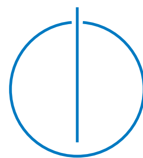
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Index-Structures for Worst-Case Optimal Join Algorithms

Tobias Schmidt



DEPARTMENT OF INFORMATICS

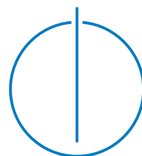
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Index-Structures for Worst-Case Optimal Join Algorithms

Indexstrukturen für worst-case optimale Joinalgorithmen

Author: Tobias Schmidt
Supervisor: Prof. Dr. Thomas Neumann
Advisor: Michael Freitag, M.Sc.
Submission Date: 15.10.2019



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, Tobias Schmidt

Abstract

Query evaluation is a key component of every database management system. Complex queries often require joins involving more than two relations. Standard query evaluation plans split such queries into successive pairwise joins. This approach, however, does not guarantee worst-case optimality, i.e., the runtime of the query evaluation plan is not asymptotic to the maximum result size of the query. Over the past ten years, research has intensified in the area of worst-case optimal join algorithms. Our work is based on the results of Ngo, Porat, Ré, and Rudra, who designed a join algorithm with worst-case running time. For the LEAPFROG TRIEJOIN, a worst-case optimal join algorithm implemented by LogicBlox, we present four optimized index-structures. The index-structures are evaluated with regard to their performance and applicability in practice. Furthermore, we developed two new variants of the LEAPFROG TRIEJOIN: the HASHBASED TRIEJOIN and the HYBRID TRIEJOIN. Both variants reduce runtime complexity and allow further optimizations. For the new join algorithms, we introduce and evaluate two index-structures. Finally, we present and analyze the results of several experiments and discuss the strengths and limitations of the join algorithms and their index-structures.

Contents

Abstract **iii**

1. Introduction **1**

2. Join Algorithms **3**
2.1. Pairwise Join Plans and the Triangle Query 3
2.2. Generic Worst-Case Optimal Join Algorithm 5
2.3. Leapfrog Triejoin 7
2.4. Hashbased Triejoin 11
2.5. Hybrid Triejoin 14

3. Index-Structures **17**
3.1. Sorted Index-Structures 17
3.1.1. Flat Array 17
3.1.2. Trie Array 19
3.1.3. Bitset Array 21
3.1.4. B-Tree 23
3.2. Unsorted Index-Structures 25
3.2.1. Multichain Hash Table 25
3.2.2. Hybrid Hash Table 28

4. Experiments **31**
4.1. Queries 31
4.1.1. Interleaved Query 31
4.1.2. Hypercube Query 32
4.1.3. Twitter Query 32
4.2. Setup 33
4.3. Evalution 34
4.3.1. Interleaved Query 34
4.3.2. Hypercube Query 36
4.3.3. Twitter Query 38
4.4. Discussion 40

5. Future Work **43**
5.1. Hashbased Triejoin and Multichain Hash Table 43
5.2. Hybrid Triejoin and Hybrid Hash Table 44

6. Related Work **45**

7. Conclusion **47**

Appendix **49**

A. Algorithms **50**
A.1. Leapfrog Triejoin 51
A.2. Hashbased Triejoin 52
A.3. Hybrid Triejoin 53

List of Figures **55**

List of Tables **57**

Bibliography **59**

1. Introduction

With the rise of big data, social networks, and graph databases, the efficient evaluation of database queries on large datasets is becoming increasingly important. The standard approach for evaluating database queries are *query evaluation plans*. A query evaluation plan specifies, in particular, the order in which the relations are joined (*join order*). The join order can be decisive for the runtime of the query. Finding an efficient join order is the task of the query optimizer. Ideally, the execution time of the best evaluation plan is asymptotic to the result size of the query. However, even if an efficient join plan exists, there is no guarantee that the query optimizer will find it [1].

Current database systems mainly use pairwise join algorithms, i.e., algorithms that obtain two input relations and produce one output relation [2, ch. 18]. However, there exist worst-case scenarios in which any order of pairwise joins is suboptimal, i.e., not asymptotic to the result size of the query.

These problems intensified research in the field of worst-case optimal join algorithms that have execution times asymptotic to the maximum result size of the query. In 2012 Ngo et al. presented a worst-case optimal join algorithm [3] which is asymptotically faster for some queries than query evaluation plans based on pairwise joins. Another worst-case optimal join algorithm is the Leapfrog Triejoin [4]. The commercial database system LogicBlox implemented this algorithm in 2009 [5].

Like some pairwise join algorithms, the Leapfrog Triejoin uses specialized index-structures. These structures provide efficient access to materialized relations and strongly influence the runtime of the algorithm. In this thesis, we present several index-structures for the Leapfrog Triejoin and for two variants of the algorithm (the Hashbased Triejoin and the Hybrid Triejoin). Three benchmarks compare the algorithms and analyze the performance of the different index-structures.

The thesis is organized as follows. Chapter 2 introduces a motivating example and provides a detailed description of the worst-case optimal join algorithms. Chapter 3 presents the index-structures for the Leapfrog Triejoin, the Hashbased Triejoin, and the Hybrid Triejoin. Chapter 4 describes and analyzes the results of the conducted experiments. In the last three chapters, we present future work and related work and draw a conclusion.

2. Join Algorithms

This chapter first gives an example of a query for which no optimal query evaluation plan based on pairwise join algorithms exists. Afterward, a generic representation for worst-case optimal join algorithms developed by Ngo, Ré, and Rudra in [6] is presented and analyzed. The following sections discuss the LEAPFROG TRIEJOIN, the HASHBASED TRIEJOIN, and the HYBRID TRIEJOIN. Compared to query evaluation plans based on pairwise joins, these algorithms guarantee runtimes asymptotic to the maximum result size of the query. Besides, the interfaces for the index-structures are specified, and runtime constraints are established.

2.1. Pairwise Join Plans and the Triangle Query

The standard approach for evaluating queries in database systems are pairwise join plans. Such plans break the query into a sequence of pairwise joins and executes them in a tree-like order. Figure 2.1 shows a right-deep join plan for the query $R \bowtie S \bowtie T$. First, it calculates the intermediate result $R \bowtie S$, which is then joined with the third relation T. Pairwise join algorithms are, for example, the BLOCK-NESTED LOOP JOIN, HASH-JOIN, and SORT-MERGE JOIN. [2, ch. 8]

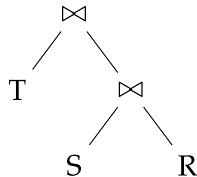


Figure 2.1.: Right-deep join plan for the query $R \bowtie S \bowtie T$

The HASH-JOIN obtains two relations: a build input and a probe input. The build input is smaller than the probe input. First, the algorithm indexes the build input using a hash table (build phase). In the second step, the algorithm iterates the probe input and searches for every tuple a join partner in the hash table (probe phase). In the right-deep join plan, R is the probe input for the first join, and S is the build input. The second join obtains T as build input and the intermediate result $R \bowtie S$ as probe input. The cost of the build phase is reduced by indexing the two smallest relations (S and T). [2, ch. 8]

Modern database systems optimize this approach by pipelining the join plan. Instead of materializing the intermediate result $R \bowtie S$, a tuple from R is passed through the entire join tree. In the given example, the tuple first searches for a join partner in the

```
SELECT R.a, S.b, T.c
FROM R, S, T
WHERE R.b = S.b
AND S.c = T.c
AND T.a = R.a
```

Figure 2.2.: SQL representation of the triangle query $Q_\triangle$

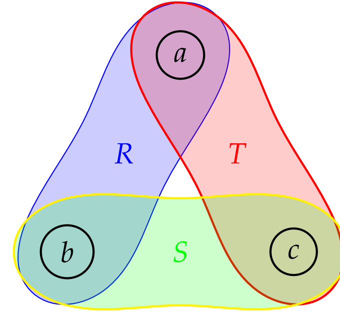


Figure 2.3.: Hypergraph representation of the triangle query $Q_\triangle$

hash table of the relation S. If a join partner is found, it immediately probes the second hash table. In the case of a match, the algorithm adds the joined tuple to the result set [2, ch. 18]. Chapter 4 compares worst-case optimal join algorithms with pipelined right-deep join plans based on the Hash-Join algorithm.

An inherent problem with pairwise join plans is the size of the intermediate result. In general, it can be larger than the query result and the input relations by orders of magnitude. The so-called triangle query $Q_\triangle$ illustrates this problem. The query interprets the edges in a directed Graph $G = (V, E)$ as binary relations $R(a, b)$, $S(b, c)$ and $T(c, a)$ and performs a natural join:

$$Q_\triangle = R \bowtie S \bowtie T \tag{2.1}$$

Figure 2.2 and Figure 2.3 show the representation of query $Q_\triangle$ in SQL notation and as hypergraph. The three relations are identical to the set of edges E: $R = S = T = E$. A tuple (x, y) represents an edge from node x to node y. The size of each relation equals the number of edges in graph G: $|R| = |S| = |T| = |E| = n$.

From a graphical perspective, the query searches for paths of length three in the graph G with the same start and end node, i.e., triangles. Atserias, Grohe, and Marx established in [7] a tight upper bound for the size of the result: $\mathcal{O}(n^{3/2})$.

For the runtime of a query execution plan using pairwise joins, however, the size of the intermediate result is equally important. Since the three relations are identical, the join order does not matter, and the join plan from Figure 2.1 can be used. First, the plan performs the join $R \bowtie S$. For every edge in the graph G, it searches for a second edge that starts where the other one ended. Therefore, the intermediate result lists all paths of length two in the graph. There exist $\mathcal{O}(n^2)$ paths of length two. The second join searches for a third edge that connects the end node with the start node of the path. Consequently, the runtime of the pairwise join plan is bounded by the size of the intermediate result $\mathcal{O}(n^2)$ and not by the size of the result set $\mathcal{O}(n^{3/2})$.

The following instance of the three relations $R(a, b)$, $S(b, c)$ and $T(c, a)$ illustrates this fact. The domain of the attributes a, b and c is $\{0, 1, \dots, m\}$ with $m \in \mathbb{N}$. R, S and T are

defined as following (a subset of the hypercube dataset from Section 4.1.2):

$$R(a,b) = S(b,c) = T(c,a) = \{0,1,\ldots,m\} \times \{0\} \cup \{0\} \times \{0,1,\ldots,m\} \tag{2.2}$$

Then, the intermediate result $Q_{R \bowtie S}$ and the final result $Q_{\triangle}$ are of the form:

$$Q_{R \bowtie S}(a,b,c) = \{0,1,\ldots,m\} \times \{0\} \times \{0,1,\ldots,m\} \cup \{0\} \times \{0,1,\ldots,m\} \times \{0\} \tag{2.3}$$

$$\begin{aligned} Q_{\triangle}(a,b,c) = & \{0\} \times \{0\} \times \{0,1,\ldots,m\} \cup \{0\} \times \{0,1,\ldots,m\} \times \{0\} \\ & \cup \{0,1,\ldots,m\} \times \{0\} \times \{0\} \end{aligned} \tag{2.4}$$

The size of R, S, and T is $n = 2m + 1$, $|Q_{R \bowtie S}| = (m+1)^2 + m \in \Theta(n^2)$ and $|Q_{\triangle}| = 3m + 1$. Any pairwise join plan for the three relations has execution times in $\Omega(n^2)$ and is, therefore, not worst-case optimal for the triangle query $Q_{\triangle}$.

2.2. Generic Worst-Case Optimal Join Algorithm

Unlike pairwise join plans, a worst-case optimal join algorithm achieves execution times asymptotic to the maximum result size of the query. In the case of the triangle query $Q_{\triangle}$, $\mathcal{O}(n^{3/2})$ is an upper bound for the size of the query result and for the execution time. A pairwise join plan for the triangle query performs the join along the relations/edges in a graph. It first enumerates all paths of length 2 and then filters the intermediate result for triangles.

Worst-case optimal join algorithms follow a different approach. Instead of decomposing the query by relations, they sequentially determine the possible values for the join attributes. The result set of the triangle query $Q_{\triangle}$ for a directed graph $G = (V, E)$ consists of triples $(a,b,c) \in V^3$ with edges from a to b, from b to c, and from c to a. The three relations $R(a,b)$, $S(b,c)$ and $T(c,a)$ are identical to the set of edges E in G, i.e., $E = R = S = T$.

The following notation is used: $T'(a,c) = \{(a,c) \colon (c,a) \in T\}$ denotes the inverse relation of $T(a,c)$; For a parameter x, $R[x]$ is the filtered version of R, i.e., $\{(x,b) \colon (x,b) \in R\} = R \ltimes x$.

A worst-case optimal join algorithm performs the following steps for the triangle query $Q_{\triangle}$:

- The algorithm first searches possible nodes for attribute a. Every node has to possess at least one incoming and one outgoing edge:

$$a \in \pi_a(R) \cap \pi_a(T')$$

- For every possible node a, the algorithm then searches possible nodes for attribute b. Every node has to satisfy two conditions: an incoming edge from node a exists, and there is at least one outgoing edge. In other words:

$$b \in \pi_b(R[a]) \cap \pi_b(S)$$

  If the intersection is empty, the algorithm proceeds with the next value for a. Otherwise, it continues with the last attribute c in the triple.

- At this point, the algorithm has found nodes a and b with an edge from a to b and there exists at least one edge that ends in a and at least one edge that starts in b. Now the algorithm has to find all nodes c that connect b with a, i.e., there exists an edge from b to c and from c to a:

$$c \in \pi_c(S[b]) \cap \pi_c(T'[a])$$

The triples (a, b, c) are added to the result set, and the algorithm proceeds with the next value for b.

Ngo, Ré, and Rudra generalized this approach in [6] to a recursive worst-case optimal join algorithm. The in-memory query processing engine LevelHeaded demonstrated the effectiveness of the algorithm. Algorithm 1 shows the Generic-Join algorithm used in LevelHeaded [8]. It was slightly modified to improve readability.

Algorithm 1 Generic-Join(V_Q, E_Q, t)

1: $I = \{v_1\}$ *with* v_1 *first element in* V_Q
2: $E_I := \{e \in E_Q \colon e \cap I \neq \emptyset\}$
3: $L := \bigcap_{e \in E_I} \pi_I(R_e[t])$
4: **if** $|V_Q| = 1$**:**
5: **return** L
6: **else:**
7: $Q := \emptyset$
8: **for** $t_v \in L$**:**
9: $Q_t :=$ Generic-Join($V_Q - I$, E_Q, $t \circ t_v$)
10: $Q := Q \cup \{t_v\} \times Q_t$
11: **end for**
12: **return** Q
13: **end if**

The Generic-Join algorithm obtains the hypergraph representation $G_Q = (V_Q, E_Q)$ of the query and a tuple t. The tuple is initially empty. Figure 2.3 shows the hypergraph representation of the triangle query. The set of vertices V_Q in G_Q contains all attributes that appear in the SQL query. If two attributes are part of an equi-join condition, they are mapped to the same vertex. The relations in the query correspond to the hyperedges in E_Q. An hyperedge connects all attributes from one relation that appear in the query. R_e denotes the relation that belongs to the hypergraph edge e. The expression $R_e[t]$ filters the elements in the relation R_e for the parameters from the tuple t: $R_e[t] = R_e \ltimes t$.

The first three lines determine some values needed in this step:

- I: attribute considered in this step
- E_I: relations containing the attribute I
- L: possible values for the attribute I

If V_Q contains only one attribute, the recursion ends and set of possible values for I is returned. Otherwise, the algorithm determines the values for the remaining attributes. It iterates over every possible value for the current attribute and invokes the GENERIC-JOIN. The recursive call obtains the remaining attributes and the extended tuple t. The result of the recursive call is added to the result of the current recursion step. If the algorithm cannot find any values for the remaining attributes, it returns an empty set. The set V_Q specifies the so-called *global attribute order*. For this purpose, the algorithm interprets V_Q as ordered set. The attributes are evaluated according to this order.

The efficient calculation of the set intersection $\bigcap_{e \in E_I} \pi_I(R_e[t])$ is a prerequisite for worst-case optimality [6]. Furthermore, the algorithm has to determine the filtered elements $R_e[t]$ efficiently. A trie-like index-structure offers this property. The following algorithms differ in the method used to calculate the set intersection and the employed index-structures.

2.3. Leapfrog Triejoin

LogicBlox is one of the first commercial database systems to implement a worst-case optimal join algorithm. Veldhuizen described the so-called LEAPFROG TRIEJOIN developed by LogicBlox in [4] and proved its worst-case optimality (up to a log-factor). The algorithm is an instance of the worst-case optimal GENERIC-JOIN algorithm. It uses a variant of the SORT-MERGE join algorithm to calculate the set intersection from Algorithm 1 (line 3). The input relations are sorted according to the global attribute order and are allocated in a trie-like index-structure.

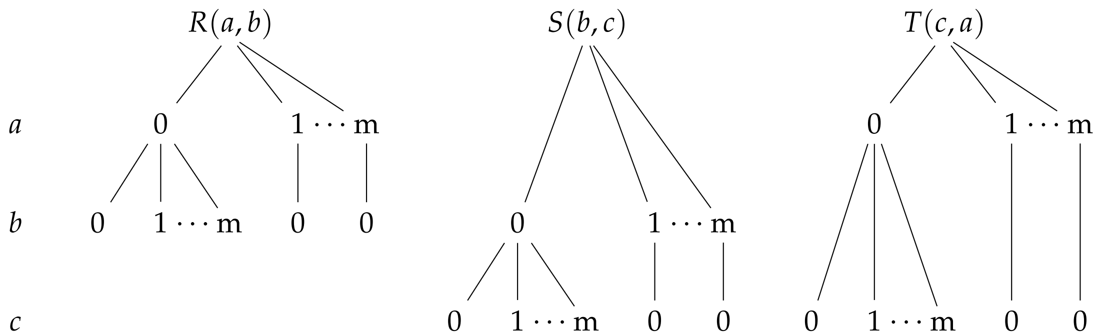


Figure 2.4.: Trie representation for an instance of $R(a,b)$, $S(b,c)$ and $T(c,a)$ from Equation 2.2 (the global attribute order is $[a,b,c]$)

Figure 2.4 shows the relations from Equation 2.2 as tries. The trie divides the relations into several levels according to the global attribute order. For example, the relation $T(c, a)$ consists of tuples (c, a). Therefore, a 2-level trie is required. The first level stores all distinct values for the attribute $a \in \pi_a(T)$ in ascending order. The children of the inner nodes are all values for attribute c, with $(c, a) \in T$. A path from the root to a leaf node represents a tuple from T.

Veldhuizen proposed a simple interface in Table 2.1 for navigating the levels of a trie and traversing the elements [4]. A trie iterator over the materialized input relations must comply with the runtime bounds from Table 2.1; n is the number of tuples in the indexed relation. The functions `next()` and `seek()` traverse the trie horizontally, and the `open`- and `up`-functions allow vertical navigation. Initially, the iterator is positioned at the root of the trie.

Table 2.1.: Trie iterator interface for sorted datasets

Function	Description	Complexity
`key()`	Returns the element at the current iterator position.	$\mathcal{O}(1)$
`next()`	Advances the iterator to the next element.	$\mathcal{O}(\log(n))$
`seek(key)`	Moves the iterator to the least upper bound for `key` or positions the iterator at the end if no such element is found.	$\mathcal{O}(\log(n))$
`atEnd()`	Returns `true` if the iterator is at the end, i.e., all elements on the current level have been visited.	$\mathcal{O}(1)$
`open()`	Proceeds to the smallest child on the next level.	$\mathcal{O}(\log(n))$
`up()`	Returns to the parent on the previous level.	$\mathcal{O}(\log(n))$

The following sequence of operations moves the iterator through the trie of $R(a, b)$. Invoking the `open`-function two times positions the iterator first at $(0, _)$, and then at $(0, 0)$. The `seek(m)` method moves the iterator to $(0, m)$. A subsequent `next()` call results in the iterator being `atEnd()`. The `up`-function returns it to $(0, _)$. The combination of `next()` and `open()` moves the iterator to $(1, _)$ and then positions it at the leaf node $(1, 0)$.

The `seek`-function must obtain a key that is equal or greater than the element at the current iterator position. The `open`-, `next`-, `seek`- and `key`-functions perform the expected operations only if the iterator is not at the end. The `up`-function returns the iterator to the parent node. Invoking `up()` at the root or `open()` at a leaf node has no effect on the iterator.

Figure 2.5 shows an extract of the LEAPFROG TRIEJOIN algorithm. The complete implementation is available in Appendix A in Figure A.1. The `leapfrog-search-`

```
globals: array Iter, integer size, boolean atEnd, integer pos, integer depth

leapfrog-search():
    max_key:= Iter[(pos - 1) mod size].key()
    while true:
        min_key := Iter[pos].key()
        if max_key == min_key:
            return
        else:
            Iter[pos].seek(max_key)
            if Iter[pos].atEnd():
                atEnd := true
                return
            end if
            max_key := Iter[pos].key()
            pos := (pos + 1) mod size
        end if
    end while
end

leapfrog-init():
    size := Iter.size()
    if ∃p ∈ [size]: Iter[p - 1].atEnd():
        atEnd := true
    else:
        atEnd := false
        pos := 0
        sort Iter by key() of each iterator
        leapfrog-search()
    end if
end
```

Figure 2.5.: Leapfrog Triejoin implementation

function is the basic building block of the Leapfrog Triejoin algorithm. It is responsible for calculating the set intersection from the Generic-Join algorithm. However, it does not materialize the result but determines with every call the next element in the intersection. The `leapfrog-init`-function is called once at the beginning and sorts the array `Iter` in ascending order. The `Iter` array stores references to every trie iterator for the considered attribute. The elements in the relations are accessed using the functions `next()`, `seek()`, and `key()`.

Figure 2.6 shows the principle of the `leapfrog-search`-function. It intersects/joins three sorted datasets A, B, and C using a variant of the Sort-Merge join algorithm called Leapfrog Join. First, the three relations are sorted according to the value of the smallest element. In the given example, the `Iter` array changes its order from $[A, B, C]$ to $[B, A, C]$. Hereafter, the algorithm searches for the first element in the intersection. The iterator for B performs a `seek(2)` and advances to 3; the iterator for A performs a `seek(3)` and advances to 5; the iterator for C performs a `seek(5)` and advances to 7. This is repeated until all iterators are positioned at the same key. For finding the next key in the intersection, one iterator is moved forward using `next()`, and the iterators perform `seek`-operations until a match is found or one iterator is at the end.

The Leapfrog Join algorithm intersects arbitrarily many unary relations $A_0, \ldots, A_{(k-1)}$ in

$$\mathcal{O}(\min(|A_0|, \ldots, |A_{(k-1)}|) \cdot \log(\max(|A_0|, \ldots, |A_{(k-1)}|) / \min(|A_0|, \ldots, |A_{(k-1)}|))).$$

Veldhuizen proved this upper-bound in [4]. The logarithmic complexity of the `seek`-

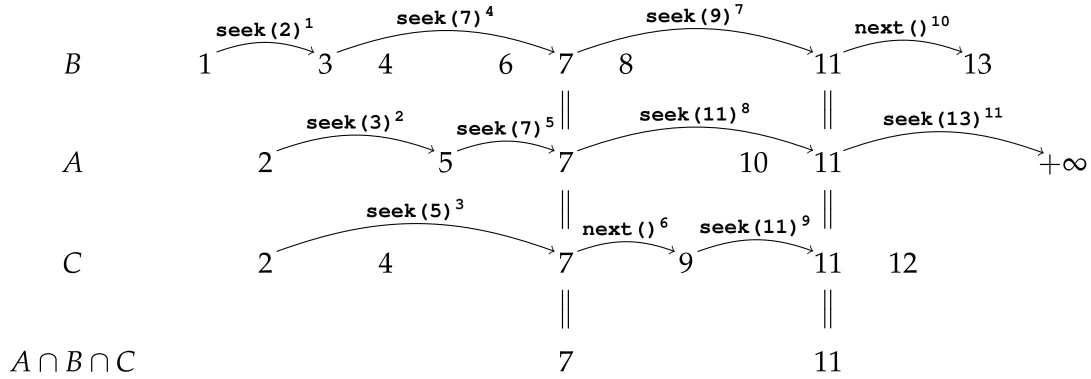


Figure 2.6.: LEAPFROG JOIN for three arbitrary relations A, B and C. The superscript number denotes the order in which the operations are executed; $+\infty$ indicates that the iterator is at the end.

function causes the log-factor in the runtime bound.

The smallest iterator is moved forward until it equals the iterator with the largest key or is larger. At the beginning of every iteration, the following two invariants hold:

$$A_{\texttt{pos}}\texttt{.key()} = \min(A_0\texttt{.key()},\ldots,A_{(k-1)}\texttt{.key()})$$
$$A_{\texttt{(pos - 1) mod size}}\texttt{.key()} = \max(A_0\texttt{.key()},\ldots,A_{(k-1)}\texttt{.key()})$$

The iterators must be sorted in ascending order so that the invariants hold at the beginning (after calling the `open`-function). The function `leapfrog-init()` ensures this. The `leapfrog-search`-function ends if one iterator is at the end, or all iterators point to the same element:

$$A_0\texttt{.key()} = \cdots = A_{(k-1)}\texttt{.key()}$$

The LEAPFROG TRIEJOIN algorithm provides the result of the join query not as materialized dataset, but as trie iterator. It implements the same interface functions as the iterators for the materialized input relations. The functions `next()` and `seek()` move the smallest iterator forward and then call `leapfrog-search()` in order to find the next match. The variable `atEnd` is accessed by the corresponding function, and the `key`-function returns the value of the current match. The functions `open()` and `up()` select the correct trie iterators for each level and open/close them. The implementation of these functions is shown in Appendix A in Figure A.1.

The maximum result size of the query is an upper bound (up to a log-factor) for the time required to extract the data using the interface functions. The log-factor reflects the cost of seeking for an upper bound in sorted data. Most trie iterators for materialized relations implement this operation using binary search. The trie iterator for the query result provides the joined data in ascending order and does not materialize any immediate or final result. In Veldhuizen's paper, the input relations are allocated in B-Trees. Section 3.1 presents further index-structures that support trie iterator interface.

2.4. Hashbased Triejoin

A fundamental assumption in the Leapfrog Triejoin is that all relations are sorted according to the global attribute order. In general, this will not be the case. Keeping the relations sorted at any time increases the costs for insert, update, and delete operations. In addition, different join queries can impose different variable orderings. Therefore, sorting the relations before joining them may be required. It is well known that the worst-case lower bound for comparison-based sorting algorithms is $\Omega(n \cdot \log(n))$ [9, p. 193]. Consequently, sorting the relations may have a noticeable impact on the execution time of the join.

An index-structure whose creation is linear to the size of the materialized relations in combination with a worst-case optimal join algorithm may reduce this impact. The following extension of the Leapfrog Triejoin, the so-called Hashbased Triejoin, processes unordered relations and is worst-case optimal. Moreover, there is no need for sorting the input relation. Thus, no sorting algorithm is needed for building the index-structures. The iterator interface for the Leapfrog Triejoin was slightly altered to support unsorted index-structures:

Table 2.2.: Trie iterator interface for unsorted datasets

Function	Description	Complexity
`key()`	Returns the element at the current iterator position.	$\mathcal{O}(1)$
`next()`	Advances the iterator to the next element.	$\mathcal{O}(1)$
`jump(key)`	Searches for the given `key`; if the element is found, the function moves the iterator to the element and returns `true`. Otherwise, it returns `false`.	$\mathcal{O}(1)$
`length()`	Returns the expected amount of work to traverse all elements.	$\mathcal{O}(1)$
`atEnd()`	Returns `true` if the iterator is at the end, i.e., all elements on the current level have been visited.	$\mathcal{O}(1)$
`open()`	Proceeds to the first (not necessarily smallest) child on the next level.	$\mathcal{O}(1)$
`up()`	Returns to the parent on the previous level.	$\mathcal{O}(1)$

For the Hashbased Triejoin, `jump()` replaces the `seek`-function. The `jump`-function searches for the given key and returns the result. Furthermore, it repositions the iterator to the obtained key. The `length`-function returns the expected amount of work to traverse all elements. In the case of materialized relations, this value corresponds to the number of children of the parent node.

Ken Ross proposed in [4] an unsorted index-structure that supports the required

iterator functions: His idea for a relation $R(a,b)$ is to maintain a hash table for $\pi_a(R)$. For each $a \in \pi_a(R)$, maintain a hash table for $R[a] = \{b : (a,b) \in R\}$. Each entry in the hash table for $\pi_a(R)$ contains a pointer to the hash table for $R[a]$ (Similarly for k-ary relations with $k > 2$). In the case of 2-ary relations, this approach allocates $|\pi_a(R)| + 1$ hash tables. Section 3.2.1 presents an index-structure that maintains for a k-ary relation only k hash tables.

Figure 2.7 shows the two main components of the Hashbased Triejoin. Similar to the Leapfrog Triejoin, the `hashbased-search`-function performs the major part of the work. It uses an extended Hash-Join to calculate the set intersection from Algorithm 1 (line 3). For every key in the first relation, the function checks whether it is in all other relations using the `jump`-function. The first iterator in the `Iter` array is always the one with the least `length()` value. The `hashbased-init`-function ensures this by reordering the iterators.

```
globals: array Iter, integer size, boolean atEnd, integer depth

hashbased-search():
    while not Iter[0].atEnd():
        key := Iter[0].key()
        equals := true
        for p ∈ [size - 1] and equals:
            equals := Iter[p].jump(key)
        end for
        if equals:
            return
        else:
            Iter[0].next()
        end if
    end while
    atEnd := true
end

hashbased-init():
    size := Iter.size()
    if ∃p ∈ [size]: Iter[p - 1].atEnd():
        atEnd := true
    else:
        atEnd := false
        find p with Iter[p].length() minimal
        length := Iter[p].length()
        swap Iter[0] and Iter[p]
        hashbased-search()
    end if
end
```

Figure 2.7.: Hashbased Triejoin implementation

The `hashbased-search`-functions calls for the first iterator the functions `atEnd()`, `key()` and `next()` $\Theta(\min(|A_0|, \ldots, |A_{(k-1)}|))$-times. In each iteration, the `jump`-function is executed not more than $(k-1)$ times. All called functions take constant time. As a result, the `hashbased-search`-function has runtime complexity $\Theta(\min(|A_0|, \ldots, |A_{(k-1)}|))$ for intersecting the relations and is asymptotically faster than the `leapfrog-search`-function. Veldhuizen's worst-case optimality proof for the Leapfrog Triejoin algorithm from [4] also applies to the Hashbased Triejoin algorithm. The worst-case optimality of the Hashbased Triejoin directly follows from the proof.

The implementation of the `next`-, `atEnd`-, `open`- and `up`-functions for the Hashbased

TRIEJOIN algorithm is almost identical to the LEAPFROG TRIEJOIN algorithm. The `jump`-function of the HASHBASED TRIEJOIN calls the `jump`-function in all iterators and forwards the obtained key. If any iterator returns `false`, the method aborts and returns `false` as well. The full implementation is available in Appendix A in Figure A.2.

The following example shows that the HASHBASED TRIEJOIN without reordering is not worst-case optimal. The `hashbased-init`-function rearranges the `Iter` array for the current depth so that the iterator with the smallest number of elements comes first. Decisive for the runtime analysis is the number of keys used for probing in `hashbased-search()`. Figure 2.8 shows the keys considered for this task in the triangle query $Q_\triangle$ from Equation 2.4. Three different situations are depicted. Figure 2.8a shows the optimal case with reordering. In this case, `hashbased-search()` probes a linear number of keys. The other two figures list the keys used for probing when the `Iter` array is not reordered.

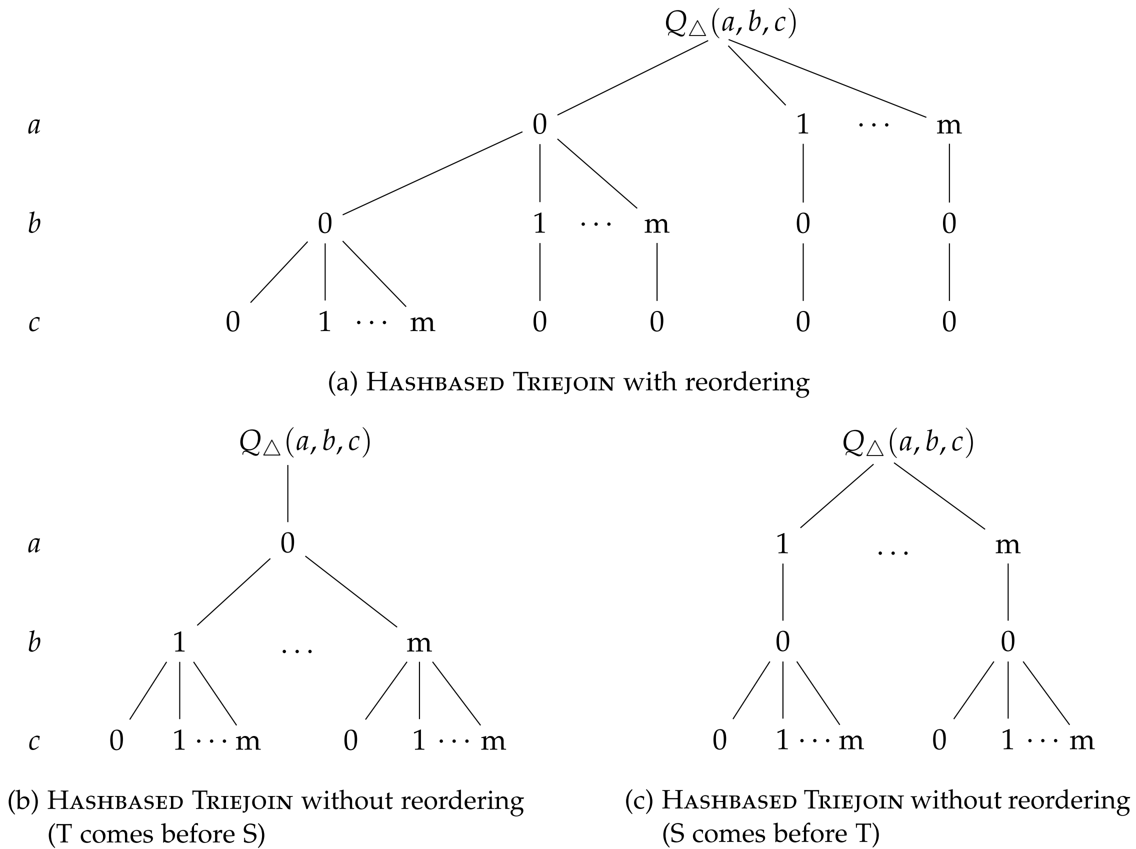


Figure 2.8.: Trie representation of the keys used for probing by the `hashbased-search`-function for the triangle query $Q_\triangle$ from Equation 2.4 (the global attribute order is $[a,b,c]$).

The HASHBASED TRIEJOIN without reordering does not alter the `Iter` array. Consequently, the three relations, R, S, and T, retain their order from the start of the algorithm. For the analysis, two cases are of interest:

1. T comes before S in the `Iter` array (Figure 2.8b): For the first level, attribute a is set to value 0. On the second level, the children of node 0 in $R(a,b)$ ($\{0,1,\ldots,m\}$) are joined with $\pi_b(S)$. The result of this intersection is $\{0,1,\ldots,m\}$. If attribute $b \in \{1,\ldots,m\}$ the `hashbased-search`-function uses the children of node 0 in $T'(a,c)$ for probing $S(b,c)$ on the third trie level. Node 0 in $T'(a,c)$ has $(m+1)$ children. Therefore, `jump()` in the iterator for $S(b,c)$ is called $(m+1)$-times. This is repeated m-times for all values of attribute b. Consequently, the `hashbased-search`-function invokes $\Theta(m^2)$-times `jump()`.

2. S comes before T in the `Iter` array (Figure 2.8c): On the first level, $\pi_a(R)$ and $\pi_a(T')$ are intersected. The result is the set $\{0,1,\ldots,m\}$. Thereafter, the algorithm determines the values for attribute b. If $a \in \{1,\ldots,m\}$, the value of b is always 0. On the third level, the children of $T'(a,c)$ and $S(b,c)$ are joined. $S(b,c)$ is opened on the first trie level for value 0. Node 0 in $S(b,c)$ has $(m+1)$ children. Since S comes before T in the `Iter` array, the elements from S are used for probing. Therefore, `jump()` in the iterator for $T'(a,c)$ is called $(m+1)$-times. This is repeated m-times for all values of attribute a. Consequently, the `hashbased-search`-function invokes $\Theta(m^2)$-times `jump()`.

In both situations, the `hashbased-search`-function executes `jump()` $\Theta(m^2)$-times. Hence, a lower bound for the runtime is $\Omega(m^2)$. One of the two cases has to happen, and therefore, the algorithm without reordering is not worst-case optimal. The HASHBASED TRIEJOIN with reordering dynamically changes the order of the `Iter` array and is, therefore, worst-case optimal.

2.5. Hybrid Triejoin

The HYBRID TRIEJOIN combines the LEAPFROG TRIEJOIN and HASHBASED TRIEJOIN. In comparison to the HASHBASED TRIEJOIN, it takes advantage of ordered relations. At the center of the algorithm is the observation that sorting small partitions of a dataset is considerably faster than sorting the entire dataset.

Given the trie representation from Figure 2.4, the inner nodes split the data into multiple smaller sets of leaf nodes. Each set of leaf nodes builds a partition and is sorted in ascending order. However, there is no ordering between the inner nodes demanded. In such a layout, the relation is called partially sorted.

The HYBRID TRIEJOIN merges partially sorted input relations. The underlying index-structures expose the data through a combination of the iterator interface from LEAPFROG TRIEJOIN and HASHBASED TRIEJOIN. If a relation contains k join attributes, the first $k-1$ attributes are accessed using the iterator interface from the HASHBASED TRIEJOIN. For

```
globals: array SortedIter, array UnsortedIter, integer sorted_size, integer unsorted_size,
boolean atEnd, integer pos

hybrid-search():
    while not atEnd:
        if sorted_size > 0:
            key := SortedIter[pos].key()
            max_key := SortedIter[(pos - 1) mod sorted_size].key()
            while key ≠ max_key:
                SortedIter[pos].seek(max_key)
                if SortedIter[pos].atEnd():
                    atEnd := true
                    return
                else:
                    max_key := SortedIter[pos].key()
                    pos := (pos + 1) mod sorted_size
                    key := SortedIter[pos].key()
                end if
            end while
        else:
            key := UnsortedIter[0].key()
        end if

        equals := true
        for p ∈ [unsorted_size] and equals:
            equals := UnsortedIter[p - 1].jump(key)
        end for

        if equals:
            return
        else if sorted_size > 0:
            SortedIter[pos].next()
            pos := (pos + 1) mod sorted_size
            atEnd := SortedIter[pos].atEnd()
        else:
            UnsortedIter[0].next()
            atEnd := UnsortedIter[0].atEnd()
        end if
    end while
end
```

Figure 2.9.: Hybrid Triejoin implementation

the last join attribute, the elements are stored in ascending order. The interface from the LEAPFROG TRIEJOIN is used to traverse the data on the last level.

An exception is the case $k = 1$: instead of sorting the entire relation, it remains unsorted and is accessed using the iterator interface from the HASHBASED TRIEJOIN.

Like the LEAPFROG TRIEJOIN and HASHBASED TRIEJOIN, the HYBRID TRIEJOIN searches for an element in the intersection of all iterators using the `hybrid-search`-function from Figure 2.9. Two arrays store the sorted and unsorted iterators. In order to find a match, the LEAPFROG JOIN algorithm first finds a key that is in all sorted iterators. If no sorted relation is available for the current level, the algorithm uses the current key from the first unsorted iterator. Then, it ensures that all unsorted iterators also contain the key. If this is not the case, the procedure starts from the beginning with another key.

The `hybrid-init`-function is a combination of the `leapfrog-init`- and `hash-based-init`-function. It sorts the `sortedIter` array and positions the smallest unsorted iterator at the front of the `unsortedIter` array. The implementation of the other functions is similar to the LEAPFROG TRIEJOIN and the HASHBASED TRIEJOIN (see Figure A.3 in Appendix A).

Although the presented HYBRID TRIEJOIN assumes that only the leaf nodes are sorted, it is possible to sort the inner nodes as well. Section 3.2.2 introduces a new index-structure that manages partially sorted data. In real-world applications, the time spent on sorting the data may reduce drastically compared to the structures used in the LEAPFROG TRIEJOIN.

3. Index-Structures

At the center of the presented join algorithms are the iterators that traverse the relations. A performant implementation of these iterators and the underlying data-structures is decisive for the actual performance of the algorithms. This chapter presents several sorted index-structures for the Leapfrog Triejoin and two unsorted index-structures for the Hashbased Triejoin and the Hybrid Triejoin. All index-structures are built from unordered arrays containing n tuples. As simplification, every tuple contains exclusively join attributes and integer values. Furthermore, the datasets do not contain duplicates.

3.1. Sorted Index-Structures

The Leapfrog Triejoin uses only sorted index-structures. The structures differ in the way the tuples are stored and in the implementation of the corresponding iterators. Although these index-structures can also be used in the Hashbased Triejoin and the Hybrid Triejoin, they do not meet the demanded runtime complexity. The Trie Array and Bitset Array structures are optimizations of the Flat Array. In addition, a standard tree structure, the B-Tree, is considered.

3.1.1. Flat Array

The most basic sorted index-structure is the Flat Array. It stores the tuples in a sorted array. Building this structure from an unordered array requires no additional steps or memory other than sorting the data. The Flat Array structure is demonstrated based on the relation $R(a,b)$ from Equation 2.2 ($m = 2$). Figure 3.1 shows the dataset in the trie representation, and Figure 3.2 illustrates the layout of the structure in memory.

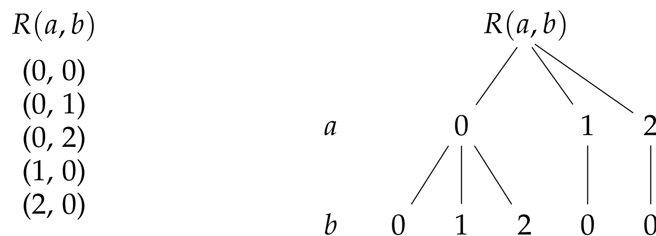
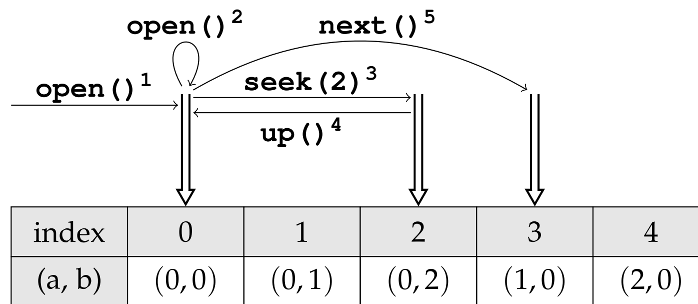


Figure 3.1.: Relation $R(a,b)$ from Equation 2.2 ($m = 2$)



index	0	1	2	3	4
(a, b)	$(0,0)$	$(0,1)$	$(0,2)$	$(1,0)$	$(2,0)$

Figure 3.2.: Flat Array for the relation $R(a, b)$ from Equation 2.2 ($m = 2$, see Figure 3.1). The superscript number denotes the order in which the operations are executed.

Figure 3.2 also shows the state of a trie iterator while executing the given sequence of operations. The first call to `open()` positions the iterator at index 0. The second call to `open()` gives access to attributes a and b but does not change the position. The iterator is now on the second trie level. The operation `seek(2)` advances the iterator forward to index 2. Then `up()` returns the iterator to the position where `open()` was called (index 0). After invoking `next()` the iterator moves to index 3. The new index is found by searching for the element $(0, +\infty)$.

The trie iterator always points to the first element that has a unique value for the attribute on the current level. In the concrete example, the iterator can access any tuple when opened for the attributes a and b. However, the iterator cannot access the tuples $(0, 1)$ and $(0, 2)$ when opened for attribute a. Instead, it points to the tuple $(0, 0)$.

A trie iterator for the Flat Array obtains a reference to the sorted array of tuples. Besides, it stores the current depth, the current position, a vector of indexes, and a boolean value. The vector saves the return points when moving upwards in the trie. The boolean value indicates whether the iterator is at the end. The implementation of the iterator has to comply with the interface definitions and runtime constraints established by the Leapfrog Triejoin.

The functions perform the following steps:

`key()` accesses the array at the current position and extracts the correct element from the tuple.

`next()` if the iterator is opened for the last attribute, it increments the position of the iterator in the array. Otherwise, it performs a seek operation. Before returning, the iterator checks whether it reached the end and updates the boolean value.

`seek(key)` uses binary search to find the given key in the remaining part of the array. Before returning, the iterator checks whether it reached the end and updates the boolean value.

`atEnd()` returns the boolean value indicating whether the iterator reached the end.

open() increments the current depth and appends the current position to the vector of indexes.

up() decrements the current depth, sets the boolean value to `false` and restores the iterator position from the vector of indexes.

The binary search algorithm finds the given element or a least upper bound in $\mathcal{O}(\log(n))$. Consequently, `next()` and `seek()` take $\mathcal{O}(\log(n))$ and the other functions consume $\mathcal{O}(1)$ time.

The Leapfrog Triejoin exhibits dense access patterns. For a `seek()` operation, the distance between the new iterator position and the previous position is probably small. The binary search algorithm can exploit this observation. Before the algorithm searches the entire remaining array, it looks at the first k elements in the array and tests whether the k-th element is greater than or equal to the sought key. If this is the case, the key is among the first k elements. Therefore, the binary search algorithm considers only a subarray of length k. Despite this optimization, the asymptotic runtime of the `next`- and `seek`-functions do not change.

Essentially, the Flat Array is simply a sorted array of tuples. Sorting the tuples, therefore, dominates the runtime for building the index-structure. A suitable sorting algorithm performs this process in $\mathcal{O}(n \cdot \log(n))$. However, inserting elements into a sorted array is relative expensive. First, the correct position is determined, and then the subsequent elements are moved one position to the right. Therefore, every insert takes on average $\mathcal{O}(n)$ time.

Chu, Balazinska, and Suciu use the Flat Array structure in their distributed version of Leapfrog Triejoin, the so-called Tributary Join [10]. The algorithm builds the index-structure for every join from scratch. The authors identified the building of the index-structure as a bottleneck. Therefore, the Flat Array is preferred over a B-Tree since sorting an array is faster than building the tree.

3.1.2. Trie Array

Wu, Zinn, Aref, and Yalamanchili optimized the Leapfrog Triejoin and the Flat Array for General Purpose GPUs [11]. The authors use the so-called Trie Array as index-structure. Figure 3.3 shows the Trie Array for the relation $R(a, b)$ from Equation 2.2 ($m = 2$). The structure directly resembles the trie layout. All elements on the same level in the trie representation are stored in one array. Furthermore, it needs to know where the children of a node start and end in the subsequent array.

For a k-ary relation, the Trie Array allocates k arrays. The attributes from the trie representation are distributed over the arrays according to the global attribute order. In addition, $(k-1)$ ptr arrays store the indexes where the children of a node are. If i is the index of some node in the array for attribute a, then the elements at index $ptr[i-1]$ (inclusive) to $ptr[i]$ (exclusive) in the array for attribute b are the children of this node.

Figure 3.3 also shows the sequence of operations discussed in Section 3.1.1. The first two operations open the iterator for the attributes a and b. First, the iterator is positioned

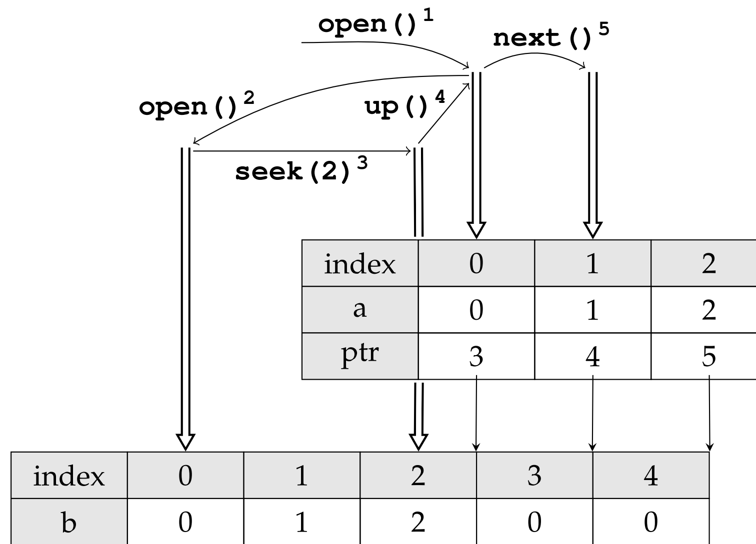


Figure 3.3.: Trie Array for the 2-ary relation $R(a, b)$ from Equation 2.2 ($m = 2$, see Figure 3.1). The superscript number denotes the order in which the operations are executed.

at index 0 in the array for attribute a, and then it accesses the first element in the array for attribute b. The following call to `seek(2)` moves the iterator to the third element in the second array (index 2). Hereafter, `up()` returns the iterator to the array for attribute a at index 0. Then `next()` advances the iterator to index 1 in the first array.

The implementation of the trie iterator for this structure is similar to that for the Flat Array. The iterator obtains references to all arrays and *ptr* arrays. A vector of indexes stores the position in each array. The trie interface functions are implemented as follows:

`key()` accesses the array for the current depth at the current position and returns the value.

`next()` increments the position in the array for the current depth.

`seek(key)` uses binary search to find the obtained key or the least upper bound in the current interval.

`atEnd()` checks whether the index for the current depth is greater than or equal the index stored in the *ptr* array of the parent node.

`open()` increments the depth and sets the index for the next level to the start of the children. This value is extracted from the *ptr* array on the current level.

`up()` decrements the depth and restores the iterator position from the vector of indexes.

The Trie Array offers several advantages over the Flat Array. Above all, it reduces the runtime of the `next`-function from $\mathcal{O}(\log(n))$ to $\mathcal{O}(1)$. Although the complexity

of the `seek`-function is still $\mathcal{O}(\log(n))$, the runtime will improve as well. The Flat Array searches the entire remaining array for the given key. The Trie Array, however, searches only in the children of the parent for the obtained key. Depending on the data distribution, the Trie Array also consumes less memory than the Flat Array.

Building the Trie array requires three steps. First, it sorts the input array of tuples ($\mathcal{O}(n \cdot \log(n))$). Then, one iteration over the sorted array determines the sizes of the arrays on each level ($\mathcal{O}(n)$). The third step allocates the required arrays and copies the values from the input arrays into the new arrays. Simultaneously, the values for the *ptr* arrays are computed and written to memory ($\mathcal{O}(n)$).

3.1.3. Bitset Array

The Bitset Array is an optimization of the Trie Array for dense datasets. It compresses several integer values into a tuple consisting of an offset value and a bitset. For example, the tuple $(12, \texttt{1011})$ represents the three 32-bit integers 12, 13, and 15. In the binary representation of 12, 13 and 15, the highest 30 bits are identical, while the lowest two bits differ. The first 30 bits serve as an offset, and the last two bits are interpreted as indexes for a bitset of size four. The bits at the corresponding positions in the bitset are set to true. The relational query processing engine EmptyHeaded inspired this approach [12].

Figure 3.4 shows the tuples from $R(a, b)$ in the bitset representation. For illustration, the tuples $(2, 17)$ and $(14, 11)$ were added. At this point, the elements were just converted to tuples consisting of offsets and bitsets, and no compression happened. Figure 3.5 compresses the relation by merging bitset tuples with the same offset. Like the Trie Array, a *ptr* array stores the indexes of the children in the subsequent array. However, the array does not contain the indexes themselves, but vectors of indexes. The reason for this is that a bitset tuple represents more than one integer. Therefore, every element has to keep track of multiple sets of children.

index		$(0,0)$	$(0,1)$	$(0,2)$	$(1,0)$	$(2,0)$	$(2,17)$	$(14,11)$
a	offset	0	0	0	0	0	0	12
	bitset	`0001`	`0001`	`0001`	`0010`	`0100`	`0100`	`0100`
b	offset	0	0	0	0	0	16	8
	bitset	`0001`	`0010`	`0100`	`0001`	`0001`	`0010`	`1000`

Figure 3.4.: Bitset representation of the relation $R(a, b)$ from Equation 2.2 ($m = 2$, see Figure 3.1). For illustration, the tuples $(2, 17)$ and $(14, 11)$ were added.

The sequence of operations from the previous sections also illustrates the behavior of an iterator for the Bitset Array. In comparison to the Trie Array, the iterator for the bitset representation accesses fewer elements. First, it performs two `open()` operations. After that, the iterator is positioned at the first element in both arrays. The subsequent call to

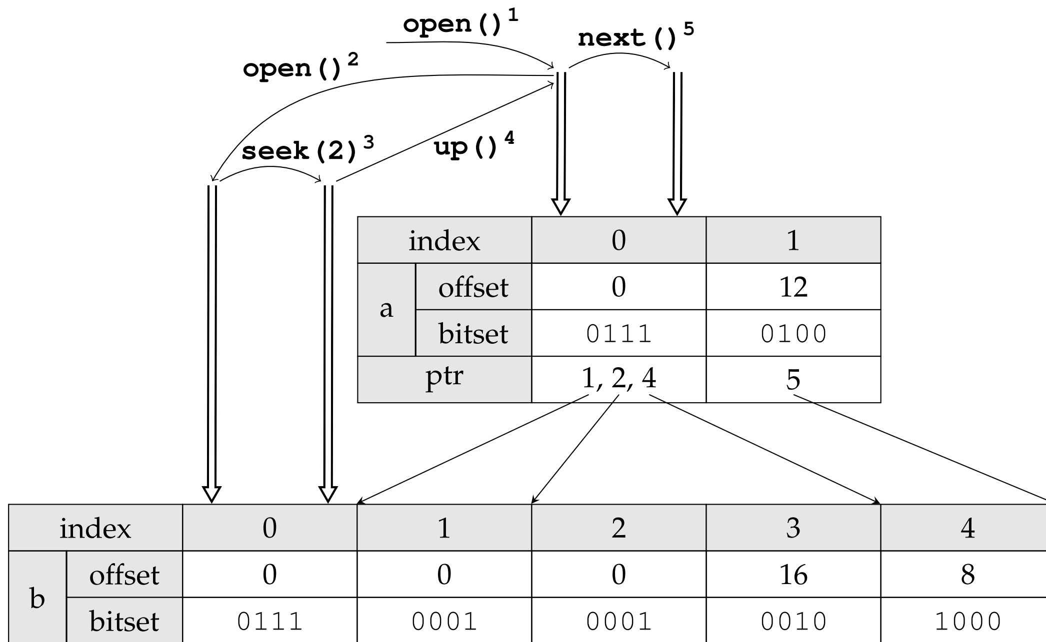


Figure 3.5.: Bitset Array for the relation $R(a, b)$ from Equation 2.2 ($m = 2$, see Figure 3.1). For illustration, the tuples $(2, 17)$ and $(14, 11)$ were added. The superscript number denotes the order in which the operations are executed.

`seek(2)` keeps the iterator at the same element. Within the bitset, the iterator is moved forward to the third entry. The same applies to the `next()` operation in the first array. The iterator is advanced to the next set bit in the current bitset (second entry).

The implementation of the iterator for the Bitset Array is in most parts identical to that of the Trie Array. Some functions have been adapted to the bitset layout:

`key()` returns the tuple containing the offset and bitset at the current position, instead of a regular integer.

`next()` moves to iterator to the next set bit in the current bitset. If no such bit is found the iterator is advanced to the next element in the array.

`seek(key)` finds the correct element in the array using the offset. Then it positions the iterator at the correct index within the bitset.

The Leapfrog Triejoin, as described in Section 2.3, is applied to the offset values. In the case of a match, all bitsets are intersected using the bitwise **and** operation. The complexity of the interface functions does not change compared to the Trie Array. In some cases, however, the performance of the join improves drastically.

In general, the last p bytes are converted into a bitset for 2^p elements. However, if the bitsets are too large, memory may be wasted. In modern processors, the general-purpose registers are 64-bit wide, and the SSE and AVX extensions add support for 128-bit and

256-bit vector registers. Therefore, reasonable values for p are 6, 7, and 8. Chapter 4 evaluates the Bitset Array for the three mentioned sizes.

Building the Bitset Array is almost identical to building the Trie Array. The conversion from regular integers to bitsets is done in the third step. This operation takes linear time and the complexity of the third step is still $\mathcal{O}(n)$. In total, building the structure costs $\mathcal{O}(n \cdot \log(n))$ (sorting the data implies this runtime bound).

Aberger, Lamp, Olukotun, and Ré use the Generic-Join algorithm from Section 2.2 in their query processing engine LevelHeaded. LevelHeaded uses a combination of the Trie Array and the Bitset Array to store the data. LevelHeaded converts only the first level of the trie to bitsets. The other levels are regular integers. This approach originates from the observation that the elements in the first level of a trie are typically dense, whereas the elements on the other levels are usually sparse [8].

3.1.4. B-Tree

A drawback of all three array structures is the time spent on inserting, deleting, and updating elements. After building the structure, these operations cost on average $\mathcal{O}(n)$ time. Therefore, the index-structures are either built on the fly for every join (like the Tributary Join does [10]) or assume that the data will not change. A structure that inserts and deletes elements in $\mathcal{O}(\log(n))$ could process modifications more efficiently. As a result, there is no need to rebuild the structure for every join or to make the data immutable.

The family of B-Tree structures fulfills these requirements. Searching, inserting, and deleting elements from a B-Tree costs $\mathcal{O}(\log(n))$. A B-Tree is a balanced search tree, i.e., the distance between the root node and all leaf nodes is identical. The integer $t \geq 2$ denotes the degree of the B-Tree. Every node except the root node contains at least $(t-1)$ elements and at most $(2t-1)$ elements. In addition, each inner node possesses $(s+1)$ children, where s denotes the number of elements stored in the node. [9, ch. 18]

Figure 3.6 shows the B-tree for the relation $R(a,b)$ from Equation 2.2 ($m = 2$). The tree has degree $t = 2$ and consists of one root node and two leaf nodes. The same operations as in the previous sections, illustrate the behavior of the iterator. Upon initialization, the iterator is placed at the smallest element (the first entry in the leftmost leaf). The first and second operations open the iterator for attributes a and b. Then `seek(2)` positions the iterator at the smallest upper bound for the tuple $(0,2)$. The current node does not contain this tuple. Thus, the iterator returns to the root node and starts searching from there. The `up()` call returns the iterator to the position where `open()` was called. As in the Flat Array, the `next()` method performs a seek operation for the element $(0,+\infty)$. The search starts at the root node and finds a least upper bound in the second leaf.

The implementation of the iterator is almost identical to that of the Flat Array. It stores a reference to the root node, the current depth, and a vector of positions in the B-Tree. Apart from the `next`- and `seek`-functions, none of the implementations was changed:

`next()` if the iterator is opened for the last attribute, it increments the position in the

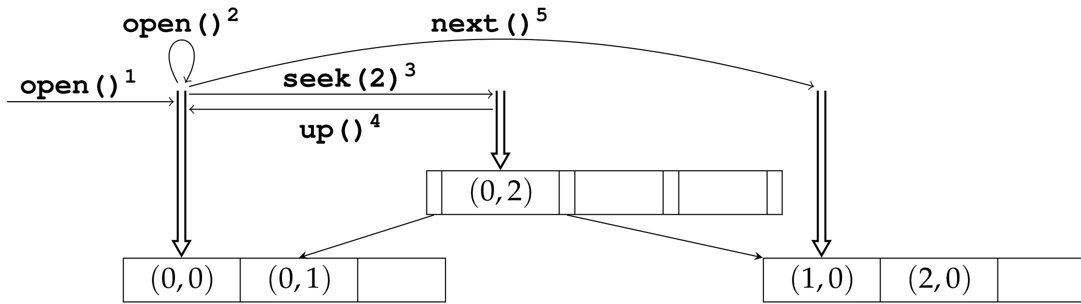


Figure 3.6.: B-Tree of degree 2 for the relation $R(a, b)$ from Equation 2.2 ($m = 2$, see Figure 3.1). The superscript number denotes the order in which the operations are executed.

current tree node. The new position can be larger than the number of elements stored in the node. In this case, the iterator takes the next element from the parent node. If the iterator is not opened for the last attribute, it performs a seek operation.

seek(key) starts at the root node and searches for given key or a least upper bound. The function uses binary search to find the position of the sought key within a node.

As before, `seek()` is optimized for dense access patterns caused by the LEAPFROG TRIEJOIN. The function first searches for the given key in the current B-Tree node or one of its children. If the last element in the current node is greater than or equal to the obtained key, the iterator does not return to the root node. Instead, it starts searching from the current node onwards. Nevertheless, the complexity of the `next`- and `seek`-functions is $\mathcal{O}(\log(n))$.

Chapter 4 evaluates the B-Tree for two different degrees. The first degree is chosen so that the leaves are not larger than 64 bytes (the size of a cache line). In the second variant, the nodes of the B-Tree fit into a 4 KB page. Although B-Trees are intended as permanent index-structures, the time required to build the tree is measured as well. For the 64 bytes variant, the time for building the index-structure will increase due to memory allocation. However, the number of cache misses for the `seek`-function reduces since the binary search algorithm accesses only one cache line. Veldhuizen suggested B-Trees as index-structures for the LEAPFROG TRIEJOIN in [4].

Although inserting n elements into a B-Tree costs $\mathcal{O}(n \cdot \log(n))$, it is faster first to sort the array and then copy the elements into the B-Tree. This approach is called *bulk-loading*. In particular, bulk-loading ensures that the height and the number of nodes in the tree are minimal. Copying the elements from the sorted array to the B-Tree requires one iteration ($\mathcal{O}(n)$). Sorting the array still dominates building the structure ($\mathcal{O}(n \cdot \log(n))$).

3.2. Unsorted Index-Structures

The runtime constraints for the trie iterators established by HASHBASED TRIEJOIN are tighter than those of the LEAPFROG TRIEJOIN. In contrast to the `next`- and `seek`-functions from the LEAPFROG TRIEJOIN, the `next`- and `jump`-functions from the HASHBASED TRIEJOIN demand constant runtimes. Sorted index-structures do not meet this requirement. Hash tables, however, are known for their constant lookup and insert time. The index-structures for the HASHBASED TRIEJOIN and the HYBRID TRIEJOIN utilizes hash tables to guarantee the runtime constraints. The HASHBASED TRIEJOIN uses the Multichain Hash Table. In comparison to the sorted index-structures, building the Multichain Hash Table is possible in linear time. The index-structure for the HYBRID TRIEJOIN, the Hybrid Hash Table, is like the join algorithm itself a combination of unsorted and sorted structures.

3.2.1. Multichain Hash Table

The Multichain Hash Table provides an index over unsorted arrays. It does not change the order of the array and solely adds further information to simplify navigation for the trie iterator. A trie iterator for the HASHBASED TRIEJOIN must execute all interface functions in $\mathcal{O}(1)$ time. For some functions, like the `key`- and `up`-functions, it is straightforward to guarantee this runtime constraint. For sorted index-structures, the iterators store the position of the current element using a pointer or array index. The iterator for the Multichain Hash Table adapts this approach and provides a pointer to the current element in the unsorted array. The `key`-function reads the element at the position of the pointer in $\mathcal{O}(1)$. For the `up`-function, previous trie iterator implementations store the return points in a vector. When `up()` is called, the iterator returns to the latest return point. This approach also works for the Multichain Hash Table.

The `jump`- and `next`-functions, however, need additional structures to guarantee the runtime constraints. The `jump`-function checks whether the given key is in the unsorted array. As the iterator cannot search for the element in the entire array, it uses a hash index. Hash tables can perform a lookup operation in linear time. For the relation $R(a, b)$ from Equation 2.2 ($m = 2$), the Multichain Hash Table maintains a hash index over the five tuples from the dataset. When the iterator is opened for attribute a and b, it uses this hash index for the lookup. However, when the iterator is opened for attribute a, the hash index for the tuples (a, b) cannot be used. Instead, a second hash table indexes the projection $\pi_a(R)$ (the first level of the trie). The two tables on the left and right side of Figure 3.7 represent these hash tables. The table on the left side indexes $\pi_a(R)$. The hash table on the right side provides an index over the entire relation $R(a, b)$.

In an unsorted array, the neighbors of a node from the trie representation can be anywhere. Therefore, the `next`-function needs some extra information for finding the next element. All elements with the same parent node lie on a so-called chain. For the first level of the trie, one chain is needed. It connects all children of the root node, i.e., the elements $(0, _)$, $(1, _)$ and $(2, _)$. For the second level, three independent chains link

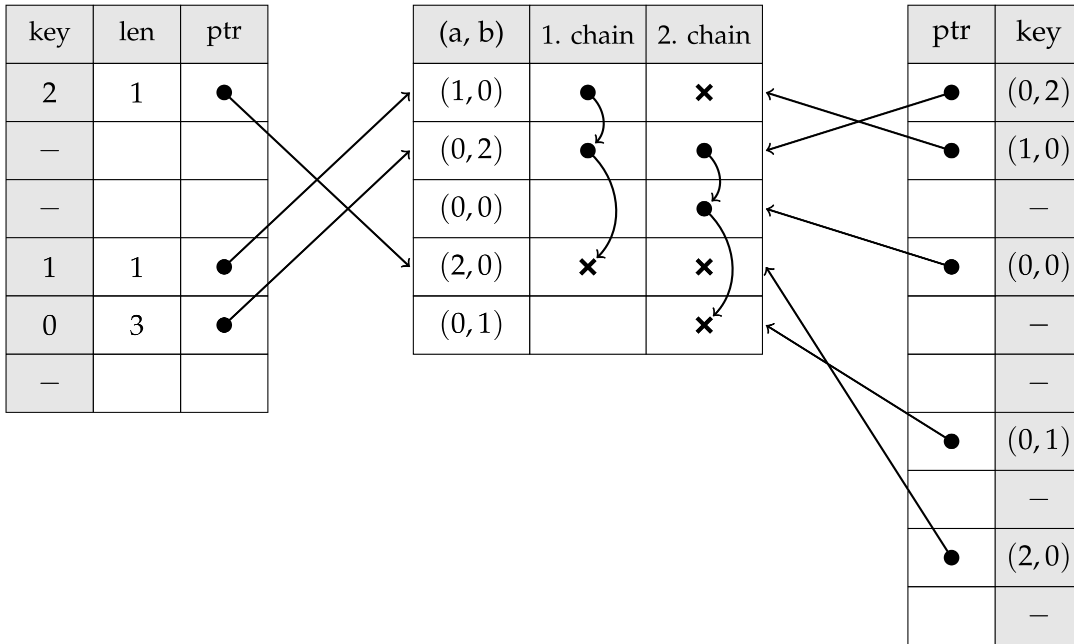


Figure 3.7.: Multichain Hash Table for the relation $R(a, b)$ from Equation 2.2 ($m = 2$). The second and third column in the tuple array store the chains. The crosses mark the end of the chain.

the elements together: one chain for $(0, 0)$, $(0, 1)$ and $(0, 2)$, one for $(1, 0)$ and another for $(2, 0)$. All chains are implemented as a linked list. For every element, a pointer to its successor is stored. The `next`-function follows the pointer from the current element to the next element. A special value marks the end of a chain.

The table in the middle of Figure 3.7 shows the unsorted array with two additional columns for the chains. The column for the first chain contains the linked list for $\pi_a(R)$, i.e., connecting the elements $(0, _)$, $(1, _)$ and $(2, _)$. The chain starts at the first element and ends at the fourth. The elements $(0, 0)$ and $(0, 1)$ are not part of this chain since $(0, 2)$ already represents the value $(0, _)$. The second column contains the three chains for the children of $(0, _)$, $(1, _)$ and $(2, _)$, i.e., connecting the leaves. The three chains start at index 0, 1, and 3. Only the second chain has more than one element and connects the children of $(0, _)$.

The `length`-function returns the number of elements in every chain. The start and length of the chains in the second column are appended to the entries in the left hash table. The hash tables do not store the keys, but pointers to the element they index.

For a k-ary relation $U(a_1, \ldots, a_k)$, the Multichain Hash Table maintains k hash tables over the projections $\pi_1(U)$, …, $\pi_k(U)$. The expression $\pi_i(U)$ denotes the projection $\pi_{a_1,\ldots,a_i}(U)$. The entries in the hash tables store pointers to the elements in the unsorted array. Additionally, for every element $(b_1, \ldots, b_i) \in \pi_i(U)$ $(i \in \{1, \ldots, k-1\})$ the hash

table for $\pi_i(U)$ contains the number of elements in

$$\pi_{i+1}(\{(a_1,\dots,a_k)\colon (a_1,\dots,a_k) \in U \wedge a_1 = b_1 \wedge ... \wedge a_i = b_i\})$$

Furthermore, the Multichain Hash Table allocates k columns for the chains. Let $i \in \{1,\dots,k\}$ and $(b_1,\dots,b_{i-1}) \in \pi_{i-1}(U)$, then the i-th column contains the chains connecting the elements in

$$\{a_i\colon (a_1,\cdots,a_i) \in \pi_i(U) \wedge a_1 = b_1 \wedge \cdots \wedge a_{i-1} = b_{i-1})$$

The pointers in the hash index over $\pi_i(U)$ ($i \in \{i,\dots,k-1\}$) denotes the start of a chain in the $(i+1)$-th column. The hash tables also store the length of the corresponding chain. For every chain, such an entry in a hash table exists.

The iterator for this structure stores the current depth and a vector of pointers into the array. The vector saves the return points when moving upwards in the trie. The implementation must match the interface definitions and runtime constraints established by the `Hashbased Triejoin`. The functions perform the following operations:

key() follows the current pointer and extracts the correct element from the tuple.

next() replaces the current pointer with the value stored in the chain column at the current pointer position.

jump(key) performs a lookup in the corresponding hash table for the given key. If an entry is found in the hash table, the current pointer is updated.

length() returns the length of the chain that the iterator currently follows. The hash table stores this value with the pointers to the start of the chain.

atEnd() compares the current pointer with a special value that marks the end of a chain.

open() increments the current depth and extracts the start pointer and length of the chain from the corresponding hash table.

up() decrements the current depth and restores the latest pointer from the vector of pointers.

The lookups in the hash table and array accesses take $\mathcal{O}(1)$ time [9, ch. 18]. Consequently, `next()` and `jump()` run in constant time. The same holds for the other functions.

In general, any hash table implementation that meets the runtime constraints can be used. In this example, the size of the hash table is twice the number of entries stored in the table. The size of the hash tables is reduced by storing array indexes instead of pointers. If the number of elements is small enough, this optimization replaces a 64-bit pointer with a smaller 32-bit integer.

In contrast to sorted index structures, the Multichain Hash Table is built in linear time. Upon creation, the Multichain Hash Table initializes the hash tables and allocates memory for the chains. Inserting the elements in the hash tables takes one iteration over the array ($\mathcal{O}(n)$). The insert operation ($\mathcal{O}(1)$) is performed k times for every tuple. Simultaneously the entries for the chains are filled, and the corresponding start and length values of the chain are inserted into the hash tables.

The lookup operation on hash tables implies random memory access patterns. As a result, almost any access to the hash table, the unsorted array of tuples, and the chain entries causes a cache miss. The number of misses is reduced by storing the tuples and the corresponding chain entries in the same row. The Multichain Hash Table copies the tuples from the unsorted array into a new array with additional columns for the chain entries.

3.2.2. Hybrid Hash Table

In the Multichain Hash Table, the number of random memory accesses is inherently higher than in sorted index structures. A direct consequence of this access pattern is a higher number of cache misses. Three different types of random memory accesses occur in the Hashbased Triejoin on the Multichain Hash Table:

1. the lookup operations on hash tables,
2. the subsequent array accesses following the pointers from a hash table entry, and
3. the iteration over the elements in the chains.

The use of hash tables makes the first kind of cache misses inevitable. The indirection implied by the pointers, however, causes the second and the third type of misses. The pointers are used for retrieving the key of the hash table entry. The Hybrid Hash Table eliminates this kind of cache miss by storing the key in the hash table entry.

In the Multichain Hash Table, the elements in a chain are spread over the entire array. Therefore, a linked list is needed that connects all children of a node. When traversing the linked list, the iterator jumps through the unsorted array. This access pattern causes the third kind of cache misses. A vector that stores the elements of a chain one after another eliminates the random access pattern and reduces the number of cache misses. The Hybrid Hash Table adds to every hash table entry a vector for all children of the node.

Figure 3.9 shows the Hybrid Hash Table structure for a 3-ary relation. In this example, the structure indexes the result of the triangle query $Q_\triangle(a, b, c)$ from Equation 2.4 ($m = 2$) instead of the 2-ary relation $R(a, b)$. Figure 3.8 lists the tuples in $Q_\triangle$ and shows the relation in trie representation. The hash table on the left side of Figure 3.9 uses the attribute a as key. Every entry stores the children of the key in a vector. The hash table on the right sides provides an index over the tuples (a, b). The corresponding vector holds the children on the third level, i.e., attribute c. Most notably, the vectors in the first

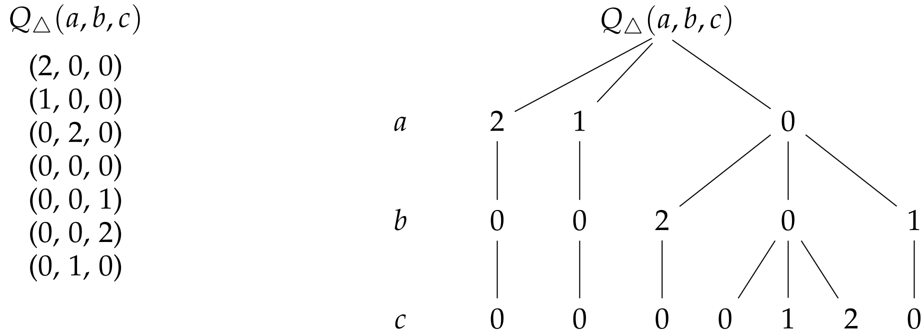


Figure 3.8.: Partially sorted relation $Q_{\triangle}(a,b,c)$ from Equation 2.4 ($m = 2$)

hash table store the values unsorted, whereas the vectors in the second hash table are sorted. Figure 3.8 only sorts the nodes on the third level as well. This layout is called partially sorted.

The sorted vectors replace the hash table for (a, b, c). There are two reasons for this approach: First, the third hash table over the entire relation increases memory consumption. Second, the third hash table would store the relation twice. The second hash table already contains all elements from Q in the form of a key and the corresponding vector of children. Consequently, the sole purpose of the third hash table is to provide an index over the leaf nodes. A sorted vector indexes the leaf nodes and reduces the memory consumption significantly. Although it is not possible to find an element in O(1), the binary search algorithm scans the array in logarithmic time. In comparison to fully sorted index-structures, the Hybrid Hash Table offers the advantage that only partitions are sorted instead of the entire relation.

For a k-ary relation $U(a_1, \cdots, a_k)$, the Hybrid Hash Table consists of $(k-1)$ hash maps over the projections $\pi_1(U)$, $\cdots$, $\pi_k(U)$. The i-th hash map indexes the tuples $(b_1, \cdots, b_i) \in \pi_i(U)$ and stores the children of $(b_1, \cdots, b_i)$, i.e., the set

$$\pi_{i+1}(\{(a_1, \cdots, a_k) \colon (a_1, \cdots, a_k) \in U \wedge a_1 = b_1 \wedge \cdots \wedge a_i = b_i\})$$

in a vector. The partially sorted data layout is achieved by sorting the vectors in $(k-1)$-th hash map.

The structure assumes that the inner nodes partition the leaf nodes into sufficiently small subsets. Sorting these subsets is considerably faster than sorting the entire data set. However, the case $k = 1$ is an exception: there are no inner nodes that partition the array. The Hybrid Hash Table uses a hash set in this case since sorting an array is asymptotically slower than building a hash table. The memory overhead is not severe as none of the elements is stored twice.

If the iterator is opened for the first trie level, `next()` iterates the hash table that indexes $\pi_1(U)$. On the lower levels, the function iterates the corresponding vector. The `jump`-function performs a lookup in the hash table for the current depth. The `seek`-function is available if the relation is not unary and the iterator is opened for the

key	data
2	[0]
–	
–	
1	[0]
0	[2, 0, 1]
–	

key	data
(2, 0)	[0]
–	
(0, 1)	[0]
–	
(0, 0)	[0, 1, 2]
(1, 0)	[0]
–	
–	
(0, 2)	[0]
–	

Figure 3.9.: Hybrid Hash Table for the relation $Q_\triangle(a, b, c)$ from Equation 2.4 ($m = 2$). The left hash table indexes $\pi_a(Q_\triangle)$ and the right one indexes $\pi_{a,b}(Q_\triangle)$.

lowest level in the trie representation. In this case, the vector is sorted, and the iterator uses binary search for finding the given key.

In the case of a unary relation, the Hybrid Hash Table is built in one step. It takes one iteration over the array to insert all elements into the hash set ($\mathcal{O}(n)$). If $k > 1$, one additional step is required. The first iteration fills the hash maps and the vectors. For every element, $(k - 1)$ insert operation are performed. The second step sorts the elements in the vectors of the last hash map. It iterates over all entries and uses a comparison-based sorting algorithm on the vectors. Under the assumption, that the insert-operation in a vector takes linear time, the first step executes in $\mathcal{O}(n)$. The complexity of the second step depends on the size of the partitions. A lower bound for the runtime is $\mathcal{O}(n)$ if every vector in the last hash-table contains one element. The worst-case scenario occurs when a single vector consists of n elements. In this case, sorting the vector takes $\mathcal{O}(n \cdot \log(n))$ time. Therefore, the worst-case runtime for building the Hybrid Hash Table is $\mathcal{O}(n \cdot \log(n))$.

4. Experiments

This chapter presents and analyzes the results of the conducted benchmarks. Three different queries assess the performance of the join algorithms and index-structures. The first two sections describe the used datasets and the environment in which the benchmarks were executed. The third section visualizes and interprets the results. Lastly, we summarize our results and discuss the strengths and limitations of the individual index-structures.

4.1. Queries

Three different queries evaluate the various aspects of the employed join algorithms and index-structures. The Interleaved Query assesses the performance of the join algorithms for a single set intersection. The Hypercube Query demonstrates that worst-case optimal join algorithms are asymptotically faster than pairwise join plans. The last query based on the Twitter dataset measures the performance of the algorithms using a real-world example. Furthermore, the Twitter Query is used to compare the worst-case optimal join algorithms with pairwise join plans for the triangle query.

4.1.1. Interleaved Query

The Interleaved Query is defined as follows:

$$Q_{Interleaved} = R_{Interleaved} \bowtie S_{Interleaved} \bowtie T_{Interleaved}$$

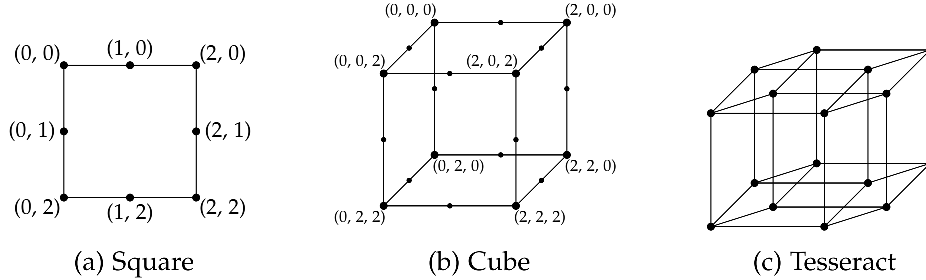
with

$$\begin{aligned} R_{Interleaved}(x) &= \{3 \cdot x \colon x \in \{0, \dots, n-1\}\} \\ S_{Interleaved}(x) &= \{3 \cdot x + 1 \colon x \in \{0, \dots, n-1\}\} \\ T_{Interleaved}(x) &= \{3 \cdot x + 2 \colon x \in \{0, \dots, n-1\}\} \end{aligned}$$

The query joins three 1-ary relations. Therefore, all three join algorithms perform a set intersection on the datasets. The number n denotes the size of the relations. Obviously, the three sets are disjoint, and the result of the query is the empty set. This distribution is a worst-case situation for LEAPFROG TRIEJOIN. The algorithm has to seek $(3 \cdot n)$-times, whereas the HASHBASED TRIEJOIN and the HYBRID TRIEJOIN call only n-times the `next` and `jump`-functions.

4.1.2. Hypercube Query

The Hypercube Query constructs the outlines of a 4-dimensional hypercube. It obtains all points on the edges of a 2-dimensional square (see Figure 4.1a) and performs several self-joins to produce the points on the edges of a higher-dimensional cube. Figure 4.1b and Figure 4.1c show a 3-dimensional and 4-dimensional hypercube.



(a) Square (b) Cube (c) Tesseract

Figure 4.1.: 2D, 3D and 4D Hypercube

For a d-dimensional hypercube the following expression list all points on the edges:

$$H_m^d(x_1, \ldots, x_d) = \bigcup_{i \in [d]} \{(x_1, \ldots, x_d) \colon x_i \in \{0, 1, \ldots, m\}, \forall j \in [d] \setminus \{i\} \colon x_j \in \{0, m\}\}$$

The factor m denotes the number of points on a single edge and controls the size of the set. The triangle query constructs the points on the edges of a 3-dimensional cube from three 2-dimensional squares:

$$H_m^3(x_1, x_2, x_3) = H_m^2(x_1, x_2) \bowtie H_m^2(x_1, x_3) \bowtie H_m^2(x_2, x_3)$$

For the benchmark, the triangle query is extended and lists all cliques with four vertices, i.e., the points on the edges of a 4-dimensional hypercube:

$$\begin{aligned} Q_{Hypercube}(x_1, x_2, x_3, x_4) = & H_m^2(x_1, x_2) \bowtie H_m^2(x_2, x_3) \bowtie H_m^2(x_1, x_3) \bowtie \\ & H_m^2(x_1, x_4) \bowtie H_m^2(x_2, x_4) \bowtie H_m^2(x_3, x_4) \end{aligned}$$

This query is a 6-way self-join on H_m^2 and produces the result H_m^4. The number of elements in the input relation is $|H_m^2| = 4m$ and in the result $|H_m^4| = 32m - 16$. As before for the triangle query, a pairwise join plan has to perform a join of the form $H_m^2(x_i, x_j) \bowtie H_m^2(x_j, x_k)$ $(i < j < k)$. This join returns $2m^2 + 8m - 2$ elements and, therefore, a pairwise join plan has to deal with a quadratic intermediate result (compared to the size of the input relation H_m^2).

4.1.3. Twitter Query

The third query finds all directed triangles in the Twitter dataset using the triangle query from Section 2.1. The Twitter dataset $T(a, b)$ models the follower-followee connection

from the social media platform Twitter as directed graph. The query is defined as follows:

$$Q_{Twitter}(x, y, z) = T(x, y) \bowtie T(y, z) \bowtie T(z, x)$$

The query is evaluated for five different sizes of the dataset. Table 4.1 lists the sizes and the number of elements in the intermediate result $T(x, y) \bowtie T(y, z)$. Although the intermediate result is in some cases considerably larger than the query result, a quadratic growth is not observable. The five datasets are subsets of a snapshot of the Twitter network taken by the SNAP group in 2010 [13].

Table 4.1.: Sizes for the Twitter Query

input relation	intermediate result	query result
2 500	4 233	1 230
25 000	178 591	85 690
250 000	2 844 862	1 651 549
2 500 000	9 466 368	2 112 612
25 000 000	119 997 074	4 059 071

4.2. Setup

The experiments were conducted on a desktop computer with an Intel Core I7-3770 CPU and 32 GByte of RAM. The open-source library Google Benchmark[1] measures the runtime and settles the number of iteration for every benchmark. The library assigns at least five minutes of runtime to each benchmark and reports the average CPU time. All experiments were evaluated for single-core performance and compiled with GCC 7.4.0. All attributes in the datasets fit into a 32-bit integer.

The Multichain Hash Table and Hybrid Hash Table require two additional libraries: robin-map[2] and CityHash[3]. The robin-map library is a C++ implementation of a fast hash map and hash set with robin hood hashing. The CityHash algorithm is used for hashing the keys. The Hybrid Hash Table uses the standard vector[4] for dynamic size arrays in its hash maps.

For sorting the datasets, we apply the sort-function from the C++ standard library[5].

[1] https://github.com/google/benchmark
[2] https://github.com/Tessil/robin-map
[3] https://github.com/google/cityhash
[4] https://en.cppreference.com/w/cpp/container/vector
[5] https://en.cppreference.com/w/cpp/algorithm/sort

4.3. Evalution

We now evaluate the three queries from Section 4.1 for the index-structures from Chapter 3. The benchmarks measure the time spent on building the index-structures from the input relations (build phase) and the time needed to perform the join (join phase). For the first two queries, we consider the LEAPFROG TRIEJOIN, the HASHBASED TRIEJOIN, and the HYBRID TRIEJOIN from Chapter 2. For the third query, we compare the runtimes with a pipelined HASH-JOIN implementation.

4.3.1. Interleaved Query

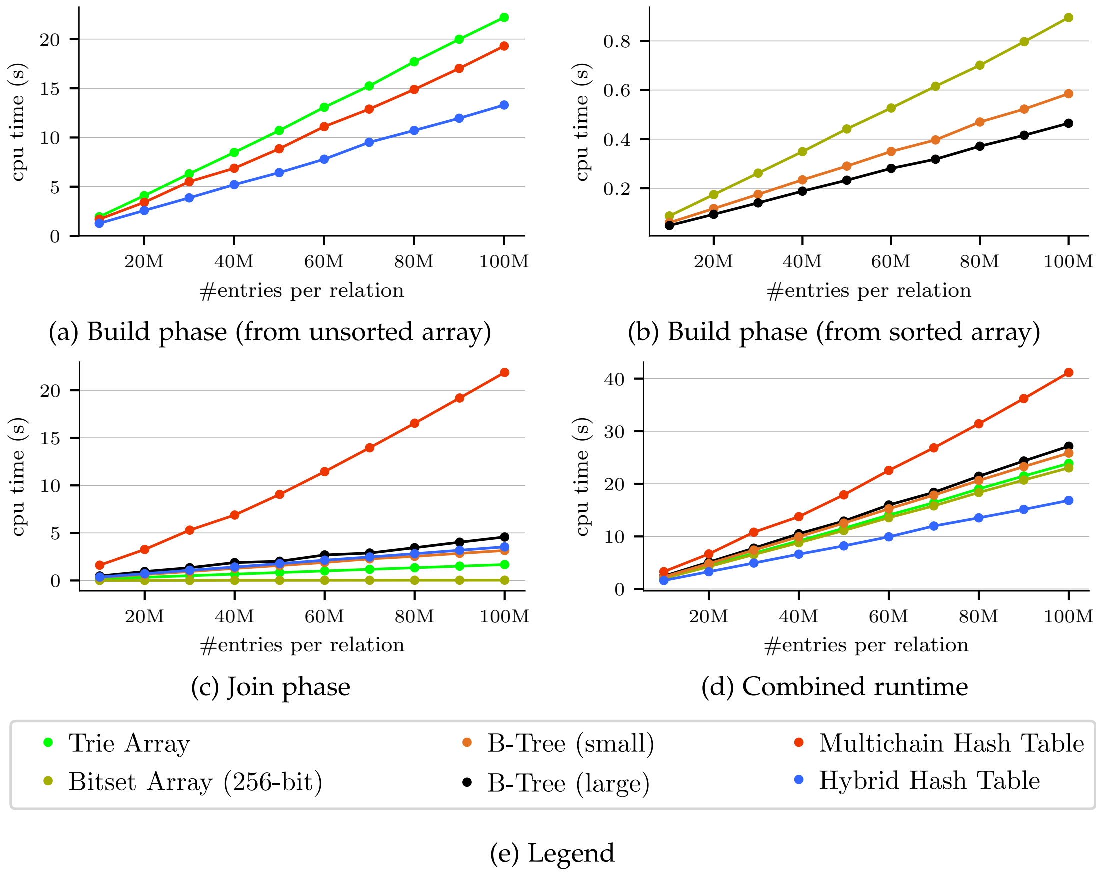
We evaluate the Interleaved Query for all unsorted and sorted index-structures. Figure 4.2 shows the result of the benchmark for 10 million to 100 million elements per relation. However since all input relations are 1-ary, both the Flat Array and the Trie Array correspond to a sorted array, and only one of them is displayed. Furthermore, the diagrams present only the best-performing Bitset Array. In this case, the parameter p equals 8, and the index-structure uses bitsets of size 256.

The query joins three different datasets; hence, for all three relations, an index-structure is needed. Figure 4.2a compares the build phase of the Trie Array with the Multichain Hash Table and the Hybrid Hash Table. The time spent on creating the index-structures seems to grow linearly. However, the CPU time for the Trie Array increases faster than for the hash tables. The additional log factor for sorting the data causes this behavior. Building the Multichain Hash Table takes longer than the Hybrid Hash Table since maintaining references is more expensive. A reference causes an additional indirection when calculating the hash or checking for equality in case of a hash collision.

Figure 4.2b compares the construction of the different sorted index-structures from a sorted array. The Trie Array corresponds to a sorted array in a 1-ary relation and is thus not considered. As already explained in Section 3.1, the index-structures are built in one or two iterations from a sorted array. The large B-Tree stores up to 512 32-bit integer keys in every node. The small B-Tree with 16 keys per node consists of more nodes, and therefore, more time is needed to build it. The runtimes for creating Bitset Arrays with 64-, 128-, or 256-bit wide bitsets vary by less than one percent.

Joining the three relations demonstrates that the HYBRID TRIEJOIN, in combination with the Hybrid Hash Table, can compete with the LEAPFROG TRIEJOIN (see Figure 4.2c). The CPU time is between the large and the small B-Tree. Joining the large B-Trees is slower since the `seek`-function has to search in larger nodes. The HASHBASED TRIEJOIN with the Multichain Hash Table, on the other hand, shows a considerably worse runtime behavior. It is roughly four times slower than the HYBRID TRIEJOIN. The indirection caused by the references slow the join down.

The LEAPFROG TRIEJOIN based on the Trie Array and the Bitset Array perform best. Table 4.2 gives a more detailed comparison for joining the Level Array and the three different Bitset Arrays. In particular, it illustrates the benefits of vector instructions. With



(a) Build phase (from unsorted array)

(b) Build phase (from sorted array)

(c) Join phase

(d) Combined runtime

(e) Legend

Figure 4.2.: Interleaved Query benchmark results

every increase of the bitset size, the CPU time is almost reduced by factor 2. Compared to the Trie Array, the Bitset Array (64-bit) is just 16 times faster. The reason for this is the overhead paid for managing the bitsets.

Table 4.2.: CPU time for the Bitset Arrays in the join phase for the Interleaved Query

size	Trie Array	Bitset 64	Bitset 128	Bitset 256
20M	335.3ms	19.3ms	10.3ms	7.1ms
40M	669.6ms	38.6ms	20.5ms	12.7ms
60M	1004.6ms	57.8ms	31.3ms	19.1ms
80M	1339.6ms	77.4ms	41.1ms	25.4ms
100M	1674.6ms	96.6ms	51.5ms	30.4ms

Figure 4.2d combines the join phase and the build phase. For the Leapfrog Triejoin, the Bitset Array (256-bit) is the fastest index-structure. The time required for sorting the datasets dominates the overall execution time. For the Hybrid Hash Table, the ratio

between building and joining is more balanced and the structure outperforms the other index-structures and join algorithms.

4.3.2. Hypercube Query

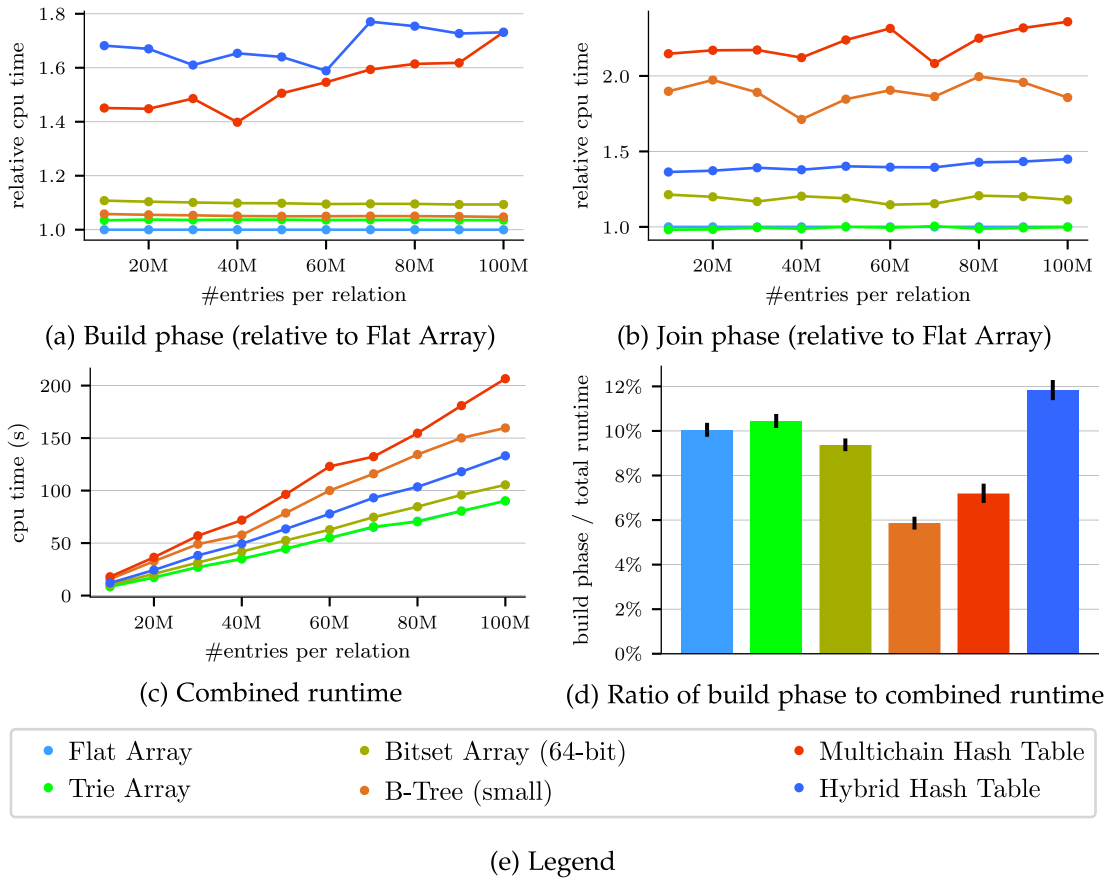
This benchmark focuses on building index-structures for binary relations and evaluating a query that performs multiple join operations. The Hypercube Query performs a 6-way self-join, hence only one index-structure is needed. As in the previous benchmark, the large B-Tree is slower and is not considered. We execute the query for ten instances of the Hypercube dataset H_m^2 with $m \in \{2500000 \cdot i \colon i \in \{1, \ldots, 10\}\}$. Therefore, the input relation contains 10 million to 100 million entries. Even for the smallest datasets, a pairwise join plan yields an intermediate result of more than 12 trillion tuples. As it is not possible to execute the join within a reasonable period of time, only worst-case optimal join algorithms are evaluated. Figure 4.3a shows the result of building the index-structures. The diagram depicts the runtimes relative to the Flat Array.

This time, sorting the dataset is faster than filling the Multichain Hash Table or the Hybrid Hash Table. Since every tuple in the dataset consists of two integers, the Multichain Hash Table has to maintain two hash indexes. In particular, it has to perform the insert operation twice per tuple. Although the Hybrid Hash Table manages just one hash map, it has to fill and sort the vectors. For small datasets, it is faster to fill Multichain Hash Table, but two different trends can be observed.

The build time for the Multichain Hash Table increases continuously in comparison to the Flat Array, whereas for the Hybrid Hash Table, it decreases. The spike between 60 million and 70 million entries per relation is an exception from this observation. For 70 million entries the Hypercube dataset contains 17.5 million distinct values on the first trie level. The hash map in the Hybrid Hash Table has space for at least two times more elements (35 million entries). However, the size of the robin map is always a power of two. As a result, for 35 million entries, the robin map allocates memory for more than 67 million keys. For 30 million entries it allocated only memory for 33 million keys. The increase of the robin map causes the spike in the diagram. Building the other sorted index-structures from the Flat Array requires 5% to 10% extra work.

For the Hypercube Query, the smallest bitset (64-bit) yields the best results. The second level of the trie for the Hypercube dataset is very sparse, i.e., in most cases it consists only of two children. Since vector instructions are slower than instructions on standard registers, the 128-bit and 256-bit Bitset Arrays slow the Leapfrog Triejoin down.

The results of the join phase are shown in Figure 4.3b relative to the Flat Array. The Leapfrog Triejoin in combination with the Flat Array and the Trie Array perform almost identical. The reason for this is the optimized binary search. Before the algorithm searches the entire remaining array, it first checks whether the sought element is in the immediate surrounding. For the Hypercube Query, this assumption holds in most cases, and the `seek`- and `next`-functions on the Flat Array take almost constant time. The Hybrid Triejoin is approximately 40% slower than the best Leapfrog



(a) Build phase (relative to Flat Array)

(b) Join phase (relative to Flat Array)

(c) Combined runtime

(d) Ratio of build phase to combined runtime

(e) Legend

Figure 4.3.: Hypercube Query benchmark results

Triejoin. However, it is faster than the Leapfrog Triejoin based on B-Trees. As with the Interleaved Query, the Hashbased Triejoin in combination with the Multichain Hash Table is outperformed due to the high number of cache misses.

Figure 4.3c and Figure 4.3d show the combined runtime. The result of the query is 8 times larger than the dataset H_m^2. The join phase thus dominates the overall runtime. The ratio between the build phase and combined runtime is on average less than 12%. Figure 4.3d shows the mean value and standard deviation. The three array structures, the Flat Array, the Trie Array, and the Bitset Array, perform best. Although it takes longer to build the Hybrid Hash Table, the faster join phase results in a better total runtime compared to the B-Trees. The Hashbased Triejoin, in combination with the Multichain Hash Table, has the highest runtime for this query.

4.3.3. Twitter Query

The Twitter Query demonstrates the performance of the three TRIEJOINS for a real-world workload and compares them to a pipelined HASH-JOIN. For the B-Tree index-structure and the Bitset Array, we consider only the best-performing configurations (small B-Tree and 64-bit Bitset Array). The query joins the relations $T(a, b)$, $T(b, c)$ and $T(c, a)$ under the global attribute order $[a, b, c]$. Although a 3-way self-join is performed, we need to inverse the third relation $T(c, a)$, i.e., $T'(a, c) = \{(a, c) \colon (c, a) \in T\}$. Consequently, two index-structure are allocated: one for the relations $T(a, b)$ and $T(b, c)$ and one for the inversed relation $T'(a, c)$.

The first benchmark examines the build phase for the Hybrid Hash Table more closely. In Section 3.2.2, we identified two steps for building the index-structure:

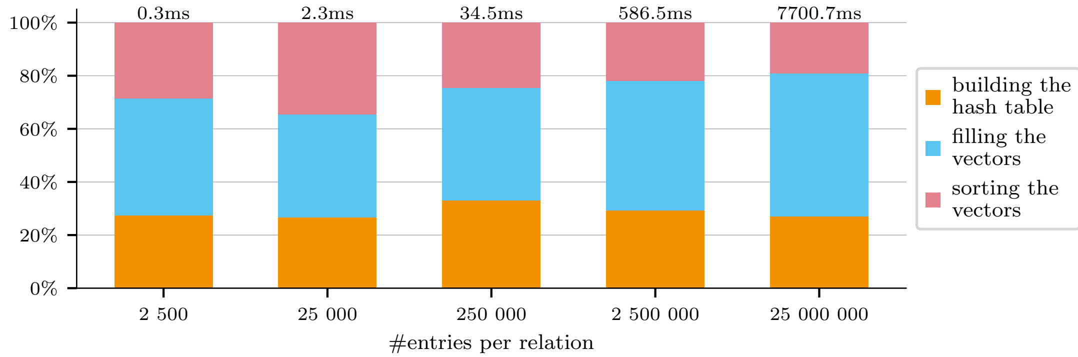
1. inserting the tuples from the relations into the hash map and the vectors and
2. sorting the vectors.

For the first step, we further distinguish between inserting the keys into the hash map (building the hash table) and appending the elements to the vectors (filling the vectors). Figure 4.4a presents the time spent on these three operations. The time required for allocating and building the hash map is constantly around 25% of the total build time. From 25 thousand to 25 million entries in the Twitter dataset, the time for sorting the vectors reduces. Most of the time is spent on appending elements to the vectors (40% - 50%).

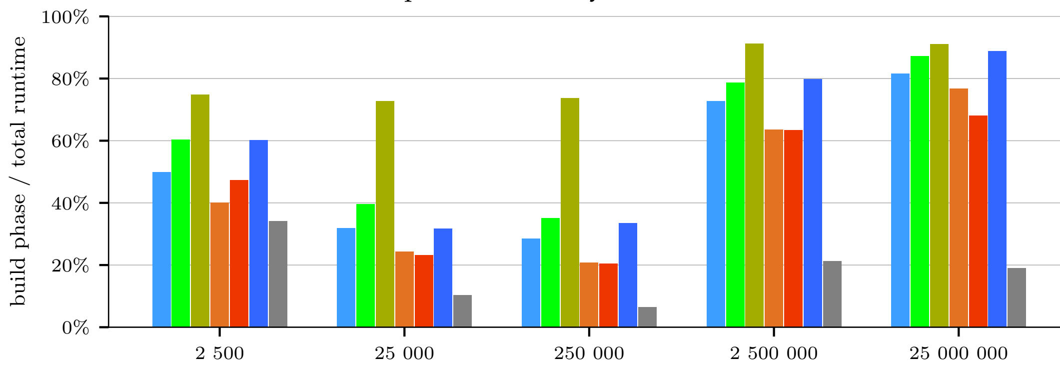
Table 4.3 lists the combined runtimes for building the index-structures and performing the join. Figure 4.4b shows the ratio of the build phase to the total runtime. Both the Flat Array and the Trie Array behave similarly. For the three smallest datasets, the impact of the build phase reduces from 50% / 60% to 30% / 35%. For the two largest datasets, the build phase is once again the dominating factor. The explanation can be found in Table 4.1. For 25 000 and 250 000 entries per relation, the number of triangles is several times larger than the size of the input relations. Therefore, the join phase dominates the combined runtime. For the other input sizes, the query result is relatively small, and the build phase consumes more time compared to the join phase.

The same holds for the small B-Tree, the Multichain Hash Table, and the Hybrid Hash Table. If the query result is larger than the Twitter dataset, the join phase dominates. Otherwise, the build phase is the primary factor. Compared to HASHBASED TRIEJOIN and LEAPFROG TRIEJOIN in combination with B-trees, the HYBRID TRIEJOIN seems to be faster. Figure 4.4c confirms this assumption. It shows the time spent on the build phase and join phase for 25 million edges in the Twitter dataset. In exchange for a fast join phase, the build phase for the Hybrid Hash Table takes longer.

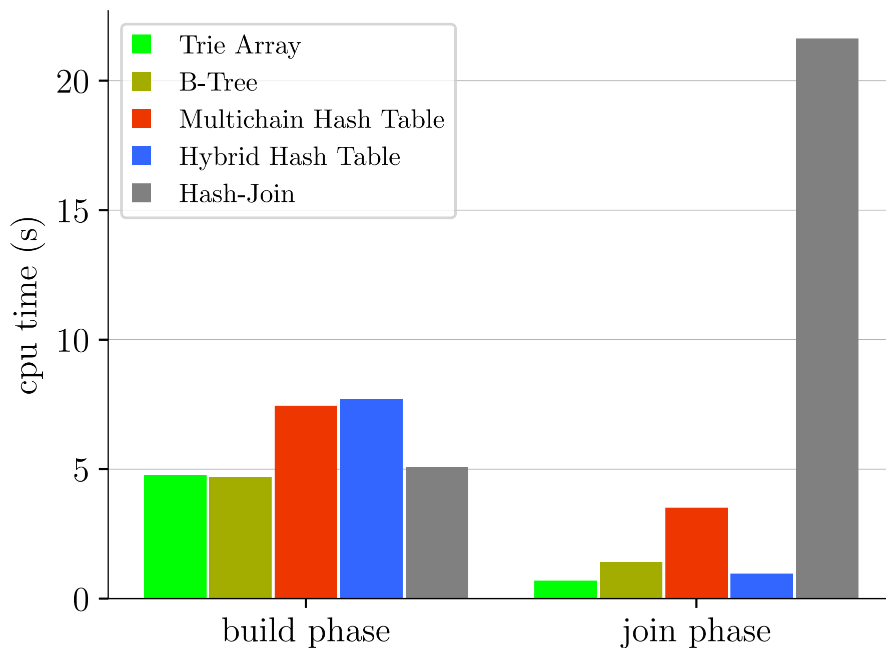
The build phase of the Bitset Array consumes a constantly high proportion of the total runtime. However, this index-structure achieves the best runtimes for the join phase. The same applies to the HASH-JOIN with the exception that here the join phase dominates permanently. In contrast to the worst-case optimal join algorithms, the size of



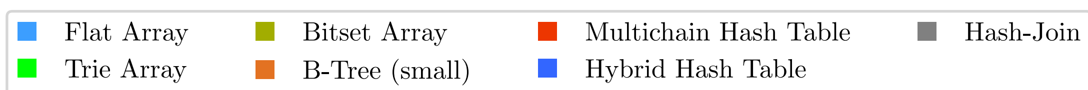
(a) Build phase for the Hybrid Hash Table



(b) Ratio of build phase to combined runtime



(c) Build and join phase for 25 000 000 entries per relations



(d) Legend

Figure 4.4.: Twitter Query benchmark results

Table 4.3.: Combined runtimes for the Twitter Query

	2 500	25 000	250 000	2 500 000	25 000 000
Flat Array	0.43ms	9.4ms	120.0ms	547.4ms	5589.8ms
Trie Array	0.39ms	8.0ms	102.9ms	524.9ms	5464.4ms
Bitset Array (64-bit)	0.41ms	5.1ms	56.8ms	536.1ms	5877.5ms
B-Tree (small)	0.56ms	12.6ms	168.9ms	639.3ms	6094.0ms
Multichain Hash Table	0.36ms	7.8ms	165.1ms	978.9ms	10949.8ms
Hybrid Hash Table	0.42ms	7.2ms	103.1ms	735.2ms	8664.5ms
Hash-Join	0.33ms	11.5ms	280.7ms	1920.0ms	26681.3ms

the query result is not decisive but rather the size of the intermediate result. Therefore, the Hash-Join is slower from 25 000 entries per relation onwards. For the largest dataset, the join phase takes roughly 5 to 10 times longer than that of the other join algorithms.

In conclusion, the Hybrid Triejoin and the Leapfrog Triejoin are better suited for large query results. This situation minimizes the impact of building the index-structures and increases the benefits of the fast join phase. In the case of small query results, the Leapfrog Triejoin based on the Flat Array or the Trie Array achieves better runtimes.

4.4. Discussion

We evaluated the six index-structures and the corresponding join algorithm for three different queries. Taking into account the results of our experiments and the theoretical analysis of the index-structure, we can give the following summary:

The Flat Array has the shortest build time and average join time among the sorted index-structures and is well suited for building on-the-fly. However, modifications like inserting, updating, or deleting elements are expensive. Therefore, the index-structure is built anew if the input relations changes.

The Trie Array provides a good tradeoff between the build phase and the join phase. Like the Flat Array, in most cases, the index-structure cannot be used throughout multiple queries.

The Bitset Array works best with dense datasets. In the Twitter Query, it outperforms the other structure in the join phase. However, the time required to build the structure is considerably higher. The best results are achieved for queries where the result is larger than the input relations.

The B-tree's biggest advantage is that it can efficiently handle modifications caused by transactions. As a result, the structure can be used throughout multiple queries. The join phase for this index-structure is roughly two times slower than that of the other sorted index-structure.

The Multichain Hash Table, in combination with the HASHBASED TRIEJOIN, has the best asymptotic complexity in the build and join phase but runs slowest in practice. It remains to be seen whether the structure with the optimizations from Chapter 5 can compete with the LEAPFROG TRIEJOIN.

The Hybrid Hash Table is, in most cases, faster than the LEAPFROG TRIEJOIN based on B-Trees. However, for smaller datasets, it still takes longer to build the structure than sorting the entire relation. Like the B-Tree, the Hybrid Hash Table can handle insert, update or delete operations more efficiently than one of the array structures.

From these results, we observe two main characteristics of index-structures for worst-case optimal join algorithms:

1. The time spent on building the index-structures must not be underestimated. Depending on the size of the relations and the query result, the build phase may dominate the execution time. As a result, building the index-structures every time from scratch is not feasible in practice. The B-Tree, the Multichain Hash Table and the Hybrid Hash Table avoid this problem by reducing the costs for insert, delete or update operations in comparison to the Flat Array, the Trie Array or the Bitset Array.

2. However, the efficient implementation of these operations leads to reduced performance in the join phase. Furthermore, we observed an increased memory consumption with the usage of B-Trees or hash tables.

5. Future Work

For both the Hashbased Triejoin, in combination with the Multichain Hash Table, and the Hybrid Triejoin, in combination with the Hybrid Hash Table, future work can be conducted. The Hybrid Triejoin, in particular, offers further potential for optimizations. It remains to be seen to what extent it can profit from existing data-structures in modern database systems.

5.1. Hashbased Triejoin and Multichain Hash Table

A limiting factor for the Hashbased Triejoin is the high number of cache misses during the join phase. Michael Freitag suggested to use radix partitioning before building the Multichain Hash Table in order to increase spatial locality. In particular, all elements in a chain would reside in the same partition. As a result, the next-function causes fewer cache misses when iterating the elements of a chain. Whether this optimization also affects the performance of the jump-function remains to be seen.

Currently, the Multichain Hash Table uses the CityHash algorithm as hash-function. For a tuple consisting of k-elements, the hash-function is called k times. In every iteration, the hash is calculated for the previous elements and one additional element. An optimized hash-function that reuses the hash for the previous elements and appends the new element could reduce the time required to evaluate the hash-function.

Another optimization incorporates parts of the join phase into the build phase. For the Twitter Query, we first build the index-structure for $T(a,b)$ and then we build the index-structure for $T'(a,c)$ (the inverse of $T(c,a)$). In the join phase, the Hashbased Triejoin first computes the intersection of $\pi_a(T(a,b))$ with $\pi_a(T'(a,c))$. This step can be optimized as follows: While building the index-structure for $T'(a,c)$ check for every value of a, whether it is in $\pi_a(T(a,b))$. If so, add it to the index-structure for $T'(a,c)$. The jump-function performs the check efficiently by executing a lookup operation in the Multichain Hash Table for $T(a,b)$. As a result, the index-structure for $T'(a,c)$ contains only the values for a that are also in $T(a,b)$, i.e., the result of the set intersection. The first set intersection from the join phase is moved to the build phase. Furthermore, the size of the second index-structure is reduced. Alternatively, a Bloom filter can be used to reduce the cost of the lookup operation.

In the current configuration, the Hashbased Triejoin and the Multichain Hash Table exploit only single-core performance. The join algorithm solely performs read operations on the index-structure. Therefore, it should be straightforward to parallelize the Hashbased Triejoin. It will be interesting to investigate how well the build phase

and the join phase scale in a multi-core setup.

5.2. Hybrid Triejoin and Hybrid Hash Table

The Hybrid Triejoin currently assumes that only the last level of the trie is sorted. At the center of the algorithm is the assumption that the inner nodes partition the leaf nodes in small partitions. However, this may not be true for all datasets or attribute orders. Some nodes may have several thousand or million children, and sorting these is expensive. We propose a different approach. Currently, the level in the trie decides whether to sort the children of a node. A better choice may be the number of children a node possesses. For example, if a node in the trie has

- $\leq$ 16 children: store them in an unsorted array (iterate the entire array for finding a key).
- $\leq$ 1000 children: store them in a sorted array (use binary search for finding a key). If the elements are dense enough, store them in a bitset array.
- $>$ 1000 children: store them in a hash table (perform lookup-operation in the hash table for finding a key).

Finding the best limits and index-structures for this approach is left to future work.

Another optimization comes to mind investigating Figure 4.4a from Section 4.3.3. The diagram shows that roughly 75% percent of the build phase for the Twitter Query is spent on inserting the elements into the vectors and sorting them. Index-structure that are automatically sorted after every insert may reduce the build time. For this purpose, sorted tree data-structures like Red-Black Trees or B-Trees come to mind.

Furthermore, the optimization presented in the last section for the Hashbased Triejoin and the Multichain Hash Table also apply to the Hybrid Triejoin and the Hybrid Hash Table.

6. Related Work

At the center of this paper are the GENERIC-JOIN and the LEAPFROG TRIEJOIN algorithm. Although the LEAPFROG TRIEJOIN is an instance of the GENRIC-JOIN algorithm, LogicBlox developed the algorithm independently. Ngo, Porat, Ré, and Rudra first described a worst-case optimal join algorithm in [3] and refined their ideas later in [6] to the GENRIC-JOIN algorithm. Meanwhile, Veldhuizen described in [4] the LEAPFROG TRIEJOIN and the interface for the trie iterators. Besides, he proposed a B-Tree as possible index-structure.

Chu, Balazinska, and Suciu evaluated the LEAPFROG TRIEJOIN in a massively parallel architecture consisting of multiple servers. Initially, the HyperCube algorithm distributes the relations efficiently. Then every server sorts the obtained data and evaluates the query using the LEAPFROG TRIEJOIN. This paper motivated the Flat Array index-structure we introduced in Section 3.1.1. The authors chose this structure due to its small build time.

Wu, Zinn, Aref, and Yalamanchili developed in [11] the Trie Array index-structure from Section 3.1.2. The author optimized the LEAPFROG TRIEJOIN for General Purpose GPUs. The Trie Array allowed them to implement the trie iterator more efficiently. The complexity of `next()` reduces to $\mathcal{O}(1)$, and although the `seek`-function still takes $\mathcal{O}(\log(n))$ time, it will be faster than the `seek`-function for the Flat Array or the B-Tree. Our results from the Twitter Query confirm the improved runtime of the trie iterator for the Trie Array. However, it takes longer to build this structure.

The next sorted index-structure, the Bitset Array, was inspired by the papers on the query processing engines EmptyHeaded [12] and LevelHeaded [8]. EmptyHeaded introduced the idea of the bitsets. Furthermore, Aberger, Tu, Olukotun, and Ré used dictionary encoding in order to increase the density of the data and the effectiveness of the bitsets. EmptyHeaded already used the GENERIC-JOIN algorithm for evaluating joins. LevelHeaded refined the techniques from EmptyHeaded. In particular, the paper came up with the observation that only the first level of the trie is dense. Furthermore, the authors evaluated LevelHeaded for queries from business intelligence and linear algebra and compared it to other relational database engines. Their results indicate that worst-case optimal join algorithms are particularly well suited for queries from linear algebra.

7. Conclusion

This thesis evaluated different index-structures for their strengths and limitations in the context of worst-case optimal join algorithms. First, we gave an example of a query that pairwise join plans cannot evaluate efficiently. We then discussed the generic worst-case optimal join algorithm from [6]. A concrete instance of this algorithm, the Leapfrog Triejoin, is used in the commercial database system LogicBlox. The algorithm also defines a simple iterator interface that accesses the elements in the relation in a sorted manner. Besides, it establishes runtime bounds for the interface function so that the algorithm is worst-case optimal (up to a log-factor) [4].

We considered three array-like index-structures, the Flat Array, the Trie Array, and the Bitset Array, and a B-Tree. For three different queries, the time for building and joining the index-structures was measured. The array-like structures outperform the B-Tree in all three queries. The Twitter Query and the Interleaved Query suggest that bitsets can reduce the runtime for the join phase drastically.

Furthermore, this thesis extended the Leapfrog Triejoin to unsorted index-structures. With the Hashbased Triejoin, we presented an algorithm that is asymptotically faster than the Leapfrog Triejoin. The algorithm adds two new functions to the trie iterator interface and tightens the complexity bounds for the other interface functions. The index-structure we developed for this join algorithm also improves the runtime complexity of the build phase. The Multichain Hash Table avoids sorting the materialized relations and only uses hash tables that are built in linear time.

The Hybrid Triejoin exploits both sorted and unsorted index-structures. Depending on the index-structure, the algorithm uses the iterator interface from the Leapfrog Triejoin or the Hashbased Triejoin. The Hybrid Hash Table combines hash tables and sorted arrays. In comparison to the Multichain Hash Table, the Hybrid Hash Table reduces the number of indirections and cache misses. However, the structure still sorts partitions of the materialized relations. In general, it is not built in linear time. Although the runtime complexity of the Hybrid Triejoin is worse than that of the Hashbased Triejoin, our results show that the Hybrid Triejoin is in practice faster.

The results of the experiments led to the following conclusions. It is not practical to build index-structures over materialized relations for every query from scratch. Therefore, the index-structures have to deal with modifications caused by transactions efficiently. However, index-structures that allow modifications, like inserting, deleting, or updating elements, perform worse in the join phase.

Appendix

A. Algorithms

A.1. Leapfrog Triejoin

```
globals: array Iter, integer size, boolean atEnd, integer pos, integer depth

leapfrog-search():
    max_key:= Iter[(pos - 1) mod size].key()
    while true:
        min_key := Iter[pos].key()
        if max_key == min_key:
            return
        else:
            Iter[pos].seek(max_key)
            if Iter[pos].atEnd():
                atEnd := true
                return
            end if
            max_key := Iter[pos].key()
            pos := (pos + 1) mod size
        end if
    end while
end

leapfrog-init():
    size := Iter.size()
    if ∃p ∈ [size]: Iter[p - 1].atEnd():
        atEnd := true
    else:
        atEnd := false
        pos := 0
        sort Iter by key() of each iterator
        leapfrog-search()
    end if
end

key():
    return Iter[pos].key()
end

atEnd():
    return atEnd
end

next():
    Iter[pos].next()
    if Iter[pos].atEnd():
        atEnd := true
    else:
        pos := (pos + 1) mod size
        leapfrog-search()
    end if
end

seek(seek_key):
    Iter[pos].seek(seek_key)
    if Iter[pos].atEnd():
        atEnd := true
    else:
        pos := (pos + 1) mod size
        leapfrog-search()
    end if
end

open():
    depth := depth + 1
    for each iter at current depth:
        iter.open()
    end for
    leapfrog-init()
end

up():
    for each iter at current depth:
        iter.up()
    end for
    depth := depth - 1
end
```

Figure A.1.: Full Leapfrog Triejoin implementation

A.2. Hashbased Triejoin

```
globals: array Iter, integer size, boolean atEnd, integer depth

hashbased-search():
    while not Iter[0].atEnd():
        key := Iter[0].key()
        equals := true
        for p ∈ [size - 1] and equals:
            equals := Iter[p].jump(key)
        end for
        if equals:
            return
        else:
            Iter[0].next()
        end if
    end while
    atEnd := true
end

hashbased-init():
    size := Iter.size()
    if ∃p ∈ [size]: Iter[p - 1].atEnd():
        atEnd := true
    else:
        atEnd := false
        find p with Iter[p].length() minimal
        swap Iter[0] and Iter[p]
        hashbased-search()
    end if
end

key():
    return Iter[0].key()
end

length():
    return Iter[0].length()
end

next():
    Iter[0].next()
    hashbased-search()
end

jump(key):
    for p ∈ [size]:
        if not Iter[p - 1].jump(key):
            return false
        end if
    end for
    return true
end

atEnd():
    return atEnd
end

open():
    depth := depth + 1
    for each iter at current depth:
        iter.open()
    end for
    leapfrog-init()
end

up():
    for each iter at current depth:
        iter.up()
    end for
    depth := depth - 1
end
```

Figure A.2.: Full HASHBASED TRIEJOIN implementation

A.3. Hybrid Triejoin

```
globals: array SortedIter, array UnsortedIter, integer sorted_size, integer unsorted_size, boolean atEnd, integer pos

hybrid-init():
    unsorted_size := UnsortedIter.size()
    sorted_size := SortedIter.size()
    if any iterator is atEnd():
        atEnd := true
        return
    else:
        atEnd := false
        pos := 0
        sort SortedIter by key() of each iterator

        find p with UnsortedIter[p].length() minimal
        length := UnsortedIter[p].length()
        swap UnsortedIter[0] and UnsortedIter[p]

        mixed-search()
    end if
end

key():
    if sorted_size > 0:
        return SortedIter[0].key()
    else:
        return UnsortedIter[0].key()
    end if
end

length():
    return UnsortedIter[0].length()
end

next():
    if sorted_size > 0:
        SortedIter[pos].next()
        atEnd := SortedIter[pos].atEnd()
        pos := (pos + 1) mod sorted_size
    else:
        UnsortedIter[0].next()
        atEnd := UnsortedIter[0].atEnd()
    end if
    hybrid-search()
end

seek(key):
    SortedIter[pos].seek(key)
    atEnd := SortedIter[pos].atEnd()
    pos := (pos + 1) mod sorted_size
    hybrid-search()
end

jump(key):
    if sorted_size > 0:
        SortedIter[pos].seek(key)
        atEnd := SortedIter[pos].atEnd()
        pos := (pos + 1) mod sorted_size
    else:
        UnsortedIter[0].jump()
        atEnd := UnsortedIter[0].atEnd()
    end if
    hybrid-search()
    return (not atEnd) && key == key()
end

atEnd():
    return atEnd
end

open():
    depth := depth + 1
    for each iter at current depth:
        iter.open()
    end for
    hybrid-init()
end

up():
    for each iter at current depth:
        Iter[p].up()
    end for
    depth := depth - 1
end
```

Figure A.3.: Full Hybrid Triejoin implementation

List of Figures

2.1. Right-deep join plan for the query $R \bowtie S \bowtie T$ 3
2.2. SQL representation of the triangle query $Q_\triangle$ 4
2.3. Hypergraph representation of the triangle query $Q_\triangle$ 4
2.4. Trie representation for an instance of $R(a,b)$, $S(b,c)$ and $T(c,a)$ 7
2.5. Leapfrog Triejoin implementation 9
2.6. Leapfrog Join for three arbitrary relations A, B and C 10
2.7. Hashbased Triejoin implementation 12
2.8. Trie representation of the keys used for probing by the `hashbased-search`-function 13
2.9. Hybrid Triejoin implementation 15

3.1. Relation $R(a,b)$ from Equation 2.2 ($m = 2$) 17
3.2. Flat Array for the relation $R(a,b)$ 18
3.3. Trie Array for the relation $R(a,b)$ 20
3.4. Bitset representation of the relation $R(a,b)$ 21
3.5. Bitset Array for the relation $R(a,b)$ 22
3.6. B-Tree of degree 2 for the relation $R(a,b)$ 24
3.7. Multichain Hash Table for the relation $R(a,b)$ 26
3.8. Partially sorted relation $Q_\triangle(a,b,c)$ 29
3.9. Hybrid Hash Table for the relation $Q_\triangle(a,b,c)$ 30

4.1. 2D, 3D and 4D Hypercube 32
4.2. Interleaved Query benchmark results 35
4.3. Hypercube Query benchmark results 37
4.4. Twitter Query benchmark results 39

A.1. Full Leapfrog Triejoin implementation 51
A.2. Full Hashbased Triejoin implementation 52
A.3. Full Hybrid Triejoin implementation 53

List of Tables

2.1. Trie iterator interface for sorted datasets 8
2.2. Trie iterator interface for unsorted datasets 11

4.1. Sizes for the Twitter Query 33
4.2. CPU time for the Bitset Arrays in the join phase for the Interleaved Query 35
4.3. Combined runtimes for the Twitter Query 40

Bibliography

[1] W. Xu. “Toward Human-Centered AI: A Perspective from Human-Computer Interaction.” In: *Interactions* 26.4 (June 2019), pp. 42–46. ISSN: 1072-5520. DOI: 10.1145/3328485.

[2] A. Kemper and A. Eickler. *Datenbanksysteme: Eine Einführung*. De Gruyter Studium. De Gruyter Oldenbourg, 2015. ISBN: 9783110443752.

[3] R. K. Mothilal, A. Sharma, and C. Tan. “Explaining Machine Learning Classifiers through Diverse Counterfactual Explanations.” In: *CoRR* abs/1905.07697 (2019). arXiv: 1905.07697.

[4] D. Wootton and C. Feldman. “The diagnosis of pneumonia requires a chest radiograph (x-ray) – yes, no or sometimes?” In: *pneumonia: A Peer Reviewed Open Access Journal* 5 (June 2014), p. 1. DOI: 10.15172/pneu.2014.5/464.

[5] H. Q. Ngo. “Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems.” In: *CoRR* abs/1803.09930 (2018). arXiv: 1803.09930.

[6] A. Holzinger, C. Biemann, C. S. Pattichis, and D. B. Kell. “What do we need to build explainable AI systems for the medical domain?” In: *CoRR* abs/1712.09923 (2017). arXiv: 1712.09923.

[7] A. Atserias, M. Grohe, and D. Marx. “Size Bounds and Query Plans for Relational Joins.” In: *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*. FOCS ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 739–748. ISBN: 978-0-7695-3436-7. DOI: 10.1109/FOCS.2008.43.

[8] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks.” In: *CoRR* abs/1703.10593 (2017). arXiv: 1703.10593.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[10] S. Chu, M. Balazinska, and D. Suciu. “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*. SIGMOD ’15. Melbourne, Victoria, Australia: ACM, 2015, pp. 63–78. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2750545.

[11] H. Wu, D. Zinn, M. Aref, and S. Yalamanchili. “Multipredicate Join Algorithms for Accelerating Relational Graph Processing on GPUs.” In: Sept. 2014.

[12] C. R. Aberger, A. Nötzli, K. Olukotun, and C. Ré. "EmptyHeaded: Boolean Algebra Based Graph Processing." In: *CoRR* abs/1503.02368 (2015). arXiv: `1503.02368`.

[13] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014.