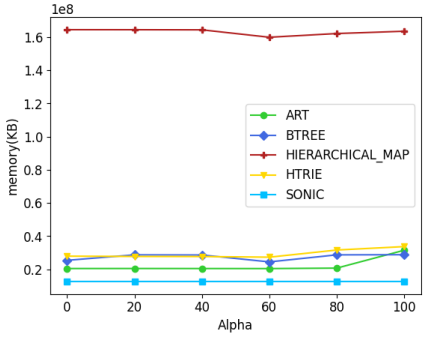

Figure 18: Memory Usage

In the case of WCOJ algorithms, the index is built ad-hoc for a single query and the build cost needs to be amortized over that query. As the breakdown shows, the total time for the WCOJ approaches is dominated by the build time while the Binary Join spends most of the time in the lookup operation.

5.16 Real Data Evaluation

To further evaluate the Generic Join, we used a real dataset, Wikipedia vote network, Epinions trusts network, Facebook and Twitter social networks data to compare the performance of the Generic Join with different indices. Table 1 illustrates triangle counting results using Generic Join (GJ) with different indices (BTree, HAT-Trie, Sonic, and Hierarchical Map) comparing to Binary Join (BJ), Hash-Trie Join (HTJ), and EmptyHeaded (EH) and the results show that Generic Join with Sonic is the fastest in most of the experiments.

We used Join Order Benchmark Light (JOB-Light) [47] data to evaluate the join algorithm for different indices. In this experiment, only joined attributes are being indexed and all combinations of tables are covered in the join queries. Table 1 demonstrates that the binary join outperforms the WCOJ algorithms, because this is not a worst-case situation.

Table 1: Cycle Counting Performance for real data (s)

Join	Facebook	Wikipedia	Epinions	Twitter	JOB
BJ	1.997	4.404	25.685	61.228	**8.57**
GJ^{BTree}	0.06	0.288	0.547	4.839	18.86
GJ^{HTrie}	0.995	1.513	8.631	20.027	21.46
GJ^{Sonic}	**0.029**	**0.042**	**0.256**	2.275	14.25
GJ^{HMap}	0.045	0.118	0.386	3.06	18.31
HTJ	0.037	0.509	0.874	2.945	13.25
EH	10.61	12.43	17.7	33.73	NA
Umbra	0.025	0.071	0.343	**2.05**	9.453

6 RELATED WORK

Ngo et al. [38] proposed a novel algorithm to process natural join queries for worst-case complexity. Their algorithm, known as NPRR, is based on the work of Atserias, Grohe, and Marx [11] and the defined optimal bound, called *AGM* bound, of the size of conjunctive queries. The NPRR algorithm relies on prefix matching of tuples and requires an index that supports the lookup operations efficiently for the algorithm's running time to satisfy the AGM bound. Despite theoretical proof of the NPRR algorithm, which has been used in other inequalities[16, 21, 33], no practical implementation for the algorithm was provided. Ngo et al., by delivering the NPRR algorithm and their extensive theoretical work, provided a foundation for subsequent work on worst-case optimal join (WCOJ) algorithms such as "beyond worst-case algorithms" [6, 8, 14, 29, 37] and operators beyond joins [4, 26–28].

Veldhuizen [46], claimed that the NPRR algorithm presented by Ngo et al. in [38] is not, in fact, worst-case optimal and proposed a new approach, called Leapfrog Triejoin. Later, Ngo et al. [39] proved both algorithms are special cases of a general algorithm which had been proposed in a study by Ngo et al. [40]. Nguyen et al. [41] examined both Leapfrog Triejoin and the join algorithm presented in [38] on graph datasets, and both methods have similar performance.

Various implementations for the WCOJ algorithm have been proposed for different hardware and applications but their limitations, such as being query-specific or expensive precomputation for persistent indexes, prevent their exploitation [3–5, 20]. Recently, Freitag et al. [22] proposed a new hash-trie-based WCOJ algorithm and employed it within the Umbra DBMS [36] for both graph processing applications and general-purpose join queries. Furthermore, they developed a hybrid query optimizer to combine the classic binary and the new WCOJ within the same query

plan. However, their approach is a specialization of the Generic Join [39] algorithm and does not take into consideration the AGM bound for the sub-problems.

Despite always being worst-case optimal, WCOJ algorithms are highly dependent on the order of join attributes. The NPRR [38] algorithm employs a heuristic approach while others take different approaches such as cost-based heuristic models [9, 46], or relying on built-in optimization models within the DBMS [22].

With respect to indexing our work is related to hash tables that provide fast point lookup, but do not support prefix or range queries, as opposed to trees or tries. Intending to improve the classical simple probing methods and reduce the number of collisions, Robin Hood hashing [18] has been proposed. Facebook F-14 table [44] is a 14-way probing hash table which is memory and CPU efficient. Tries offer a trade-off that uses more memory for the sake of faster lookups than trees. Askitis et al. [10] proposed a cache-conscious trie structure for variable-length string management. Recently, Zhang et al. [48] designed a new data structure, Fast Succinct Trie, and developed a Succinct Range Filter that supports both point lookups and range lookups. Freitag et al.[22] used a built-in hash-trie data structure in Umbra for WCOJ algorithm. Although the index is not customized for WCOJ algorithms and some of the optimization techniques impose negative effects on the join performance.

7 FUTURE WORK

Improving multi-threaded version and optimizing the number of locks based on the workload is very beneficial. Additionally, balancing the workload among the threads using methods such as work-stealing would be interesting.

A second line of work would be to broaden the usefulness of the Sonic index by auto-configuring for optimal performance with respect to the query and data type. Furthermore, it can be used for different applications. Specifically, the Leapfrog Trie Join algorithm requires a trie-like interface to an index structure. Such an interface could be provided in a straight-forward manner by sorting the input before building Sonic.

8 CONCLUSION

While worst-case optimal join algorithms live up to their name in terms of robustness, their performance is significantly worse than that of classic join algorithms. One of the key components to making them performance-competitive is an index structure that performs all the operations fast. To contribute such an index, we developed *Sonic*that combines a fast build phase with best-of-breed lookup performance. Supported by Sonic-indices, the worst-case optimal Generic Join algorithm performs up to 2.5 times better than using Sonic's fastest competitor. This performance gain transforms the Generic Join algorithm from an approach that is "academically interesting" to one that is "practically useful".

REFERENCES

[1] 2017. A cache-conscious trie 2017. https://tessil.github.io/2017/06/22/hat-trie.html

[2] aappleby Aappleby. 2016. Aappleby/smhasher: Automatically exported from code.google.com/p/Smhasher. https://github.com/aappleby/smhasher

[3] Christopher R Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. 2017. LevelHeaded: Making Worst-Case Optimal Joins Work in the Common Case. *arXiv preprint arXiv:1708.07859* (2017).

[4] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* (2017).

[5] Christopher R Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. Old techniques for new join algorithms: A case study in RDF processing. In *2016 IEEE 32nd International Conference on Data Engineering Workshops (ICDEW)*. IEEE.

[6] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. 2016. FAQ: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*.

[7] Abseil Abseil. 2017. Abseil/Abseil-CPP: Abseil Common Libraries (C++). https://github.com/abseil/abseil-cpp

[8] Kaleb Alway, Eric Blais, and Semih Salihoglu. 2019. Box covers and domain orderings for beyond worst-case join processing. *arXiv preprint arXiv:1909.12102* (2019).

[9] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM.

[10] Nikolas Askitis and Ranjan Sinha. 2007. HAT-trie: a cache-conscious trie-based data structure for strings. In *Proceedings of the thirtieth Australasian conference on Computer science-//volume 62*. Australian Computer Society, Inc.

[11] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size bounds and query plans for relational joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. IEEE.

[12] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size bounds and query plans for relational joins. *SIAM J. Comput.* (2013).

[13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE.

[14] Paul Beame, Paraschos Koutris, and Dan Suciu. 2016. Worst-Case Optimal Algorithms for Parallel Query Processing. *arXiv preprint arXiv:1604.01848* (2016).

[15] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. https://panthema.net/tlx, retrieved Oct. 7, 2020.

[16] Béla Bollobás and Andrew Thomason. 1995. Projections of bodies and hereditary properties of hypergraphs. *Bulletin of the London Mathematical Society* (1995).

[17] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. 1999. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*.

[18] Pedro Celis, Per-Ake Larson, and J Ian Munro. 1985. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. IEEE.

[19] Tzi-cker Chiueh and Dhruv Pilania. 2005. Design, implementation, and evaluation of a repairable database management system. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE.

[20] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*.

[21] Fan RK Chung, Ronald L Graham, Peter Frankl, and James B Shearer. 1986. Some intersection theorems for ordered sets and graphs. *Journal of Combinatorial Theory, Series A* (1986).

[22] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment* (2020).

[23] Goetz Graefe. 2012. New algorithms for join and grouping operations. *Computer Science-Research and Development* (2012).

[24] Oded Green, Robert McColl, and David A Bader. 2012. GPU merge path: a GPU merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*.

[25] Martin Grohe and Dániel Marx. 2014. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms (TALG)* (2014).

[26] Manas Joglekar, Rohan Puttagunta, and Christopher Ré. 2015. Aggregations over generalized hypertree decompositions. *arXiv preprint arXiv:1508.07532* (2015).

[27] Manas R Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*.

[28] Mahmoud Abo Khamis, Ryan R Curtin, Benjamin Moseley, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Functional Aggregate Queries with Additive Inequalities. *ACM Transactions on Database Systems (TODS)* (2020).

[29] Mahmoud Abo Khamis, Hung Q Ngo, Christopher Ré, and Atri Rudra. 2016. Joins via geometric resolutions: Worst case and beyond. *ACM Transactions on Database Systems (TODS)* (2016).

[30] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* (2015).

[31] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases.. In *ICDE*.

[32] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[33] Lynn H Loomis and Hassler Whitney. 1949. An inequality related to the isoperimetric inequality. *Bull. Amer. Math. Soc.* (1949).

[34] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment* (2019).

[35] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* (2011).
[36] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*.
[37] Hung Q Ngo, Dung T Nguyen, Christopher Re, and Atri Rudra. 2014. Beyond worst-case analysis for joins with minesweeper. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*.
[38] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case Optimal Join Algorithms. *arXiv preprint arXiv:1203.1952* (2012).
[39] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *Journal of the ACM (JACM)* (2018).
[40] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. *arXiv preprint arXiv:1310.3314* (2013).
[41] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q Ngo, Christopher Ré, and Atri Rudra. 2015. Join processing for graph patterns: An old dog with new tricks. In *Proceedings of the GRADES'15*. ACM.
[42] Thomas Schank and Dorothea Wagner. 2005. Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms*. Springer.
[43] Nikita Shamgunov. 2014. The MemSQL In-Memory Database System.. In *IMDM@ VLDB*.
[44] Nathan BronsonXiao Shi, Nathan Bronson, and Xiao Shi. 2019. Open-sourcing F14 for memory-efficient hash tables. https://engineering.fb.com/developer-tools/f14/
[45] Tessil. 2019. Tessil/robin-map. https://github.com/Tessil/robin-map
[46] Todd L Veldhuizen. 2012. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481* (2012).
[47] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2020. Are we ready for learned cardinality estimation? *arXiv preprint arXiv:2012.06743* (2020).
[48] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. ACM.