DEPARTMENT OF MATHEMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Mathematics in Data Science

Adaptive Radix Trees for Trigram Indexing in Main Memory Database Systems

Nasiba Zokirova

DEPARTMENT OF MATHEMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Mathematics in Data Science

Adaptive Radix Trees for Trigram Indexing in Main Memory Database Systems

Adaptive Radix-Bäume für die Trigramm-Indizierung in Hauptspeicherdatenbanksystemen

Author:	Nasiba Zokirova
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Adrian Riedl, M.Sc.
Submission Date:	June 2, 2025

I confirm that this master's thesis in mathematics in data science is my own work and I have documented all sources and material used.

Munich, June 2, 2025 Nasiba Zokirova

Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Dr. Thomas Neumann, for giving me the opportunity to pursue and complete this master's thesis under his guidance.

Special thanks to my advisor, Adrian, for believing in my abilities, for his patience in explaining complex concepts, and for his thoughtful direction and advice at every stage of the work. His insights, encouragement, and high standards have been invaluable throughout this journey.

Finally, I must thank the rainy German weather, its unrelenting clouds and showers provided the perfect backdrop for focused study, and without it, I might never have found the discipline to see this project through to the end.

Abstract

In this thesis, we examine the feasibility of replacing PostgreSQL's traditional B-tree structured entry tree in Generalized Inverted Index (GIN)-based trigram indexes with an in-memory Adaptive Radix Tree (ART) to accelerate substring searches over textual data. Full-text and pattern matching, particularly LIKE expressions are fundamental yet often performance-critical operations in database systems.

We reimplement PostgreSQL's trigram indexing mechanism in C++, faithfully reproducing trigram extraction, posting-list storage (inline and tree-based), and the B-tree based entry tree construction. We then substitute the B-tree with an ART entry tree. Our implementation stays maximally loyal to PostgreSQL's model and bulk-load interfaces, enabling a direct comparison under realistic workloads.

To evaluate performance, we benchmark both index variants on TPC-H dataset at scale factors 1 and 10, measuring bulk-load execution time, memory footprint, and LIKE-query latency across short, medium, and long patterns. We also analyze the contribution of entry tree traversal to end-to-end query time and verify theoretical depth bounds for both structures.

Our findings demonstrate that, with a static data, in in-memory setting, ART can serve as a drop-in replacement for B-tree entry layers without degrading performance. This thesis contributes a detailed design and evaluation of an ART-based GIN entry layer, offering database developers a viable path to leverage adaptive in-memory structures for high-throughput substring search workloads.

Contents

1. Introduction

Object-relational database systems combine the expressive power of object models with the proven reliability of relational engines. Among these platforms, PostgreSQL has emerged as the most popular database for two consecutive years in the StackOverflow Developer Survey[1]. Its open-source, C-based codebase, publicly maintained on GitHub, provides an ideal research testbed for extending and refining core database functionality.

At the heart of any high-performance database lies its suite of indexing structures. Hash indexes excel at exact-match lookups; bitmap indexes support rapid multi-column filtering; inverted indexes power full-text search and other pattern-matching queries. As the volume of online text continues to escalate, efficient substring search, often expressed via SQL's `LIKE` operator, remains a vital capability. For instance, retrieving all records whose "description" contains the substring "cake" represents a common yet challenging workload.

This thesis investigates whether the inverted indexing can be optimized for `LIKE` pattern matching in PostgreSQL. We begin by dissecting the internal architecture of GIN and re-implement its core components in C++. Our primary obejctive is to replace GIN's traditional B-tree-based entry tree with the in-memory ART, a trie variant that adapts node layouts to the actual data distribution. Prior work by Leis *et al.* [1] demonstrated ART's superiority over cache-sensitive B-tree variants (e.g. CSB$^+$-tree) for random key lookups; we perform a head-to-head comparison of the classic B-tree entry tree versus our ART-based entry tree in the specific context of PostgreSQL's trigram-indexing mechanism for LIKE pattern matching.

This substitution is particularly well-justified in today's environments, where large main memories can hold the entire data. Under these circumstances, the traditional B-tree's reliance on fixed-size pages and disk-oriented node splitting imposes unnecessary memory and pointer overhead. By contrast, ART's adaptive node layouts shrink to fit each node's actual degree, reducing wasted space and improving cache locality when the index resides entirely in main memory.

The contributions of this work are threefold. First, we design and implement an ART-based entry tree for GIN-trigram indexing, replicating the core data structure and lookup logic of PostgreSQL's entry layer in an in-memory setting. Secondly, we conduct a benchmark study comparing build times and lookup latency between the classic B-tree entry structure and our ART-enhanced index. Lastly, we characterize the text data in which ART could offer tangible performance gains over B-tree in the context of trigram indexing.

In the chapters that follow, we review relevant preliminary theory, detail our implementation of the modified GIN, describe our experimental methodology, and analyze the resulting performance trade-offs.

[1] `https://survey.stackoverflow.co/2024/technology#1-databases`

1.1. Foundations

In 2006, at the PostgreSQL Anniversary Summit, Teodor Sigaev and Oleg Bartunov formally introduced GIN and defined it as follows[2]:

An inverted index is an index structure storing a set of (key, posting list) pairs, where 'posting list' is a set of documents in which the key occurs. Generalized means that the index does not know which operation it accelerates. It works with custom strategies, defined for specific data types. GIN is similar to GiST and differs from B-Tree indices, which have predefined, comparison-based operations.

Their initial motivation centered on accelerating full-text search and multi-type array operations. Specifically, the first operator classes they presented included:

- One-dimensional arrays (array[T]), supporting
  - `&&` Overlap
  - @ Contains
  - ~ Is contained by
- `Tsearch2` - the core full-text search module
- `Intarray` - for enhanced integer-array support

For example, a query such as:

```
SELECT '{apple, banana, cherry}'::text[] && '{banana, strawberry}'::text[];
```

uses the `&&` operator to test whether the two arrays overlap (in this case, returning true).

GIN was incorporated into PostgreSQL 8.2[3], and in subsequent releases Sigaev extended its reach to support the HStore module. He also refined GIN's conceptual definition to emphasize its "inverted" nature[4]:

Termin 'inverted' comes from the theory of fulltext search. Usual (direct) index stores pairs id of document and some representation of text of document. Inverted index stores pairs of word from document and its id. So, in direct index there is only one and only one entry for each document. In inverted index – as much as words in document.

Concurrently, Guillaume Smet, together with Oleg Bartunov and Teodor Sigaev, added a GIN operator class to the `pg_trgm` extension, introducing trigram indexing to accelerate both text-similarity and pattern-matching queries[5]. This enhancement complemented the existing GiST operator class by providing a GIN-based execution path, further broadening GIN's applicability across diverse text-retrieval workloads.

[2] http://www.sigaev.ru/gin/Gin.pdf
[3] `https://www.postgresql.org/docs/current/gin.html`
[4] http://www.sigaev.ru/gin/hstore.pdf
[5] `https://www.postgresql.org/message-id/1d4e0c10702211600v7e0761c7ja533b949f6f79cad@mail.gmail.com`

1.2. Motivation

Our thesis is grounded in the observation that specialized, in-memory index structures can substantially outperform traditional B-tree variants, especially as modern servers afford enough main memory to keep entire indexed data resident in memory.

Viktor Leis, Alfons Kemper, and Thomas Neumann [1] introduced the Adaptive Radix Tree in 2013 as a main-memory index structure that dynamically chooses between four node layouts - Node4, Node16, Node48, and Node256, based on the actual number of children at each node. By matching node representation to occupancy, ART achieves a high average fan-out and a shallow tree, resulting in superior lookup and range-scan performance compared to cache-sensitive B-tree variants (e.g. the CSB^+-tree) in in-memory settings.

Building on the ART paradigm, Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis [2] presented the Height-Optimized Trie(HOT) at SIGMOD 2018. HOT extends adaptive node layouts by selecting the number of key bits examined per node to maximize fan-out under skewed or variable-length key distributions. Its node implementations, augmented with SIMD-accelerated lookups, deliver both faster search times and reduced memory consumption relative to ART and cache-optimized B-trees, especially for string workloads.

These works motivate our hypothesis that replacing GIN's B-tree–based entry tree with an ART-style structure can yield substantial gains for in-memory, trigram-indexed databases and operations on them.

1.3. Structure

The core of this work begins by introducing the notations and key definitions that will be referenced throughout the thesis. Chapter 2 offers a comprehensive exploration of the state-of-the-art implementations used as benchmarks in this research. 2.3 provides a detailed explanation of the data structures and the underlying mechanisms of the GIN. In a similar fashion, we then delve into the novel ART structure.

In Chapter 3, we show our implementation setup and approach in C++ with some modifications to accommodate our setup. This chapter also covers our main potential optimization proposal, the replacement of B-tree structure with the ART.

Chapter 4 describes the comparison experiments and discusses the results. There are three main results of this work. The first one addresses the building time of the index when implemented with the ART instead of B-tree. The second result involves the application of indexing in LIKE pattern matching for text data. It demonstrates the performance of GIN with the integration of ART into it. The final results obtain conditions - particular data distributions under which the ART - based entry tree might outperform its B-tree counterpart, highlighting the scenario where the ART is the preferred underlying structure.

Chapter 5 concludes this work and discusses open topics in a similar line of research.

2. Preliminaries

This chapter lays the theoretical groundwork for our investigation by first reviewing the fundamental data structures that underpin our implementations, namely, B-trees and the ART. We present formal definitions, properties, and the principal algorithms associated with each structure, emphasizing those aspects most pertinent to this work.

Building on this foundation, we then trace the evolution of PostgreSQL's GIN. After a historical overview, we dissect GIN's core components and data-flow mechanisms, with a discussion of its practical applications. We pay particular attention to trigram-based indexing, which serves as the primary use case for our subsequent modifications.

Finally, we examine LIKE pattern-matching process, detailing how GIN facilitates efficient substring-searches within the database engine.

2.1. B-tree

One of the data structures fundamental to our study is a **B-tree**. B-trees were first introduced by Rudolf Bayer and Edward McCreight in 1972 [3] to address the prohibitive cost of disk I/O per operation on the data. A single disk access can take as much as 5 ms, so minimizing the number of reads is critical. By employing a high fanout B-trees reduce tree height and thus for example, the number of costly disk I/O operations required to locate a given record. Today, main memory exhibits latencies that, relative to CPU speed, approach those of disks in the 1970s. Consequently, the same high fanout characteristics that made B-trees indispensable for on-disk storage also yield performance benefits for in-memory data structures. The original paper by Bayer and McCreight [3] defines B-trees as follows:

Definition 2.1.1 *Let $h \geq 0$ be an integer, k a natural number. A directed tree T is in the class $\tau(k, h)$ of B-trees if T is either empty $(h = 0)$ or has the following properties:*

- *Each path from the root to any leaf has the same length h, also called the height of T, i.e., h = number of nodes in path.*
- *Each node except the root and the leaves has at least $k + 1$ children. The root is a leaf or has at least two children.*
- *Each node has at most $2k + 1$ children.*

We define the order m of the tree by $m := 2k + 1$, so that each node contains between $\lceil m/2 \rceil$ and m children.

An example of a B-tree with order 5 is given by Figure 2.1. This is a B-tree of order $m = 5$. Each node may has at most $m - 1 = 4$ keys and at most $m = 5$ children.

For a B-tree of order m storing N keys, the height h satisfies

$$h \leq \log_{\lceil m/2 \rceil}\left(\frac{n+1}{2}\right).$$

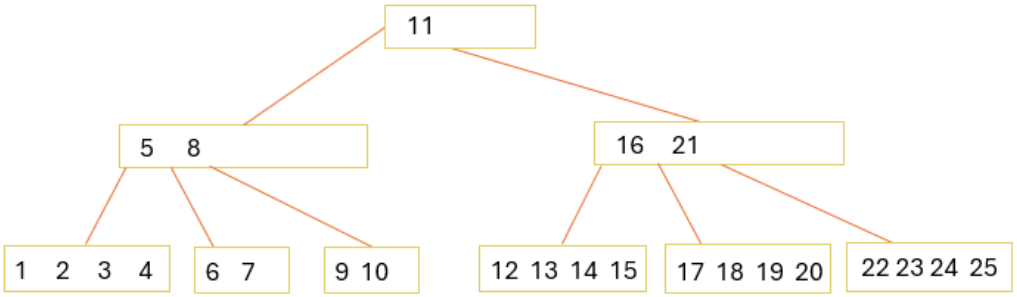

Figure 2.1.: a Simple B-tree example: both internal nodes and leaves may or may not contain the data pointers.

The true benefit of B-trees lie in the reducing constant factors. By choosing the order m so that its m child pointers and $m - 1$ keys exactly occupy one cache-line-sized block at the top of the memory hierarchy, we maximize the likelihood of the cache hits. Maintaining the tree's balance typically requires only very localized adjustments. One gains a deeper understanding of why this is the case by examining the operations involved, which we shall discuss as next.

Our primary object of study - the so-called entry tree, however, demonstrates several of the hallmark B$^+$-tree properties. B$^+$-tree refines the classic B-tree in three fundamental respects:

1. Internal nodes in B$^+$-trees store only separator keys and child pointers; all actual data pointers (or record references) live in the leaf level vs. in B-tree every node stores both keys and if they exist, the associated record pointers.

2. Additionally, the internal nodes may include pointers to their adjacent siblings, allowing straightforward movement across nodes at the same level and easing split and merge operations.

3. In a B$^+$-tree the leaves are linked in key order, forming a "data list" that supports efficient in-order and range traversals by pointer chasing alone.

As we want to construct a B$^+$-tree from scratch, the first operation we review is bulk loading. In this work, we construct a B$^+$-tree via the bottom-up bulk loading method, which proceeds in three phases:

1. **Building the leaves:** Divide the sorted key list into consecutive groups of up to $2k$ keys (where k is the parameter from Definition 2.1.1). Each group becomes one leaf node, storing its keys in order and exactly filling the node to capacity (except possibly the last leaf).

2. **Assembling the internal nodes:** Extract the first key of every child except the first one to form the separator key list (of size up to $2k$). Create an internal node whose pointers reference the $r \leq 2k+1$ children and whose $r-1$ keys are exactly those separators, in order.

3. **Level iteration:** Treat the newly formed internal nodes as the "leaves" of the next higher level, and repeat step 2 until a single root node remains.

Figure 2.2 gives an example of a B$^+$-tree, where the internal nodes contain the keys only as seperators. Their actual storage together with the data pointers occurs at the bottom part of the tree, in the leaves.

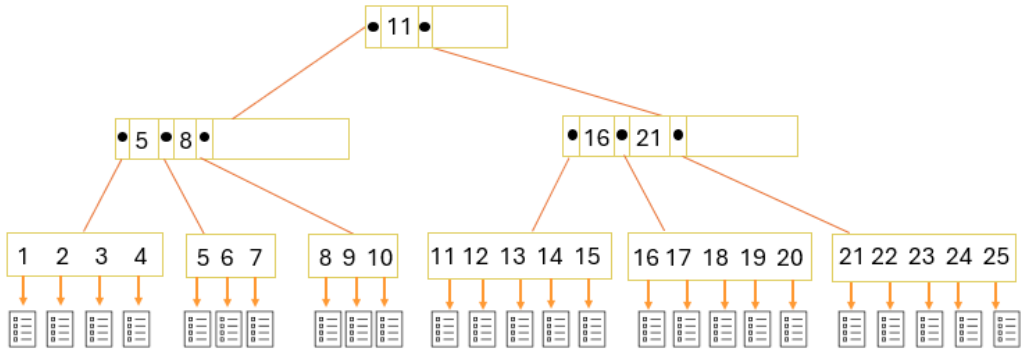

Figure 2.2.: B$^+$-tree example: internal nodes consist of seperator keys and child pointers, while leaves contain the keys and the associated data pointers.

By consuming the sorted key array in cache- or block-sized batches, each leaf and internal node is written in a single, contiguous I/O, confining all writes to small, adjacent regions of memory or disk.

Building on our discussion of bulk-loading, we now turn to the lookup operation, which is central to evaluating tree performance in substring-search workloads. We omit the discussion of sequential leaf scans here, since our evaluation focuses on the binary-search–based lookup performance. A search for a key K in a B$^+$-tree of height h and maximum node capacity m proceeds as follows:

1. **In-node binary search:** At the current node x, perform a binary search over its up to $m-1$ keys to find the smallest index i such that

$$K \leq x.key_i, \quad \text{or set } i = n+1 \text{ if } K > x.key_n.$$

2. **Key comparison:**
   - If $i \leq n$ and $K = x.key_i$, return the associated record or posting-list pointer.
   - Otherwise, proceed to step 3.
3. **Tree descent:** Follow the ith child pointer (or $(n+1)$th if K exceeds all keys) and repeat from step 1 until the key is found or a leaf is reached without a match.

A lookup follows a single root-to-leaf path of height h accessing at most h nodes. Each node is stored in one block, so the search incurs at most h cache or disk-block misses. Within each node, the binary search on a packed key array stays entirely within that block, maximizing cache hits.

B-tree, and by extension B^+-tree serves as the foundational indexing mechanism within GIN, a role we will explore in greater depth in the following chapters. With this groundwork laid, we now turn our attention to an alternative in-memory structure: the ART.

2.2. Adaptive Radix Tree

The *trie* or *radix tree* data structure dates back to Fredkin's 1960 work on information retrieval [4] and was later formalized by Knuth [5] as follows:

> "A trie is essentially an M-ary tree whose nodes are M-place vectors with components corresponding to digits or symbols. Each node at level l represents the set of all keys that begin with a certain sequence of l symbols; the node specifies an M-way branch, depending on the $(l+1)$th symbol."

Definition 2.2.1 *A* radix tree *on an alphabet Σ is a rooted tree in which:*

- *Each edge is labeled by a nonempty string over Σ.*
- *No node has two outgoing edges whose labels begin with the same symbol.*
- *Every key $k \in \Sigma$ corresponds to exactly one path from the root whose concatenated edge labels equal k.*

Leis et al. [1] enumerate several distinguishing properties of radix trees:

- The tree height grows with the maximum key length rather than the total number of keys.
- No rebalancing is required: all insertions yield the same tree shape up to edge-label compression.
- Keys are implicitly stored in lexicographic order.
- Keys are represented by the path of their corresponding leaf, hence are implicitly stored via these paths.

Although radix tries deliver lookups whose depth depends solely on the maximum key length and require no rebalancing, a naive implementation wastes memory and underutilizes caches. In an array-based node, allocating one pointer slot per possible symbol (e.g. 256 for byte-wise tries) leads to large sparsely filled structures. Its adaptive variant, first presented by Viktor Leis, Alphons Kemper and Thomas Neumann [1] - ART addresses this by selecting among multiple node layouts, each tailored to the node's current child count, thereby cutting memory overhead and enhancing cache locality.

ART enhances the classical radix tree by dynamically adapting each node's layout to its actual number of children, thereby reducing memory overhead and improving cache efficiency. As visualized in Figure 2.3, the adaptive inner nodes of ART are designed as follows:

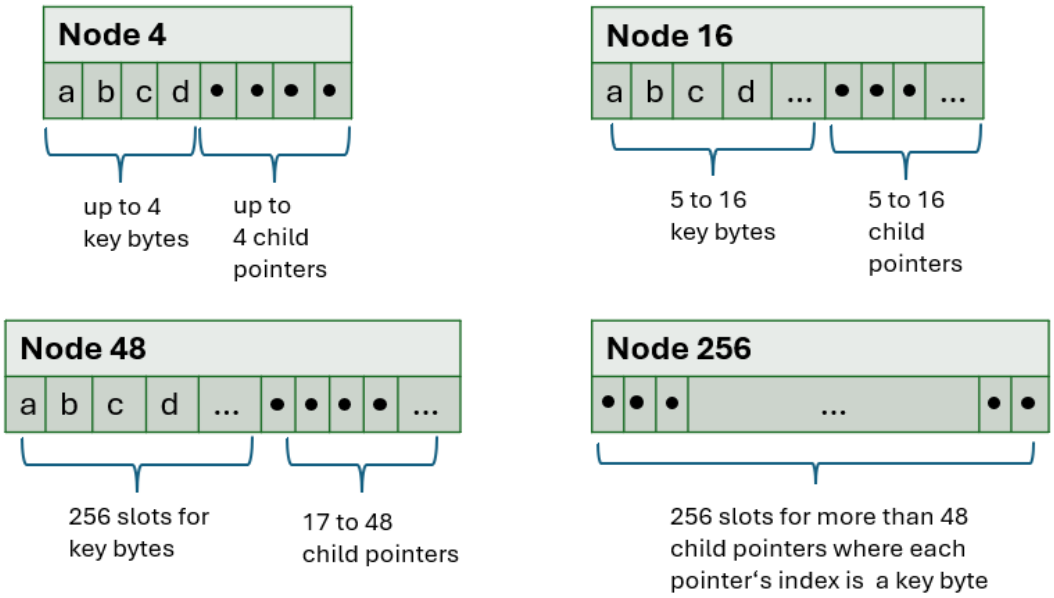

Figure 2.3.: Layouts of ART's four inner-node types. For Node4 and Node16, the letters ('a', 'b', 'c', 'd') represent sorted key-byte values and the • indicate child pointers. In Node48, the 256-entry array indexes key bytes into a compact 48-pointer array. Node256 is a flat 256-pointer array indexed directly by byte value.

Node4: This is the most compact internal node, holding no more than four child pointers. It maintains two parallel arrays of size 4: one for the key bytes and one for the corresponding child pointers, and keeps the keys in sorted order to facilitate lookups.

Node16: Designed for moderate fan-out (5 to 16 children), Node16 also uses two arrays each with 16 slots: for keys and pointers. Lookups can be accelerated via binary search, or on modern CPUs via SIMD-based parallel byte comparisons to quickly locate the matching key.

Node48: Beyond 16 children, storing keys explicitly becomes inefficient. Node48 replaces the key array with a 256-element byte-to-index map, where each entry points into a compact 48-element child pointer array. This indirection reduces memory compared to a full 256-

pointer table, since each map entry only needs one byte.

Node256: When a node holds between 49 and 256 children, ART uses a flat 256-entry pointer array, indexed directly by the key byte. This offers constant-time child access with no extra lookups, and if most entries are occupied, it remains space-efficient since it stores only pointers.

ART inner nodes encode key bytes (or compressed prefixes) and maintain child-pointer arrays while all associated payloads are stored in the leaves. Assuming each key is unique, there are three common strategies for storing or referencing those leaf-level payloads:

- **Single-Value Leaf Nodes:** Each value lives in its own dedicated leaf node.
- **Multi-Value Leaf Nodes:** Here, leaf nodes are structured much like the inner nodes (e.g., Leaf4, Leaf16, Leaf48, Leaf256), but they store batches of values instead of child pointers.
- **Mixed Pointer/Value Slots:** If values can fit into a machine pointer (for example, when indexing fixed-size tuple IDs), each inner-node pointer slot can be reused to hold either a child pointer or a value. A single extra tag bit (or pointer tagging) disambiguates the two.

Similar to building B^+-tree, we are interested in the bulk load algorithm of ART for this thesis. To build an ART in one pass over a sorted list of tuples, we apply a top-down, divide-and-conquer procedure that

1. **Terminates early for small groups:** If the current bucket contains at most the upper limit of leaf items (denoted as LEAF_THRESHOLD) or all bytes of the keys have been consumed, we stop splitting and emit a leaf-level structure: Single item goes to a lone ART leaf vs. multiple items goes to a tiny Node4 whose children are individual ARTLeaf nodes, each keyed by the byte at the current depth (or zero if the key has no more bytes).
2. **Partitions by the next key byte:** We create 256 sub-lists (one for each possible byte value). Each key-value pair in the current bucket is placed into the list corresponding to its byte at position depth (treating missing bytes as 0). The number of non-empty sub-lists defines the node's fan-out at this level.
3. **Chooses the "right-sized" inner node:** Based on the observed fan-out, we pick one of the four node types:
   - ≤ 4 children, then Node4
   - 5 to 16 children, then Node16
   - 17 to 48 children, then Node48
   - ≥ 49, the Node256

   This guarantees each node uses just enough space to represent its direct branches.

4. **Recursively builds child subtrees:** For each byte value b whose partition is non-empty, we invoke the bulk-loader on that sub-list with depth+1. Once the recursive call returns a root pointer, we link it into the parent as follows:
   - Node4/Node16: append the byte to keys[] and the subtree pointer to children[].
   - Node48: record the slot index in childIndex[b] and store the pointer in the compact children[] array.
   - Node256: place the subtree pointer directly at children[b].
5. **Returns the newly constructed node:** Returns the newly constructed node. After wiring up every non-empty branch, we hand the filled node back to our caller, which in turn attaches it to its own parent, and so on up to the root.

Once we know how to build the ART, we turn to the next operation, which allows checking its lookup performance. Leis et.al [1] presents the search algorithm for ART which can be summarized in the following 4 steps.

1. **Prefix Validation:** Before examining a child pointer, ART nodes may compress long runs of identical key-bytes into a short "prefix" buffer. When you arrive at any inner node, you must first verify that the next few bytes of your search key exactly match this stored prefix. If the remaining search key is shorter than the prefix, or if any byte differs, you can immediately conclude the key does not exist. Matching the prefix in one contiguous memory comparison allows ART to skip many levels of trivial one-byte branching, speeding up common "long common-prefix" scenarios.
2. **Key-Exhaustion Check:** After successfully consuming the compressed prefix, you advance your position within the key. If you now find that you have no bytes left in the search key (i.e. you've already matched every key byte), yet you're still at an inner node, the lookup must fail. Inner nodes never represent complete keys by themselves—they only encode partial paths. Reaching an inner node with no key bytes remaining means there is no corresponding leaf below.
3. **Child-Pointer Selection:** With at least one byte still to process, you take the next byte value, call it b, and ask the current node to tell you which child pointer to follow:
   - Node4 scans its small keys[] array linearly (up to 4 comparisons) to find an entry equal to b.
   - Node16 compares b against all 16 stored bytes at once using a single SIMD instruction, producing a bitmask; the position of the matching bit gives you the child index.
   - Node48 uses a 256-entry lookup table: childIndex[b] yields either an index into its compact 48-element pointer array or a sentinel (0xFF) if no child exists for b.
   - Node256 simply indexes directly into its 256-element children[] array at position b; a null pointer means no branch.

If the node indicates there is no child for b, the search terminates with "not found." Otherwise you recurse into that child, incrementing your depth by one (since you consumed one more key byte).

4. **Leaf Validation:** Eventually, the recursion will reach a leaf node (or the algorithm will have fallen off earlier). A leaf node holds the full remaining key (the exact sequence of bytes from root to leaf) along with its associated value. At this point you do a final byte-for-byte comparison of the entire stored key versus the query key: If they match in both length and content, you return the stored value. If they differ, you return "not found."

In summary, the ART lookup proceeds in four phases: prefix validation, key-exhaustion check, child-pointer selection, and leaf comparison, each designed to exploit the tree's adaptive node layouts for maximum speed. In our implementation (see Chapter 3), we simplify this sequence by omitting the prefix-validation step, since we do not employ prefix compression in our ART bulk loading.

With the ART structure and its core search mechanics established, we now turn to acquire some preliminary knowledge about the internal architecture of GIN.

2.3. GIN Architecture

In this chapter, we analyze the internal architecture of the GIN. The GIN framework comprises multiple integrated and hierarchical subsystems, which we categorize into three distinct component types (see Figure 2.4). We first delineate the function of each component and their interactions, then focus in depth on the principal structure of interest for this thesis - the entry tree. All subsequent descriptions are derived from the most recent PostgreSQL source code available on GitHub[1].

2.3.1. Building mechanisms

The centerpiece of the GIN index is the **entry tree**, a specialized variant of the B-tree whose precise correspondences and departures are analyzed in the next section. Its structure is composed of two page types: entry pages and data pages. Entry pages function as the tree's internal nodes, each housing an ordered sequence of separator keys, one per distinct indexed value, together with node-specific metadata that accelerates traversal and flags logically deleted entries. By routing lookup operations to the correct child pointers, entry pages facilitate rapid descent through the hierarchy. At the leaf level, data pages store the actual posting lists or, when postings are further organized in subtrees, pointers to those subordinate structures. While the entry tree establishes the principal indexing scaffold, its operation is supported by supplementary pages, most notably, the meta page - which maintains global index metadata and configuration parameters. In Figure 2.4, the portion outlined in orange specifically denotes the entry tree.

[1] `https://github.com/postgres/postgres/tree/master/src/backend/access/gin`

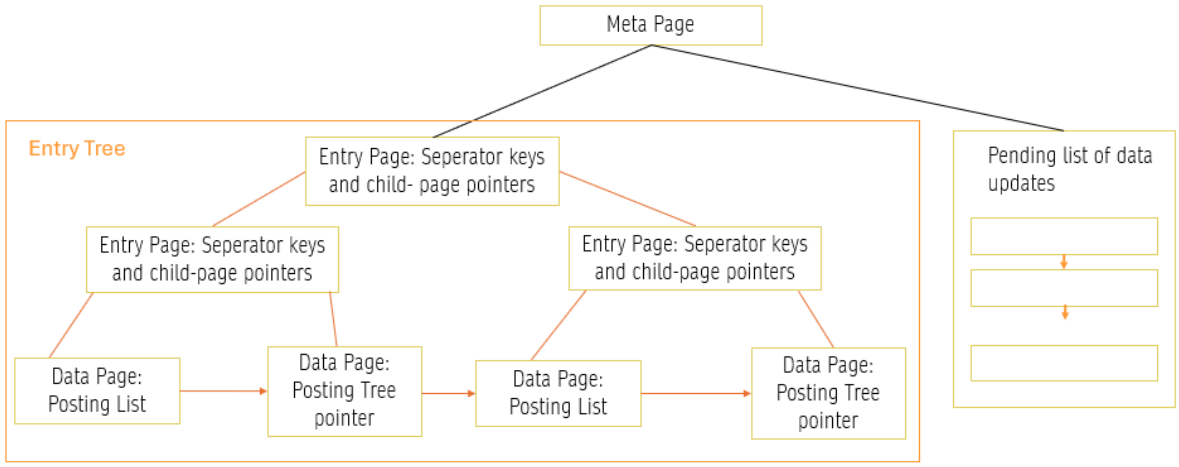

Figure 2.4.: High-level layout of GIN: a single meta page points to a hierarchy of entry pages (internal nodes) that store separator keys and page pointers, which in turn lead to data pages holding either posting lists or pointers to posting-tree roots. A separate pending-list chain captures new entries awaiting merge.

Meta page encapsulates all metadata required to construct, update, and maintain the entry tree over its lifetime, but most importantly it stores the pointer to the root of the entry tree. It records critical transactional and versioning information - such as the Write-Ahead Log (WAL) location at which the index was last modified and an index-specific version number, to ensure consistency and support crash recovery. Furthermore, the meta page tracks statistics and thresholds used by the index's maintenance routines (e.g., vacuum or cleanup), enabling automated decisions about when to merge auxiliary data into the primary structure. Meta page also maintains the data incoming from the so-called "pending list" which is another building block of the GIN. In Figure 2.4, the meta page is shown at the very top of the diagram, emphasizing its role as the entry point to the index's internal hierarchy.

Pending list underpins GIN's fast update strategy by staging incoming tuple entries in a simple on-disk list rather than immediately inserting them into the entry tree. This batched approach defers maintenance work until the list exceeds a configurable threshold (e.g., `gin_pending_list_limit`), at which point all accumulated entries are bulk-merged into the primary index structure.Although the pending list operates independently, periodic integration is essential to ensure that the entry tree reflects the most recent modifications. In Figure 2.4, the pending list is depicted on the right side of the diagram.

When updates are infrequent or the indexed dataset remains immutable, neither the meta page nor the pending list contributes to runtime performance; the entry tree alone suffices for all query and access operations. As our evaluation employs a static workload and concentrates exclusively on the entry tree's structural characteristics, the remainder of this thesis will confine its analysis to the entry tree - examining in detail its construction and behavior.

2.3.2. Structure of the entry tree

A precise understanding of the entry tree is a prerequisite for our proposed structural modifications and their subsequent evaluation. Below, we analyze and summarize to which extent the entry tree is of At its foundation, the entry tree faithfully implements the canonical B-tree model, inheriting its core invariants:

- **Uniform leaf depth.** All leaf nodes reside at the same level, ensuring that every key lookup traverses an identical number of levels.
- **Sorted in-node ordering.** Within each node, keys are stored in strictly increasing order, which permits O(log m) binary search over m entries per node.
- **Bounded height via node-capacity constraints.** By enforcing minimum and maximum occupancy thresholds on each node and performing splits or merges only when necessary, the tree's height remains comparatively small even as the number of entries grows.

These B-tree properties guarantee balanced structure, predictable performance, and efficient traversal. Later in this section, we will detail how the entry tree augments the classical B-tree framework to satisfy GIN-specific requirements, and then explore its parallels with the B^+ B^*-tree variants.

Building on its B-tree foundation, the entry tree adopts a B^+-tree-like organization by partitioning pages into two specialized roles. Entry pages function purely as internal nodes, containing only separator keys and child pointers and never actual tuple references, thereby maximizing fan-out and maintaining minimal height. In contrast, data pages serve as the leaf level, housing all posting-list entries or pointers to subordinate posting structures. This design concentrates every record pointer at the leaves, while internal pages remain dedicated to navigation, mirroring the classic B^+-tree's separation of routing and storage. In addition to segregating routing and storage into entry (internal) and data (leaf) pages, the entry tree exhibits some other hallmark B^+-tree features:

- **Sibling (right-link) pointers in internal nodes.** Alongside their child pointers, entry pages maintain a link to their right sibling at the same level. These right-links both simplify re-balancing and decouple page splits from long-held parent locks, improving concurrency.
- **Leaf-page chaining.** Each data page includes pointers to its immediate neighbors, both predecessor and successor. This doubly-linked leaf list enables truly in-order scans, queries can traverse contiguous leaves without re-ascending to the root, just as in classical B^+-trees.

In summary, the GIN entry tree combines B^+-tree design elements, most notably separator-only internal keys and right-link pointers for concurrency with the classic B-tree algorithms for splitting, merging, and redistribution. The result is a compact, depth-bounded structure that supports efficient point lookups and range scans while keeping implementation overhead

low. Because our investigation centers on bulk loading and binary-search traversal, we will henceforth treat the entry tree as a B-tree variant whose internal nodes contain only separator keys.

2.4. LIKE pattern matching

In this chapter, we examine SQL's `LIKE` predicate – our principal use case for the modified entry tree. We begin by formalizing the semantics of `LIKE` pattern matching, then discuss the various execution strategies available, and finally describe how PostgreSQL's GIN index implements and accelerates these queries.

In traditional relational systems, `LIKE` enables lightweight substring and wildcard searches on text attributes, obviating the need for full-text or external search infrastructure. The pattern language relies on two metacharacters:

- Percent sign (%) matches any sequence of zero or more characters.
- Underscore (_) matches exactly one arbitrary character.

For example, the predicate

```
WHERE column_name LIKE '%cake%'
```

returns all rows whose column_name contains the substring "cake" anywhere within its value, while

```
WHERE column_name LIKE 'c_ke'
```

matches exactly four-character strings beginning with "c" and ending with "ke" (e.g. "cake" "coke" "cuke" etc.). Subsequent discussions explore pattern classification, execution pathways (sequential scan, index-supported search), and the specific trigram-based GIN mechanism that underlies our entry-tree optimization.

In this work, we focus on the first class of patterns, those containing percent-sign wildcards.

If evaluated naively, compel a full table scan, incurring a high cost. To handle these efficiently, PostgreSQL provides a trigram-based GIN index via the `pg_trgm` extension. Each text value is decomposed into overlapping three-character substrings (trigrams), and the index records, for each trigram, the set of row identifiers in which it appears. At query time, the planner extracts the "required" trigrams from the literal segments of the LIKE pattern and probes their posting lists in the GIN entry tree. Only the intersection of these lists—typically a small candidate set is then rechecked against the full pattern, thereby avoiding a full scan of the base table.

To enable this mechanism, one creates the index as follows:

```
CREATE INDEX gin_idx ON table USING gin (column_name gin_trgm_ops);
```

Subsequent LIKE '%...%' queries on column_name will leverage this trigram index for efficient candidate selection before performing any final exact-match verification.

This pipeline constitutes the LIKE pattern-matching workload we use to evaluate our modified entry tree, with particular attention to bulk-load and query performance. In Chapter 3, we implement and instrument the following three phases, enabling a head-to-head comparison between the standard B-tree–based entry tree and an ART-based variant:

1. Initially, PostgreSQL invokes `gin_extract_value_trgm` which is defined in `pg_trgm` module[2], to decompose each text value into its set of three-character substrings ("trigrams"). Each distinct trigram is mapped to an integer key and added to the entry tree's data pages.

2. Once a `LIKE` predicate appears, the planner calls `gin_extract_query_trgm` to scan the pattern, ignore the '%', and emit only those trigrams drawn from its literal segments.

3. For each required trigram, the entry tree is traversed from root to leaf yielding the list of row identifiers in which that trigram appears. The individual lists are intersected to produce a small set of candidate rows guaranteed to contain every required trigram. Since trigram filtering may admit false positives (e.g. the required trigrams occur out of order), GIN's consistency callback `gin_trgm_consistent` flags the scan for recheck.

For the remainder of this thesis, we refer to this GIN-based workflow simply as trigram indexing, reflecting its central role in our performance study.

[2] `https://github.com/postgres/postgres/blob/master/contrib/pg_trgm/trgm_op.c`

3. Implementation and Experimental Setup

In order to investigate the performance of trigram indexing within a GIN framework and to support targeted benchmarking, we have developed two distinct implementations of the entry tree. We focused on trigram indexing with its application on LIKE pattern matching. First, we developed a C++ reimplementation of GIN entry tree for trigram indexing based on PostgreSQL's original codebase. Second, we modified the index mechanism by replacing the standard entry tree with an Adaptive Radix Tree (ART) structure. Lastly, we implemented LIKE pattern matching using indexes. This chapter is organized into three sections, each section is focused on one of these implementations.

3.1. Implementation of trigram indexing and LIKE queries

3.1.1. Trigam generation

In our in-memory implementation of the trigram index, data processing begins with loading the table contents and transforming each row into a structured format (i.e. a `TableRow` object containing an identifier and a text field). We extract indexable tokens, in this case - trigrams from the text using the `trigram_generator` function. For this step, we followed the official pg_trgm documentation[1], which defines a trigram as follows:

> *A trigram is a group of three consecutive characters taken from a string.*

Additionally, the documentation explains that the PostgreSQL's trigram generation process discards non-alphanumeric characters and pads each word with two spaces at the beginning and one at the end. We followed the same procedure in our implementation of `trigram_generator` function. Specifically, our implementation replicates the trgm_op.c from the original codebase where `generate_trgm` returns an array of sorted and de-duplicated trigrams.

Example 3.1.1 `trigram_generator` processes the string 'gold' as ' g', ' go', 'ld ', 'gol', 'old'.

As our data is made of rather simple English words with whitespaces only, all appearing in lowercase format, we don't include the implementation of the cleaning of alphanumeric characters. In addition to this, our data also doesn't include any multibyte UTF-8 characters (e.g. ä, ö, ü), we skip the implementation of `compact_trigram` function that ensures 3-byte representation of a trigram in exceptional cases such as those with multibyte UTF-characters.

[1] https://www.postgresql.org/docs/current/pgtrgm.html

The next step is only relevant for the original entry tree with a B-tree structure and is to be skipped for ART-based entry tree. This stage mimicks pg_trgm module's `trgm2int` function. `trgm2int` encodes each trigram into a 32-bit integer by packing the three characters into their successive byte positions. Specifically, it converts each character to its unsigned char representation, and through bitwise operations (shifting and OR-ing), combines them into a single integer value. These integer representations are collected into a dynamically allocated vector of unique integer representations of the input trigrams. In essence, `trgm2int` provides a compact numerical representation of text segments (which is advantageous for indexing and comparison operations).

Example 3.1.2 With `trgm2int`, one of the trigrams from above, 'gol' becomes:

$$(g \ll 16)|(o \ll 8)|l = 103 \times 65536 + 111 \times 256 + 108 = 6750208 + 28416 + 108 = 6778732$$

considering its unsigned char representations: 'g' (103), 'o' (111), 'l' (108).

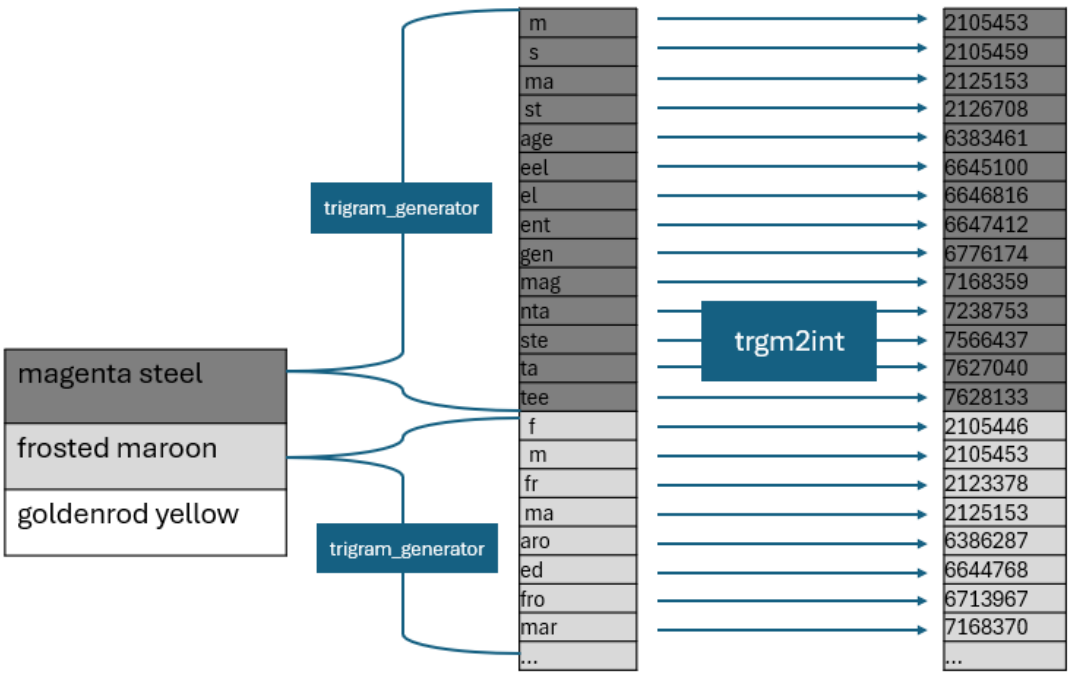

Figure 3.1.: Two-stage preprocessing of input strings into integer tokens for trigram indexing. Input strings are first tokenized into trigrams; in the B-tree variant, each trigram is then encoded to an integer ID via trgm2int, whereas the ART-based entry tree ingests the raw trigram strings directly..

When processing a table, `trigram_generator` is invoked once per table row to produce a sorted, deduplicated set of trigrams. `trgm2int` goes through this set and encodes each trigram into an integer. Hence, the output at each call yields a sorted, duplicate-free array of trigram integers for one row at a time. At the conclusion of this phase, each input row is

represented by its own sorted vector of distinct trigram integers, all stored in one array of trigram integers. See Figure 3.1 for a graphical overview of this stage.

At the same time, we record the corresponding row identifier (TID) under each trigram key in a hash-map structure. While PostgreSQL typically represents a TID as a composite of a block number and an offset, thereby pinpointing a row's precise physical location on disk, we opt to simplify this structure to a single integer field (rowId) in our in-memory environment.

Once all rows have been processed, we sort each vector of TIDs in ascending order. Such an ordering enables efficient downstream set operations such as intersections. Finally, we copy the hash-map into a key-ordered map to guarantee deterministic traversal in subsequent stages. The output of pre-processing stage is a sorted map of trigram integers together with a vector of TIDs associated with them. Figure 3.2 illustrates this output as the continuation of the above example. In the upcoming sections, we restrict our discussions to trigram integers

2105453	2105446	2105459	2125153	...
[1,2...]	[2,...]	[1,...]	[1,2,...]	...

Figure 3.2.: The final preprocessing output for the B-tree based entry tree is the sorted map of trigram integers and the vector of TIDs which contain them. For the ART-entry tree the mapping consists of trigram strings and vector of TIDs. This map is further processed into index tuples in the next section.

wherever they are relevant at; the ART variant using trigrams as they are, is analogous.

3.1.2. Construction of trigram indexing

In this stage, the implementation executes the core routine of constructing index tuples, which are fundamental to our indexing mechanism. As there is a limited amount of academic clarification about index tuples in the context of GIN, we have taken the initiative to formally define them in this work.:

Index Tuples. *An Index Tuple is an adaptive data structure that encapsulates an indexing key along with its associated posting data as a value. It stores each key uniquely and dynamically manages its unique postings by either embedding a compact, inline posting list when the number of postings is small or by referencing a more complex posting tree when the postings exceed a predetermined threshold.*

Such a dual representation not only facilitates efficient memory utilization and query performance but also ensures robust resource management through smart pointers and precise size computations for each tuple.

A critical decision during tuple formation is the choice of posting data representation. We use the parameter `GinMaxItemSize` to delineate the boundary between inline storage and tree-based storage. Specifically, if the number of TIDs associated with a given key exceeds `GinMaxItemSize`, the algorithm invokes the posting tree's `bulkLoad` routine on the full TID

vector. The resulting tree's root pointer then becomes the tuple's posting-data reference; otherwise, the TIDs remain embedded inline within the tuple. Such a mechanism ensures efficient use of memory and predictable lookup performance across varying posting-list sizes. Figure 3.3 visualizes a general structure of an index tuple where the two different versions of the pointer are possible depending on the size of posting data and `GetMaxItemSize`.

Posting tree construction occurs by either bulk-loading or incremental insertion. In standard GIN implementations, bulk-loading is employed for the initial index build, followed by incremental updates: insertions, deletions, or modifications managed via a pending list. Because our dataset is static, we only perform a single bulk aggregation of all posting data. Consequently, the preprocessed trigram-to-TID mappings described above suffice to drive the index-tuple creation.

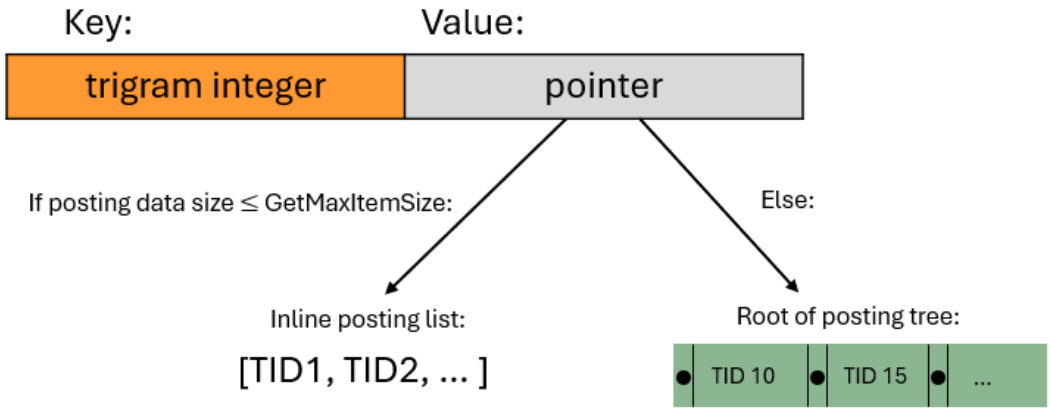

Figure 3.3.: Structure of an index tuple. The key of the tuple is the trigram integer obtained in the previous section. The pointer can point to either posting list or the root of the posting tree.

If the posting data size remains below or equal to `GinMaxItemSize`, an inline posting list is constructed and the value of the index tuple is updated to the posting list's smart pointers.

PostgreSQL computes `GinMaxItemSize` at compile time by dividing the usable space on an 8 KB page into three equal parts, one each for the posting list, downlinks, and pending list, after rounding page-overhead structures to machine-word alignment. In equations:

```
usable_bytes = BLCKSZ - MAXALIGN(SizeOfPageHeaderData + 3 * sizeof(ItemIdData))
                      - MAXALIGN(sizeof(GinPageOpaqueData))

GinMaxItemSize = Min(INDEX_SIZE_MASK, MAXALIGN_DOWN(usable_bytes/3)
```

Considering our in-memory setup as well as the simplified version of TIDs (and the entry tree as the next section describes), the definition above yields a threshold of roughly 1700 to

2000 four-byte TIDs. Taking into account this value, the observed repetitiveness in tpch data, and the lengths of our posting lists, all index tuples' posting data in our case defaults to a posting tree representation. To prevent this outcome, a fixed `GinMaxItemSize`=20 000 is used and interpreted directly as a maximum number of TIDs to fit into an inline posting list.

In a more mature system, one could replace this hard-coded limit with an adaptive mechanism that profiles posting list statistics at a build time (or even dynamically) and selects an optimal threshold. Such feedback-driven tuning offers a promising avenue for balancing memory usage against retrieval performance and is left as future work.

The logic for constructing each index tuple is encapsulated in the function `GinFormTuple()`. Next we present the pseudocode that illustrates this function with threshold checks, and the conditional creation of inline posting lists or posting trees.

```
function GinFormTuple(key, postingData):
    newTuple->key = key;
    IF numTIDs > GinItemMaxSize THEN
        tree ← new PostingTree()
        tree ← bulkload(postingData)
        newTuple.postingTree ← tree
        newTuple.postingSize ← numTIDs
    ELSE
        newTuple.postingList ← postingData
        newTuple.postingSize ← numTIDs
    END IF
    return newTuple
```

Because our primary focus is analyzing trigram indexing in the context of LIKE-pattern queries, we implemented the necessary access methods on index tuples. In particular, search operations require rapid retrieval of the full set of row identifiers (TIDs) for a given key. The architecture of `GinFormTuple()` directly supports this requirement by enabling a concise and efficient extraction routine as follows:

```
function GetPostingData(tuple):
    if tuple.postingList is not null then
        result ← tuple.postingList.tids
    else if tuple.postingTree is not null then
        result ← tuple.postingTree.getTIDs()
    end if
    return result
```

This method selects between the inline posting list and the tree-based representation as appropriate. To appreciate its implementation nuances, we next examine the structure and traversal algorithm of the posting tree, particularly its `getTIDs()` method.

Posting tree structure and methods

Posting trees build upon the core concepts of *B*-trees that focus on balanced node organization and logarithmic-time lookups while tailoring their structure to the demands of inverted–index workloads. Unlike classical *B*-trees, which store complete key–value records at every node, posting trees reserve their leaf nodes solely for sorted arrays of TIDs. Internal nodes, by contrast, maintain only the separator keys and pointers to direct searches. This specialization both reduces memory overhead and accelerates the set-based operations (for example, intersection and union) that are central to full-text and trigram search.

To instantiate this design in code, we must choose concrete values for node fan-out and leaf capacity. In the interest of clarity and simplicity, we fix the internal branching factor of all posting trees at $B = 16$. We also define three compile-time thresholds to govern leaf-node sizing during both bulk-loading and dynamic updates:

```
GinPostingListSegmentMaxSize    = 1 600 bytes
GinPostingListSegmentTargetSize =   800 bytes
GinPostingListSegmentMinSize    =   400 bytes
```

In PostgreSQL's on-disk GIN implementation, these values are derived from the standard 8 KB page size and an assumed 6-byte TID, yielding a fan-out of roughly 1 300–1 400 entries per leaf. Here, we instead choose values that ensure a worst-case tree height of two to three levels. Dividing by our 4-byte TID size gives the following leaf-count parameters with their interpretations:

- `LeafMaxCount` = 400 is the maximum number of TIDs in a leaf before splitting.
- `LeafTargetCount` = 200 is the number of TIDs each leaf is intended to hold under bulk-load.
- `LeafMinCount` = 100 the minimum leaf size; if a leaf falls below this threshold after removal or partial loading, it will be merged with an adjacent leaf (provided the combined size does not exceed).

These parameters strike a balance between shallow tree depth and efficient node utilization, providing predictable performance for both initial bulk loads and any subsequent updates.

To bulk load the posting tree, first TIDs are grouped into leaf nodes under given constraints:

```
function buildLeafNodes(sortedTIDs):
    leaves ← empty list
    n ← length(sortedTIDs)
    i ← 0
    while i < n do:
        count ← LeafTargetCount
        if (i + count > n) then:
```

```
            count ← n - i
        if (leaves is not empty) and (count < LeafMinCount) then:
            lastLeaf ← last element in leaves
            append sortedTIDs[i ... n-1] to lastLeaf.keys
            break the loop
        if (count > LeafMaxCount) then:
            count ← LeafMaxCount
        leaf ← new BTreeNode(leaf = true)
        leaf.keys ← sortedTIDs[i ... i+count-1]
        append leaf to leaves
        i ← i + count
    return leaves
```

Then, the internal tree levels are built recursively as follows:

```
function buildInternalLevel(children):
    if length(children) = 1 then:
        return children[0]
    parents ← empty list
    B ← 16
    n ← length(children)
    i ← 0
    while i < n do:
        count ← min(B, n-i)
        parent ← new BTreeNode(leaf = false)
        parent.children ← children[i ... i+count-1]
        for j from 1 to count - 1 do:
            append children[i + j].keys[0] to parent.keys
        append parent to parents
        i ← i + count
    return buildInternalLevel(parents)
```

The primary method of the posting tree utilized in this work is getTIDs(), which, given a pointer to the root node, retrieves all row identifiers stored within the tree. This method performs a recursive traversal starting from the root and aggregates all TIDs located in the leaf nodes. The procedure is outlined below:

```
function getTIDs(node):
    result ← empty list
    if node is null then
        return result
    if node.isLeaf then
```

```
        return copy of node.keys
    end if
    for each child in node.children do
        childTIDs ← getTIDs(child)
        append all elements of childTIDs to result
    end for
    return result
end function
```

In an online deployment, once the index has been bulk loaded, new records may be inserted and existing records deleted. To accommodate such modifications, posting trees support incremental insertion and deletion operations. A key component of these operations is node splitting (and its counterpart, node merging), whereby the tree reorganizes itself to maintain balance by partitioning overfull nodes and adjusting separator keys accordingly. In addition, redistribution between neighboring nodes may occur when a node becomes underfull, ensuring that the tree's height remains minimal and that search performance stays optimal. Although not currently implemented, a comprehensive design would also include concurrency control measures, such as latch coupling or lock crabbing, to safely coordinate splits and merges under concurrent updates. Because our implementation targets a static dataset, we have not implemented these dynamic update routines.

Posting trees primarily serve to limit memory consumption when representing very large posting lists. Each posting tree is constructed a single time during index creation and is only accessed when its corresponding key is looked up in the entry tree - the overarching structure we implement as next.

3.1.3. Entry tree formation, its structure and methods

The entry tree constitutes the topmost level of GIN index, integrating all subordinate components into a coherent search structure. Its primary function is to organize the set of indexed keys, which in our case are the integer-encoded trigrams, into a balanced search tree, thereby enabling efficient lookup operations across all distinct keys. As described in Section 2.3, we model the entry tree as a conventional B-tree in which each leaf node contains a contiguous array of index tuples. Each index tuple associates a single key with a pointer to its posting data, which may be stored as an inline posting list (as a compact list of tuple identifiers) or as a separate posting tree for large lists.

In contrast to auxiliary posting trees, which focus on optimizing storage and set-based operations for individual keys, the entry tree must efficiently route lookups across the entire key space. To support this global navigation while maintaining a simple implementation, we introduce a single configuration parameter—`EntryLeafMaxCount`, that defines both the maximum number of keys per leaf node and, by extension, the fan-out for internal nodes:

Because the entry tree is the focal point of our performance experiments, we have retained PostgreSQL's original design philosophy wherever practical. In the on-disk implementation, the maximum number of key–pointer pairs per page is computed by dividing the available

page space, after subtracting the page header, by the size of each index tuple entry (rounded up for alignment) plus its line-pointer overhead:

```
(BLCKSZ - SizeOfPageHeaderData)/(sizeof(IndexTupleData) + sizeof(ItemIdData))
```

Under typical parameters (an 8 KB page, 4 bytes for the key, 8 bytes for the pointer), this calculation yields on the order of 500 entries per node, producing a very shallow tree for trigram indexing. To introduce greater structural complexity for both analysis and benchmarking, we instead enforce a fixed leaf capacity of 20 keys (and, by extension, 21 child pointers) by defining `EntryLeafMaxCount` = 20. Applied to the 454 distinct trigrams in the TPC-H's part.tbl table, this setting requires an entry tree of height three, as derived from the B-tree depth formula presented in Section 2.3.

For the sake of simplicity, we identify the fan-out value of the internal nodes with values between `MAX_KEYS = EntryLeafMaxCount` and `MIN_KEYS = EntryLeafMaxCount/2`. The entry tree is constructed via a bottom-up bulk-loading algorithm which we reviewed in Section 2.1 through the following three functions:

Building the leaves: We partition the sorted list of index tuples into leaf nodes, each containing up to `EntryLeafMaxCount` entries:

```
function buildLeafNodes(tuples):
    leaves ← empty list
    n ← length(tuples)
    i ← 0
    while i < n do
        leaf ← new EntryTreeNode(isLeaf = true)
        count ← min(n - i, EntryLeafMaxCount)
        for j from 0 to count - 1 do
            leaf.keys.append  (tuples[i + j].key)
            leaf.pointers.append(tuples[i + j].pointer)
        end for
        leaves.append(leaf)
        i ← i + count
    end while
    return leaves
end function
```

Internal-Level Assembly: We don't chain the sibling leaves unlike the PostgreSQL implementation for the sake of simplicity. Once the leaves are built, we iteratively group child nodes into parents. Each internal node can reference from `MIN_KEYS` up to `MAX_KEYS + 1` children; its separator keys are the first key of each child beyond the first:

```
function buildInternalLevel(children):
    if length(children) = 1 then
        return children[0]
```

```
    end if
    parents ← empty list
    n ← length(children)
    i ← 0
    while i < n do
        parent ← new EntryTreeNode(isLeaf = false)
        count  ← min(n - i, EntryLeafMaxCount + 1)
        for j from 0 to count - 1 do
            child ← children[i + j]
            parent.children.append(child)
            if j > 0 and child.keys is not empty then
                parent.keys.append(child.keys[0])
            end if
        end for
        parents.append(parent)
        i ← i + count
    end while
    return buildInternalLevel(parents)
end function
```

Iteration: Now we put the two functions together to build the B-tree bottom-up. At first layer, the algorithm assigns the leaves and then proceeds to identify key seperators and upper levels until the root is identified.

```
function bulkLoad(tuples):
    leaves ← buildLeafNodes(tuples)
    root   ← buildInternalLevel(leaves)
end function
```

To retrieve the index tuple for a given key, we perform binary search method which we reviewed in Section 2.1.

```
function searchEntryTree(root, K):
    node ← root
    while node is not a leaf do
        i ← binary_search_smallest_index(node.keys, K)
        if i < length(node.keys) and node.keys[i] = K then
            return node.values[i]
        end if
        node ← node.children[i]
    end while
    i ← binary_search_smallest_index(node.keys, K)
    if i < length(node.keys) and node.keys[i] = K then
        return node.values[i]
```

```
    end if
    return null
end function
```

Figure 3.4 depicts the resulting entry tree after bulk loading (in our implementation the leaf starting with the seperator key is on the right of it). The illustration shows three levels: the root, two internal nodes, and 23 leaf nodes. Each leaf contains up to 20 keys and their corresponding tuple pointers, while the internal and root nodes store only separator keys to guide the traversal.

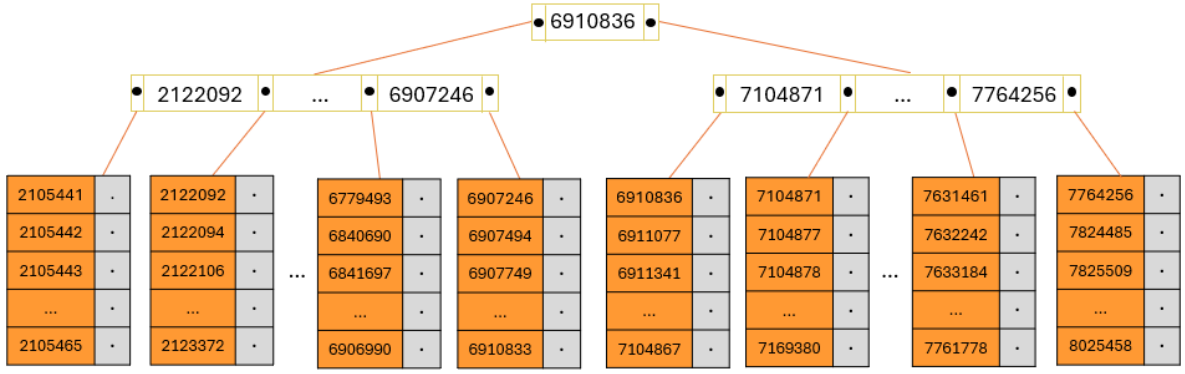

Figure 3.4.: Entry tree structure after processing and bulk-loading the part table. Leaves are color-coded (orange and gray), and each internal separator key is chosen so that the leaf in its right subtree begins with that key.

In summary, our entry tree implementation faithfully preserves the key architectural and algorithmic features of PostgreSQL's original GIN design. In particular, it maintains:

1. Separation of Concerns: By decoupling key-to-posting lookup (entry tree) from posting data storage and manipulation (posting trees), our indexing achieves both rapid key search and flexible, high-performance handling of high-cardinality postings.
2. Fixed fan out for cache efficiency: Internal nodes group multiple child pointers under common separator keys, allowing the tree to remain shallow (typically two to three levels).
3. Logarithmic-Time Key Lookup: By maintaining its nodes in sorted order, the entry tree ensures $O(\log N)$ search complexity for N indexed keys.

These properties emerge from the entry tree's B-tree like structure, where we also imposed clear node capacity limits, and a modular separation of index responsibilities. Although such a design scales effectively, we claim that further optimization is possible. In the next chapter, we explore replacing the entry tree with an ART structure, offering a novel alternative that challenges the status quo.

3.2. Modifying entry tree's structure with ART

The entry tree, as established in Section 2.3, serves as the global routing layer of the GIN index, organizing integer-encoded trigrams into a balanced, disk-resident B-tree. While B-trees provide predictable performance, their node capacities and search paths are ultimately constrained by fixed fan-out values and by the trade-off between fan-out and node-depth. ART, by contrast, dynamically tailors node representations to the actual set of key bytes at each trie depth, yielding a memory-efficient, in-memory structure with very low search path lengths.

In this section, we describe how we replace the traditional B-tree entry layer with ART, preserving the logical semantics of an entry tree – namely, a mapping from each distinct trigram key to its corresponding posting data pointer. While doing so, we leverage ART's adaptive node layouts to reduce average lookup times. We retain our previous abstraction of an index tuple (key + posting-pointer), and simply restructure the tree layers that link these tuples.

When constructing the ART over our trigram index, the preprocessing differs only in one respect: we retain each trigram as its original three-character byte sequence rather than encoding it into a 32-bit integer via `trgm2int`. Consequently, every key in the ART consists of exactly three bytes, and the tree's maximum depth is fixed at three levels.

- **Level 1:** The root node's slots correspond to all possible first-byte values that actually occur as the initial character of any trigram in the dataset. In practice, this includes the whitespace character and all alphanumeric characters present at the start of at least one trigram. Entries are maintained in ascending byte-value order. See for example, the ART root in Figure 3.6, which we obtained for part table of TPC-H dataset.

- **Level 2:** For each root-level child, the ART allocates slots to every second-byte value that follows the given first byte in at least one trigram. Thus, a child of the root corresponding to byte 'b' will have pointers only for those bytes 'x' for which the trigram '(b,x,*)' appears in the data.

- **Level 3:** Finally, each second-level node points to leaf entries consisting of all third bytes that complete a valid trigram. In this way, a path of length three in the ART exactly corresponds to one of the dataset's trigrams.

Because certain characters (for example, 'z') are infrequent [6] as initial trigram bytes, their corresponding root-level slots may lead directly to leaves (i.e. second-level nodes are omitted). In other words, if no other trigram begins with '(z,*,*)' beyond the first byte, the ART bulk-loader creates a two-level branch rather than allocating an empty intermediate node. Although this resembles path compression, our implementation achieves it implicitly during bulk loading by only instantiating nodes for byte values that actually occur at each position.

Our ART implementation employs the standard node categories: `Node4`, `Node16`, and `Node48`, for both the root and all internal levels. Although the ART specification also defines

a `Node256` type, we omit it in practice because our trigram data never requires more than 48 distinct child pointers at any single node. (Nonetheless, the constructor for `Node256` remains in our codebase to support future extensions.)

In addition to this, modes are promoted dynamically from one category to the higher based on fan-out: when the number of distinct next-byte values attached to a given prefix exceeds the capacity of the current node type, we allocate the next larger node category and migrate its children accordingly. For example, rare initial bytes such as 'x', 'q', and 'z' –which [6] identifies as the least frequent in English. We can also see this in Fig 3.5. Such characters typically remain in `Node4` or `Node16` configurations, reflecting their low branching factor in the trigram set.

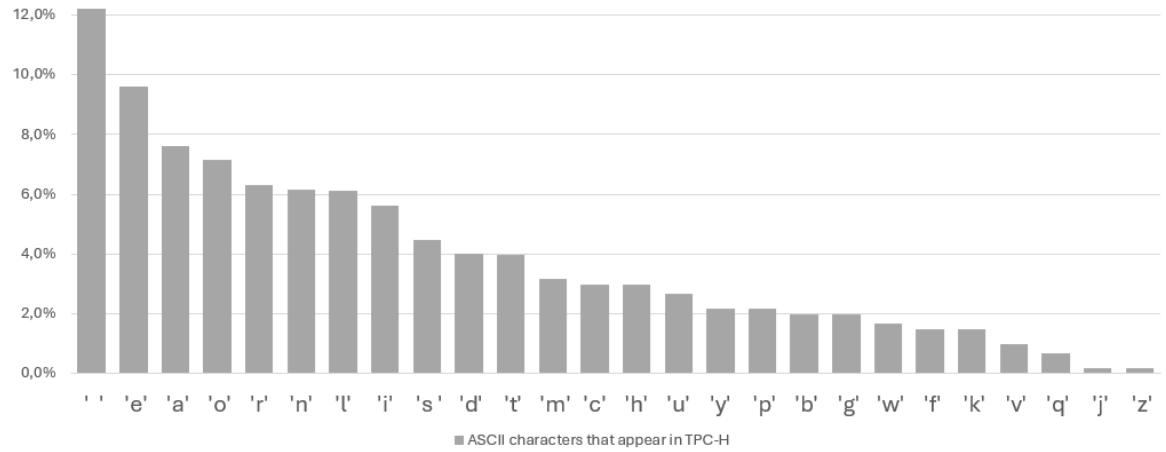

Figure 3.5.: The relative distribution of ASCII characters in part.tbl.

Section 2.2 reviewed several approaches to storing leaf values in an ART. For our trigram index, we adopt a single-value leaf strategy, in which each leaf node directly holds a pointer to its associated `IndexTuple`. Figure 3.6 illustrates a portion of the resulting ART structure, highlighting both standard multi-child nodes and the compressed paths introduced by our single-child optimization.

In ART-based entry tree, full trigram keys are replaced by their individual byte-components. This modified structure diverges from the original B-tree entry tree in three principal ways:

- 1. At depth $d \in \{1, 2\}$, each internal node partitions its children according to the dth byte of the full three-byte trigram key, rather than by 32-bit integer values.
- 2. Instead of enforcing fixed minimum and maximum fan-out bounds, the ART entry tree employs node types of varying capacities (`Node4`, `Node16`, `Node48`). In practice, most internal nodes fall into `Node4` or `Node16`, with occasional `Node48` usage—particularly at the root and its immediate child corresponding to the whitespace padding byte.
- 3. Each leaf in the ART entry tree contains exactly one full trigram (three-byte) key and a pointer to its posting data, in contrast to the B-tree entry tree's packed leaves that store multiple key–pointer pairs per node.

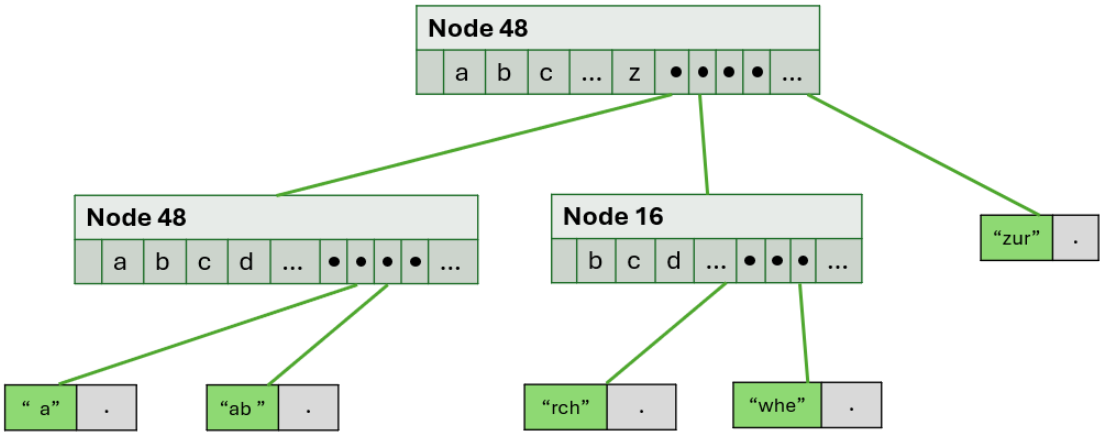

Figure 3.6.: ART structure after preprocessing and bulk-loading the part table. Leaves, color-coded in neon green consist of 1 tuple at a time, while the root and internal levels can be of different node types. Custom path shortening is applied to connect the root with the single child directly in the case of ’z’ as the initial character.

Assuming that all `IndexTuple` objects have been constructed, the ART entry tree implements bulk loading and lookup as we already described in Section 2.2, using the following methods (Due to its considerable length, the full pseudocode is omitted.):

Radix partition. At each tree depth $d \in \{0, 1, 2\}$, we distribute the set of key–tuple pairs into 256 buckets: one for each possible byte value, using the following loop:

```
for (const auto& item : items) {
unsigned char b = (depth < (int)item.first.size())
                  ? item.first[depth]
                  : 0;
partitions[b].push_back(item);}
```

Inner Node Type Choice. After partitioning, we select the smallest ART node type that can accommodate the non-empty buckets:

```
nonEmptyCount ← number of buckets with at least one item
if nonEmptyCount < 5 then
    node ← new ARTNode4()
else if nonEmptyCount < 17 then
    node ← new ARTNode16()
else if nonEmptyCount < 49 then
```

```
        node ← new ARTNode48()
    else
        node ← new ARTNode256()
    end if
```

Each bucket then becomes a child pointer of the newly created node, and we recurse on each bucket to populate deeper levels.

With the tree fully constructed through recursive bucket partitioning, we now turn to the search mechanism that navigates these adaptive nodes. **Search Mechanism.** Our search implementation follows the same routine as presented by [1], which we already described in Section 2.2 with node-type-specific optimizations:

- `ARTNode4`: Linear scan over up to four keys.

```
function ARTNode4_search(node, key, keyLen, depth):
    d <- depth
    if d >= keyLen then
        return null
    b <- key[d]
    for i from 0 to node.count - 1 do
        if node.keys[i] = b then
            return ARTNode4_search(node.children[i], key, keyLen, d + 1)
    return null
```

- `ARTNode16`: Parallel comparison of up to sixteen keys using SSE instructions. This procedure uses SSE2 intrinsics to broadcast the probe byte, compare it in parallel against up to 16 node keys, mask out unused slots, and select the matching child index for continued recursion.

```
function ARTNode16_parallel_search(node, key, keyLen, depth):
    d <- depth
    if d >= keyLen then
        return null
    b <- key[d]
    cmp_key <- SIMD_set1_byte(b)
    key_vec <- SIMD_load128(node.keys)
    cmp_mask <- SIMD_compare_equal(cmp_key, key_vec)
    valid_mask <- (1 << node.count) - 1
    bits <- SIMD_movemask(cmp_mask) & valid_mask
    if bits == 0 then
        return null
    idx <- count_trailing_zeros(bits)
    return ARTNode16_parallel_search(node.children[idx], key, keyLen, d + 1)
```

- `ARTNode48`: Two-array lookup, where one array maps byte values to child indices.

```
function ARTNode48_search(node, key, keyLen, depth):
    d <- depth
    if d >= keyLen then
        return null
    b <- key[d]
    idx <- node.childIndex[b]
    if idx = 0xFF then
        return null
    return ARTNode48_search(node.children[idx], key, keyLen, d + 1)
```

- `ARTNode256`: Direct indexed access via a 256-entry child pointer array. We omit the pseudocode for this node type as it's not used in our application.

In the following section, we integrate these lookup routines into our 'LIKE'-pattern search pipeline.

3.3. LIKE Pattern Matching Implementation

This section presents the end-to-end design and implementation of a trigram-based substring search mechanism, realized atop two distinct trigram indexing's entry-tree variants: a conventional B-tree-based structure and an ART-based structure. Both variants employ an identical four phase processing pipeline, enabling a controlled comparison of their performance characteristics.

We focus on patterns of the form:

$$\%x_1\%x_2\%...\%x_k\% \tag{3.1}$$

where each x_i denotes a literal substring to be matched. The required trigrams are those three-character sequences that must appear, potentially with intervening characters, within any candidate row.

Trigram Generation. In this phase, the input pattern is parsed to extract the ordered set of "required" trigrams, following the procedure of Section

Index-Based Candidate Retrieval. Each trigram is looked up in the selected entry tree index. For the B-tree variant, the trigram is first encoded to a 32-bit integer and passed to `entryTree.search(key)`. For the ART variant, the raw three-byte sequence is used in the call `artRoot->search(bytes, length)`. The retrieved `IndexTuple` is then probed via `getPostingData()` function to obtain its sorted vector of tuple identifiers (TIDs).

Posting-List Intersection. The individual posting lists, each sorted by TID, are merged using a standard linear-time intersection algorithm. The result is a reduced set of candidate rows containing all required trigrams.

Final String Verification. For each candidate TID, the corresponding text value is fetched from the base table. A final check confirms that the literal substrings $x_1, x_2, \ldots, x_k$ appear in the correct order and relative positions, thus satisfying the full semantics of the `LIKE` pattern.

The only substantive distinction between the two indexing mechanisms lies in the lookup method for a single trigram:

- **B-tree–Based Entry Tree:**

```
IndexTuple* tup = entryTree.search(encodedKey);
```

- **ART-Based Entry Tree:**

```
IndexTuple* tup = artRoot->search(byteSequence, 3);
```

All subsequent steps: posting-list extraction, intersection, and verification remain identical across both variants.

Such a unified implementation framework provides the essential basis for the performance evaluation presented in the next chapter, where build-time and query-time metrics are compared for the B-tree and ART entry tree approaches.

4. Experimental Setup and Main Results

In this chapter, we assess how replacing the conventional B-tree-based GIN entry tree with the ART in the context of trigram indexing affects both index construction time and LIKE pattern matching performance across datasets of varying scale.

In the implementation and analysis chapter, we demonstrated that PostgreSQL's entry tree is inherently shallow. Our experimental results further indicate that substituting it with an ART produces no appreciable change in either query execution or index build performance. We begin by describing the experimental environment and methodology, and then present a comprehensive analysis of the results.

4.1. Experimental Setup

4.1.1. Hardware specification

All experiments were conducted on a single core (one NUMA node) of a dual-socket server equipped with two Intel Xeon E5-2680 v4 CPUs clocked at 2.40 GHz. The system is outfitted with 256 GB of DDR4 RAM. It runs a 64-bit Ubuntu 22.04.1 LTS installation with a Linux 5.15.0-56-generic kernel. Trigram indexing and LIKE pattern matching are developed in C++ and built with g++ 14.2.0 using -O3 optimizations. The times are measured by high resolution clock in seconds and milliseconds.

4.1.2. Data and Queries

We conducted our evaluation of both B-tree- and ART-based entry tree implementations using two variants of the TPC-H dataset, corresponding to scale factors 1 and 10. Our analysis focused exclusively on the product_name column of the part table as the representative text field. In the following sections, we characterize these datasets and detail the query workload employed in our benchmarking experiments.

The SF1 dataset comprises 200,000 tuples, whereas the SF10 dataset contains two million tuples, representing a tenfold increase in cardinality. Despite this difference in size, both datasets exhibit identical lexical diversity: 92 distinct terms derived from the product_name values. Applying the trigram extraction procedure outlined in Section

The frequency distribution of the most prevalent trigrams is presented below:

Based on our analysis of varying trigram frequencies, we consider three different multiple expression patterns for our work:

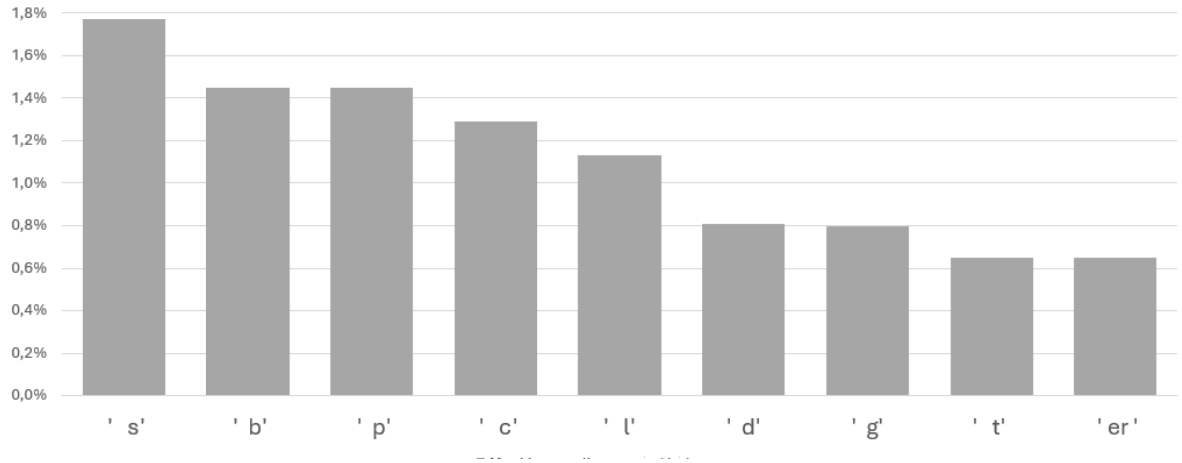

Figure 4.1.: The relative distribution of trigrams in part.tbl.

short expressions: The search string consists of two trigrams.

```
select count(*) from part where p_name LIKE '%mon%ros%'
```

This query results in 2052 rows.

medium expressions: The search expression consists of one longer string and one trigram.

```
select count(*) from part where p_name LIKE '%chocolate%mon%'
```

This query results in 704 rows.

long expressions: The search string consists of two full words.

```
select count(*) from part where p_name LIKE '%lavender%almond%'
```

This query results in 246 rows.

The number of rows matching a given LIKE pattern defines its selectivity: higher selectivity indicates fewer matching rows. We therefore rank our three example patterns by increasing selectivity as follows:

```
'%lavender%almond%' > '%chocolate%mon%' > '%mon%ros%'
```

where ">" reads "has higher selectivity than". This ranking frames our expectation that more selective patterns will benefit more from trigram indexing, while less selective ones stand to gain the least.

4.1.3. PostgreSQL performance

Leveraging PostgreSQL's `pg_trgm` extension, we now compare planner cost estimates for these three patterns on the original database setup. Using the `EXPLAIN` command under identical system parameters, we record the optimizer's estimated cost both with and without a GIN trigram index enabled. Table 4.1 presents these cost figures, expressed in PostgreSQL's internal units, and the corresponding cost optimization ratios.

Pattern	No-Index Cost	Index-Based Cost	Cost Optimization
`%mon%ros%`	6 171.99	3 521.74	1.8×
`%chocolate%mon%`	6 169.91	231.24	26.7×
`%lavender%almond%`	6 169.91	268.96	23.0×

Table 4.1.: Planner cost comparison for three LIKE patterns on `part.tbl` (TPC-H SF=1), with and without a GIN trigram index.

Several insights emerge from Table 4.1. First, the planner's cost estimates for all three patterns are essentially identical in the absence of an index, reflecting the constant expense of a full sequential scan. Second, the benefit of trigram indexing is most pronounced for medium selective predicates, with cost improvement exceeding 20x for the longer patterns, and yet remains nontrivial even for less selective query. Moreover, index-assisted planner cost does not scale linearly with either pattern length or selectivity. If cost were proportional to the number of wildcard segments or the fraction of matching rows, one would anticipate that the '%mon%ros%' pattern having fewer trigrams and lower selectivity would incur substantially lower indexed cost than the longer '%chocolate%mon%' or '%lavender%almond%' patterns. Instead, we observe:

1. A modest 1.8× speedup for '%mon%ros%', despite its minimal trigram count.
2. A dramatic 26.7× and 23.0× improvement for the two longer patterns, even though their increased literal length and larger trigram sets could have imposed greater lookup overhead.

Hence, the planner cost and, by extension, actual execution time reflects a complex interplay of factors beyond mere pattern length or overall selectivity. Key contributors include the absolute number of trigrams extracted, the size of each trigram's posting list, and the way these trigrams are distributed across the dataset. In particular, long literal segments, by virtue of producing more distinct trigrams, can generate some rare n-grams (and hence small posting lists), which in turn reduce the residual post-filter workload. Conversely, short or otherwise common trigrams may return large posting lists that dominate the remaining cost. Thus, pattern length or selectivity hardly suffice to predict the magnitude of index-based cost reduction. These findings suggest intriguing avenues for future exploration; a detailed treatment, however, lies beyond the scope of this work. Query definitions and output results from these exploratory analyses are provided in Appendix A.1 for reference. Finally, we

present a speculative extrapolation for the TPC-H SF10 dataset. Given the approximately linear increase in planner cost observed at SF = 1, one would predict that the parallel sequential-scan plan's cost would rise by an order of magnitude to around 61700 cost units. Likewise, bitmap-index plans would scale roughly in proportion to their matching-row counts, approximately 2460 for '%lavender%almond%', 7040 for '%chocolate%mon%', and 20520 for '%mon%ros%'. Under these projections, the relative cost optimization offered by the GIN-trigram index would remain largely unchanged: over 2× for least selective patterns, about 26× for moderate selectivity, and roughly 23x in the most selective case. We emphasize, however, that these are purely speculative; validating them on actual SF = 10 data is left for future work.

4.2. Experimental Results and Analysis

In this chapter, we report the outcomes of our comparative study between the PostgreSQL-native B-tree entry tree (baseline) and our ART-based implementation. We executed each of the key tasks: bulk index loading and short, medium, and long LIKE pattern searches 50 times to ensure statistical stability. The results for SF1 over 50 iterations are reported in Appendix A.2.

We begin by contrasting the bulk-load times of the two index structures, then proceed to evaluate their performance on the LIKE–pattern matching workload.

Our findings demonstrate that, for a fixed trigram vocabulary (as in our smaller-scale tests), the underlying tree structure, whether B-tree or ART, has negligible impact on both bulk-load time and query latency. Furthermore, we supplement our empirical findings with a theoretical assessment, including quantitative calculations, which corroborates that PostgreSQL's B-tree implementation is sufficiently performant for trigram-indexing applications. A more exhaustive investigation involving larger or more heterogeneous datasets is left to future work.

4.2.1. Index Bulk Loading

Bulk loading represents the fundamental first step in constructing any GIN index, whether built offline or maintained online. To assess the performance implications of our two entry-tree designs (the native B-tree and the ART-based variant), we measured their bulk-load times on TPC-H datasets at scale factors 1 and 10. The results are summarized in Tables 4.2 and 4.3.

Irrespective of the tree structure, the dominant cost during bulk loading arises from scanning the full set of n index keys. In the ART implementation, each of the three trie levels requires a complete, linear pass over those keys, yielding three sequential scans in total. By contrast, the B-tree loader performs:

1. A single pass over all n sorted keys to build the leaf pages, resulting in the cost equal to $\mathcal{O}(n)$.

2. One pass across the $\lceil n/\text{MAX\_KEYS} \rceil$ leaf pages to compute separator keys (cost $\mathcal{O}(n)$, since each leaf holds up to MAX_KEYS entries).

3. A final scan over the $\lceil n/\text{MAX\_KEYS} \rceil$ parent-node keys with the cost of $\mathcal{O}(n/\text{MAX\_KEYS})$.

ART bulk-loading incurs $\mathcal{O}(3n)$ work (three full passes). B-tree bulk-loading incurs $\mathcal{O}(2n) + \mathcal{O}(n/\text{MAX\_KEYS})$ work (two full passes plus one small internal scan).

Because our experiments use fixed, three-byte trigram keys and moderate data volumes, these constant-factor differences in scan count balance out at SF = 1, yielding virtually identical bulk-load times (Table 4.2). Only at larger scale (SF = 10), when the dataset is ten times bigger, does the extra full-scan in the ART loader begin to show up: the B-tree variant—performing two rather than three full passes over n runs about 6 % faster (Table 4.3).

Variant	Mean	Std. Dev.	Min	Max
B-tree	1.667	0.021	1.638	1.721
ART	1.768	0.023	1.732	1.827

Table 4.2.: Bulk-loading time statistics at TPC-H SF=1 for B-tree–based and ART-based entry trees.

Our results for TPC-H scale factor 10 look slightly differently with B-tree's bulk-loading time being faster by 1 second, which evaluates to 6% of the baseline time. This slowdown can partially be explained by the big O complexity computations above, but also might be due to cache effects.

Variant	Mean	Std. Dev.	Min	Max
B-tree	16.747	0.276	16.380	17.692
ART	17.997	0.297	17.624	18.897

Table 4.3.: Bulk-loading time statistics at TPC-H SF=10 for B-tree–based and ART-based entry trees.

To understand the memory requirements of our two entry-tree variants, we recorded the peak resident set size (RSS) during the bulk-load phase using GNU `time -v`. Despite ingesting a 245 MB input file at TPC-H SF 10, both the B-tree and ART loaders reached a maximum RSS of approximately 1.8 GB (1 858 MB for B-tree and 1 861 MB for ART). This elevated footprint arises from several contributing factors beyond the raw table data. Most of the work until the trigram extraction is the same,

- **In-Memory Table Representation:** We maintain two in-memory arrays of rows (the raw `vector<Row>` plus the transformed `vector<TableRow>`), consuming roughly 400–500 MB.

- **Posting-List Aggregation:** Building trigram-key-to-TID maps allocates per key vector-TID containers with geometric growth, adding over 200 MB.

- **Posting Trees:** For high-cardinality keys, the construction of B-tree-based posting trees incurs additional heap allocations and vector buffers, which together contribute hundreds of megabytes.
- **Entry Tree Structures:** The bulk-load routines allocate node objects ($\approx$ 26 for B-tree, $\approx$ 486 for ART) and temporary partition buffers (256 vectors per recursion level), further increasing RSS.
- **Allocator Overhead and Fragmentation:** The default C++ allocator maintains multiple memory arenas and per-object metadata, leading to fragmentation that can easily add hundreds of megabytes above the sum of live allocations.

Taken together, these components explain why the bulk-load process temporarily occupies nearly 1.8 GB of physical memory. The negligible 3 MB difference between the two variants (approx. 0.2 %) indicates that our custom pooling and data-layout optimizations for ART successfully contain its larger node count within the same overall footprint as the B-tree implementation.

To isolate the time spent in the core bulk-load algorithm from the remainder of the process, we instrumented the loader to separately measure runtime of the bulk-load invocation, encompassing all in-process data transformations and memory allocations. The ART's bulk loading time amounted to 0.23 milliseconds compared to B-tree's 0.01 milliseconds. These measurements show that the core bulk-load algorithm itself contributes only a vanishingly small fraction of the total load time at SF = 10, on the order of tenths of a millisecond versus tens of seconds end-to-end. When compared to the full bulk-load durations ($\approx$ 17 000 ms), these correspond to less than 0.002 % of the total runtime in both cases. The higher core time for ART reflects its three sequential dataset scans (one per trie level), whereas the B-tree loader's single-pass leaf construction yields an order-of-magnitude lower core cost. Nevertheless, because even the ART's 0.23 ms is negligible relative to the tens of seconds spent in I/O, memory allocation, and database-engine overhead, the choice of entry-tree structure has virtually no impact on overall bulk-load performance in our SF = 10 workload.

One further point merits attention which affects the bulk loading's execution time. In the ART design, the tree height remains fixed at three levels for all three-byte trigrams, independent of the vocabulary size. By contrast, a B-tree will grow in height once its distinct-key count exceeds the per-page fanout limit. In PostgreSQL's implementation, each leaf or internal page can accommodate approximately 500 key-pointer entries (see Section 3.1.2), so even text corpora with several thousand unique trigrams will rarely require more than three levels. At the end of this chapter, we conclude with detailed calculations that confirm this behavior for realistic trigram vocabularies.

4.2.2. Pattern Matching

In this subsection, we present the results of our LIKE–pattern matching experiments on the TPC-H dataset at scale factor SF10, reporting all timings in milliseconds for enhanced precision. We decompose each query into two phases: the *Index Search*, which measures only

the cost of traversing the entry tree (B-tree or ART) to retrieve candidate posting lists, and the *Full Verification*, which encompasses the index search, posting-list intersection, and final string verification against each candidate row. Results for SF1 exhibit the same relative behavior at roughly one-tenth the absolute latencies and are therefore omitted for brevity. By isolating these components on SF10 data, we can more clearly assess the impact of the underlying tree structure on search latency, independent of pattern-extraction and filtering overheads.

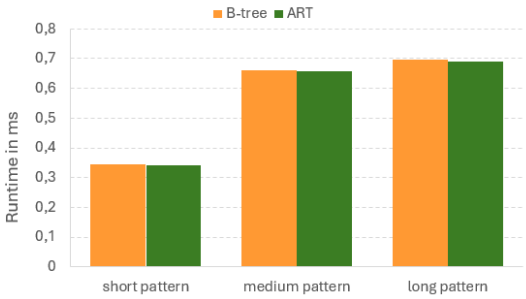

Figure 4.2.: Search-phase latency for B-tree and ART entry trees on the TPC-H SF10 dataset, measured for short, medium, and long LIKE patterns.

Figure 4.2 presents the search phase latency measurements(averaged over 50 runs) for short, medium, and long patterns, comparing the two entry tree implementations. For each pattern length, the mean latency of the two structures differs by at most a few microseconds, and their standard deviations are likewise comparable. This close correspondence indicates that the choice of tree layout (B-tree vs. ART) has a negligible impact on the cost of retrieving posting lists once trigrams are extracted. The short pattern incurs approximately 0.34 ms of search-phase work, the medium pattern around 0.65 ms, and the long pattern about 0.69 ms, almost exactly double the short-pattern cost in both cases. The narrow spread of the measurements (standard deviations of only a few microseconds) demonstrates that the search-phase is highly repeatable, with minimal influence from caching artifacts or system jitter.

Next, we formalize this observation with an asymptotic analysis showing that both ART and B-tree lookups execute in $\mathcal{O}(1)$ time under our experimental constraints.

ART's lookup cost is $\mathcal{O}(1)$ with respect to the total number of trigrams. Because each key is exactly three bytes long, the tree height is fixed at three levels, and each node lookup—whether in `Node4`, `Node16`, `Node48`, or `Node256`—executes in $\mathcal{O}(1)$ time. Consequently, a search always incurs exactly

$$3 \times \mathcal{O}(1),$$

which remains $\mathcal{O}(1)$ in n.

In the case of the B-tree, the cost analysis is similarly constant-time with respect to n. Each node (and each leaf page) contains up to a fixed—constant—number of key separators, denoted by `MAX_KEYS`. A binary search within such a node therefore costs

$$\mathcal{O}\big(\log(\texttt{MAX\_KEYS})\big) \;=\; \mathcal{O}(1),$$

since `MAX_KEYS` is a compile-time constant. In our experiments on a dataset with 454 trigrams, the B-tree has exactly three levels (root, one internal level, and leaf). Consequently, a full lookup visits three nodes, each in $\mathcal{O}(1)$ time, yielding

$$3 \times \mathcal{O}(1) \;=\; \mathcal{O}(1)$$

Node Type	Lookup Mechanism	Cost
Node4	Sequential loop over up to 4 keys, comparing each to the target byte	$\mathcal{O}(1)$
Node16	Single SSE instruction performing parallel comparison of 16 keys	$\mathcal{O}(1)$
Node48	Array lookup `childIndex[b]` to test the presence of a child pointer	$\mathcal{O}(1)$
Node256	Direct array access of 256 child pointers indexed by the target byte value	$\mathcal{O}(1)$
Leaf	Fixed-length `memcmp` of 3 bytes to compare full key versus lookup key	$\mathcal{O}(1)$

Table 4.4.: Per-node and leaf lookup mechanisms and their asymptotic costs in the ART.

overall. This matches the ART's three-level, constant-time lookup cost.

The second component of our pattern-matching evaluation incorporates the complete verification pipeline posting list intersection and literal-substring checks, allowing us to quantify the proportion of total execution time attributable to the entry-tree lookup phase. All timings are reported in milliseconds.

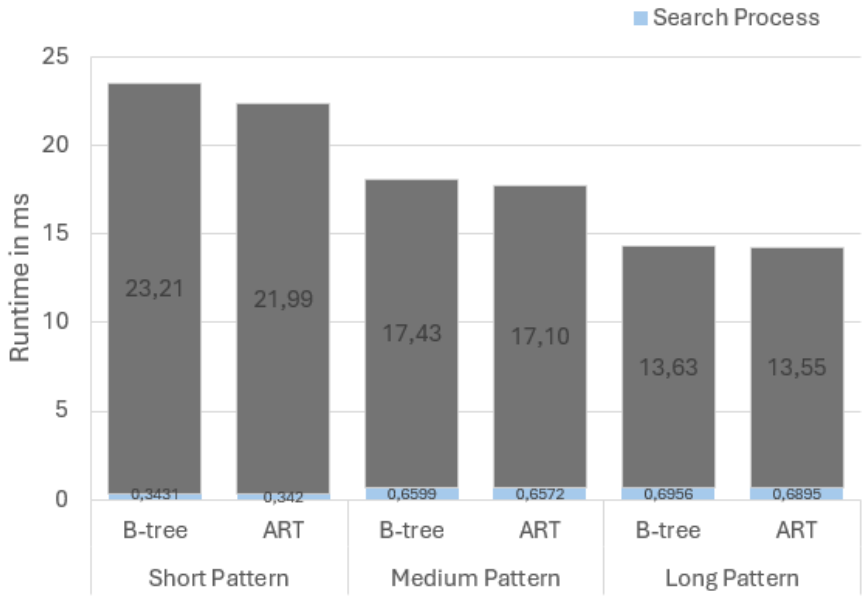

Figure 4.3.: The runtime of the tree search algorithms as part of the full pattern matching processes for short, medium and long patterns for B-tree and ART. Each pattern length clusters B-tree and ART's performance metrics into one section.

Figure 4.3 illustrates the total query durations for B-tree and ART across the three pattern lengths. As anticipated, the overall runtimes for each pattern exhibit minimal divergence between the two tree structures. We decompose these totals to determine the fraction consumed by the index-search phase: The lookup stage accounts for approximately 1% of the total runtime in the B-tree variant and 2% in the ART variant for short patterns, both implementations devote roughly 4% of their end-to-end time to tree traversal for medium

length patterns, and the index-search cost rises to about 5 % of the overall execution time for both B-tree and ART for long patterns. These results confirm that in our implementation the bulk of the query cost derives from posting-list intersection and string verification rather than the entry-tree lookup itself. Moreover, the steadily increasing lookup fraction—from 1–2 % for short patterns up to 5 % for long patterns suggests a roughly linear relationship between the number of trigram lookups and the index-search time. To rigorously validate this linear scaling and isolate any non-linear overheads, future work should collect additional data points on an in-memory testbed with varied pattern complexities.

Based on this analysis, a B-tree's lookup time will only surpass that of the ART when the tree's height becomes sufficiently large. To quantify what "sufficiently large" means, we performed calculations and present the resulting values below.

B-tree's depth in trigram indexing. Recall from Section 2.1 that the height h of a B-tree storing n distinct keys, where each node can hold up to MAX_KEYS children but must have at least $\lceil \text{MAX\_KEYS}/2 \rceil$ children when non-root, satisfies the bound

$$h \;\le\; \log_{\lceil \text{MAX\_KEYS}/2 \rceil}\!\Big(\frac{n+1}{2}\Big).$$

In PostgreSQL's default configuration, a B-tree page can hold approximately MAX_KEYS = 500 key-pointer entries, so

$$t \;=\; \Big\lceil \frac{500}{2} \Big\rceil \;=\; 250$$

is the minimum number of children for any non-root node. Hence, to achieve height $h \ge 4$, one requires

$$n \;>\; 2\,t^3 \;=\; 2 \times 250^3 \;=\; 31{,}250{,}000$$

distinct keys. Because the universe of all possible three-byte trigrams is

$$256^3 = 16{,}777{,}216,$$

and real-world text corpora contain far fewer than 31.25 million unique trigrams, a PostgreSQL trigram index will remain at height three (or below) in all practical scenarios.

For comparison, using the simpler, and overly optimistic approximation

$$h = \Big\lceil \log_{\text{MAX\_KEYS}}\, n \Big\rceil,$$

one would need

$$n > 500^3 = 125{,}000{,}000$$

keys to reach $h \ge 4$. In either analysis, no realistic dataset exceeds these thresholds, so the B-tree height for trigram indexing remains three levels.

5. Conclusion and Outlook

5.1. Conclusion

This work set out to evaluate whether an in-memory ART could optimally replace PostgreSQL's traditional B-tree-based entry layer in a GIN-induced trigram index with potential performance improvements. We first reimplemented PostgreSQL's trigram indexing mechanism in C++, encompassing trigram extraction, index-tuple formation (with inline posting lists or posting-trees), and a B-tree–based entry tree (in Chapters 3.1 and **??**). We then introduced an ART variant for the entry tree (Chapter 3.2), leveraging ART's adaptive node layouts and some path-shortening optimizations for three-byte trigram keys. For both implementations, we also built LIKE pattern matching to evaluate and compare each entry tree's performance on substring search workloads.

Our experimental study yielded the following key findings:

1. Bulk-load performance: On SF=1, B-tree and ART loaders complete in 1.667 s vs. 1.768 s respectively; on SF=10, they finish in 16.747 s vs. 17.997 s (6% slower for ART).

2. Memory footprint: Both variants consume ≈1.86 GB RSS during bulk-load, with only a 3 MB (0.2 %) difference, indicating that ART's higher node count is offset by compact node layouts and shared allocator arenas.

3. Lookup latency: Search-phase times for short, medium, and long LIKE patterns on SF10 differ by mere microseconds between B-tree and ART, each exhibiting equally low lookup cost due to fixed tree heights.

4. Overall query cost: Entry-tree traversal accounts for only 1–5 % of end-to-end LIKE query times; the dominant costs remain posting-list intersection and final string verification (Figure 4.3).

5. Theoretical depth bound: A B-tree with per-page fan-out ≈500 remains at three levels for up to 31 million distinct trigrams, matching ART's fixed three-level depth.

In summary, replacing the B-tree entry layer with ART for trigram indexing in a static, in-memory setting yields negligible performance differences. While ART introduces one extra full scan during bulk-load, its richer node representations and constant-time byte-wise lookups match B-tree performance for practical trigram vocabularies.

5.2. Outlook

Although our empirical results did not ultimately favor the ART-backed entry tree over the well-tuned B-tree implementation, they highlight several avenues for further inquiry. First, our study was confined to a static dataset with a limited, repetitive vocabulary; evaluating the ART integration on more complex, semi-structured collections such as `JSON` documents subject to frequent updates may reveal use cases in which its adaptive node layouts deliver clearer benefits or expose unforeseen maintenance costs.

Second, we deployed a generic ART implementation without GIN-specific enhancements, while the B-tree variant benefits from decades of tailored optimizations within PostgreSQL. Developing ART node representations, bulk-load algorithms, and concurrency controls that align with GIN's transactional model could close the performance gap or even tip the balance in ART's favor.

Moreover, prior work by Leis et al. [1] and Binna et al. [2] has shown that, for pure random-key lookups and certain in-memory scenarios, ART can significantly outperform even cache-sensitive B-tree variants. Nevertheless, these performance gains are highly sensitive to workload characteristics—such as key distribution, access patterns, and update rates and to the specific optimizations applied to each structure. A systematic, use-case-driven comparison across a broad spectrum of application domains and index configurations remains an important open question, and one that we leave to future work.

A. Appendix

A.1. SQL Commands for PostgreSQL Performance Results

Below are the SQL commands we used to establish results from the Section on PostgreSQL performance 4.1.3.

Listing A.1: SQL statements for index drop/create and query plans

```
-- Drop existing GIN index if present
DROP INDEX IF EXISTS gin_idx;

-- Warm-up queries before index creation
SELECT COUNT(*) FROM part WHERE p_name LIKE '%mon%ros%';
EXPLAIN SELECT COUNT(*)
  FROM part
  WHERE p_name LIKE '%chocolate%mon%';
EXPLAIN SELECT COUNT(*)
  FROM part
  WHERE p_name LIKE '%lavender%almond%';

-- Create GIN-trigram index
CREATE INDEX gin_idx
  ON part USING gin (p_name gin_trgm_ops);

-- Benchmark queries after index creation
EXPLAIN SELECT COUNT(*)
  FROM part
  WHERE p_name LIKE '%mon%ros%';
EXPLAIN SELECT COUNT(*)
  FROM part
  WHERE p_name LIKE '%chocolate%mon%';
EXPLAIN SELECT COUNT(*)
  FROM part
  WHERE p_name LIKE '%lavender%almond%';
```

A.2. Results for TPC-H SF1

Below are the runtime results over 50 iterations of bulk loading, and looking up different patterns in the B-tree-based entry tree.

Table A.1.: Bulk-loading and candidate-retrieval times per iteration (in seconds)

Iteration	Bulk load	Short	Medium	Long
1	1.77048	0.000354698	0.00054197	0.000572872
2	1.80535	0.000353752	0.000747839	0.00057218
3	1.75649	0.000356545	0.00052364	0.000576729
4	1.74158	0.000355821	0.000531456	0.000573107
5	1.79414	0.000359904	0.000524046	0.000585157
6	1.74961	0.000354984	0.000531163	0.000586291
7	1.7534	0.000356437	0.00053127	0.000570335
8	1.76194	0.000353684	0.000748508	0.000575084
9	1.7791	0.000352668	0.000525726	0.000571528
10	1.74108	0.0003677	0.00052128	0.000576459
11	1.74429	0.000369015	0.000527644	0.000577882
12	1.77672	0.000359051	0.000528019	0.000577767
13	1.77827	0.000353855	0.000530238	0.000575232
14	1.8008	0.00035421	0.000525089	0.000578354
15	1.76831	0.000368845	0.00055289	0.000601296
16	1.77365	0.0003689	0.000523677	0.000580257
17	1.76419	0.000371195	0.000530994	0.000583348
18	1.73192	0.000353338	0.000536825	0.00056997
19	1.75953	0.000354606	0.000536158	0.000568079
20	1.78681	0.000355378	0.000540876	0.000570988
21	1.76604	0.000355805	0.000529154	0.000578442
22	1.77708	0.000368133	0.000759514	0.000573099
23	1.76058	0.000353865	0.000530072	0.000568639
24	1.76173	0.000372399	0.000524349	0.000576293
25	1.76773	0.000357419	0.000676769	0.000576439
26	1.75496	0.000353249	0.000521957	0.000567104
27	1.80314	0.000371665	0.00060927	0.000575617
28	1.74372	0.000349866	0.000534425	0.000580976
29	1.74192	0.000354214	0.000524054	0.000569971
30	1.73606	0.000351666	0.000531024	0.000569809
31	1.75507	0.000372309	0.000522159	0.000574563
32	1.7985	0.000354861	0.000526944	0.000567087
33	1.75229	0.000350489	0.000523521	0.000572216
34	1.75285	0.000372412	0.000527184	0.000585685
35	1.76892	0.000354305	0.000532429	0.000569199
36	1.78655	0.000354314	0.000555308	0.000574652
37	1.75211	0.000361773	0.000526838	0.000580441
38	1.82745	0.000354442	0.000531839	0.00057062
39	1.73837	0.000360556	0.000523705	0.000588951
40	1.76053	0.000374886	0.000521476	0.000573152
41	1.82714	0.000353779	0.000525765	0.000573019
42	1.81672	0.000354731	0.000524001	0.000567359
43	1.74187	0.000350462	0.000529018	0.000570938
44	1.76102	0.000352569	0.000533271	0.000570337
45	1.79373	0.00035648	0.000523829	0.000574477
46	1.74749	0.000360536	0.000524753	0.000581705
47	1.76825	0.00035505	0.000536596	0.000575653
48	1.74531	0.000354068	0.000531049	0.000580402
49	1.76335	0.000355941	0.000526334	0.000577997

50	1.78596	0.000354572	0.000525827	0.000569947

And the equivalent results for the ART:

Table A.2.: Bulk-loading and candidate-retrieval times per iteration (in seconds)

Iteration	Bulk load	Short	Medium	Long
1	1.64479	0.000405698	0.000524234	0.000593683
2	1.66143	0.000381984	0.000530043	0.000581807
3	1.70873	0.00038594	0.000535162	0.00057657
4	1.64783	0.000399293	0.000521192	0.000580513
5	1.68649	0.000383253	0.000533799	0.000596478
6	1.68522	0.000382004	0.000757024	0.00058361
7	1.64793	0.000381721	0.000530674	0.000567976
8	1.64883	0.000381611	0.00055001	0.000606029
9	1.66705	0.00040135	0.000523535	0.000589564
10	1.67604	0.000395202	0.000533083	0.000579892
11	1.66758	0.000405547	0.000521375	0.000793853
12	1.65463	0.00038704	0.000531032	0.000593517
13	1.70739	0.000396293	0.00053457	0.000594285
14	1.64678	0.000382939	0.000533683	0.000571486
15	1.6861	0.000381832	0.000525084	0.000586193
16	1.6663	0.000385877	0.000532696	0.000575089
17	1.66489	0.000408671	0.000529768	0.000588957
18	1.66081	0.000407513	0.000525245	0.000580406
19	1.65646	0.00039277	0.000545998	0.000589317
20	1.67621	0.000381005	0.00053118	0.00059401
21	1.67086	0.000393373	0.000519891	0.000802925
22	1.67805	0.000393517	0.000539488	0.000581704
23	1.712	0.000403133	0.000540325	0.000621546
24	1.65838	0.000393883	0.000539244	0.000574137
25	1.63783	0.000399041	0.000534427	0.000602415
26	1.65306	0.00038807	0.000529176	0.000578229
27	1.64597	0.000382967	0.000526426	0.000593388
28	1.66722	0.000385367	0.000535835	0.000569462
29	1.65283	0.000380157	0.000530247	0.000571801
30	1.65045	0.000384984	0.00054213	0.000574195
31	1.6936	0.000624888	0.000532293	0.000578799
32	1.6835	0.000412125	0.000524513	0.000794839
33	1.64648	0.000414338	0.000522949	0.000589637
34	1.65409	0.000382186	0.000533087	0.000569212
35	1.63855	0.00039269	0.000523436	0.000590114
36	1.72104	0.000409624	0.000519496	0.000587377
37	1.66719	0.000406541	0.000528983	0.000839514
38	1.64272	0.000377654	0.000531786	0.000575293
39	1.63798	0.000412269	0.000524079	0.000583004
40	1.70401	0.00037997	0.000527493	0.000573478
41	1.69128	0.000391613	0.000520466	0.000573154
42	1.66671	0.000381397	0.000536524	0.00060111
43	1.66467	0.000393076	0.000524431	0.000590087
44	1.63944	0.000381837	0.000530783	0.000585828
45	1.66606	0.000385315	0.000528856	0.000577735
46	1.67919	0.000381825	0.00052912	0.000588368
47	1.67291	0.000405968	0.00052271	0.000581908
48	1.6538	0.000388315	0.000543603	0.000572305
49	1.68614	0.000407722	0.000531603	0.000589131
50	1.66976	0.000398799	0.000519242	0.000585177

List of Figures

List of Tables

Bibliography

[1] V. Leis, A. Kemper, and T. Neumann. "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases". In: (2013). DOI: https://db.in.tum.de/~leis/papers/ART.pdf.

[2] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. "HOT: A Height Optimized Trie Index for Main-Memory Database Systems". In: *SIGMOD'18: 2018 International Conference on Management of Data* (2018), pp. 521–534. DOI: https://dl.acm.org/doi/10.1145/3183713.3196896.

[3] R. Bayer and E. M. McCreight. "Organization and Maintenance of Large Ordered Indices". In: *Acta Informatica* 1.3 (1972), pp. 173–189. DOI: 10.1007/BF00288933.

[4] E. FREDKIN. "Trie memory". In: *Communications of the ACM 3* (1960), pp. 490–499. DOI: https://dl.acm.org/doi/10.1145/367390.367400.

[5] D. E.Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching*. Addison Wesley, 1973, page 481.

[6] R. E. Lewand. *Cryptological Mathematics*. Mathematical Association of America, 2000.