UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

Hashed CompactLTJ: Improving the Efficiency of Compact Leapfrog Triejoin in Graph Databases

PROPUESTA DE TESIS PARA OPTAR AL TÍTULO DE MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

Santiago de Chile
Julio 2024

1. Introduction

Graph databases are powerful tools for modeling and managing complex, interconnected data. These databases represent relationships between entities as edges connecting nodes in a graph, allowing for intuitive and flexible querying of patterns. A common approach to represent graph data is as triples of subject, predicate, and object (SPO), where joins are used to identify patterns in the data.

However, many systems rely on pairwise join algorithms, which are inefficient, particularly for large-scale datasets and queries involving graph structures. For example, in queries like the triangle query (R(a,b) ⋈ S(b, c) ⋈ T(c, a), with $|R| = |S| = |T| = n$), pairwise joins often require $O(n^2)$ time due to large intermediate results, despite the optimal bound being $O\left(n^{\frac{3}{2}}\right)$. This inefficiency arises because they process joins sequentially, leading to large intermediate results that could be minimized by filtering across all relations simultaneously [10].

Algorithms like Leapfrog Triejoin (LTJ) have become a standard for worst-case optimal (WCO) performance by filtering across all relations simultaneously, avoiding the inefficiencies of pairwise joins [11]. However, its main disadvantage is that it requires storing the tuples in tries for the six possible orders of the attributes, which causes high space usage. Because of this, many systems choose to implement versions that are not optimal in the worst case (non-wco) [2].

Given the exponential growth of data, it is crucial to design compact data structures that maintain a balance between space and query efficiency. Compact structures seek to reduce data size to near entropy levels without compromising query efficiency [8: p.2-3]. CompactLTJ is a compressed version of LTJ that significantly reduces the space required, achieving up to 75% space savings compared to traditional (wco) implementations, without losing efficiency in most queries [2].

However, CompactLTJ has certain limitations that affect its time efficiency. In particular, during the join process, one phase involves identifying the children of a node in the trie through exponential search, which introduces a logarithmic factor into the overall query complexity [2]. While CompactLTJ optimizes its performance in terms of alternation complexity, the overhead from this search can still become significant.

In this research, we explore a new variation of CompactLTJ that replaces the exponential search with a hashing-based method. Hashing offers the potential to improve temporal performance by reducing the search overhead during joins. While this approach may not achieve the theoretical alternation complexity of the original algorithm, it ensures that the intersection cost stays within the minimum size of the two child sets being intersected, as is common practice in many modern database systems.

Our goal is to see if this variant with hashing achieves better results in practice compared to the original version, without compromising its compactness, and to see if this trade-off can be correlated with the alternation complexity of the queries.

2. Related Works and Concepts

In this section, we review the key concepts and works fundamental to our research. We begin with the AGM bound and worst-case optimality (WCO), which define theoretical limits for join algorithms. Next, we introduce alternation complexity, which measures the inherent difficulty of solving intersection problems based on dataset characteristics. We then examine the Leapfrog Triejoin (LTJ) algorithm, the foundation for CompactLTJ. Following this, we explore CompactLTJ itself, the basis for our proposed variant. Finally, we discuss minimal perfect hashing functions (MPHF), a crucial component in our enhancement.

2.1. AGM Bound and Worst-Case Optimality

2.1.1. AGM Bound

The AGM bound defines the maximum number of tuples in a join query result for any database instance with the same size and structure. This bound represents the join as a hypergraph, where each attribute is a node and each relation is a hyperedge connecting relevant attributes. Using a *fractional edge cover*, which allows edges to take values between 0 and 1, as long as the sum of values for edges covering each attribute is at least 1, the AGM bound calculates an upper limit by taking the product of the sizes of each relation raised to its fractional cover value[1] [3, 6].

For example, for the triangle query $Q = R(A, B) \bowtie S(B, C) \bowtie T(C, A)$, the AGM bound provides an upper limit of $\sqrt{|R| \cdot |S| \cdot |T|}$ on the result size. In Figure 1, each relation R, S, and T is represented as an edge covering attributes A, B, and C. By setting fractional cover values $x_R = x_S = x_T = \frac{1}{2}$, we satisfy the condition that each vertex is covered with a sum of at least 1. This bound gives an efficient estimate of the maximum output size for the query [10].

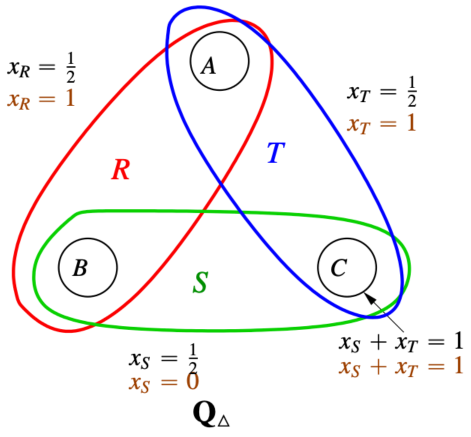


Figure 1: Fractional edge cover for the triangle query [10].

2.1.2. Worst-Case Optimality

Let $R_1, \ldots, R_n$ be relations of a database instance D. An algorithm to compute $\bowtie_{i=1}^{n} R_i$ is WCO if it takes time T and there exist $R_{1'}, \ldots, R'_n$ with R'_i having the same attributes as R_i and same size, such that $|\bowtie_{i=1}^{n} R_i| = \Omega(T)$. In other words, a WCO algorithm for computing a join may not always be the

[1] Mathematically, the AGM bound is defined as $Q^* = \Pi_{\{F \in E\}} |R_F|^{x_F}$, where Q^* is the maximum join size, R_F are the relations, and x_F are the fractional edge cover values constrained by $\Sigma_{\{F: v \in F\}} x_F \geq 1, \forall v \in V, x_F \geq 0$ [3, 6].

fastest for a specific query instance, however, if it is WCO, then while a particular query might be solved more quickly with another approach, there exists a similar query (same relation sizes and attributes) where it is impossible to achieve a faster solution than with the WCO algorithm [3, 6].

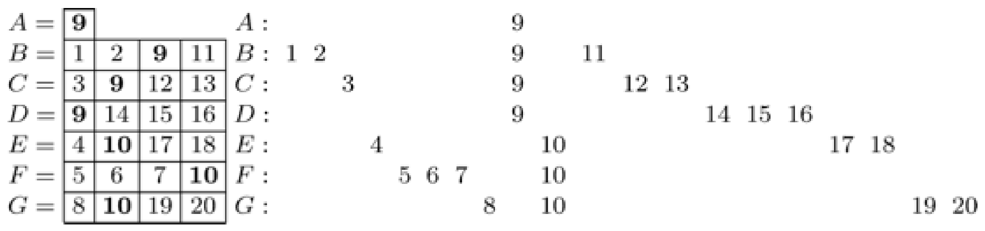
It was shown that a join algorithm is WCO if it has a running time of $\tilde{O}(Q^*)$, where Q^* is the AGM bound for the query, and the tilde notation hides possible polylogarithmic factors [3, 6].

The first WCO join algorithm was developed by Ngo, Porat, Ré, and Rudra (NPRR) [9]. Soon after, Leapfrog Triejoin (LTJ) [11] was introduced as a more straightforward approach that also meets WCO guarantees, providing a practical option for systems seeking worst-case optimality in join processing.

2.2. Alternation Complexity

Alternation complexity measures the difficulty of solving an intersection problem by counting the minimal number of intervals, or “switches”, needed to confirm which elements are shared across sets. This metric provides a lower bound on the number of checks an algorithm requires, enabling comparisons between deterministic and randomized approaches. By capturing how often the dataset composition changes, alternation complexity offers insights into algorithm efficiency, especially for data that is sparse or unevenly distributed [4].

To illustrate this, we will go through its definition and a guiding example. This example will help show how algorithms with low alternation complexity can reduce unnecessary checks, especially when data is sparse or unevenly distributed.



$A =$	**9**			
$B =$	1	2	**9**	11
$C =$	3	**9**	12	13
$D =$	**9**	14	15	16
$E =$	4	**10**	17	18
$F =$	5	6	7	**10**
$G =$	8	**10**	19	20

```
A :                      9
B : 1 2                  9    11
C :       3              9        12 13
D :                      9              14 15 16
E :         4              10                     17 18
F :           5 6 7        10
G :                    8   10                           19 20
```

Figure 2: Example instance of the intersection problem showing seven sorted arrays (A through G) [4].

Let U be a total order set. An *instance* of the intersection problem is a collection of k ordered sets $A_1, \ldots, A_k$ over U. The *intersection* of the instance is $\bigcap A_i$. For example, in Figure 2, the intersection of the instance is empty.

Any algorithm computing the intersection must verify two things: that each element in the output belongs to all k arrays, and that no elements have been omitted. This is done using a partition certificate, which serves as proof that only the elements in the output could be in the intersection [4].

A *partition certificate* is a partition $\left(I_j\right)_{j \leq \delta} \subseteq U$ such that:

1. $\forall I \in \left(I_j\right), \quad I = \{x\} \Rightarrow x \in \bigcap A_i$
2. $\forall I \in \left(I_j\right), \quad |I| > 1 \Rightarrow \exists A \in (A_i) : I \cap A = \emptyset$

For example, in Figure 2, a possible partition certificate is $\left(I_j\right) = (-\inf, 9), [9, 10), [10, +\inf)$.

The *alternation* δ of an instance $(A_1, ..., A_k)$ is the minimal number of intervals forming a partition certificate. Mathematically, $\delta = \min\{|(I_j)| : (I_j) \text{ is a partition certificate for } (A_i)_{i \leq k}\}$. In the example, the alternation is 3, as the partition certificate has three intervals and no smaller certificate exists [4].

This metric provides a way to analyze how efficiently an algorithm performs intersection operations by counting the minimum number of comparisons, capturing the "switches" in dataset overlap. Alternation complexity might offer a better performance bound in more realistic data scenarios—particularly when data is sparse or unevenly distributed—helping us understand trade-offs between different join algorithms.

2.3. Leapfrog Triejoin

Leapfrog Triejoin (LTJ) is a straightforward yet widely adopted worst-case optimal (WCO) algorithm that efficiently computes joins by eliminating one attribute at a time rather than one relation, as traditional join approaches do. LTJ achieves this by iterating over possible values in the output and branching on subsets that match each value. To operate efficiently, LTJ requires relations to be stored in a trie structure. Although LTJ remains WCO regardless of the elimination order, performance can vary significantly, and storing each relation in all attribute orders can lead to high space requirements [2, 11].

2.3.1. Trie Construction for Multi-attribute Relations

LTJ uses tries to represent each relation. The order in which attributes are joined is given by a specified variable order, which the trie structure respects. For a relation $R(a, b)$, this structure allows efficient traversal from one attribute value to the next (e.g., values of a followed by their corresponding b values). If the initial trie order does not match the query's variable order, a different trie structure with the attributes reordered must be used, for example, for a query $Q(a, b, c) \leftarrow R(a, b), S(b, c), T(c, a)$, with (a, b, c) as the query order, we would need to use the trie $T'(a, c)$ instead of $T(c, a)$ to match the query order. This means that in the worst case, we need to build or store tries for all possible orders of the attributes for each relation [11].

On Figure 3, we can see the trie structure for a relation $A(x, y, z)$, where the first level corresponds to possible values of x, the second level to values of y that follow each x, and the third level to values of z that follow each y.

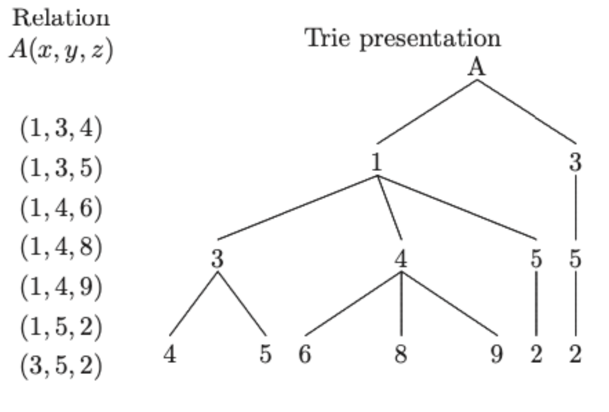


Figure 3: Trie structure example for a relation $A(x, y, z)$. [11].

2.3.2. Join Execution

When performing a join, such as $Q(a, b, c) \leftarrow R(a, b), S(b, c), T(a, c)$, LTJ constructs leapfrog joins for each variable based on the variable order. In this example, with the variable order $[a, b, c]$:

1. LTJ performs a leapfrog join over the projections $R(a, -)$[2] and $T(a, -)$ to identify valid a values.
2. For each a, it performs a leapfrog join on $R_a(b)$[3] and $S(b, -)$ to locate valid b values.
3. For each b, it performs a leapfrog join on $S_b(c)$ and $T_a(c)$ to find valid c values.

This hierarchical elimination of joins avoids the need for intermediate results and reduces redundant checks, ensuring worst-case optimality [11].

2.3.3. Leapfrog Join

In a leapfrog join, each relation is represented as an iterator over sorted keys. Given a set of relations A, B, C with values as shown in Figure 4, the leapfrog join algorithm maintains a pointer for each iterator, positioning each pointer at the smallest key. It then advances the smallest pointer to match the largest, "*leapfrogging*" each pointer until all pointers align at the same key. This technique avoids redundant checks and unnecessary computations, enabling LTJ to quickly identify empty intersections in scenarios where a pairwise join would otherwise produce a large intermediate result [11].

For example, if $A = \{0, ..., 2n - 1\}$, $B = \{n, ..., 3n - 1\}$, and $C = \{0, ..., n - 1, 2n, ..., 3n - 1\}$, a pairwise join would yield n results, while LTJ detects the empty intersection of $A \cap B \cap C$ in $O(1)$ steps [11]. This is an example of where having a good alternation complexity can drastically reduce the number of comparisons needed to find the intersection of the iterators.[4]

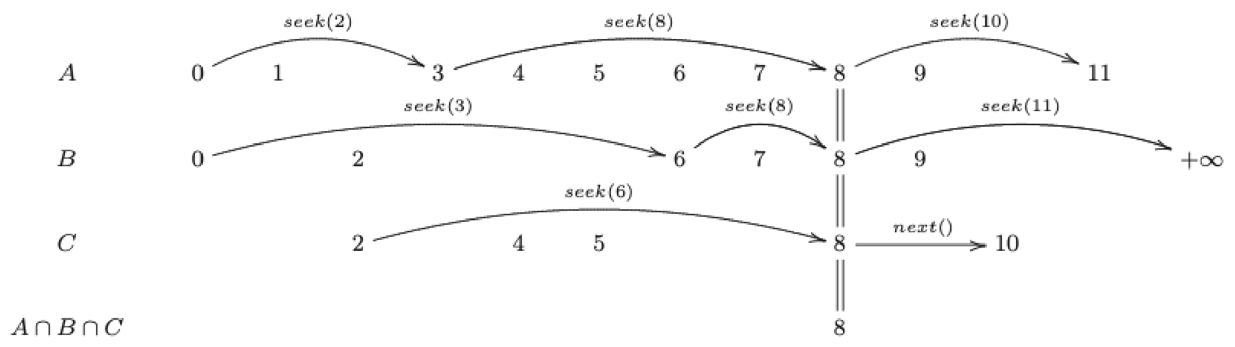


Figure 4: Leapfrog join example with three unary relations [11].

Indeed, LTJ's ability to avoid intermediate results and redundant checks makes it a powerful and efficient approach for worst-case optimal joins. The main drawback of LTJ is the high space usage due to the trie structures, which needs all possible orders of the attributes for each relation. To address this limitation, CompactLTJ was developed as a compressed version of LTJ that significantly reduces the space required while maintaining WCO guarantees [2].

[2]The notation $R(a, -)$ denotes the projection of R on attribute a

[3]The notation $R_a(b)$ denotes the curried version of R on attribute a with value b (i.e., $R_a(b) := \{b \mid (a, b) \in R\}$)

[4]For this example, a possible minimal partition certificate is $(-\inf, 2), [2, 6), [6, 8), \{8\}, (8, 10), [10, +\inf)$, with an alternation of $\delta = 6$.

2.4. CompactLTJ

CompactLTJ achieves its space efficiency by representing the trie structure and its edge labels separately. The trie topology is encoded using the **Level-Order Unary Degree Sequence (LOUDS)** representation (a compact trie) [7], while the edge labels are stored in a compact array. These optimizations allow CompactLTJ to use approximately 25% of the space required by traditional LTJ implementations, while maintaining comparable query performance [2].

Next, we will study the implementation in more detail.

2.4.1. Trie Topology

The LOUDS representation, central to the space efficiency of CompactLTJ, encodes the topology of an n-node trie using a bitvector T. Each node with d children is represented by the sequence $0^d 1$, reflecting its degree. The trie is traversed level by level (breadth-first), and the bitvector sequentially encodes the degree of each node. Unlike the original LOUDS approach, CompactLTJ omits leaf nodes from the encoding, further reducing the space requirements to exactly one bit per edge (represented as $0^{d-1}1$). This optimization is feasible because all leaves in the trie are at the same depth [2].

In Figure 5, we can see an example. The traditional LOUDS encoding is shown in red, while the CompactLTJ encoding is shown in blue, resulting in $T = \underbrace{001}_{P}\ \underbrace{0001\ 1\ 1}_{S}\ \underbrace{1\ 1\ 1\ 1\ 00001\ 0001}_{O}$

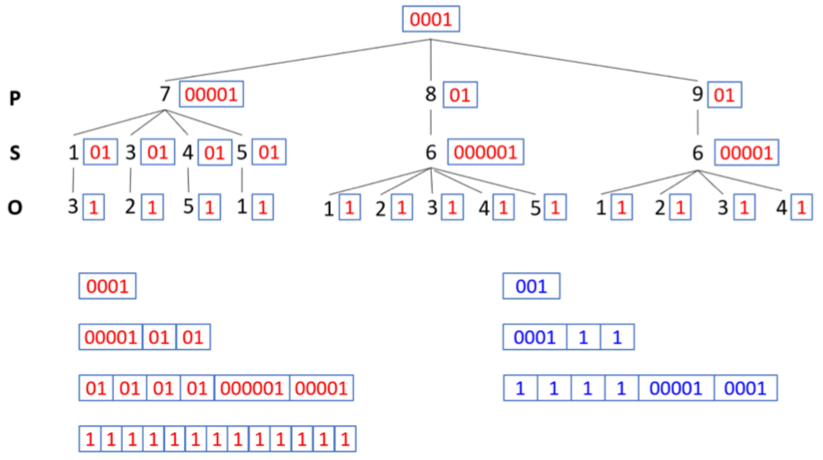


Figure 5: LOUDS representation of a trie topology [2].

2.4.2. Trie Navigation

In order to understand how CompactLTJ implements tries traversal, we need to review the main bitmap operations it uses. The two key operations are ***select*** and ***selectnext***, which are used to navigate the trie structure efficiently.

The following table summarizes these operations:

Operation	**Description**	**Time Complexity**	**Space Complexity**	**Example (using T)**
$\mathbf{select}_b(T, i)$	Finds the position of the i-th occurrence of bit b (0 or 1) in the bitvector T.	O(1)	$o(n)$	$\text{select}_1(T, 3) = 8$
$\mathbf{selectnext}_b(T, i)$	Finds the position of the next occurrence of bit b after index i in T.	O(1)	$o(n)$	$\text{selectnext}_1(T, 4) = 7$

Table 1: Summary of bitmap operations in CompactLTJ.

These operations allows us to compute $\text{child}(v, i)$ as the i-th child of node v in the trie, and $\text{degree}(v)$ as the number of children of node v. In this representation, each node v corresponds to the encoding $T[v + 1, ..., \ v + \text{degree}(v)]$, in other words, a node v identifies a subsequence $0^{d-1}1 \in T$ starting at position $v + 1$. For example, in the trie of Figure 5, $v = 3$ identifies the subsequence 0001 that corresponds to the first child of the root [2].

Trie navigation operations are defined as follows: [2]

1. $\mathbf{child}(v, i) = \mathbf{select}_1(T, v + i)$. For example, the second child of the root is $\text{u} = \text{child}(0, 2) = \text{select}_1(T, 0 + 2) = 7$.
2. $\mathbf{degree}(v) = \mathbf{selectnext}_1(T, v + 1) - v$. For example, the number of children of the first child of the root ($v = 3$) is $\text{degree}(3) = \text{selectnext}_1(T, 3 + 1) - 3 = 7 - 3 = 4$. And we can also see that $v = 3$ identifies the subsequence $T[3 + 1, 3 + 4] = T[4, 7] = 0001$.

These operations allow CompactLTJ to navigate the trie efficiently, descending from the root to the desired nodes. Now the only remaining step is to find the edge labels associated with each child, which we will cover next.

2.4.3. Edge Labels

The labels on the edges of the trie are stored in a compact array L, where each label occupies $\lceil \log U \rceil$ bits, with U being the size of the universe of constants. The labels are stored in level-order, aligned with the traversal of the trie. For any node v, the labels of its children are stored consecutively in L as shown below:

$$T = \ 001 \ \ 000111 \ \ 1111000010001$$

$$L = \ 789 \ \ 134566 \ \ 3251123451234$$

This allows efficient access during join operations. For example, in Figure 5, the edge labels for the children of the root node ($v = 0$) are $L[v + 1, v + \text{degree}(v)] = L[1..3] = \{7, 8, 9\}$.

Having covered all of these components, we can now study how CompactLTJ performs trie navigation and join operations efficiently.

2.4.4. Join Process

During the join process in CompactLTJ, the algorithm traverses the compact trie structures for all relations simultaneously. For a given node v, the algorithm identifies its children in the bitvector T using the compact operations $\text{child}(v, i)$ and $\text{degree}(v)$, which give access to the indices of its children and their labels stored consecutively in L.

To perform the join, the algorithm intersects the sets of children from the relevant tries. This intersection step ensures that only the values common across all participating relations are considered, effectively pruning invalid branches early. The edge labels in L provide the constants needed for this intersection, enabling the algorithm to move to the next level of the trie only when a match is found. This process is repeated iteratively, binding one variable at a time until all variables are resolved, producing a complete solution to the query.

2.4.5. Advantages and Limitations

CompactLTJ achieves a remarkable balance between space and time efficiency. It uses just 3.3 times the space of raw triple data, compared to 14 times for traditional WCO implementations like MillenniumDB. Additionally, its query performance is on par with the fastest WCO systems, often outperforming them in terms of median query times [2].

CompactLTJ spends most of its time performing intersection operations, which are the core of its execution. To remain compact, it uses exponential search during these operations instead of storing additional information, such as hash tables. This trade-off minimizes space usage but introduces a logarithmic factor in query complexity. In traditional approaches, storing extra data to speed up intersections is not a concern, as compactness is not a priority; however, for CompactLTJ, such an approach could compromise its space efficiency. Our research aims to explore whether replacing exponential search with hashing can improve intersection efficiency while preserving CompactLTJ's compactness. Additionally, we will investigate how to implement hashing in a way that is both time-efficient and space-efficient for this context.

2.5. Minimal Perfect Hashing Functions (MPHF)

Since our dataset is static, we can leverage Minimal Perfect Hash Functions (MPHF) to assign unique, collision-free hash values to keys with minimal space overhead. These functions are ideal for scenarios requiring efficient lookups and compact storage [8: p.92-96].

2.5.1. Space and Time Complexity of MPHF

MPHF offer efficient operations with $O(1)$ query time, making them highly suitable for scenarios where fast lookups are critical. In terms of space complexity, MPHF can be constructed with as little as $2.46n + o(n)$ bits[5], achieving nearly optimal storage requirements while maintaining their performance [8: p.92-96].

2.5.2. Construction of Perfect Hash Functions

MPHFs can be constructed using various approaches, each with different trade-offs. Here, we describe a method based on a randomized Las Vegas approach, which ensures correctness with high probability. This construction is particularly promising for our study due to its minimal space requirements and suitability for static datasets [5]. The key components of the construction include:

[5]Plus, $s'n$ or $s'm$ bits for satellite data, depending on the construction method, where s' is the number of bits needed to store each satellite value [8: p.92-96].

- **Parameter m**: The size of the hash table is chosen as $m = \left\lceil \frac{c \cdot n}{3} \right\rceil \cdot 3$ for a constant $c > 1.23$, ensuring that the construction succeeds with a constant number of retries and high probability [8: p.92].
- **Hash Functions**: Three independent hash functions, h_0, h_1, h_2, are used to map each key x to a triplet $e_x = (h_0(x), h_1(x), h_2(x))$. Each hash function outputs values within disjoint intervals of $[0, m - 1]$[6] [8: p.92].
- **Sorting and Allocation**: The values $v \in [0, m - 1]$ must satisfy certain criteria to ensure unique mappings for all keys. The construction involves sorting triples and allocating values to ensure that conflicts are resolved probabilistically, where the size of m helps bound the number of retries. Full details on the construction process can be found in the literature [8: p.92-93].

The process generates two auxiliary arrays:

1. $G[v]$: An array determining which of the three hash functions (h_0, h_1, h_2) should be used for each key. (See $h(x)$ definition below.)
2. $V[v]$: Indicates whether a value has already been visited during the construction.

The actual hash function $h(x)$ is computed as:

$h(x) = h_j(x)$, where $j = (G[h_0(x)] + G[h_1(x)] + G[h_2(x)]) \bmod 3$ and G is an array built during the construction process [8: p.93].

In other words, at the end of the construction process, each key x is assigned a unique hash value $h(x)$ by selecting one of the three hash functions. This modular arithmetic ensures a deterministic and collision-free mapping of keys to their hash values.

2.5.3. Reducing Space Overhead

There is a way to reduce the space overhead of G from $2.46n$ bits to exactly $2n$ bits. This can be done by exploiting the fact that G ranges from 0 to 3, mod 3. By rewriting $G[v] = 3$ as $G[v] = 0$, we can encode the values of G using values in the range $[0, 2]$, which allows to store G as trits, reducing the space overhead to $2n$ bits [8: p.95].

The only drawback of this approach is that the space used for satellite data is $s'm$ instead of $s'n$, where s' is the number of bits needed to store each satellite value, meaning that this method is particularly useful when we do not have satellite data, or it uses a small number of bits per value.[7] [8: p.95].

2.5.4. Relevance to Trie-Based Joins

MPHF could play an important role in optimizing trie-based joins by addressing the logarithmic overhead of exponential search during the intersection phase. By potentially aligning computational costs with the size of the smaller child set and preserving the compactness of the structure, MPHF offers a promising way to enhance the efficiency of intersection operations as part of the broader CompactLTJ approach.

[6]More specifically, the intervals are $\left[0, \frac{m}{3} - 1\right]$, $\left[\frac{m}{3}, \frac{2m}{3} - 1\right]$, and $\left[\frac{2m}{3}, m - 1\right]$.

[7]According to the literature, if $s' > 2$, it is better to use the method that uses $2.46n + s'n + o(n)$ bits [8: p.95].

3. Problem Statement

CTLJ represents a significant advance in reducing the space cost associated with WCO join algorithms, by leveraging compact data structures, CLTJ achieves a remarkable reduction in space usage, decreasing the size to only 25% of traditional WCO implementations, while being 2 to 3 times faster than non-WCO systems. This reduction in space without significantly compromising query times makes CLTJ a promising option for large-scale graph databases [2].

However, the exponential search used in the trie navigation phase of CLTJ introduces a logarithmic factor into the query complexity, resulting in $\delta(Q)$[8] $\cdot O(\log n)$. To address this, we propose replacing exponential search with a hashing-based mechanism that reduces the cost to min_size $\cdot\, O(1)$, aligning computational costs with the size of the smallest set being intersected. It is not immediately clear which approach will perform better, as the trade-off depends on data characteristics: hashing may perform better in uniform datasets with smaller intersections, while the original method could be more efficient when alternation complexity is lower. This research will evaluate these trade-offs in real-world datasets while exploring whether hashing can achieve these benefits without significantly compromising the compactness of CLTJ. However, implementing hashing in a compact and practical way introduces its own set of challenges, which we address in this study.

One promising direction to achieve these goals is the use of Minimal Perfect Hash Functions (MPHF). These functions are particularly suited to static datasets, offering efficient lookups while keeping space overhead minimal. Unfortunately, it is not as simple as directly applying MPHF on node children, because it would involve creating many hash tables, some of them very small, leading to a sublinear space overhead that, while asymptotically not reported ($o(n)$), could become significant, potentially dominating the total space usage and making the solution worse than the original, negating the benefits of the approach due to excessive weight.

A potential solution to mitigate the explosion in size is to use a single hash function for all the tables. However, it is not clear if a perfect solution with the Las Vegas approach can be quickly found—this needs further analysis. Another alternative is to explore creating only a few hash functions that would suffice for all the tables, in such a way that the number of hash functions scales gracefully with the number of tables, keeping the construction time low. Yet another approach is to apply hashing exclusively for large lists, adopting a hybrid strategy.

An interesting idea is to use a single hash table per level of the trie instead of multiple small ones, significantly reducing the overhead caused by small tables. However, this introduces a new challenge: efficiently navigating the children (siblings) of a node, as this information is not directly available from the hash table, unlike in the trie representation, which naturally supports this navigation.

This research will explore and compare different methods to implement and optimize hashing in the context of CompactLTJ, focusing on balancing spatial efficiency and query performance. Additionally, we will investigate whether these performance characteristics correlate with the alternation complexity of queries, particularly for real-world data. By examining the relationship between query patterns, data distribution, and different CompactLTJ variants, we aim to provide practical insights for improving graph query processing methods.

[8] The alternation complexity of the query

4. Research Questions

- Which approach yields better performance for CompactLTJ: optimizing alternation complexity (incorporating a logarithmic factor due to the exponential search) or utilizing hashing techniques?
- How can the compactness of CompactLTJ be maintained when introducing hashing, minimizing additional space overhead? Is it feasible to avoid significant space increases when integrating hashing?
- How does data distribution affect the different types of queries in the context of CompactLTJ, particularly when comparing the hashing and original approaches? Does the alternation complexity of queries influence the performance of each method?

5. Hypothesis

Replacing exponential search with a hashing-based search mechanism in the navigation phase of CompactLTJ will improve the temporal performance of queries in graph databases without compromising spatial efficiency. Meanwhile, the alternation complexity of queries will play a significant role in determining the performance of each method, with hashing potentially offering better efficiency for certain query patterns that involve high alternation complexity.

6. Goals

6.1. General Goals

The main goal of this research is to study the effect of replacing the exponential search in the trie navigation phase of CompactLTJ with a hashing-based search mechanism using Minimal Perfect Hash Functions (MPHF). This study aims to explore whether this alternative approach can provide a viable, space-efficient option for large-scale graph databases, while retaining the worst-case optimality (WCO) benefits of CompactLTJ. Additionally, we aim to analyze how the alternation complexity of queries affects the performance of CompactLTJ with hashing compared to the original implementation.

6.2. Specific Goals

- Design an algorithm to compute the alternation complexity of the benchmark queries, enabling an in-depth analysis of query characteristics.
- Implement a tailored version of Minimal Perfect Hash Functions (MPHF) suitable for use within CompactLTJ.
- Benchmark and analyze the performance of this modified version of CompactLTJ in comparison to the original implementation, as well as other wco and non-wco join algorithms.
- Investigate the relationship between alternation complexity and the performance of both versions of CompactLTJ, identifying scenarios where hashing or exponential search performs better.

7. Methodology

The research will be carried out in several phases, aimed at implementing and evaluating the proposed enhancement in CompactLTJ, using C++ for its efficiency and ability to integrate with existing libraries.

- Perform an exhaustive review on compact data structures and joins algorithms in graph databases, LTJ and CompactLTJ.
- Study the current implementation of CompactLTJ, with special emphasis on the trie navigation phase and exponential search.
- Review practical schemes for constructing Minimal Perfect Hash Functions (MPHF), evaluating alternatives and selecting the most suitable ones for the context of CompactLTJ.
- Design a hashing-based search scheme to replace the exponential search in tries, selecting the most efficient hashing strategy in terms of time and space trade-off.
- Implement the new version of CompactLTJ with the hashing-based search mechanism.
- Select large graph datasets and design a set of queries of different patterns (acyclic, cyclic, simple, etc.).
- Benchmarking:
  - Compare the performance of the new CompactLTJ with the original version and other wco and non-wco algorithms using *WDBench*: a real-world benchmark for knowledge graphs based on Wikidata [1].
  - Measure average execution time per query.
  - Evaluate space usage (bytes per triple).
  - Compare performance across different levels of query type.
  - Calculate alternation complexity for each query.
- Compare the results obtained with popular systems such as Jena LTJ, MillDB, RDF-3X and Blazegraph, evaluating improvements in temporal and spatial efficiency.
- Write up the conclusions of the work, including analysis of results, comparisons with other approaches and suggestions for future work.

8. Expected Results

We anticipate that the proposed enhancement using a hashing-based search mechanism will provide valuable insights into the trade-offs between different strategies for join processing in CompactLTJ. Specifically, while the hashing approach might not always outperform the exponential search in every scenario, we aim to understand better which conditions favor each method. For uniform data distributions, hashing is expected to offer better efficiency, as the benefits of optimizing alternation complexity are less significant in these cases. On the other hand, for real-world, non-uniform datasets, our results should highlight practical trade-offs, showing the balance between time efficiency and maintaining alternation complexity. This analysis will help identify when one approach is more suitable than the other, contributing to the development of more effective strategies for space-efficient graph query algorithms. Additionally, we will share all implementations and benchmarking scripts as open-source on GitHub, making it easier for others to build upon this work and collaborate.

Bibliography

[1] Renzo Angles, Carlos Buil Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoč. 2022. WDBench: A Wikidata Graph Query Benchmark. In *ISWC 2022 – The Semantic Web*, 2022. Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-19433-7_41

[2] Diego Arroyuelo, Gonzalo Navarro, Daniela Campos, Carlos Rojas, Adrián Gómez-Brandón, and Domagoj Vrgoč. 2024. Space & Time Efficient Leapfrog Triejoin. In *7th Joint Workshop on Graph Data Management Experiences & Network Data Analytics (GRADES-ND '24)*, June 2024. Association for Computing Machinery. https://doi.org/10.1145/3661304.3661898

[3] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, 2008. 739–748. https://doi.org/10.1109/FOCS.2008.43

[4] Jérémy Barbay and Claire Kenyon. 2008. Alternation and Redundancy Analysis of the Intersection Problem. *ACM Transactions on Algorithms* 4, 1 (March 2008). https://doi.org/10.1145/1328911.1328915

[5] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2013. Practical Perfect Hashing in Nearly Optimal Space. *Information Systems* 38, 1 (March 2013), 108–131. https://doi.org/10.1016/j.is.2012.06.002

[6] Martin Grohe and Dániel Marx. 2014. Constraint Solving via Fractional Edge Covers. *ACM Trans. Algorithms* 11, 1 (August 2014). https://doi.org/10.1145/2636918

[7] Guy Jacobson. 1989. Space-efficient Static Trees and Graphs. In *30th Annual Symposium on Foundations of Computer Science (FOCS)*, 1989. IEEE. https://doi.org/10.1109/FOCS.1989.74538

[8] Gonzalo Navarro. 2016. *Compact Data Structures: A Practical Approach.* Cambridge University Press. Retrieved from https://doi.org/10.1017/CBO9781316588284

[9] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case Optimal Join Algorithms. Retrieved from https://arxiv.org/abs/1203.1952

[10] Hung Q. Ngo, Christopher Re, and Atri Rudra. 2013. Skew Strikes Back: New Developments in the Theory of Join Algorithms. Retrieved from https://arxiv.org/abs/1310.3314

[11] Todd L. Veldhuizen. 2013. Leapfrog Triejoin: a worst-case optimal join algorithm. Retrieved from https://arxiv.org/abs/1210.0481