SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics: Data Engineering And Analytics

Implementing External Ternary Joins

Ahmed Diab

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics: Data Engineering And Analytics

Implementing External Ternary Joins

Author:	Ahmed Diab
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Altan Birler, M.Sc.
Submission Date:	14.04.2025

I confirm that this master's thesis in informatics: data engineering and analytics is my own work and I have documented all sources and material used.

Munich, 14.04.2025	Ahmed Diab
Location, Submission Date	Author

Abstract

Efficient processing of multi-way join queries is a critical challenge in both relational and graph databases, especially as data scales beyond main memory.This thesis investigates the evaluation of *triangle queries* — three-way cyclic joins that are prevalent in graph analytical databases —under constraints of limited memory. Traditional query engines decompose a triangle query into binary joins, which can generate enormous intermediate results and incur excessive I/O overhead. To overcome these limitations, we design and implement a **Grace Ternary Join** algorithm within a push-based execution model for graph analytical database. Our approach extends the Grace hash join paradigm to handle three input relations simultaneously, thus materializing triangle results without storing large intermediate relations. We evaluate the proposed system on real-world graph datasets, comparing its performance against DuckDB [1] (analytical database executing the same triangle query as two binary joins) and a traditional Grace-based binary hash join baseline [2]. The results demonstrate that our Grace Ternary Join can significantly reduce execution time in scenarios with heavy intermediate result blow-up: For instance, the proposed approach demonstrates a speedup of up to 1.8× when processing the Twitter dataset, particularly when the Intermediate-Data-to-Input-Size Ratio exceeds 200x in comparison to DuckDB [1]. This enhancement is particularly significant in the context of large social network graphs, where traditional binary plans would typically result in the materialization of substantial intermediate results. Under varying memory constraints, we analyze the trade-offs between the ternary and binary approaches. The findings highlight that while the ternary join excels in mitigating intermediate data explosion, it incurs overhead from recursive partitioning when memory is extremely scarce. Overall, this work advances the state of the art by introducing a novel external memory three-way join operator for triangle queries, providing insights into its benefits and limitations. Key contributions include the new push-based join execution model of Ternary Operator, a comprehensive implementation of an external Grace Ternary Join, and an in-depth experimental study that charts the strengths and weaknesses of 3-way versus binary join processing for triangle queries.

Contents

1 Introduction

As the amount of interconnected data grows in areas like social networks and knowledge bases, there is an increasing need for systems that can quickly and efficiently process queries [3]. One key analytical operation these systems perform is join query, which essentially means linking data from different sources by matching up common attributes. In graph analytical database, many important queries can be expressed as multi-way joins. For instance, a *triangle query* finds all triples of entities that form a triangle (three pairwise-related nodes) in a graph, and can be formulated as a three-way self-join in relational database terminology. Such triangle computations are central to analytics like discovering communities or computing clustering coefficients. However, processing a triangle query at scale is challenging: conventional relational database systems break the 3-way join into a sequence of binary joins, which often leads to materializing very large intermediate results [4]. When joining three relations $R(a, b)$, $S(b, c)$, and $T(a, c)$ to enumerate triangles, a binary-plan will first join two of them (say $R \bowtie S$) producing an intermediate relation that could be orders-of-magnitude larger than the initial inputs. This intermediate then must be joined with the third relation, incurring heavy computation. The explosion of intermediate data not only wastes memory and storage but also degrades performance, especially if the data exceeds main memory and must be spilled to disk.

In this thesis, we tackle the above problem by developing a specialized **push-based** query execution engine with a focus on an efficient ternary join operator for triangle queries. Modern analytical databases increasingly employ push-based (produce-consume) execution models in which operators directly propagate result tuples to parent operators, in contrast to the classic pull-based Volcano iterator model [5]. The push-based design minimizes call overhead and enhances CPU cache locality. Although our proposed approach does not incorporate pipeline parallelism at this stage, the architecture has been thoughtfully designed to facilitate its integration in future research endeavors. Building on this paradigm, we introduce a novel **Grace Ternary Join** strategy that extends the external Grace hash-based binary join algorithm [2] to three relations. Our approach partitions the data from all three input tables on appropriate join attributes so that matching tuples from R, S, and T end up in corresponding partitions. By doing so, an entire triangle can be discovered within a single partition, avoiding the need to ever materialize a full join of any two relations across the whole dataset. The join operator executes in two phases (analogous to build and probe): it first *builds* hash-based partitions for the input distribution among all buckets, then *probes* these partitions to find matching triplets forming triangles. Crucially, if a partition still cannot fit in memory (due to data skew or extremely limited memory), the operator applies recursive partitioning to that partition - a recursive invocation of the Grace Ternary Join on that subset of the data. This ensures the algorithm can handle very large datasets (out-of-core execution) by breaking the problem into manageable chunks.

We implemented the proposed execution model and join operator in C++ as a prototype query engine for computing the number of triangles. The engine manages data in a block-oriented manner (pages of size 16KB each) to optimize I/O and cache performance.

This thesis contributions can be summarized as follows:

- We propose a novel *Grace Ternary Join* algorithm for efficient triangle query execution. To our knowledge, this is one of the first external-memory hash join approaches specifically tailored to a three-way cyclic join, extending the classical Grace hash-based binary join to multiple inputs. This operator avoids the materialization of large intermediate join results by processing three relations in one unified operation.

- We use the push-based query pipeline execution to support the new ternary join operator.

- We conduct a comprehensive experimental evaluation on 5 real-world graph datasets. We compare our ternary join against (i) a state-of-the-art in-memory analytical database (DuckDB [1]) as single threaded query processing for executing the same 3-way join query using its default binary join plan, and (ii) a baseline external Grace binary hash join [2] that computes the triangle query via two consecutive binary joins. The evaluation examines performance under varying dataset sizes and memory conditions.

- We provide an in-depth analysis of the results, demonstrating that our approach can outperform traditional strategies in many cases. In particular, on datasets where the binary join plan would produce extremely large intermediate outputs, our ternary join runs significantly faster by eliminating most of the I/O associated with writing and reading those intermediates. We also analyze scenarios where our approach incurs overhead (e.g., due to recursive partitioning under very tight memory or skew), thereby highlighting its limitations and the conditions under which binary joins might remain preferable.

This thesis is organized as follows: Chapter 2 provides the background on graph databases and join algorithms, formally defining join operations and discussing how graph queries like triangle finding relate to relational joins. Chapter 3 surveys related work, including classic external binary hash joins (Grace, Hybrid, Adaptive) and modern multi-way join approaches, to position our work in the context of existing research. Chapter 4 presents the design of our push-based execution model and introduces the architecture of the ternary join operator, explaining how it processes input data. Chapter 5 details important implementation aspects of our join strategy, such as the structure for managing partitions (buckets) and the mechanisms for recursive partitioning. Chapter 6 then empirically evaluates the performance of the proposed approach. We describe the experimental setup, datasets (including social network graphs of varying sizes), and benchmark results comparing the Grace Ternary Join with DuckDB [1] and the Grace binary join [2] under different memory configurations. Finally, Chapter 7 concludes the thesis, summarizing our findings, discussing the limitations of the current approach.

2 Background

This chapter introduces the basic ideas behind database join operations and the systems that support them. We start by looking at graph databases, where data is stored as nodes connected by edges. In these systems, you can directly follow the links between pieces of data instead of relying on complex join operators like those used in relational databases. This approach often makes handling multi-hop queries more efficient, although for straightforward queries, traditional relational databases might still be a better fit.

After looking at graph databases, we then moved on to explore how joins work in relational systems. We start with the simplest type—the binary join that merges two tables—and gradually move on to more advanced forms like multi-way and ternary joins. Along the way, we'll explain how these operations work, discuss the computational challenges they present, and highlight why managing the size of intermediate results is so important. Along the way, we'll also touch on worst-case optimal join algorithms and dive into key concepts like the AGM bound and fractional edge covers, which help us understand the efficiency limits of join processing.

Overall, this chapter aims to build a solid foundation in join operations, preparing for more advanced discussions on optimizing joins and improving performance in both traditional relational databases and modern graph-based systems.

2.1 Graph Databases

Graph databases represent the direct relations data as nodes and edges. The join operation in graph databases is different from the SQL join operation in relational databases; The graph databases fetch corresponding data by traversing edges (directed or undirected) between nodes. In other words, the joining queries follow the relationship (edges) in the graph (e.g., find a person's followers and the followers of the followers on X), which can jump from one node to the direct node that exists in the edge list. In contrast, the join query can be obtained for relational databases by joining the multiple edge lists (tables) using multiple join operations, which can add extra overhead for scan indices or tables for matching the keys.

For example, we can consider there is a query for followers-of-followers on X. We can start the path at a specific user node, traverse their list of followers (edges) to find direct followers, and subsequently traverse to another followers edge to find followers-of-followers. In a relational database mode, if followers are represented as a table Followers(user1, user2), fetching followers-of-followers may be essential to join Followers table: one for finding the user's followers, and the additional to fetch the corresponding followers' followers. The more complex queries that may require various joins can be retrieved by graph traversals, which gives graph databases better performance. However, there is no significant difference for simple queries as relational databases can still be more efficient, and the advantage of graph traversal may increase the cost [6].

2.2 Overview of Joins

In a relational database, join queries are fundamental operations that combine two or more tables based on a specific key or condition. For example, consider a table named **Student** with the columns **student_id**, **name**, and **semester**, and another table called **Enrollment** with the columns **student_id** and **course_id**. A join on the **student_id** column allows us to associate each student with their corresponding course. Without the join operator, we would need to denormalize the data into a single table, which would contain redundant information and be less scalable. Therefore, join operators play a crucial role in efficiently linking normalized data.

Formally, the join of two relations R and S on a certain predicate ρ . The relational join can be represented in relational algebra as follows:

$$R \bowtie_{\rho} S = \{r \circ s \mid r \in R,\ s \in S,\ \rho(r,s)\}.$$

The concatenated tuples $r \circ s$ such that r belongs to set R and s belongs to set S which satisfy condition ρ [6]. The most common example is the natural join where ρ filters the equality for one or more columns that exist in both tables, R and S. The solution of the join operator is a new table with columns of R and S. All materialized rows should satisfy the predicate ρ, which contributes to the result. If there is no match, the row may be materialized with null values for the outer join. However, inner joins ignore non-matching rows.

For illustration, assume we have two relations R and S.

Database Table R:

a	b
1	2
2	3
4	5

Database Table S:

b	c
2	3
4	5
6	7

Joined Table $R \bowtie_{\rho} S$**:**

a	b	b	c
1	2	2	3

If we apply the inner join for both tables on b column, we select all rows where r.b = s.b. Based on the previous example, The first row (a = 1, b = 2), in table R, is joined with the first row (b = 2, c = 3), in table S. This was a simple example for illustrating how the join operator works on two tables based on a predicate of the equivalent columns.

Computational Complexity : The join operator is considered one of the most expensive operators for query execution in relational databases due to the extensive intermediate results produced by binary operations. Assume table R has n tuples and table S has m tuples; the traditional nested loop join will result in quadratic complexity., since it compares every tuple in R with every tuple in S which leads to time complexity $O(n \cdot m)$. The quadratic complexity is not a practical solution for large tables; if we join two tables with a million records each, the number of comparisons will reach 10^{12}. comparisons. In real database engines, query processors use efficient join algorithms to reduce the number of comparisons between rows and eliminate the non-matching rows. The fundamental join algorithms are nested-loop join, hash join, and sort-merge join. The main differences between join algorithms can be differentiated by their performance:

1. Nested Loop Join: Assume we have two tables R and S, each row in R scans all rows in S in order to find the matching tuple with the corresponding predicate. The time complexity is quadratic by order $O(n \cdot m)$ [6].

2. Sort Merge Join: For table R and S, the tuples should be sorted first by the join keys. Afterward, there should be a linear iteration to merge the matching tuples between the sorted tables. The time complexity of sorting for both tables is by order $O(n \cdot log(n) + m \cdot log(m))$, and the merging process might be $O(n + m)$, which leads to the final asymptotic complexity with order $O(n \cdot log(n) + m \cdot log(m))$ [6].

3. Hash Join: Assume you have two tables and you need to match rows where the keys are the same. With a hash join, one table (say, table R) is first converted into a fast lookup structure—a hash table—using its join key. Building this hash table takes time roughly proportional to the cardinality R. Then, the other table (S) is scanned, and for each row, the algorithm checks the hash table to find matching rows from R. Overall, the work done is proportional to the sum of the rows in both tables. This method works best when one of the tables is small enough to fit into memory, making the process both efficient and practical [6].

We ignored the complexity of output size in the asymptotic complexity of the three join algorithms. In terms of output size, the cardinality can be so small if it has a small selectivity ratio or a huge result size with a selectivity ratio equal to one, which is equivalent to the Cartesian product of two relations. Therefore, the worst-case of joining two tables based on output results will always be quadratic with $O(n \cdot m)$. The improvement of join algorithms is crucial for eliminating non-matching results and reducing intermediate results, which is considered an overhead performance. The main objective of the query optimizer is to figure out the optimal (or suboptimal) join order to reduce intermediate results. The complexity class for finding optimal join order is NP-Hard, which grows exponentially based on the number of tables. Most of the modern database engines use dynamic programming such as DPccp [7] and DPhyp [8] to find the optimal plan. In a nutshell, modern database engines adjust the implementation of the join operation, which approaches linear execution time in practice based on the input size. The challenges occur in the presence of multiple joins or for extreme data size.

2.3 Binary Join vs. Ternary Join

The binary join operator performs the fundamental join operation between two tables, as described in Section 2.2. Formally, the binary join operator is defined as joining two input operators (which can be either relations or intermediate results) and producing the matching tuples into one output operator. Since database queries often involve joining multiple tables, join operators are considered associative. This means that the result of joining tables can be grouped in different ways: $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$. Thus, the database engine can join multiple relations simultaneously. The ternary join operation, on the other hand, specifically joins three tables as part of a singular operation. For example, suppose we have three relations: $R(A, B)$, $S(B, C)$, and $T(C, D)$. The ternary join operation can be executed by first joining tuples of R and S based on column B, and then joining tuples of S and T based on column C. In practice, the steps to join three tables can be outlined as follows: first, compute $I = R \bowtie S$, which serves as an intermediate result for the first two tables. Next, perform $I \bowtie T$ to materialize all matching tuples. The final result will include all tuples (r, s, t) such that $r \in R$, $s \in S$, and $t \in T$, in addition to equality conditions on the matching columns, specifically $r.b = s.b$ and $s.c = t.c$.

Using a ternary join operation with two binary join operators can be inefficient if the intermediate result generates a large number of tuples, especially when the subsequent binary operator produces only a few matches concerning table T. In such cases, the query processing becomes more complex due to the need to materialize and scan the numerous tuples from the intermediate result. A multi-way join can help reduce the overhead associated with materializing the intermediate results by allowing the tuples to be materialized only once. Theoretically, multi-way joins can be evaluated more efficiently than a series of binary joins. Worst-case optimal join algorithms, such as the Generic Join, join multiple relations concurrently based on their join attributes. These optimal join algorithms typically have a time complexity that is proportional to the output size, in addition to the time taken for scanning the input. Unlike some less efficient algorithms, they do not reach quadratic time complexity. In practice, the query optimizer utilizes binary joins for 3-way joins that can be executed in a specific sequential order. In graph databases, such as those using SPARQL [9], performing a single join operation on three tables is advantageous as it helps to avoid the overhead costs associated with joining the first two relations separately..

2.4 Worst-Case Optimal Joins (WCOJs) and AGM Bound

2.4.1 Join Size in Worst Case

Relational joins can materialize output tuples whose size is huge in the worst case, even if the table size is moderate. For traditional query processing, multi-joins are executed as a sequence of binary joins, which may materialize intermediate results with large sizes. The traditional example is a triangle join query, which joins three relations, $R(a, b)$, $S(b, c)$, and $T(a, c)$, joined based on their common attributes. If each relation has an equal number of tuples with size N, the worst-case of materialized tuples in the triangle query is bounded by order of $O(N^{\frac{3}{2}})$ [10]. Unfortunately, all join ordering of binary join plan for triangle query can be forced to materialize an intermediate join with an order of $O(N^2)$ in worst-case complexity even though the number of matching tuples at the end has only $O(N^{\frac{3}{2}})$ in worst-case. This gap gives the motivation to work on more research about multi-way join algorithms that can join all relations simultaneously to materialize the triangle query results in $O(N^{\frac{3}{2}})$ time, neglecting the overhead computation in intermediate results of binary operators with order $O(N^2)$.

2.4.2 Join Queries as Hypergraphs and Edge Covers

In order to analyze the output size, it is helpful to model the join query as a hypergraph. The natural join of a sequence of relations $R_1, R_2, \ldots, R_n$ by hypergraph $H = (V, E)$ where V represents a set of vertices that correspond to an attribute of a relation R_i in the query, and E represents a set of hyperedges; each hyperedge E_i corresponds relation R_i that link between attributes in R_i. For example, $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}, \{a, c\}\}$ as $H = (V, E)$ is hypergraph of triangle query for relations R, S and T. In this hypergraph, all sets of edges in E cover all vertices V. More formally, the edge cover constraint can be represented as a union of all edges, which is equivalent to a set of vertices V: $E_1 \cup E_2 \cup \cdots \cup E_m = V$. The edge cover yields a bounded matching tuple size. Intuitively, for any group of relations, the total number of materialized tuples is bounded by the product of all relations's sizes. For example, $\{R, S\}$ is an edge cover in triangle query, where attributes $\{a, b\}$ from relation R and $\{b, c\}$ from relation S as both sets cover all edges $\{a, b, c\}$ $|R \bowtie S \bowtie T| \leq |R \bowtie S| \leq |R| \cdot |S| = N^2$. Equivalently, $\{R, T\}$ and $\{S, T\}$ also cover all attributes, are bounded by N^2. The minimum size of edges (total number of tuples) yields the tight bounded size using integral covers. It overestimates the output size for triangle queries by order of $O(N^{\frac{3}{2}})$

2.4.3 Fractional Edge Covers and AGM bound

Atserias, Grohe, and Marx (AGM) [11] observed that allowing fractional (real-valued) edge covers that are strictly bounded to a tight join size. A fractional edge cover is a representation of non-negative weight $x_{R_i} \geq 0$ for each edge (relation R_i) such that $a_j \in V$, the summation of all edges' weight covering attribute a_j with at least 1. More formally, $\sum_{R_i : a_j \in \mathrm{attrs}(R_i)} x_{R_i} \geq 1$ for all attributes a_j. The fractional cover number for hypergraph H, $\rho^*(H)$, which is the minimum total weight $\sum_{i=1}^{m} x_{R_i}$ for any fractional edge cover of the hypergraph H. For any fractional cover x_{R_i}, the number of matching tuples is bounded by relation sizes powered by corresponding weights. More formally, $R = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$ is joined query of all relations and fractional edge cover with weights $(x_{R_1}, \ldots, x_{R_n})$, the total number of joins is bounded as follow: $|R| \leq \prod_{i=1}^{n} |R_i|^{x_{R_i}}$.. This AGM inequality for generalized multi-way joins for cover-based bounds. In practice, finding the optimal weight of fractional cover with weight x_{R_i} that minimizes the upper bound $\prod_i |R_i|^{x_{R_i}}$; the summation of all weights is $\rho^*(H)$. For instance, the hypergraph of triangle query has weight fractional cover $x_R = x_S = x_T = 1/2$, since each relation contributing half of covering one of two attributes as all attributes a, b, c were covered with total weight equal to $\frac{1}{2} + \frac{1}{2} = 1$. The total weight of all relations $0.5 + 0.5 + 0.5 = 1.5$; therefore AGM bound for triangle query join $|R \bowtie S \bowtie T| \leq |R|^{\frac{1}{2}}|S|^{\frac{1}{2}}|T|^{\frac{1}{2}} = N^{\frac{3}{2}}$. In conclusion, AGM bound showed the final result of the worst-case output size for any conjunction query is formalized by the fractional edge cover of its hypergraph.

2.4.4 Linear Programming Formulation and Duality

Linear programming can be effectively used to tackle fractional edge cover problems in order to determine optimal tight bounds. Consider a set of relations $R_1, R_2, \ldots, R_n$, where $|R_i|$ denotes the size of relation R_i. Our goal is to find a fractional edge cover $\{x_{R_i}\}$ that minimizes the weighted sum

$$\sum_{i=1}^{n} x_{R_i} \log |R_i|$$

subject to coverage constraints. More formally, the problem can be formulated as the following primal linear program:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} x_{R_i} \log |R_i| \\ \text{subject to} \quad & \sum_{i:\, a \in \text{attrs}(R_i)} x_{R_i} \geq 1, \quad \forall \,\text{attribute } a, \\ & x_{R_i} \geq 0, \quad i = 1, 2, \ldots, n. \end{aligned}$$

This formulation minimizes the sum $\sum_{i=1}^{n} x_{R_i} \log |R_i|$, which is equivalent to minimizing

$$\log \prod_{i=1}^{n} |R_i|^{x_{R_i}},$$

by the properties of logarithms. The use of the logarithmic function is justified by its monotonicity, ensuring that minimizing the logarithm of a product is equivalent to minimizing the product $\prod_{i=1}^{n} |R_i|^{x_{R_i}}$ itself.

The dual LP problem covers the integrative perspective by introducing the dual variable y_a for each attribute a; the dual LP is : maximize $\sum\limits_{a \in V} y_a$ subject to $\sum\limits_{a \in \text{attrs}(R_i)} y_a \leq \log |R_i|$ for each relation R_i, and $y_a \geq 0$ for each attribute a. Intuitively, each attributes a corresponds to weight y_a, which represents the attribute's contribution in the join result size; in addition, each relation R_i the sum of weights of the relevant attributes cannot exceed $\log |R_i|$

The purpose of maximizing $\sum_a y_a$ yields to maximizing $\log |R|$ (The logarithm of total join size of $R = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$). LP duality gives an assurance that at optimum, the summation of total dual weight $\sum_a y_a$ equals the total primal cost $\sum_i x_{R_i} \log |R_i|$. In fact, $2^{(\sum_a y_a)} = \prod_i |R_i|^{x_{R_i}} \leq 2^{log|R|} = |R|$, so the dual y_a values provide the same bound in an insightful way.

2.4.5 Theoretical Significance and Implications for Graph Databases

The AGM bound is a fundamental theory used in worst-case scenario analysis, determining the maximum number of materialized tuples in join queries based on the given input relations. Sequential binary join plans are not necessarily optimal for certain types of queries, as discussed in Section 2.4.1 concerning the triangle query. The theoretical aspects of the AGM results not only establish the upper limit for the maximum number of join sizes but also provide a framework for designing new algorithms that achieve this upper bound. Recent research into worst-case optimal join algorithms—such as the NPRR and Leapfrog-Triejoin algorithms—has achieved runtimes that are proportional to the AGM bound for all hypergraph queries. These algorithms avoid materializing large intermediate results by exploring the matching tuples across all relations simultaneously.

In graph databases, queries often require joining multiple triples or even several edge relations simultaneously, as seen in triangle queries. The worst-case analysis shows that graph datasets can be highly connected or skewed, leading to the possibility that a sequential binary join plan may produce large intermediate results. The AGM bound helps to analyze the worst-case number of matches that can be generated, highlighting the importance of multi-way join approaches. For instance, in triangle queries, the maximum number of matches in the worst case is $O(n^{1.5})$, where n represents the number of edges (or tuples in the relation). Worst-case optimal algorithms can achieve this bounded-output runtime. In contrast, sequential binary joins could result in a runtime of $O(n^2)$ due to the presence of intermediate paths in the worst-case scenario. Therefore, implementing an external ternary join for a graph database is expected to deliver significantly better performance in the worst-case situation.

3 Related Work

Join processing is one of the most critical operations in database systems, especially as data volumes continue to expand beyond the capacity of main memory. Over the years, researchers have proposed a diverse array of algorithms to address the inherent challenges of scalability and I/O efficiency in join operations. In this chapter, we review several foundational techniques and recent advances in join algorithms that seek to mitigate the limitations imposed by memory constraints.

We begin our discussion with external binary join algorithms, including Grace, Hybrid, and Adaptive Hash Joins. These methods extend the classic in-memory hash join by incorporating data partitioning strategies that allow efficient processing even when the input data significantly exceeds main memory. Each algorithm addresses specific challenges such as I/O overhead, data skew, and recursive partitioning, which are critical for achieving acceptable performance levels under constrained memory conditions.

Subsequently, we shift our focus to multi-way join techniques, with particular attention to worst-case optimal algorithms like Leapfrog Triejoin (LFTJ). These algorithms leverage trie-based indexing and synchronized traversal techniques to efficiently evaluate join queries—especially in complex scenarios such as graph and RDF data processing. The discussion will also touch upon the improvements and limitations associated with multi-way join approaches, including the trade-offs between index overhead and query performance.

Overall, this chapter synthesizes past and current research to provide a comprehensive view of how join processing algorithms have evolved. By understanding the strengths and weaknesses of these methods, we can better appreciate the design decisions underlying modern database systems.

3.1 Grace, Adaptive, and Hybrid Binary Join

Increasing the scale of join processing efficiently is considered an open challenge in database systems, especially when the data size exceeds the main memory. The external binary join algorithms, Grace hash join, Hybrid hash join, and Adaptive hash join all expand the in-memory hash joins to handle the constraint of main memory size. In the next sections, we will provide more details about different external binary join algorithms with formal definitions, asymptotic complexity, and comparisons for their efficiency and usage in modern database systems.

3.1.1 Grace Hash Join

Grace hash join [2] is an external hash join that partitions input tuples, so each partition should fit in the main memory. The algorithm consists of two phases. The first phase (Input Partitioning/Build): It scans all relations from both tables, R and S, which apply a hash function on the join attribute in order to distribute all input relations into B buckets on disk. Suppose a certain partition overflows the main memory size. In that case, we have to recursively re-partition all tuples inside the bucket with a different hash function h', which divides them into different partitions until they fit the main memory size. The second phase (Probe phase): Each partition with index i, the two corresponding partitions, R_i and S_i, fetched from disk to build all tuples in a hashtable with the smaller partition size. Afterward, the partition with a larger size of the tuple should be probed from the hashtable in order to materialize the matching tuples. Since the smaller partition can fit in memory, the probing can be processed without I/O operations. However, if the partition has a large skew on hashing the join attribute, we must apply grace join recursively on both partitions before probing.

Computational Complexity : The Grace binary join has an overhead I/O cost $\approx 3(|R|_p + |S|_p)$ in the first level of partitioning ($|.|_p$ is the number of pages of a relation). $2.(|R|_p.|S|_p)$ is considered for writing both relations into partitions and $(|R|_p.|S|_p)$ for reading the pages of relations in probe phase. The time complexity of Grace binary join is $(|R| + |S|)$ ($|.|$ is the input size), the hashing cost is ignored, and we assume that all hash function distributions are uniformly distributed. If a partition exceeds the memory size threshold, the I/O operations will grow linearly with additional levels of partitioning. The join algorithm is considered efficient when memory size is constrained compared to input size. Therefore, it can overcome the quadratic growth of time complexity for nested loops from hashing. Modern database engines use grace partitioning for handling equi-joins, which cannot fit the main memory as it makes many sequential I/O and reduces the random I/O operations in each partition.

Unfortunately, the Grace binary join algorithm has an overhead computation, especially in partitioning. Each tuple is written and read to the disk, which increases the number of I/O operations; the CPU cache misses can also reduce the efficiency of execution time. Suppose multiple partitions with corresponding data can fit the main memory. In that case, the join algorithm may execute inessential I/O operations by partitioning the tuples in the disk, which can be stored in the main memory directly. A new external join algorithm will be introduced in Section 3.1.2 to handle more portion of tuples in memory and reduce the execution time of the join process.

3.1.2 Hybrid Hash Join

Hybrid hash join [12] is a join algorithm combining the Grace hash partitioning and the in-memory hash join to reduce I/O operations. The main logic behind hybrid join is to utilize most of the main memory, and the remaining memory size can be used as a buffer memory to partition the remaining tuples. The algorithm partition tuples of relation R to B buckets; however, we can keep the first r buckets as resident partitions in the main memory instead of storing them in the disk as in the Grace join algorithm. The remaining $B - r$ buckets of relation R should be stored in disk since no memory is available for the remaining tuples. The same logic will be applied for relation S; before partitioning the tuple, we have to check if the matching tuples are in one of the first r buckets, which can be probed directly in the in-memory hashtable; the other tuples will be partitioned in the remaining buckets. After finalizing the partitioning phase, the remaining partitions ($i = B - r, B - r + 1, ..., B - 1$) will proceed to find the matching tuples. the partition R_i will be loaded to in-memory hashtable, and bucket S_i will probe R_i as in Grace join algorithm.

Since the first r buckets are never stored or fetched from disk, the hybrid hash join algorithm reduces the I/O operations compared to the Grace join algorithm. In the best case scenario, if the relation with the smaller input size ($min|R|, |S|$) can fit in memory to be stored to in-memory hashtable, which has similar performance of in-memory hash join algorithm. However, the hybrid join will be identical to the Grace join if the memory constraint is limited with a small size compared to the input size in the worst-case scenario.

The most challenging part of the hybrid join algorithm is the available memory size to allocate/store tuples into in-memory or disk-buffer partitions. Suppose the first r partitions allocate large memory. In that case, the remaining $B - r$ partitions may have many tuples that don't fit in memory, leading to recursive partitioning with extra passes. In contrast, the first r partitions can allocate a small memory size; thereby, I/O operations can be increased, which makes the hybrid join algorithm equivalent to the Grace join algorithm. In real database systems, the query optimizer evaluates the number of partitions and the threshold of memory size based on cardinality estimation on relation/input sizes. The asymptotic time complexity is considered the same as for Grace join by order $O(|R|.|S|)$, and I/O operations cost $2.(|R|_p + |S|_p)$ + I/O in probe phase (probing the last $B - r$ partitions that were stored in disk).

The main disadvantage of hybrid join is memory sensitivity. The main usage of main memory in hashtable in the build phase on first r buckets, if the tuples distribution is skewed in one partition with large data size (e.g. due to the existence of duplicates), the join algorithm may need to apply extra pass of partitioning or may lead to infinite recursive partitioning in case the partition will never fit in memory in presence of a huge number of tuples with duplicates in the same bucket. In real database engines, hybrid hash join is not commonly used, making it difficult to handle data skewing. In order to handle this kind of limitations, another approach was introduced named Adaptive Hash Join, which will be discussed in Section 3.1.3

3.1.3 Adaptive Hash Join

Adaptive join algorithm adapts the algorithm's behavior during execution time instead of allocating available memory size in advance by an estimator. As mentioned in the research paper [13], the adaptive join initially assumes the optimistic scenario. The smaller relation can be distributed into B buckets; all buckets fit in memory. During the build phase, if the memory threshold is reached (overflows), the algorithm will adjust some buckets to write them on the disk. The algorithm adjusts the input size and memory during runtime.

The general idea behind adaptive join is to divide all buckets into clusters that can share the buffer space. The buffer space can be split dynamically if it exceeds the memory threshold. Initially, a lot of buckets can be allocated with more sharing buffers to handle data distribution skew. During the building process, new buffer space will be allocated if the buffer overflows and we have available memory; if that is not the case, the whole cluster will be swapped out to the bucket file (on disk), which leads to freeing memory. The process will proceed until all build tuples are distributed among all clusters. If memory is very low, the algorithm behaves as Grace or Hybrid join with more recursive passes when the bucket overflows; however, the adaptive algorithm adjusts and reverts the strategy to a nested join strategy in the presence of highly skewed distribution (e.g., existence of duplicates). More specifically, the algorithm may change to block nested loop join in the presence of a buffer space, which improves performance in the worst-case scenario.

The adaptive join algorithm ensures the asymptotic time complexity in linear time, even in the presence of tiny memory size (at least by having a few buffers), in addition to handling data distribution skews. In a nutshell, the adaptive join algorithm is more robust with different data distributions, which improves performance stability that ensures the approximate optimal I/O usages based on corresponding memory constraints and uses Grace/Hybrid join with assumptions of data distribution and memory conditions.

3.2 Leapfrog Triejoin and Multi-way Joins

The joining process has recently been considered a long research challenge in terms of scalability and efficiency; the recent topics focus more on worst-case optimal joins. Atserias, Grohe, and Marx's AGM bound [14] published a tight upper bound for the matching join tuples. Ngo et al. [10] published the first worst-case optimal join algorithm (NPRR) that managed to achieve a running time that is proportional to AGM bound. This was the state-of-the-art multi-way join, and how to efficiently handle ternary joins (3-way join) as it handles cyclic queries (e.g. triangles) in execution time bounded by the total number of matching tuples. Unfortunately, NPRR's complexity made it difficult to apply in practice; therefore, Leapfrog Triejoin (LFTJ) [15] which offers simpler and practical worst-case optimal join algorithm. It organizes all relations by a multi-level trie index and then synchronizes leaps by merging the intersections of sorted lists. It finds the matching for one variable at a time and jumps to the next match in each relation's trie node until we reach all satisfied constraints. It was proved that LFTJ is worst-case optimal with logarithmic factors as for NPRR; in addition, it may outperform NPRR in other cases. LFTJ runs in order by $O(n.log(n))$ while NPRR requires $\Theta(n^{1.375})$. The main strength of LFTJ is its simplicity and effectiveness in practice, as it can be implemented with an indexed data structure (e.g., B+ trees).

3.2.1 Extensions and Applications in Graph Databases

For graph databases and RDF data, multi-way join algorithms were considered the main advantage to theoretically neglect intermediate results and reach optimal runtimes. This section discusses the Leapfrog Triejoin algorithm, the state-of-the-art multi-way join, and how to efficiently handle ternary joins (3-way joins). For the Semantic Web domain, new research by Hogan et al. [16] was published to adapt LFTJ to evaluate SPARQL graph patterns and multi-join queries on RDF triples. Their new strategy/method is integrated into Apache Jena, which speeds up complex SPARQL benchmarks. This indicates the outcomes of evaluating multi-joins in a single operation rather than traditional sequential binary joins. Other systems as EmptyHeaded [17] supported worst-case optimal joins with handling intersections using bitset indexes that combined with generalized hypertree decomposition (GHD) of input query Q. Basically, GHD is a joined tree of query Q, and each node inside the hypertree is a sub-query of Q. This approach accelerates subgraph pattern matching in-memory.

3.2.2 Index Structures with Limitations and Improvements

Although LFTJ and other worst-case optimal joins are trending approaches for graph databases, they require indexing for tuples in tries on all join attributes. In ternary join queries, all possible binding (permutation of join attributes) will be up to 6 indexes (3! for ternary joins) in order to find the optimal, efficient path to find the number of intersections to filter the matching tuples. LFTJ can be utilized by sorted indexes as B+-tree to traverse each relation in a single step; The implementation of LFTJ with B+-tree was implemented on Apache Jena for all six triple orders that require ordering of all join attributes and lookup for the next matching tuple in $O(logn)$ [16]. However, six sorted indexes require an extra space cost, which increases the usage of memory size for large graphs as it can increase the storage usage for roughly 14 times the raw data size [18]. The overhead memory space of LFTJ's trie indexes limits the efficiency of the multi-way join algorithm for large-scale graphs. Recent work by Arroyuelo et al. [18] improves the overhead space by more compact indexes representing one bit per trie edge instead of using a pointer (succinct bit-level pointers). It reduces space for about 25% of classical leapfrog index and 45%–65% of non-WCO indexes. The compressed trie approach decreases the index size and fetches matching tuples faster than all non-WCO systems. Another research [17] was proposed the ring indexing scheme that neglects the idea of storing multiple orders and uses instead one compact index with almost no extra space by simulating the needed traversal in $O(logn)$. LFTJ will run and always be bounded proportionately to the number of join results. However, poor ordering can lead to unnecessary work, which is considered an open problem in finding the optimal index ordering. In a nutshell, the database research organizations have proposed and mentioned the limitation of LFTJ in practice for index overhead and finding optimal indexing ordering to make WCOJs more feasible for graph databases.

4 Push-Based Query Execution model

Efficient query processing is important for graph database systems, which can handle complex multi-join queries. Currently, we address external ternary joins for materializing triangle queries. Such tuples represent the edge for directed or undirected graphs [19].

To achieve high performance, our system should adopt the produce-consume execution model (push-based model) [5], where operators push tuples through pipelines instead of passively pulling the data as in the iterator model (volcano model).

Each tuple in the graph database is represented by two fixed-size integers, each 8 bytes long. The data tuples are organized into a set of fixed-size pages, each measuring 16 KB. This configuration allows for the storage of up to 1024 tuples for base relations. Each page is arranged as a flattened array of 64-bit integers (`uint64_t`), as illustrated in Figure 4.2. The total length of the array is equal to the width of the tuple multiplied by the number of tuples. For base relations, the initial tuple width is set to 2, and to accommodate the 16 KB page size, the number of tuples must be less than or equal to 1024. This layout is designed to support efficient sequential I/O access from disk and to facilitate cache-friendly data retrieval.

This chapter illustrate the push-based query execution plan [5] that we used for our implementation in Chapter 5.

```cpp
struct Tuple {
   std::vector<uint64_t> values;
   Tuple(size_t tupleWidth)
        : values(tupleWidth) {}
};
```

Figure 4.1: Tuple representation in graph database

```cpp
struct Page {
  std::vector<uint64_t> elements;
  size_t tupleWidth;

  Page(size_t tupleWidth,
       size_t pageSize = 16 << 10)
       :tupleWidth(tupleWidth) {
    size_t tupleCount = pageSize /
              (sizeof(uint64_t) * tupleWidth);
    elements.reserve(tupleCount * tupleWidth);
  }
};
```

Figure 4.2: Page representation

This chapter illustrates the design and implementation of a push-based model for scan, counter, print, and ternary operators.

4.1 Operator Interface

Recent query engines utilize a push-based pipeline execution model, often referred to as the produce/consume model. In contrast to the iterator (pull) model, each operator pushes tuples upward to its parent by invoking the `consume()` function on the parent operator. The flow of data in this pipeline moves from the leaves to the root of the query plan tree and follows a recursive pattern: the parent operator calls the child's `produce()` function to initiate data processing, after which the child calls the parent's `consume()` function to retrieve all data tuples. This push-based pipeline replaces the traditional `next()` function with the produce/consume functions.

This alternative design reduces the overhead associated with the pull model and improves CPU cache utilization.

Our project implements the operator interface with three main methods: *Prepare*, *Produce*, and *Consume*. The code structure for the operator interface is shown below in Figure 4.3:

```
class Operator {
protected:
    // Number of elements per tuple
    size_t tupleWidth;
    // Parent operator to consume (push) the output tuples
    Operator* consumer;
public:
    /// Constructor
    Operator() = default;
    /// Copy constructor
    Operator(const Operator&) = delete;
    /// Destructor
    virtual ~Operator() noexcept = default;

    virtual void Prepare(Operator* parent) = 0;
    virtual void Produce() = 0;
    virtual void Consume(const Operator* child, const Page* page) = 0;

};
```

Figure 4.3: Push-based operator execution model

Produce method is called by the parent operator in order to start producing (fetching) all tuples. The current operator processes tuples by invoking the children's produce function and then calls the parent's consume function for each output tuple. The **Prepare** method is only used before query execution in order to set the consumer pointer operator from the parent operator, open file buffers, allocate hash tables, or define the join strategy. The push-based model permits pipelining as tuples are produced from lower operators (leaves) and consumed by the parent operator without any intermediate materialization unless an existent pipeline breaker (e.g., for `join` or `group-by` operators).

4.2 Core operators and Parallel Query Execution Plan

Figure 4.4 shows a simple query plan in the graph database, a Ternary join operator that contains three input scan operators for relations R, S, and T. The final output data is consumed at the end in the output (Counter/Print) operator. The produce-consume query plan execution model: Counter/Print operator produces by calling the producing method of the ternary join operator, and, subsequently, the join operator invokes the produce method of each scan operator. The three scan operators push the tuples to the join operator upwards for the build phase. Afterward, the matching tuples will be consumed to the parent operator of the ternary join inside the probe phase; in our query plan, the tuples will be consumed to the output (counter/print) operator.

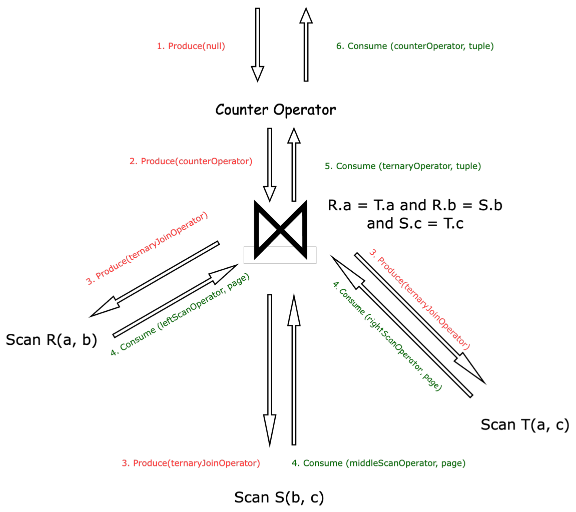

1. Produce(null)
6. Consume (counterOperator, tuple)
Counter Operator
2. Produce(counterOperator)
5. Consume (ternaryOperator, tuple)
R.a = T.a and R.b = S.b
and S.c = T.c
3. Produce(ternaryJoinOperator)
4. Consume (leftScanOperator, page)
Scan R(a, b)
3. Produce(ternaryJoinOperator)
4. Consume (rightScanOperator, page)
Scan T(a, c)
3. Produce(ternaryJoinOperator)
4. Consume (middleScanOperator, page)
Scan S(b, c)

Figure 4.4: Query Plan for the Ternary Operator – Joining Three Child Relations and Producing Matching Tuples to the Output Operator

4.2.1 Scan Operator

It is considered an input operator (leaf) node that can produce tuples for the upward operator without consuming any operators from its children. The `Prepare` method can be used to open file handlers where the relation tuples are stored and assign the appropriate parent node for processing the chunk of tuples. Concerning the Produce method, it effectively constructs pipelines by generating pages filled with deserialized tuples retrieved from disk, ready for consumption by the parent operator, specifically the ternary operator in this instance.

4.2.2 Ternary Join Operator

The ternary join is a generic implementation for the join logic of three relations; the difference depends on the delegation of the join strategy as shown in Figure 4.5, which could be an in-memory hash join, sort-merge join, or external hash join for the current thesis implementation. The external ternary join strategy will be discussed in Chapter 5 (extension to grace hash binary join, but for three relations). The main objective is to materialize all tuples (r, s, t) such that $r \in R(a, b), s \in S(b, c), t \in T(a, c)$, and matches the triangle join predicate, $r.b = s.b$, $s.c = t.c$ and $t.a = r.a$. The join operator is classified into two phases: ① **Build Phase** - consuming the input tuples from three children of the join operator (R, S, and T) organizes the tuples for the probe phase, as it depends on the join strategy; and ② **Probe Phase** - Consuming the matching tuples by probing the hash tables or bucket directories during the build phase produces the matching tuples for the parent of the join operator.

```
class TernaryJoinOperator : public Operator {
    Operator* left;
    Operator* middle;
    Operator* right;

    // The delegation of join strategy
    JoinStrategy* joinStrategy;

    // The depth of recursive external ternary join
    size_t currentLevel = 0;
    ...
}
```

Figure 4.5: Fields of Ternary Join Operator

In `Prepare` method as shown in Algorithm 1, the join operator construct the corresponding bucket directories, hashtables and file handlers by allocating and constructing the join strategy, in addition, recursively prepare the three children operators.

Algorithm 1 TernaryJoinOperator::Prepare

```
1: procedure PREPARE(consumer: Operator*)
2:     left.PREPARE(this)
3:     middle.PREPARE(this)
4:     right.PREPARE(this)
5:     self.consumer ← consumer
6:     self.joinStrategy ← CONSTRUCTJOINSTRATEGY(joinTypeEnum)
7: end procedure
```

During producing the tuples from three relations, we have to start with `Produce()` call in the build phase on children operators(R, S, and T scan operators) to fetch the entire tuples. Each page of tuples from three operators will be consumed by the join operator which will use `consumeBuildLeft(page)`, `consumeBuildMiddle(page)`, and `consumeBuildRight(page)` as mentioned in Algorithm 3. Since we are only considering the external hash join strategy, those methods will be discussed in Chapter 5.

After constructing the buckets for relations R, S, and T, we need to complete the building process as outlined in Algorithm 2. Following this, we should initiate the probe phase by calling `finishProbe()`. This function will consume the matching tuples by probing the tuples stored in the constructed buckets or hash tables. In ternary join operations, for each tuple (a, c) $\in$ T, we need to retrieve the matching tuples from R[a] and S[c]. Subsequently, we find the intersection of the two sets of tuples from R[a] and S[c] that match on the same attribute `b`. The matching tuple pair, denoted as `(r, s)` $\in R \times S$ based on attribute `b`, will be generated alongside tuple `t`. This results in a triple matching tuple `(r, s, t)` $\in R \times S \times T$. This triple will subsequently be processed by the parent operator, specifically the `Counter operator`, as outlined in our query plan presented in Figure 4.4.

Algorithm 2 TernaryJoinOperator::Produce

```
1: procedure PRODUCE
2:     Call left.Produce()                                  ▷ Build Phase Starts
3:     Call middle.Produce()
4:     Call right.Produce()
5:     Call joinStrategy.finishProbe()                      ▷ Probe Phase Starts
6: end procedure
```

Algorithm 3 TernaryJoinOperator::Consume

```
1: procedure CONSUME(child: Operator*, page: Page*)
2:     if child is left then
3:         joinStrategy.consumeBuildLeft(page)
4:     else if child is middle then
5:         joinStrategy.consumeBuildMiddle(page)
6:     else if child is right then
7:         joinStrategy.consumeBuildRight(page)
8:     end if
9: end procedure
```

4.2.3 Print/Counter Operator

Both print and counter operators are considered root/output operators, as they materialize the join results or aggregate the tuples for counting. The output operator receives input from a single operator, producing output tuples that initiate the query execution for the entire query plan. The `Consume(Operator* child, Tuple tuple)` method receives the joined tuple.

For print operatator, the tuple can be materialized using an I/O operation to display the tuple. For the *Counter* operator, it counts the matching tuples that was produced by input child operator. In our experiments detailed in Chapter 6, we use a counter operator instead of a print operator to measure latency without the extra overhead of materializing output.

Algorithm 4 CounterOperator::Consume

```
1: procedure CONSUME(child, tuple)
2:     $counter \leftarrow counter + 1$
3: end procedure
```

Algorithm 5 PrintOperator::Consume

```
1: procedure CONSUME(child, tuple)
2:     Output tuple
3: end procedure
```

5 Implementation

This chapter details the implementation of the **Grace Hash Ternary Join** strategy designed to process triangle queries in graph databases. Triangle queries, which identify all triples of tuples from relations R, S, and T that satisfy the join condition, are formally defined as:

$$\Delta_3(R, S, T) = R(a, b) \bowtie S(b, c) \bowtie T(a, c)$$

Efficient evaluation of such queries is challenging, particularly when the intermediate results generated from chained binary join operations can grow prohibitively large and overwhelm memory resources.

To address this challenge, our implementation extends the traditional Grace hash-based binary join [2] to perform a three-way (ternary) join in a single operator.The main idea is to partition the input relations according to their join attributes. This ensures that any potential join candidates (such as a triangle) are limited to specific sets of partitions, or "buckets." The number of partitions, denoted as k, is a tunable hyperparameter that can be determined by the query optimizer. The strategy is executed in two primary phases:

1. **Build Phase**: Each relation is partitioned using hash functions. The left and middle relations are partitioned into k buckets each, while the right relation is partitioned into k^2 buckets by applying a two-dimensional hash. This phase employs a robust bucket mechanism that incorporates memory buffering, temporary file persistence.

2. **Probe Phase**: The join is performed by iterating over corresponding bucket pairs from the left and middle relations and probing against the matching two-dimensional bucket from the right relation. For in-memory join candidates, hash tables are used to compute intersections on the common join attribute. In cases where bucket sizes exceed available memory, a recursive join strategy is invoked, further partitioning the data until an in-memory join becomes feasible.

This chapter begins with a high-level overview of the implementation, including detailed descriptions of the bucket structure and join strategy. Subsequent sections delve into the specifics of each phase—including pseudo-code, algorithms, and an analysis of the computational time complexity.

5.1 High-Level Overview of the Grace Ternary Join Implementation

The implemented strategy extends the classical Grace hash join to a three-way (ternary) join to process the triangle query in a single operator. The implementation is structured around two main phases—**Build Phase** and **Probe Phase**—and is underpinned by a carefully designed bucket mechanism. The following sections provide a high-level overview of the key components and design choices.

5.1.1 Bucket Structure and Management

The main component of the implementation is the bucket abstraction, which manages, buffers, and persists tuples during join execution. Each bucket is composed of:

- **Tuple Count**: Maintains metadata indicating the total number of tuples stored in the bucket, aiding in efficient tuple retrieval during the probe phase.
- **Bucket File**: A temporary file is used to persist tuples when the in-memory buffer (cached page) reaches capacity. Tuples are written sequentially as binary data, ensuring low I/O overhead and scalability.
- **Cached Page**: A fixed-size (e.g., 16KB) in-memory page that buffers incoming tuples before they are written to disk. This design reduces the number of disk writes by batching I/O operations.

The `addTuple` method is designed to flush the cached page to disk once it fills up, thereby preserving memory and maintaining efficient I/O.

5.1.2 Join Strategy: Partitioning and Build Phase

The overall join strategy is based on a multi-level partitioning scheme that aligns with the three relations:

- **Partitioning of Input Relations:**
  - The *left* relation R and the *middle* relation S are partitioned into a fixed number of buckets (e.g., $k = 32$) based on their join attributes by applying a CRC64 hash function. This hash function is implemented using CRC instructions as described by Altan Birler et al. [20]. For relation R, the hash is computed on attribute a, while for S it is computed on attribute c.
  - The **right** relation T is partitioned into k^2 buckets using a two-dimensional hash. Here, the buckets are indexed by the first $log_2(k)$ bits of both $hash(t[a])$ and $hash(t[c])$, aligning each right tuple with the corresponding left-middle bucket pair.
- **Handling Recursive Partitioning:** To address memory constraints, the current recursion depth (i.e., `currentLevel`) is used as a seed in the hash function, $CRC64(\text{value}, \texttt{currentLevel})$. This approach enables re-distribution of tuples for oversized buckets into k sub-buckets when necessary, thereby facilitating recursive ternary join processing.

5.1.3 Probe Phase and Join Execution

The probe phase is responsible for joining the partitioned tuples:

- **In-Memory Join for Small Partitions:** When a left-middle bucket pair is small enough to fit in memory, hash tables are built for each bucket. The left bucket's hash table is indexed on attribute a and the middle's on attribute c. The corresponding bucket from T is then probed, and an intersection on the join attribute (typically the intermediate attribute, b) is computed to produce the final join output.
- **Intersection Strategies:** Two intersection methods are adopted:
  - For base relations, a count-based intersection produces join results based on the frequency of matching tuples. The intersection procedure is optimized for tuples of width two, allowing retrieval without persisting the physical tuples.
  - For intermediate results, a multi-hash set approach is implemented to account for increased tuple widths. For larger tuples, it is essential to track the physical tuples to accurately materialize the matching triangles with all their values.
- **Recursive Join Execution:** If any bucket pair exceeds memory limitations, a recursive ternary join operator is invoked. In this scenario, a *BucketOperator* reads the persistent tuples of the corresponding bucket from disk, and a new instance of the ternary join (with an incremented recursion level) is created using the same partitioning strategy but with a different hash seed.

5.1.4 Asymptotic time Complexity

The asymptotic time complexity of the algorithm is analyzed as follows, disregarding the complexity associated with output materialization:

- **Build Phase Complexity**: Each tuple from R, S, and T is inserted into its corresponding bucket. With buffered insertion and constant-time disk writes upon overflow, the build phase completes in linear time, $O(|R| + |S| + |T|)$.
- **Probe Phase Complexity**: The probe phase iterates over k left buckets and k middle buckets, probing the k^2 right buckets. The overall complexity for probing is $O(|R| + k \cdot |S| + |T|)$, where the factor k arises from the nested iteration over bucket pairs.

Although the probe phase introduces additional overhead through the bucket iteration, this overhead is offset by the significant benefit of eliminating the need to materialize large intermediate results—a common performance bottleneck in sequential binary join approaches.

In summary, the implementation of the Grace Ternary Join utilizes a strong bucket mechanism along with hash partitioning and recursive refinement. It also employs a push-based probing strategy to efficiently handle triangle queries while operating under memory constraints. This approach significantly reduces the costs associated with materializing intermediate results, making it possible to execute scalable operations on large graph datasets.

5.2 Bucket Structure

Efficient management of input tuples is essential in external hash joins, especially when processing large volumes and complex queries in graph databases. To address memory constraints and enhance performance, it is vital to implement a structured bucket mechanism for effective management.

```
struct Bucket {
    // The total number of buckets
    std::size_t tuple_count;
    // The bucket file which will be used to serialize all tuples inside the file
    std::unique_ptr<TemporaryFile> bucketFile;
    // Cache the newly consumed tuples in a page before writing them into bucketFile
    std::unique_ptr<Page> cachedPage;
}
```

Figure 5.1: Bucket Structure

Each bucket is defined as follows:

- **Tuple Count**: It counts the total number of tuples that were stored inside the bucket, whether it was cached in memory or persisted inside the bucket file on disk. The counter is considered as metadata for reading the tuples from the disk in probe phase.
- **Bucket File**: We used a temporary file that will be closed and removed automatically in the destructor of the temporary file by using *RAII* idiom [21]. Basically, the bucket file is used to persist the tuples on disk if the tuples exceed cached page capacity. The tuples are written sequentially as binary data, ensuring the scalability for efficiently handling large datasets.
- **Cached Page**: It is used as a buffer for storing the tuples temporarily before serializing the buffer page into the bucket file. This reduces the number of I/O operations by writing tuples with sequential access using one I/O operation per page, which optimizes disk utilization.

Algorithm 6 Bucket::addTuple

```
1: procedure ADDTUPLE(tuple)
2:     if cachedPage is full then
3:         WRITECACHEDPAGETODISK(CACHEDPAGE, BUCKETFILE)
4:     end if
5:     cachedPage.ADDTUPLE(tuple)
6:     tuple_count ← tuple_count + 1
7: end procedure
```

The `addTuple(tuple)` method, defined in Algorithm 7, is a robust function for inserting tuples into the bucket. It ensures the tuples are inserted into the bucket at the beginning of the in-memory buffer , specifically within the cached page. If the buffer pages reach the limit of fixed page size (i.e., 16KB), it proactively writes and serializes the buffer page into a bucket file on disk ensuring the data persistence.

5.3 Join Strategy: External Grace Hash Ternary Join

```
struct GraceTernaryJoinStrategy {

        Bucket leftDirectory[NUM_BUCKETS];
        Bucke middleDirectory[NUM_BUCKETS];
        Bucket rightDirectory[NUM_BUCKETS][NUM_BUCKETS];

        // Maximum memory usage per operator in bytes
        size_t memory_size;
    ...
}
```

Figure 5.2: Grace Ternary Hash Join Structure

5.3.1 Build Phase

Initially, all three input relations are partitioned using a hash function that applies CRC instructions [20] on the join keys. The bucket index is determined by the first $\log_2(k)$ (i.e, k = $NUM\_BUCKETS$) bits of the hash value, ensuring that the number of buckets is always a power of two for simplicity to consider the first $\log_2(k)$ bits. This process is referred to as the build phase of the join strategy and will be implemented in the functions *consumeBuildLeft(childOperator, tuple)*, *consumeBuildMiddle(childOperator, tuple)*, and *consumeBuildRight(childOperator, tuple)*. For clarity, we assume the cardinalities of the three relations are in ascending order, such that $|R| \leq |S| \leq |T|$. The ordering of the relations can be managed by the query optimizer.

The number of buckets will be fixed to ensure that memory usage for all bucket allocations remains low. We will determine the number of buckets using the query optimizer, which can be defined in future work. Both the left and middle relations will each contain k buckets, while the right relation will have k^2 buckets. The rationale behind this approach is to efficiently index the left relation R using the first $\log_2(k)$ bits of hash from the attribute value a (i.e., $leftBucketIndex = \text{hash}(r[a]) \ shr \ (64 - log_2(k)))$, and to index the middle relation using the the first $\log_2(k)$ of hash from attribute value c (i.e., $middleBucketIndex =$

hash($s[c]$) shr ($64 - log_2(k)$)). Furthermore, the tuples of the right relation will be partitioned based on both $leftBucketIndex$ and $middleBucketIndex$. During the probing phase, all tuples in `rightDirectory`[$leftBucketIndex$][$middleBucketIndex$] will check for matches in `leftDirectory`[$leftBucketIndex$] and `middleDirectory`[$middleBucketIndex$]. This allows us to find matching tuples by identifying the intersection of `leftDirectory`[$leftBucketIndex$] and `middleDirectory`[$middleBucketIndex$] to retrieve all corresponding b values.

Algorithm 7 GraceTernaryJoinStrategy::consumeBuildLeft

```
1: procedure CONSUMEBUILDLEFT(page)
2:     $currentLevel \leftarrow$ GETCURRENTLEVEL
3:     for r in $page$ do
4:         $bucketIndex \leftarrow \text{CRC64}(r[a], currentLevel)\ shr\ (64 - log_2(k))$
5:         $bucket \leftarrow$ leftDirectory[$bucketIndex$]
6:         bucket.ADDTUPLE($r$)
7:     end for
8: end procedure
```

Algorithm 8 GraceTernaryJoinStrategy::consumeBuildMiddle

```
1: procedure CONSUMEBUILDMIDDLE(page)
2:     $currentLevel \leftarrow$ GETCURRENTLEVEL
3:     for s in $page$ do
4:         $bucketIndex \leftarrow \text{CRC64}(s[c], currentLevel)\ shr\ (64 - log_2(k))$
5:         $bucket \leftarrow$ middleDirectory[$bucketIndex$]
6:         bucket.ADDTUPLE($s$)
7:     end for
8: end procedure
```

In the consume build phase for left and middle relations, as shown in Algorithm 7 and 8, we need to consider the current recursive depth of the ternary join operator. This depth will be considered as the seed for the hash function to differentiate between hash functions at different levels of recursive ternary join. This differentiation helps to re-distribute tuples that belong to the same bucket within the recursive ternary operator. The tuples are partitioned using the hash function $CRC64$ on the value of $r[a]$ in left relation R and $s[c]$ in middle relation S with the seed of the recursive depth level. Given that the bucket indices range $[0 \ldots k)$, the first $log_2(k)$ bits will be considered for tuple insertion in the corresponding bucket whether for `leftDirectory` and `middleDirectory`.

Algorithm 9 GraceTernaryJoinStrategy::consumeBuildRight

```
1: procedure CONSUMEBUILDMIDDLE(page)
2:     $currentLevel \leftarrow$ GETCURRENTLEVEL
3:     for t in $page$ do
4:         $leftBucketIndex \leftarrow$ CRC64($t[a]$, $currentLevel$) $shr$ $(64 - log_2(k))$
5:         $middleBucketIndex \leftarrow$ CRC64($t[c]$, $currentLevel$) $shr$ $(64 - log_2(k))$
6:         $bucket \leftarrow$ rightDirectory[$leftBucketIndex$][$middleBucketIndex$]
7:         bucket.ADDTUPLE($t$)
8:     end for
9: end procedure
```

During the build phase of right relation T for Algorithm 9, It is essential to determine the bucket index for values $t[a]$ and $t[c]$. This step is important to find the corresponding bucket within the 2D directory of right relation. This will be considered as the only difference compared to the consumption of tuples in left and middle relations.

Algorithm 10 GraceTernaryJoinStrategy::finishProbe

```
1: for $leftBucketIndex \leftarrow 0$ to $k - 1$ do
2:     $leftBucket \leftarrow leftDirectory[leftBucketIndex]$
3:     $leftBucketHashTable \leftarrow$ constructHashTable($leftBucket$)
4:     for $middleBucketIndex \leftarrow 0$ to $k - 1$ do
5:         $middleBucket \leftarrow middleDirectory[middleBucketIndex]$
6:         $rightBucket \leftarrow rightDirectory[rightBucketIndex]$
7:         if exceedMemoryUsage($leftBucket, middleBucket$) then
8:             applyRecursiveTernaryJoin($leftBucket, middleBucket, rightBucket$)
9:         else
10:            $middleBucketHashTable \leftarrow$ constructHashTable($middleBucket$)
11:            for all $rightTuple \in rightBucket$ do
12:                $a\_value \leftarrow rightTuple$[a]
13:                $c\_value \leftarrow rightTuple$[c]
14:                $bR \leftarrow leftBucketHashTable[a\_value]$
15:                $bS \leftarrow middleBucketHashTable[c\_value]$
16:                for all $left\_middle\_tuple \in bR \cap bS$ do
17:                    consumer.consume($left\_middle\_tuple \circ rightTuple$)
18:                end for
19:            end for
20:        end if
21:    end for
22: end for
```

5.3.2 Probe Phase

When the left-middle bucket pair *(leftDirectory[leftBucketIdx], middleDirectory[middleBucketIdx])* are small enough to fit in memory, `finishProbe()` method can handle out the join for both partitions entirely in memory with no recursive hash joins. The `finishProbe()` procedure is mainly used for producing matching tuples as three-way join. The probe phase starts by iterating over all left and middle buckets. The join structure for Algorithm 10 is managed based :

1. **In-Memory Join of a Partitions** For each left-middle pair bucket, we must ensure that the input tuples in both buckets can fit in memory. In that case, two hash tables will be constructed from tuples in a left bucket and middle bucket, respectively. The hash table structure differs if the child operator is a base relation or an intermediate result as it should be indexed by the common join attribute between the current relation and right relation T. More formally, the hash table of the left relation $R(a, b)$ will be indexed by attribute a since its the common attribute with right relation $T(a, c)$. Likewise, the middle relation $S(b, c)$ will be indexed by attribute c in the hash table. The only difference is the strategy for indexing all possible b's in order to compute intersection efficiently based on the children operator:
   - **Base Relation**: Since we consider only edges (tuples) of graph database representation, there is no need to consider any other attribute to have a reference on the physical tuple. The effective solution is to have hash tables on R and S with keys a and c, respectively, and an embedded hash map of all b's with count for a number of `b` in $R[a]$ and $S[c]$. We iterate over the smaller embedded hash map size and iterate over all b's. If the larger hash map contains the corresponding b, the number of matching tuples that the parent operator should consume equals to $leftBucketHashTable[a][b] \times middleBucketHashTable[c][b]$ times, those b's will be considered as the final join result (as specified in Algorithm 11) that will be consumed by `finishProbe()` procedure.
   - **Intermediate Result**: The tuple width may be increased if the ternary join operator produces the tuples from child join operator, which makes the attribute indexing more complex. One possible strategy is to have hash tables on R and S with keys a and c, respectively, and hash map to look up on all b's with Multi-HashSet with all tuples of corresponding `b` value as specified in Algorithm 12.

Algorithm 11 Intersection Base Relations

```
function IntersectBaseRelations(leftBucketHashTable, middleBucketHashTable, a, c)
    $bLeftBucket \leftarrow leftBucketHashTable[a]$
    $bMiddleBucket \leftarrow middleBucketHashTable[c]$
    if size($bLeftBucket$) < size($bMiddleBucket$) then
        for all $b \in$ keys($bLeftBucket$) do
            if $b \in bMiddleBucket$ then
                $bCount \leftarrow bLeftBucket[b] \times bMiddleBucket[b]$
                while $bCount > 0$ do
                    yield $(a, b) \circ (b, c)$
                    $bCount \leftarrow bCount - 1$
                end while
            end if
        end for
    else
        for all $b \in$ keys($bMiddleBucket$) do
            if $b \in bLeftBucket$ then
                $bCount \leftarrow bLeftBucket[b] \times bMiddleBucket[b]$
                while $bCount > 0$ do
                    yield $(a, b) \circ (b, c)$
                    $bCount \leftarrow bCount - 1$
                end while
            end if
        end for
    end if
end function
```

Algorithm 12 Intersection Intermediate Operator

```
1: procedure IntersectIntermediateOperator($leftBucketHashTable$, $middleBucketHashTable$, $a$, $c$)
2:     $bLeftBucket \leftarrow leftBucketHashTable[a]$
3:     $bMiddleBucket \leftarrow middleBucketHashTable[c]$
4:     if size($bLeftBucket$) < size($bMiddleBucket$) then
5:         for all $b \in$ keys($bLeftBucket$) do
6:             if $b \in$ keys($bMiddleBucket$) then
7:                 for all $tuple \in (bLeftBucket[b] \times bMiddleBucket[b])$ do
8:                     yield $tuple$
9:                 end for
10:            end if
11:        end for
12:    else
13:        for all $b \in$ keys($bMiddleBucket$) do
14:            if $b \in$ keys($bLeftBucket$) then
15:                for all $tuple \in (bLeftBucket[b] \times bMiddleBucket[b])$ do
16:                    yield $tuple$
17:                end for
18:            end if
19:        end for
20:    end if
21: end procedure
```

2. **Recursive Partitioning for Oversized Buckets**: If the left and middle bucket sizes exceed memory limit, it triggers recursive ternary join on the three buckets, which are considered as matching partitions of three relations R, S, and T. The call of `applyRecursiveTernaryJoin` procedure in Algorithm 10 within line 8 creates a new ternary join operator configured with External Grace Hash Join Strategy. The three buckets will be injected as child operators of a recursive ternary join, which will be produced for the ternary join operator. The join operator's attribute *currentLevel* indicates the recursive depth, starting at 0 for the top-level join. At level 1 (the next recursive ternary join operator), the hash function uses a different seed by using `currentLevel` in `CRC64(value, currentLevel)` which makes `CRC64(value, 0)` $\neq$ `CRC64(value, 1)`, which can re-distribute the tuples of the same bucket in the old join operator to the buckets of the new join operator and ensure the uniform distribution of tuples on k buckets. For any two distinct integers i and j (where $i \neq j$), the functions `CRC64(value, i)` and `CRC64(value, j)` are highly unlikely to produce the same hash value. In other words, the probability of a collision occurring between these two functions is extremely low [20].

   The tuples of three buckets are not all in memory at once as they were persisted on disk. The *BucketOperator* is basically equivalent to *ScanOperator* as in Section 4.2.1.

The only difference is instead of producing input tuples from a persistent table; the bucket operator produces its persistent tuples from bucket file descriptor, which will be consumed by the parent operator (recursive ternary join operator)

Algorithm 13 ApplyRecursiveTernaryOperator

```
1: procedure APPLYRECURSIVETERNARYOPERATOR(leftBucket, middleBucket, rightBucket, currentLevel)
2:     leftBucketOperator ← constructBucketOperator(leftBucket)
3:     middleBucketOperator ← constructBucketOperator(middleBucket)
4:     rightBucketOperator ← constructBucketOperator(rightBucket)
5:     newCurrentLevel ← currentLevel + 1
6:     newJoinOperator ← constructTernaryJoinOperator(leftBucketOperator, middleBucketOperator, rightBucketOperator, newCurrentLevel)
7:     newJoinOperator.PREPARE(consumer)
8:     newJoinOperator.PRODUCE()
9: end procedure
```

5.4 Computational Complexity of External Grace Hash Join

We analyse the asymptotic analysis of External Grace Hash Join based on Build and Probe phase, ignoring the complexity of output materialization.

- **Build Phase**: The tuples were consumed in the ternary operator from three relations R, S, and T. Each tuple will be inserted in the corresponding bucket. At the beginning, the tuples are cached in a page of size 16KB. If the page is filled, it will be written on disk which has asymptotic complexity $O(1)$ per each tuple insertion. Therefore, the complexity of consuming the tuples of all relations in build phase is linear with order of $O(|R| + |S| + |T|)$.

- **Probe Phase**: This phase is fully handled in `finishProbe()` procedure that was mentioned in Algorithm 10. The probe phase starts by iterating over k buckets of left relation R, and for each bucket in left relation, we iterate over all k buckets of middle relation S in order to probe the matching tuples from corresponding (i, j) bucket pair index of right bucket with i_{th} left bucket and j_{th} middle bucket. The total number of operations will be equivalent to $\sum_{i=1}^{k} \left(|R|_i + \sum_{j=1}^{k} \left(|S|_j + |T|_{i,j} \right) \right)$, where $|.|_i$ denotes the cardinality of the i_{th} bucket of the corresponding relation. Since, the total number of tuples in all buckets will be equivalent to relation's cardinality; thereby, $\sum_{i=1}^{k} |R|_i = |R|$, $\sum_{i=1}^{k} |S|_i = |S|$, and $\sum_{i=1}^{k^2} |T|_i = |T|$. By substitution, the asymptotic complexity of probing the matching tuples is $O(|R| + k \cdot |S| + |T|)$, which puts an overhead with factor k (number of buckets). This indicates that using two Grace Binary Join Operators to combine three relations still maintains a better asymptotic complexity of $O(|R| + |S| + |T|)$.

6 Evaluation

In this chapter, we take a closer look at our new Grace Ternary Join algorithm and compare it with two other approaches: one that runs a triangle query using DuckDB [1] and the traditional Grace Binary Join [2]. We begin by detailing how each approach processes triangle queries, breaking down the inner workings of the ternary and binary operators and discussing the key ideas behind each strategy. Afterwards, we describe our experimental setup in detail. This includes the datasets we used, the system configurations, and the different ways we balanced data and memory. Our goal here is to create a fair environment to measure and compare the performance of each method. We ran two main experiments. The first experiment assesses the performance of the Grace Ternary Join in comparison to DuckDB, which is set up for single-threaded query processing. The second experiment investigates the trade-offs between the Grace ternary and binary join methods under various memory conditions. In both cases, we focused on execution times, the overhead of data partitioning, and scalability. Ultimately, our work highlights the strengths and weaknesses of each strategy, providing insights that we hope will contribute to advancing the field.

6.1 Query Plans

A triangle query is a type of cyclic query graph involving three relations that counts or materializes the number of triangles in a graph. In this context, a triangle consists of a set of three vertices (attributes) where each relation represents a list of edges, and these vertices are pairwise connected by direct edges. In terms of a relational database, a triangle can be expressed as a 3-way self-join, as follows:

$$\Delta_3(R, S, T) = R(a, b) \bowtie S(b, c) \bowtie T(a, c),$$

with join condition that ensure relation R and S have the same attribute `b` on condition $R.b = S.b$, relation S and T on attribute `c` within condition of $S.c = T.c$ and relation T and R on attribute a with condition $R.a = T.a$.

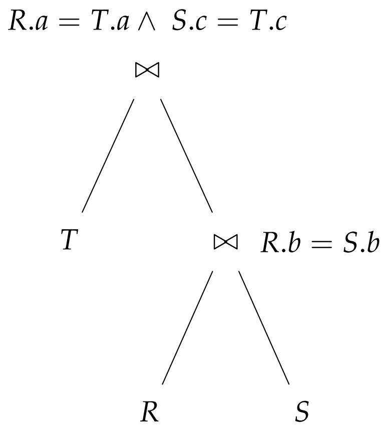


Figure 6.1: Binary Join plan on relations R, S and T

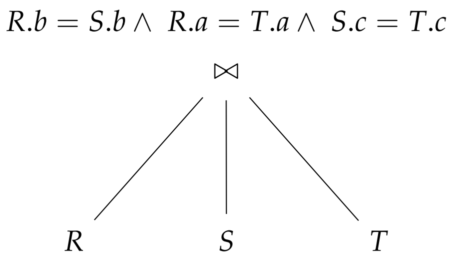


Figure 6.2: Ternary join plan on relations R, S and T

Our evaluation focuses on executing a triangle query using a sequence of two binary joins, as illustrated in Figure 6.1. This involves joining three relations through binary joins, which will be implemented with the Grace Binary Join method. We will also compare this approach to a single ternary join using the Grace Ternary Join strategy, as depicted in Figure 6.2 and described in Section 6.5.

6.2 Datasets

Dataset	# Edges	Relation Size	Intermediate Result	Intermediate Result:Relation Size Ratio	3 Relation Size	# Triangles
Facebook	88 234	1.4 MB	2 690 019	≈ 30:1	4.2 MB	1 612 010
Twitter	2 420 766	38.73 MB	628 009 402	≈ 259:1	116.19 MB	1 458 379 417
Wiki	5 021 410	80.34 MB	1 383 166 763	≈ 275:1	241.02 MB	20 307 710
Twitch	6 797 558	108.76 MB	731 816 194	≈ 107:1	326.28 MB	54 148 939
LiveJournal	68 993 773	1.10 GB	5 915 633 945	≈ 85:1	3.30 GB	946 400 853

Table 6.1: Summary of Datasets, Their Sizes, Intermediate Results, and Ratios, Along with the Number of Triangles

We evaluate the join operators based on existent datasets that were taken by SNAP group [22]. We considered the datasets that cover online social networks and communication platforms. All datasets are lists of edges; each edge contains two integers with a fixed size of 8 bytes (16 bytes per edge). Therefore, the total size of each dataset in bytes is *16 bytes × number of Edges*. The datasets that we considered are:

- **Facebook** [23]: The dataset contains ego-networks, which are subgraphs of individual Facebook users. This dataset is practical for studying the structure of social influence patterns in a well-known social media platform. The number of edges is 88234 with a size of 1.4 MB (*88234 edges × 2 integers × 8 bytes*), and the number of triangles is 1,612,010

- **Twitter** [23] (*X* now): It consists of ego-networks captured from Twitter (now as *X* now). Similarly, the dataset Twitter contains a list of followers. Each edge (a, b) is formulated as `a follows b`, but it does not guarantee that `b follows a`. Therefore, the Twitter dataset contains directed edges. The number of edges is 2,420,766 edges with size 38.73 MB and the number of triangles is 1,458,379,417.
- **Wiki**: The network has the users and discussion of Wikipedia. The nodes represent the Wikipedia users, and the directed edge (a, b) represents that user `a` edited a page of user `b`. The number of directed edges is 5,021,410 with a size of 80.34 MB, and the number of triangles is 20,307,710.
- **Twitch** [24]: Each edge in the dataset represents a mutual follower relationship between users of Twitch. The dataset contains 6,797,558 edges with a size of 108.76 MB and 54,148,839 connected triangles.
- **LiveJournal** [25]: It is community with 10 million users. The edges refer to the direct friendship in the social network between users. The number of direct friendships is 68,939,773 with size 1.10 GB and 946,400,853 triangles.

6.3 Experimental Setup:

All our experiments were conducted on an Apple M1 machine with an 8-core CPU, 16GB of RAM, and a 256GB SSD. For each dataset, the triangle join was set up as a self-join on the graph's edges using three identical relations (R, S, and T), meaning each relation contains the same set of edges as the original graph (i.e., $|R| = |S| = |T|$ for every dataset).

We implemented our approach using a Grace Ternary hash partitioning strategy with a fixed number of buckets, $k = 32$. In contrast, the baseline strategy utilizes a Grace Binary hash join with $k = 1024$ buckets. To ensure a fair comparison between the two methods, we configured the number of partitions for the probe relation (the right-hand side) to be 1024 in both cases.

This approach guarantees an even distribution of input data, allowing each partition to be processed efficiently within the available memory. Our external ternary join utilizes the `robin-map` [1] third-party hash table library, which facilitates fast hash map using open-addressing and linear robin hood hashing. To effectively measure performance, we choose not to write output tuples (triangles) to disk. Instead, we employ a counter operator to tally results, thereby eliminating I/O overhead during timing, as elaborated in Section 4.2.3.

Each experiment was conducted in single-threaded mode to guarantee that any performance differences observed were solely attributed to the algorithms themselves. DuckDB was set to operate with a single thread, and our join operators were also designed to utilize a single-threaded algorithm. The timing results presented reflect the average CPU execution time over several runs.

[1]`https://github.com/Tessil/robin-map`

For all datasets, the Grace ternary hash join operator was configured with 32 partitions for both the left and middle relations. In the case of the ternary join, the left relation (denoted as R) was partitioned based on the attribute `'a'`, the middle relation (denoted as S) was partitioned based on the attribute `'c'`, and the right relation (denoted as T) was partitioned using a combination of attributes `'a'` and `'c'`. This was achieved through a 2D bucket directory within 1024 partitions, derived from the product of `leftPartitions.size()` and `rightPartitions.size()` for the right relation, as detailed in Section 5.3.

In our experiments, we varied the data-to-memory ratio to see how our method performs under different resource constraints. This ratio tells us how the size of our dataset compares to the available memory. When the ratio is 1×, the dataset fits perfectly within memory, meaning almost all of it can be stored at once. As we gradually increase the ratio—to 2×, 4×, and up to 32×—we mimic situations where memory is more limited, which creates greater challenges for data input and output. In these tougher cases, particularly when dealing with unevenly distributed graphs or very small amounts of memory, some parts of the data (buckets) might temporarily need more memory than is available. When this happens, our approach is to break down the overloaded bucket further with an extra pass using ternary join on the matching three buckets. In simple terms, we perform a recursive Grace ternary join on that specific subset of data, the procedure detailed in Section 5.3.2.

6.4 Experiment 1: DuckDB vs. Grace Ternary Join

We first compare the performance of our external Grace Ternary Join against DuckDB [1] (a fast in-memory analytical database) on the triangle query. While DuckDB efficiently executed the triangle join query using the edge table in a single thread, our Grace Ternary Join utilized external partitioning for the same query. Although this approach required writing to disk, the partitions effectively resided in RAM with a 1:1 ratio. Table 6.2 provides a comprehensive overview of total execution times for each dataset in this baseline scenario.

Dataset	**DuckDB**	**Grace Ternary Join**
Facebook	0.146	0.311
Twitter	36.393	20.045
Wiki	35.864	32.517
Twitch	27.112	39.418
LiveJournal	644.387	726.952

Table 6.2: Total execution time (in seconds) using DuckDB vs. Grace Ternary Join (single-threaded, ratio = 1)

In our analysis of the Facebook dataset, which has 88K edges, we observe that both systems completed their tasks almost instantly, with just a tiny performance gap of 0.16 seconds. DuckDB's triangle query performed remarkably well, benefiting from the small intermediate results that kept overhead to a minimum. In contrast, our ternary join implementation had to carry out partitioning and disk I/O—even though the entire dataset fit in memory—resulting in a slightly slower performance. Overall, the difference in execution time was only a few hundred milliseconds.

On datasets where intermediate results far exceed the size of the inputs, the benefits of the specialized ternary join method become increasingly evident. In our experiments with the Twitter (2 million edges) and Wiki (5 million edges) datasets, DuckDB's binary join approach produces intermediate results that are 259× and 275× larger than the input data, respectively. DuckDB's query plan decomposes the triangle join into two binary joins, which forces the system to materialize these enormous intermediate results. In contrast, the GraceTernaryJoin strategy processes all three relations within a single operator, bypassing the need to write out and subsequently read back large temporary outputs. This integrated approach leads to significant performance benefits, with GraceTernaryJoin achieving a 1.8× speed-up on the Twitter dataset (completing in 20 seconds versus DuckDB's 36 seconds) and a 1.10× speed-up on the Wiki dataset (31.8 seconds versus 35 seconds).

On the Twitch dataset, where the intermediate results are 107 times the size of the input, we observed a reversal in performance. In this case, DuckDB finished the operation in 27.112 seconds, while our GraceTernaryJoin took about 39.418 seconds. This suggests that although our specialized ternary join can offer significant advantages for datasets producing very large intermediate outputs, its effectiveness largely depends on the specific characteristics of the data at hand.

In our analysis of the LiveJournal dataset, the differences between the two approaches become more evident. The largest LiveJournal graph contains approximately 68.9 million edges and about 946 million triangles (see Table 6.1). For this dataset, DuckDB executed the triangle query in 644.387 seconds, while our GraceTernaryJoin algorithm took around 726.952 seconds—about 13% longer in raw execution time. This difference can be mainly explained by the way DuckDB's intermediate binary join works as it produces matching tuples that end up being 85 times larger than the original input—a scale that's actually more moderate than what we see with the other datasets.

The baseline results indicate that our Grace Ternary Join can match or even exceed the performance of a leading DuckDB system for triangle joins, especially as the ratio of intermediate size to input data size increases. In environments with ample memory and a small size ratio, like Facebook, the additional overhead from partitioning may slightly impact performance. However, for larger ratios, such as those found in Twitter and Wikipedia datasets, the three-way join strategy demonstrates clear advantages.

6.5 Experiment 2: Grace Binary vs. Grace Ternary Join

We evaluate the performance of the Grace hash-based binary join approach in comparison to our ternary join method, while varying memory availability. For these experiments, we utilize our own implementation of a traditional Grace hash binary join, as described briefly in Section 3.1.1.To compute triangles, we perform a triangle query using two consecutive binary hash joins. First, we join tables R and S on their common key to produce an intermediate result. Then, we join that intermediate result with table T. This process of binary joins uses external partitioning, which enables it to handle datasets larger than the available memory by spilling data to disk. We will compare this method to the GraceTernaryJoin across various datasets at different data-to-memory ratios {1, 2, 4, 8, 16, 32}. These ratios simulate increasingly demanding memory conditions; for instance, at a ratio of 32:1, the memory available is only 3% of the data size.This scenario requires significant disk I/O and results in additional passes due to the creation of recursive join operators. For each dataset, we will provide a detailed breakdown that includes the time spent in the build phase, the probe phase, and the total execution time for both the binary and ternary approaches. The build phase involves reading and partitioning the input data. In contrast, the probe phase entails scanning the partitions and constructing hash tables to identify matches and writing out results for the first join in the binary case.

6.5.1 Facebook Dataset

Table 6.3: Execution Times for the Facebook Dataset (88K edges)

Data-to-Memory Ratio	Grace Ternary Join			Grace Binary Join		
	Build (s)	Probe (s)	Total (s)	Build (s)	Probe (s)	Total (s)
1	0.041	0.269	0.310	0.282	0.262	0.546
2	0.036	0.273	0.309	0.288	0.265	0.555
4	0.032	0,272	0.305	0.279	0.262	0.543
8	0.039	0.274	0.313	0.282	0.264	0.547
16	0.038	12.795	12.834	0.303	0.332	0.636
32	0.036	20.181	20.218	0.411	0.325	0.738

The Facebook graph consists of a relatively small number of edges (88K), which allows both Grace Binary and Grace Ternary methods to operate efficiently at ratios up to 8. For the Ternary Join at a ratio of 1, the build time is only 0.041 seconds, which represents about 13% of the total time of 0.310 seconds. In contrast, the probe phase takes 0.269 seconds, accounting for the remaining 87%. At ratios of 2 and 4, we observe a similar distribution, with total execution times remaining around 0.31 seconds or less. The Binary Join method also demonstrates consistently low times (less than or equal to 0.55 seconds) across ratios from 1 to 8, with the build and probe times roughly split in half. For instance, at ratio 1, the build time is 0.282 seconds and the probe time is 0.262 seconds.

However, starting at ratio 16, Ternary Join's probe time jumps from under a second to 12.795s, which becomes 99.7% of its total (12.834s). By contrast, Binary's total at ratio 16 remains just 0.636s, with the build (0.303s) and probe (0.332s) nearly balanced. At ratio 32, Ternary's probe dominates even more—20.181s out of 20.218s total—whereas Binary stays near 0.74s, split roughly 56% build vs. 44% probe. This disparity suggests that under severely constrained memory, Ternary Join encounters deeper recursion or skew-driven overhead, inflating its probe phase, even though the graph itself is small. By contrast, The overhead for

Binary join on Facebook dataset remains minimal, regardless of the ratio.

In summary, for ratios 1–8, both methods run quickly (under a second) on this tiny dataset. But when memory is drastically limited (16×, 32×), During the probe phase of the Ternary Join, we observe a notable rise in total processing times, varying from 12 to 20 seconds. This increase is attributed to recursive ternary joins when the joining buckets surpass memory limitations. In contrast, the Binary Join consistently completes this phase in under one second. Although the Ternary Join generally performs well at managing intermediate blow-up in larger datasets, it faces penalties due to recursion overhead when applied to a small graph with significant memory constraints.

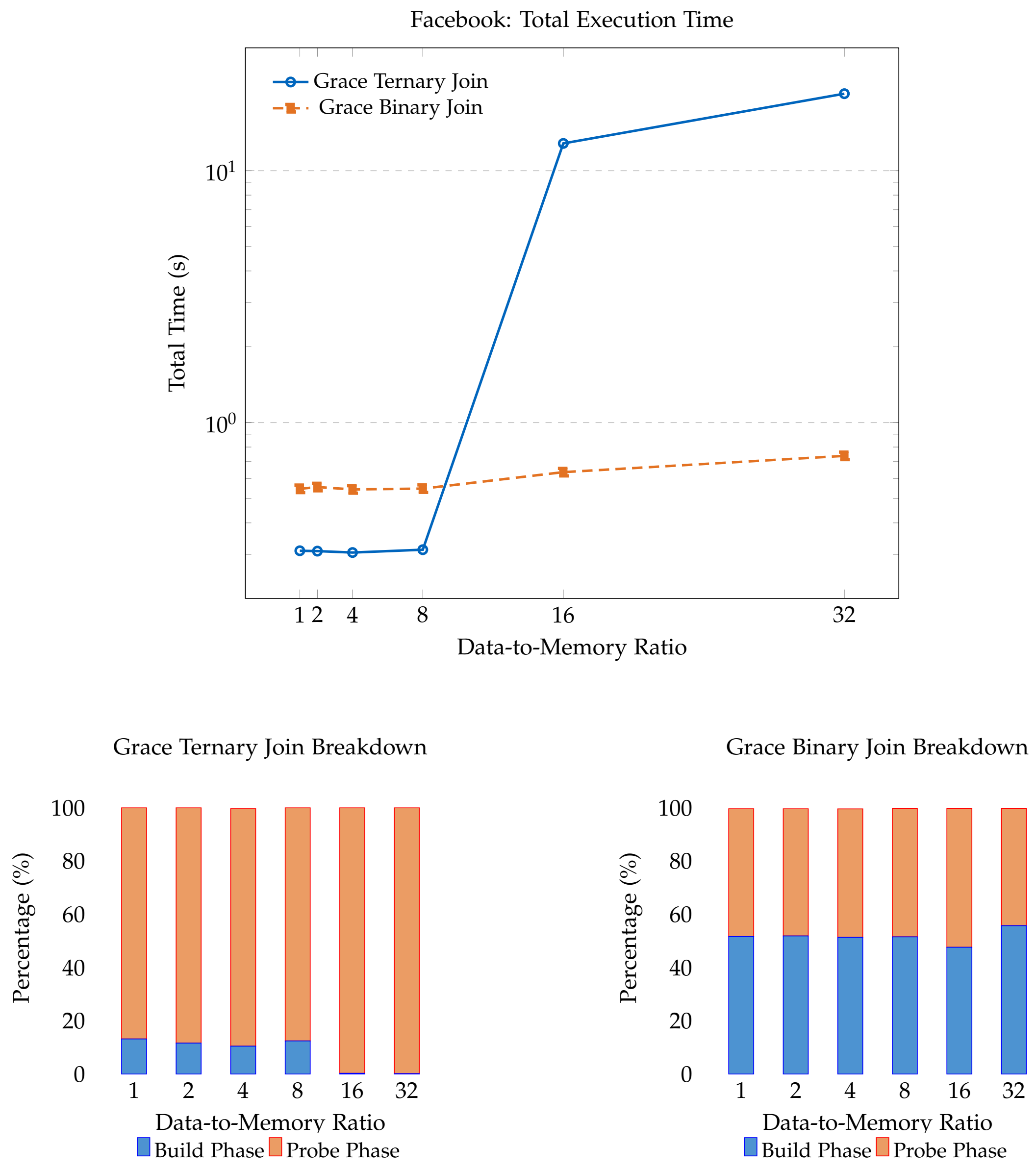


Figure 6.3: Facebook Dataset (88K edges):
Top: Log-scale line graph of Total Execution Time for Grace Ternary Join (blue circles) and Grace Binary Join (orange squares, dashed).
Bottom Left: Stacked bar chart for the Grace Ternary Join breakdown (Build vs. Probe).
Bottom Right: Stacked bar chart for the Grace Binary Join breakdown (Build vs. Probe).

6.5.2 Twitter Dataset

Table 6.4: Execution Times for the Twitter Dataset (2 million edges)

Data-to-Memory Ratio	Grace Ternary Join			Grace Binary Join		
	Build (s)	Probe (s)	Total (s)	Build (s)	Probe (s)	Total (s)
1	0.244	19.800	20.045	77.546	127.791	205.338
2	0.174	19.517	19.693	75.506	126.727	202.235
4	0.153	19.468	19.622	74.817	126.955	201.774
8	0.274	19.735	20.010	74.877	124.811	199.689
16	0.150	107.097	107.248	74.602	176.657	201.259
32	0.146	249.013	249.160	72.342	431.334	503.676

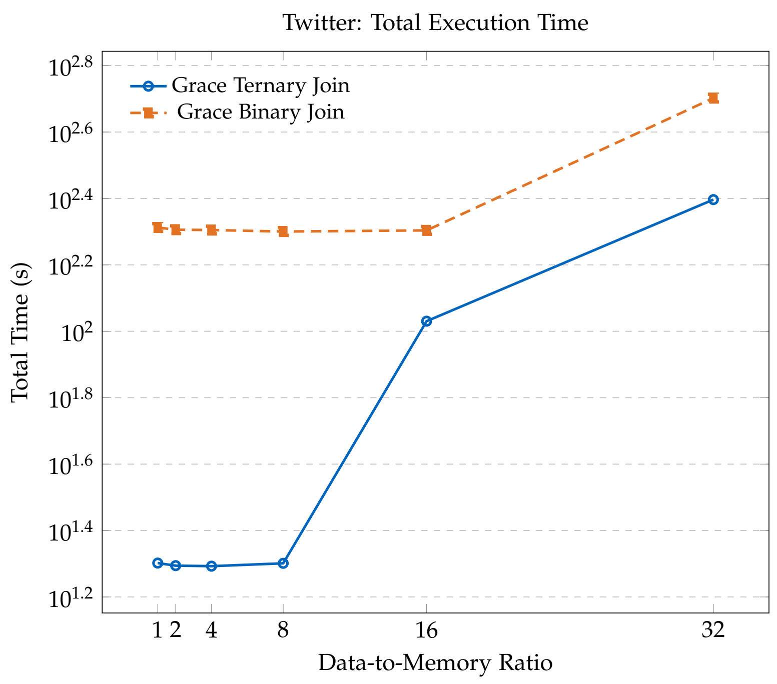
Figure 6.4 (top) shows the total execution times of Grace Ternary and Grace Binary joins on the Twitter graph, containing 2 million edges. We see that *Grace Ternary* begins around 20 seconds total for ratios $\{1, 2, 4, 8\}$ and then jumps to 107 seconds at ratio 16, finally reaching 249 seconds at ratio 32. Meanwhile, *Grace Binary* remains in the 200–205 second range from ratios 1 to 16, but it surges to 504 seconds at ratio 32. Thus, Ternary is significantly faster at all ratios, though the speedup diminishes under tight memory.

Ratio 1–8 : When the ratio is up to 8, Ternary Join completes in roughly 20 seconds, while Binary Join takes about 200 seconds—a tenfold difference. As shown in the bar charts in Figure 6.4 (bottom), Ternary spends only 1–2% of its time on building partitions, with more than 98% dedicated to the probing phase. In contrast, Binary Join devotes around 75–77 seconds (roughly 37% of its overall time) to the build phase, leaving 125–128 seconds for probing.

Ratio 16 : When the memory ratio reaches 16, the total time for the Ternary Join increases by more than five times, rising from about 20 seconds to 107 seconds. This increase is almost entirely due to the probe phase, which accounts for 99% of the overhead. In contrast, the Binary Join maintains a total time of around 201 seconds, with a division of 37% for the build phase and 63% for the probe phase. Despite the significant increase in Ternary Join's time, it still performs approximately twice as fast as the Binary Join, taking 107 seconds compared to 201 seconds.

Ratio 32 : Both approaches incur further slowdowns. Ternary's total reaches 249 seconds, whereas Binary climbs to 504 seconds. The probe phase dominates once again, at 99.9% for Ternary and 85.6% for Binary. Although Ternary's runtime rises sharply, it remains about twice as fast as Binary, confirming its resilience in limiting intermediate data materialization.

The Twitter graph is incredibly interconnected, so when using a binary approach, we end up with an overwhelming number of possible edge pairs. This situation poses a real challenge for join processing, particularly when memory is limited. However, the ternary approach cleverly sidesteps the need to create all of these extra tuples, which leads to much faster processing.



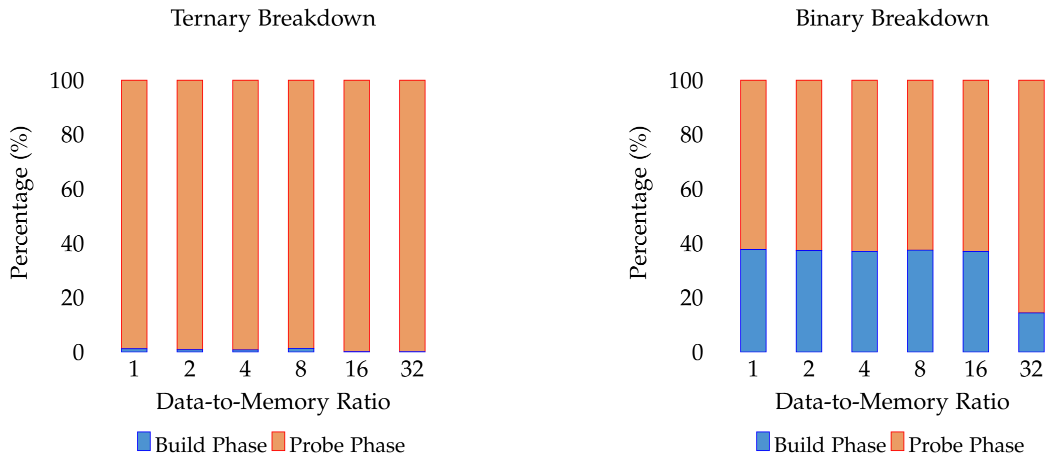


Figure 6.4: Twitter Dataset (2 million edges):
Top: Log-scale line graph of Total Execution Time for Grace Ternary Join (blue circles) and Grace Binary Join (orange squares, dashed).
Bottom Left: Stacked bar chart of the Grace Ternary Join breakdown (Build vs. Probe).
Bottom Right: Stacked bar chart of the Grace Binary Join breakdown (Build vs. Probe)

6.5.3 Wikipedia Dataset

Table 6.5: Execution Times for the Wikipedia Dataset (5 million edges)

Data-to-Memory Ratio	Grace Ternary Join			Grace Binary Join		
	Build (s)	Probe (s)	Total (s)	Build (s)	Probe (s)	Total (s)
1	0.351	32.165	32.517	47.638	36.538	84.177
2	0.370	28.958	29.330	51.590	43.125	94.717
4	0.386	31.450	31.837	49.187	43.187	92.212
8	0.323	29.852	30.176	45.745	34.215	79.962
16	0.409	212.453	212.863	51.362	40.217	91.580
32	0.393	410.591	410.983	49.381	824.371	873.753

Wikipedia dataset has around 5 million edges. As demonstrated in Table 6.5 and Figure 6.5, both Grace Ternary and Grace Binary joins achieve total execution time within two minutes for ratios up to 8. However, their performance notably diverges at ratios of 16 and 32, mainly due to differences in partitioning overhead, bucket overflows with recursive join processing overhead and the sizes of intermediate results for binary join.

Ratios 1–8 : At ratios 1 through 8, Ternary clearly outperforms Binary Join Strategy:

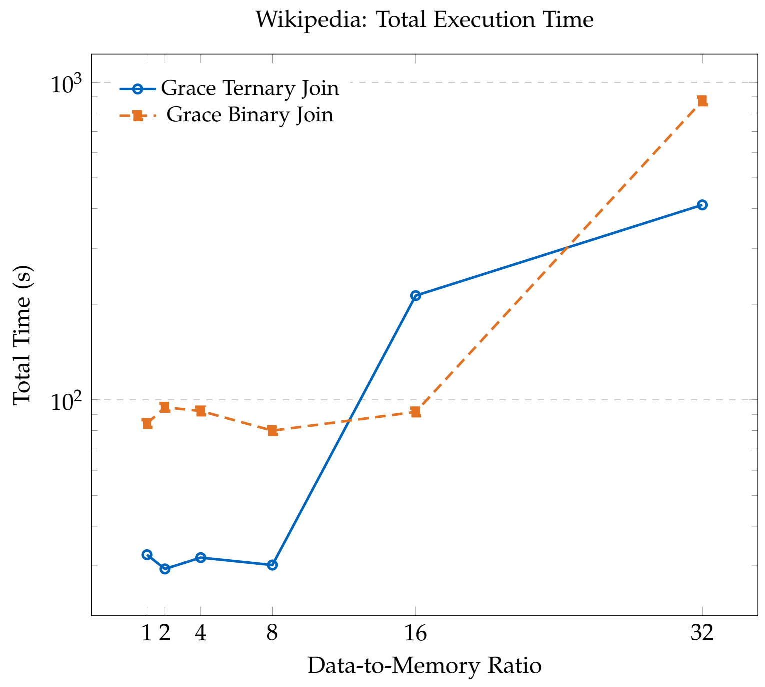
- **Ratio 1:** Ternary finishes in 32.52 s, vs. Binary at 84.18 s.
- **Ratio 2–4:** Ternary remains around 29–32 s total, whereas Binary is 92–95 s.
- **Ratio 8:** Ternary sits at 30.18 s, while Binary drops to 79.96 s (its best in these ranges).

In each of the four scenarios, Ternary significantly outperforms Binary, completing tasks in about one-half to one-third of the time. This efficiency is largely due to Binary's generation of a substantial $(R \bowtie S)$ intermediate that requires additional partitioning and probing.

Ratio 16 : A remarkable increase is observed for Ternary at a ratio of 16, where the total time rises to an impressive 212.86 seconds, with the probe phase alone accounting for 212.45 seconds—over 99% of the total time. In contrast, Binary maintains a total time of 91.58 seconds. This suggests that, at this ratio, Ternary faces significant recursion or skew, leading to a notable slowdown, while Binary's performance remains more manageable, staying under two minutes.

Ratio 32 : At a ratio of 32, Ternary achieves a processing time of 410.98 seconds, while Binary significantly escalates to 873.75 seconds—over twice as long as Ternary. This results in Binary generating a substantially large partial join output, requiring multiple passes and leading to a dominant probe phase of 824.37 seconds. Although Ternary incurs a notable probe cost of 410.59 seconds, it effectively mitigates the worst-case scenario by continuously partitioning and intersecting buckets in real time.

When memory demands are low to moderate, our Grace Ternary approach really shines—often running two to three times faster than Grace Binary. At a ratio of 16, however, Ternary experiences a noticeable slowdown (212 seconds versus Binary's 92 seconds), likely due to either data skew or the extra work from repeated recursive partitioning. But when memory becomes a scarce resource (ratio 32), Binary's extra overhead takes more time, running in 874 seconds, while Ternary manages to stay efficient at 411 seconds. Overall, these results show that Ternary not only cuts down on the amount of intermediate data written but also handles limited memory environments much more effectively than Binary.



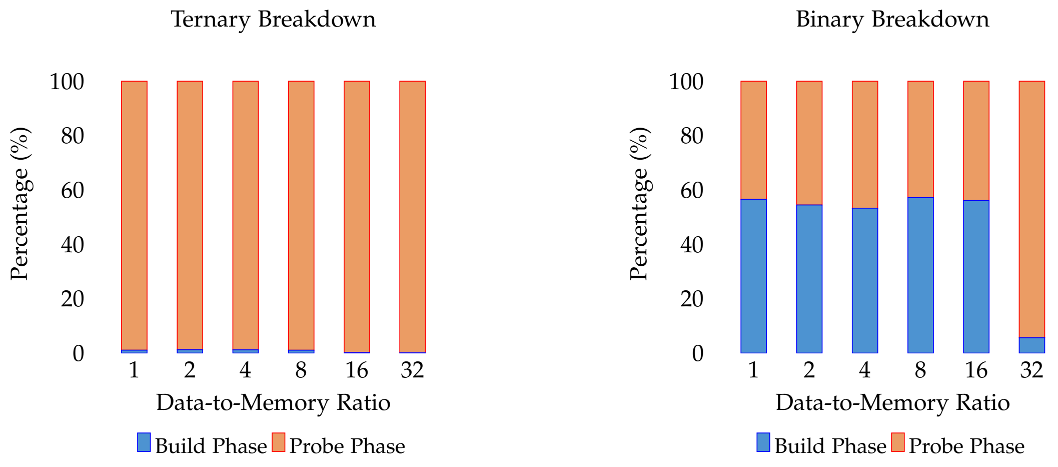


Figure 6.5: Wikipedia Dataset (5 million edges):
Top: Log-scale line graph of Total Execution Time for Grace Ternary Join (blue circles) and Grace Binary Join (orange squares, dashed).
Bottom Left: Stacked bar chart of the Grace Ternary Join breakdown (Build vs. Probe).
Bottom Right: Stacked bar chart of the Grace Binary Join breakdown (Build vs. Probe).

6.5.4 Twitch Dataset

Table 6.6: Execution Times for the Twitch Dataset

Data-to-Memory Ratio	Grace Ternary Join			Grace Binary Join		
	Build (s)	Probe (s)	Total (s)	Build (s)	Probe (s)	Total (s)
1	0.481	38.936	39.418	95.484	109.143	204.627
2	0.551	39.436	41.758	97.584	108,676	206.260
4	0.431	38.636	37.748	93.984	116.422	210.406
8	0.571	39.336	40.308	97.284	103.846	201.130
16	0.401	208.400	231.330	93.284	120.734	214.020
32	0.521	490.000	498.390	97.984	402.73	500.724

Table 6.6 and Figure 6.6 present the execution time results for the Twitch dataset, which consists of approximately 6 million edges. The data shows that *Grace Ternary* significantly outperforms *Grace Binary* when memory constraints are moderate (ratios 1 through 8). However, this performance gap narrows or even reverses at ratios beyond 16.

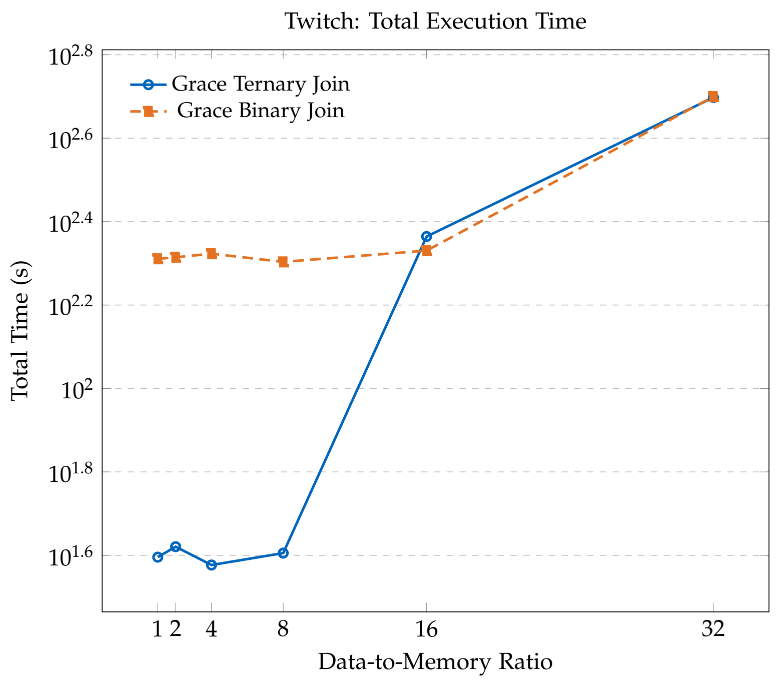
Ratios 1–8: For ratios 1, 2, 4, and 8, our observations are quite encouraging. The Ternary method consistently finishes processing in just 37 to 42 seconds, whereas the Binary method takes between 200 and 210 seconds. In other words, Ternary runs about five times faster under these moderate memory conditions. This efficiency is clear from the bottom-left bar chart in Figure 6.6: Ternary's build phase only accounts for less than 2% of the total processing time, thanks to its effective single-pass partitioning approach, while its probe phase consumes nearly all of the remaining time. In contrast, the Binary method splits its time more evenly, with the build phase taking around 45 to 48% and the probe phase 52 to 55%. The longer durations in the Binary approach are largely due to the larger intermediate results created during the partial join, which slows both its build and probe phases.

Ratio 16: At a ratio of 16, the Ternary approach takes about 231.33 seconds overall, with most of that time—208.40 seconds—spent on probing the data. On the other hand, the Binary method finishes slightly faster at 214.02 seconds. This suggests that under tighter memory conditions, Ternary may encounter deeper levels of recursion, especially with uneven data distribution, which lessens its usual advantage. Meanwhile, although Binary has its own overhead, it grows more gradually because the dataset doesn't produce overwhelmingly large intermediate results at this ratio.

Ratio 32: Both methods take over 200 seconds when memory is even less relative to the data. Ternary performs at 498.39 seconds, while Binary takes 500.72 seconds. The two approaches show similar performance, with Ternary being just 2 seconds faster overall. The bar charts reveal that Ternary spends only 0.1% of its time on building and 99.9% on probing, whereas

Binary's probing time increases to over 80%. Despite Ternary's significant increase in earlier ratios, it ultimately performs comparably to Binary in this worst-case scenario.

In summary, our evaluation shows that the Grace Ternary Join consistently outperforms the Grace Binary Join, running roughly five times faster when dealing with ratios between 1 and 8. This speed boost comes from the Ternary method's lower intermediate overhead and its efficient, one-pass partitioning strategy. However, at ratio of 16 changes slightly—the Ternary method takes about 231 seconds, compared to the Binary method's 214 seconds. This slowdown suggests that factors like limited memory and data skew can increase the cost of recursive join. When the ratio climbs to 32, both methods finish in around 500 seconds, indicating that under tight memory conditions, their performance becomes more similar. Overall, while the Ternary approach is generally more efficient by avoiding excessive intermediate writes and keeping memory usage moderate, both techniques tend to face similar challenges when the data is highly skewed or memory is very constrained.



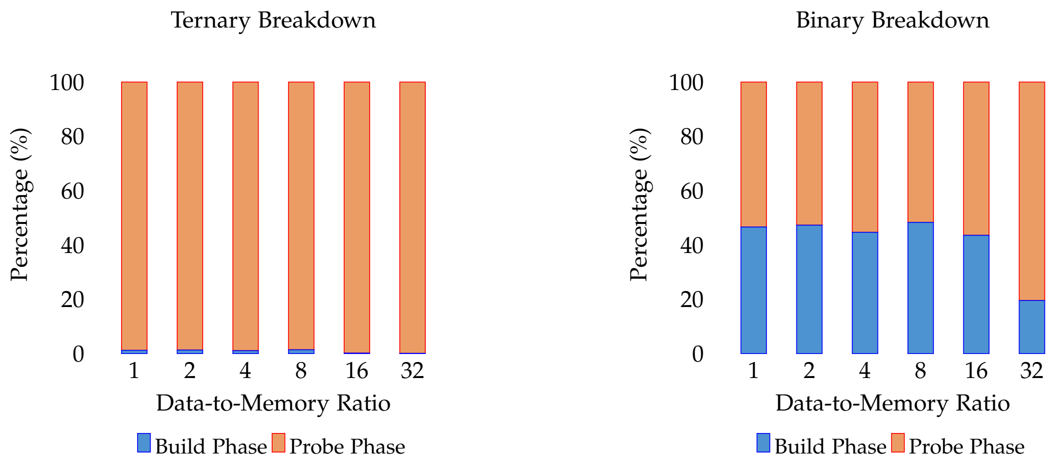


Figure 6.6: Twitch Dataset: **Top:** Log-scale line graph of Total Execution Time (s) for Grace Ternary Join (blue circles) and Grace Binary Join (orange squares, dashed).
Bottom Left: Stacked bar chart of the Grace Ternary Join breakdown (percentage of Build and Probe phases).
Bottom Right: Stacked bar chart of the Grace Binary Join breakdown.

6.5.5 Live Journal Dataset

Table 6.7: Execution Times for the LiveJournal Dataset

Data-to-Memory Ratio	Grace Ternary Join			Grace Binary Join		
	Build (s)	Probe (s)	Total (s)	Build (s)	Probe (s)	Total (s)
1	7.659	719,293	726.952	N/A	N/A	N/A
2	7.100	719.998	727.100	N/A	N/A	N/A
4	7.840	723.900	731.740	N/A	N/A	N/A
8	7.170	722.197	729.270	N/A	N/A	N/A
16	7.085	2949.400	2956.485	N/A	N/A	N/A
32	6.658	8412.100	8418.758	N/A	N/A	N/A

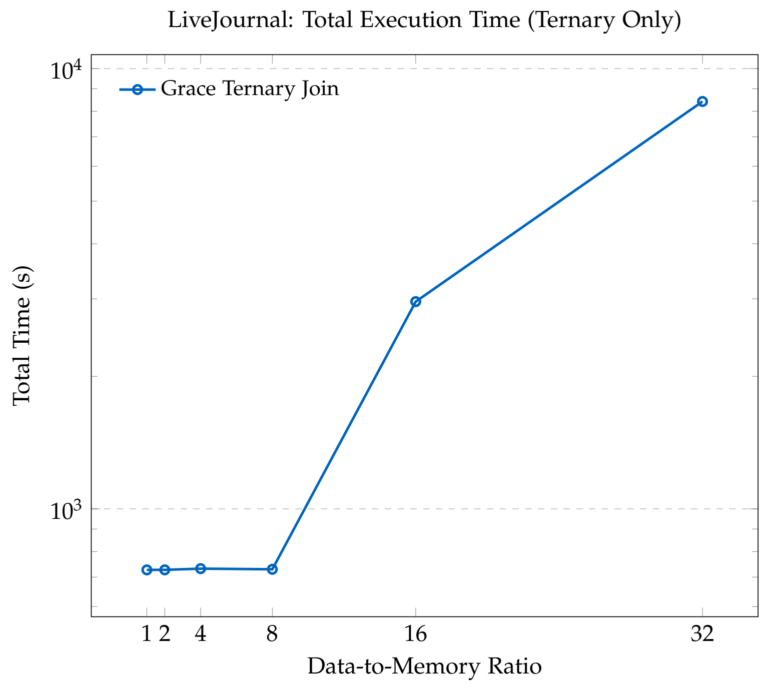
The LiveJournal social network represents our largest dataset, containing approximately 68 million edges and a very high triangle count. As shown in Table 6.7 and Figure 6.7, the *Grace Ternary Join* successfully completes the triangle query at all data-to-memory ratios. In contrast, the *Grace Binary Join* fails across the board (N/A) because it struggles to handle the intermediate results. With the number of matching tuples equating to 5 billion entries, the materialization of these intermediate results amounts to 189 GB, which exceeds our hardware capabilities. Nonetheless, the absolute runtimes for the Ternary Join are still considerable.

Ratios 1–8: For ratios between 1 and 8, the Ternary method consistently completes in about 727 to 732 seconds—roughly 12 minutes overall. The build phase is very swift, taking just 7 to 8 seconds (only 1–2% of the total time), while the probe phase consumes the bulk of the processing time, lasting between 719 and 724 seconds. The uniform runtime across these ratios indicates that no recursive partitioning is needed, as each bucket easily fits within the memory limits. Even when the memory budget is reduced, the Ternary approach continues to manage each bucket efficiently.

Ratio 16: At a ratio of 16, where the dataset is 16 times larger than the available memory, the total execution time for Ternary jumps to around 2,956 seconds, roughly four times longer than before. The bulk of this time, about 2,949 seconds, is spent in the probe phase, while the build phase remains very efficient under 8 seconds. This shows that Ternary successfully employs several levels of recursive partitioning to handle large, unevenly distributed data buckets. In addition, the high density of triangles combined with limited memory forces the system to make extra passes through the data, which significantly increases the CPU time during the probe stage.

Ratio 32: Under the strictest conditions, the Ternary strategy achieves a time of approximately 8419 seconds (over two hours). While this duration is extremely long in absolute terms, it remains the only strategy that successfully completes the task; the Grace Binary approach is not viable at any ratio. The build phase is brief, lasting only 6.66 seconds, which means that the majority of the time is spent on an extensive probing effort to address the number of potential matches.

In summary, our findings with LiveJournal illustrate that although the Grace Ternary Join may incur significant absolute runtimes under severe memory constraints, it proficiently performs in all tested scenarios by adeptly partitioning and managing skewed buckets. In contrast, the Grace Binary Join encounters critical failures due to the generation of excessively large intermediate data and the constraints of our SSD hard disk. This highlights the important insight that, for large, intricately connected graphs, the multi-way approach of the Ternary Join is not only more efficient but may indeed represent the sole effective strategy for achieving meaningful results.



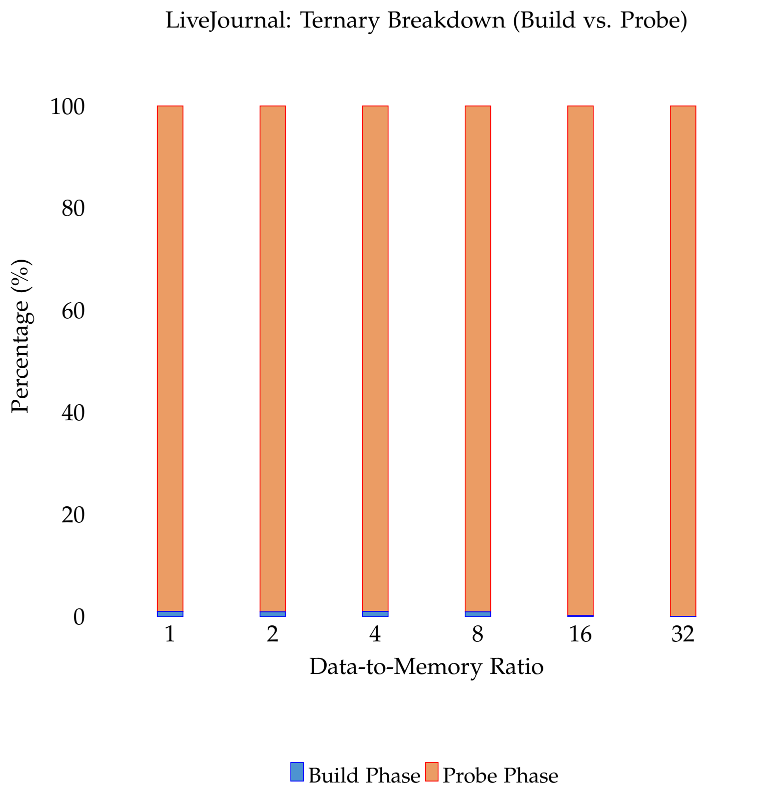


Figure 6.7: LiveJournal Dataset:
Top: Log-scale line graph of Total Execution Time (s) for the Grace Ternary Join.
Bottom: Stacked bar chart showing the breakdown of the Grace Ternary Join into Build and Probe phases.

7 Conclusion

In this thesis, we investigated the efficient execution of triangle queries – a prototypical three-way join problem – on large-scale graph data under realistic memory limitations. The core contribution of our work is the design and implementation of a push-based *Grace Ternary Join* operator that computes the triangle join in a single query operator. By extending the Grace hash join algorithm to three relations pipeline-query execution engine [5], we aimed to eliminate the redundant work and I/O overhead associated with traditional plans that compose binary joins. Our experiments and implementation confirm that this method effectively mitigates one of the major bottlenecks related to generating large intermediate results.

Through our evaluation on real-world graph datasets, we demonstrated the strengths and weaknesses of the proposed solution. The results showed that for datasets where a binary join plan would suffer from extreme intermediate result blow-up (hundreds of times larger than the input) , our Grace Ternary Join markedly outperforms the conventional approach. In particular, we observed up to a 1.8× reduction in query execution time on such workloads compared to DuckDB [1] as for Twitter dataset. This performance gain stems from our operator's ability to process all three input relations together, thereby avoiding the cost of writing large temporary join outputs to disk and reading them back in the next join stage. Even in scenarios where the advantage was less pronounced, our method matched the baseline with only minimal overhead (on the order of a few hundred milliseconds difference) , indicating that the push-based ternary join is efficient even when its benefits are not fully utilized. Moreover, the comparative study under varying memory availability provided deeper insight into each method's behavior. When enough memory was available relative to the dataset size, both the Grace Ternary and Grace Binary joins performed similarly fast on triangle queries, as expected. However, under severely constrained memory (e.g., when available memory size is only a few percent of the data size), we found that the ternary join experienced significant performance degradation on certain inputs . In our experiments with a small graph at extremely high data-to-memory ratios, the ternary operator had to invoke multiple levels of recursive partitioning, causing the join's probe phase to dominate the runtime (accounting for over 99% of total execution time at the worst ratio) . In contrast, the binary join plan handled the same scenario with negligible overhead , since each binary join dealt with smaller partitions and did not require deep recursion. This outcome highlights a key limitation: although our unified three-way join approach excels at containing intermediate data growth, it can incur extra overhead from recursive join when memory is extremely scarce or data is heavily skewed.

Reflecting on these findings, we conclude that the Grace Ternary Join is a promising technique for 3-way join processing, offering clear benefits in many practical cases of triangle discovery and similar queries. It effectively resolves the main performance issue by maintaining a fully pipelined join and avoiding large intermediate results. At the same time, the approach is not a silver bullet for all scenarios. There is an inherent trade-off between the granularity of partitioning and the cost of managing additional complexity in a single 3-way operator. Our evaluation indicates that for relatively moderate-sized intermediates or when memory is plentiful, a well-tuned binary join plan (especially in an in-memory database) can be very competitive, sometimes even faster due to its simpler processing pipeline and lack of recursive join overhead. Therefore, an ideal query processor might dynamically choose between a sequence of binary joins and a 3-way join operator based on the data characteristics and system resources.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] M. Raasveldt and H. Mühleisen. "DuckDB: an Embeddable Analytical Database". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1981–1984. ISBN: 9781450356435. DOI: 10.1145/3299869.3320212. URL: https://doi.org/10.1145/3299869.3320212.

[2] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. "Application of hash to data base machine and its architecture". In: *New Generation Computing* 1 (1983), pp. 63–74.

[3] T. Neumann and G. Weikum. "RDF-3X: a RISC-style engine for RDF". In: *Proc. VLDB Endow.* 1.1 (Aug. 2008), pp. 647–659. ISSN: 2150-8097. DOI: 10.14778/1453856.1453927. URL: https://doi.org/10.14778/1453856.1453927.

[4] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. "Adopting worst-case optimal joins in relational database systems". In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 1891–1904. ISSN: 2150-8097. DOI: 10.14778/3407790.3407797. URL: https://doi.org/10.14778/3407790.3407797.

[5] T. Neumann. "Efficiently compiling efficient query plans for modern hardware". In: *Proc. VLDB Endow.* 4.9 (June 2011), pp. 539–550. ISSN: 2150-8097. DOI: 10.14778/2002938.2002940. URL: https://doi.org/10.14778/2002938.2002940.

[6] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, Sixth Edition*. McGraw-Hill Book Company, 2011. ISBN: 978-0-07-352332-3. URL: https://www.db-book.com/db6/index.html.

[7] G. Moerkotte and T. Neumann. "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products". In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 930–941.

[8] G. Moerkotte and T. Neumann. "Dynamic programming strikes back". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: Association for Computing Machinery, 2008, pp. 539–552. ISBN: 9781605581026. DOI: 10.1145/1376616.1376672. URL: https://doi.org/10.1145/1376616.1376672.

[9] M. Galkin, K. M. Endris, M. Acosta, D. Collarana, M.-E. Vidal, and S. Auer. "SMJoin: A Multi-way Join Operator for SPARQL Queries". In: *Proceedings of the 13th International Conference on Semantic Systems*. Semantics2017. Amsterdam, Netherlands: Association for Computing Machinery, 2017, pp. 104–111. ISBN: 9781450352963. DOI: 10.1145/3132218.3132220. URL: https://doi.org/10.1145/3132218.3132220.

[10] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. *Worst-case Optimal Join Algorithms*. 2012. arXiv: `1203.1952 [cs.DB]`. URL: `https://arxiv.org/abs/1203.1952`.

[11] A. Atserias, M. Grohe, and D. Marx. *Size bounds and query plans for relational joins*. 2017. arXiv: `1711.03860 [cs.DB]`. URL: `https://arxiv.org/abs/1711.03860`.

[12] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. "Implementation techniques for main memory database systems". In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD '84. Boston, Massachusetts: Association for Computing Machinery, 1984, pp. 1–8. ISBN: 0897911288. DOI: `10.1145/602259.602261`. URL: `https://doi.org/10.1145/602259.602261`.

[13] H. Zeller and J. Gray. "An Adaptive Hash Join Algorithm for Multiuser Environments." In: *VLDB*. Vol. 90. 1990, pp. 186–197.

[14] A. Atserias, M. Grohe, and D. Marx. "Size Bounds and Query Plans for Relational Joins". In: *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '08. USA: IEEE Computer Society, 2008, pp. 739–748. ISBN: 9780769534367. DOI: `10.1109/FOCS.2008.43`. URL: `https://doi.org/10.1109/FOCS.2008.43`.

[15] T. L. Veldhuizen. *Leapfrog Triejoin: a worst-case optimal join algorithm*. 2013. arXiv: `1210.0481 [cs.DB]`. URL: `https://arxiv.org/abs/1210.0481`.

[16] A. Hogan, C. Riveros, C. Rojas, and A. Soto. "A Worst-Case Optimal Join Algorithm for SPARQL". In: *The Semantic Web – ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part I*. Auckland, New Zealand: Springer-Verlag, 2019, pp. 258–275. ISBN: 978-3-030-30792-9. DOI: `10.1007/978-3-030-30793-6_15`. URL: `https://doi.org/10.1007/978-3-030-30793-6_15`.

[17] D. Arroyuelo, A. Hogan, G. Navarro, J. L. Reutter, J. Rojas-Ledesma, and A. Soto. "Worst-Case Optimal Graph Joins in Almost No Space". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 102–114. ISBN: 9781450383431. DOI: `10.1145/3448016.3457256`. URL: `https://doi.org/10.1145/3448016.3457256`.

[18] D. Arroyuelo, D. Campos, A. Gómez-Brandón, G. Navarro, C. Rojas, and D. Vrgoč. "Space & Time Efficient Leapfrog Triejoin". In: *Proceedings of the 7th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. GRADES-NDA '24. Santiago, AA, Chile: Association for Computing Machinery, 2024. ISBN: 9798400706530. DOI: `10.1145/3661304.3661898`. URL: `https://doi.org/10.1145/3661304.3661898`.

[19] D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. "Join Processing for Graph Patterns: An Old Dog with New Tricks". In: *Proceedings of the GRADES'15*. GRADES'15. Melbourne, VIC, Australia: Association for Computing Machinery, 2015. ISBN: 9781450336116. DOI: `10.1145/2764947.2764948`. URL: `https://doi.org/10.1145/2764947.2764948`.

[20] A. Birler, T. Schmidt, P. Fent, and T. Neumann. "Simple, Efficient, and Robust Hash Tables for Join Processing". In: *Proceedings of the 20th International Workshop on Data Management on New Hardware*. DaMoN '24. Santiago, AA, Chile: Association for Computing Machinery, 2024. ISBN: 9798400706677. DOI: `10.1145/3662010.3663442`. URL: `https://doi.org/10.1145/3662010.3663442`.

[21] B. Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013.

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

[23] J. McAuley and J. Leskovec. "Learning to discover social circles in ego networks". In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 539–547.

[24] B. Rozemberczki and R. Sarkar. *Twitch Gamers: a Dataset for Evaluating Proximity Preserving and Structural Role-based Node Embeddings*. 2021. arXiv: `2101.03091 [cs.SI]`.

[25] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. "Group formation in large social networks: membership, growth, and evolution". In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. Philadelphia, PA, USA: Association for Computing Machinery, 2006, pp. 44–54. ISBN: 1595933395. DOI: `10.1145/1150402.1150412`. URL: `https://doi.org/10.1145/1150402.1150412`.