SPEED UP STREAMING SUBGRAPH ISOMORPHISM BY VECTOR DATABASES

Fangyue Chen (z5175111)

Supervisor: Zhengyi Yang

School of Mathematics and Statistics
UNSW Sydney

September 2023

Submitted in partial fulfilment of the requirements of
Data Science and Decisions

Abstract

Graph databases have become a crucial tool for representing complex relationships across various domains. Among graph analysis operations, subgraph isomorphism matching - identifying specific pattern instances within a larger graph - is a computationally challenging task, especially when dealing with large volumes of tasks. This paper introduces a novel approach to manage a batched stream of subgraph isomorphism queries, which acknowledges the limitations of existing kd-tree cache storage structures for similarity search. In a significant departure from traditional methods, we leverage advanced techniques such as Inverted File (IVF) and Product Quantization (PQ) to accelerate the k-nearest neighbors' search. This approach significantly optimizes the task processing time. This innovative integration capitalizes on high-dimensional vector representations of subgraphs obtained from established graph embedding techniques, thereby transforming the computationally rigorous subgraph isomorphism problem into a similarity search task within the vector database. This transformation boosts efficiency and scalability within realistic memory and processing constraints. Experimental outcomes using real-world datasets affirm that our methodology significantly enhances the performance of the batched streaming subgraph framework, thereby showing potential for handling extensive subgraph isomorphous query streams.

Contents

1 Introduction

Graphs, as a versatile data structure, are increasingly employed to represent intricate relationships in numerous domains such as social networks [1, 2] and biological systems [3, 4, 5]. They deviate from traditional databases by placing emphasis on relationships between entities via an edge-centric approach. This way, data is embodied through vertices and edges, imbued with various attributes to offer a holistic, interconnected view. The emergence of graph database management systems (GDBMS) has proven vital in scenarios necessitating the processing of related data and where relational contexts are crucial. This shift towards graph-based data representation has paved the way for deeper insights and revolutionized data handling. Hence, proficient management and analysis of graph-structured data has become pivotal due to the potential depth of understanding these structures offer.

Subgraph isomorphism is a fundamental operation in graph analysis that aims to identify all instances of a specific pattern, represented as a query graph Q, within a larger graph G. It stands as a more refined counterpart to subgraph matching, sharing the goal of uncovering patterns and anomalies in the data graph and thereby facilitating a deeper understanding of the underlying system. However, unlike subgraph matching, subgraph isomorphism adheres to a stricter condition. A subgraph is considered a match only if there exists a one-to-one correspondence between its vertices and edges and a subset of vertices and edges in the larger graph, preserving the exact connection structure. It is also a complex, multifaceted problem, presenting a spectrum of unique challenges across varied environments. Its solutions may differ based on whether the graph is static or dynamic, as well as on the nature and precision of queries—ranging from single, one-time queries to continuous streams or batches, and from exact to approximate matching. Given its intricacy, existing solutions are usually customized to suit the specific requirements of their settings.

Given the exacting nature of this operation, the subgraph isomorphism problem is classified as NP-Hard. This classification underscores the significant computational challenges it poses, especially when dealing with large-scale graphs. In scenarios involving large query graphs, the subgraph isomorphism problem exhibits a worst-case run-time complexity of $O(N! \cdot N^2)$, implying that the time complexity escalates factorially with the number of nodes (N). This results in it becoming computationally prohibitive for larger graphs. Conversely, for small query graphs comprising k nodes, the worst-case run-time complexity is $O(N^k)$ [6], indicating that time complexity grows exponentially with the number of nodes in the query graph. Even if the overall graph size is relatively small, this can still prove demanding for larger values of k. As the volume of data continues to grow, the challenges in efficiently handling large-scale data become increasingly pressing. This includes the urgent need to develop more efficient solutions for subgraph isomorphism, a focus that this research is poised to address.

This work is under the setting of streaming environments [6] where continuous query streams need to be processed in real time which has a wide range of usage scenarios. For instance, in bioinformatics [7, 8], researchers might periodically query a stable protein-protein interaction network to find subgraphs that match certain biological pathways of interest. In cybersecurity [9, 10], the network of computers and their connections can be viewed as a graph, and security analysts might continuously query this graph to identify patterns associated with security threats. In addition, its use also involves the field of social network analysis [11, 12, 13], transportation and logistics [14, 15], etc. Many users and systems often issue subgraph isomorphic queries simultaneously, and only the use of algorithms that efficiently handle the flow of subgraph isomorphism queries can provide timely insight and make decisions more effective. However, traditional subgraph isomorphism algorithms, designed primarily for static settings and one-time queries, are not equipped to handle such needs. This calls for frameworks that can efficiently handle streaming subgraph isomorphism queries. A significant advantage of the streaming query setting is it introduces a whole new layer of dynamism, robustness, and responsiveness.

In light of these challenges, the primary objective of this study is to enhance the performance of the streaming subgraph framework. Specifically, we focus on efficiently managing large-scale subgraph isomorphic query streams that are assumed to arrive continuously in fixed time periods. This becomes especially crucial in high-volume settings, where batch processing of queries is often the norm. This thesis will articulate the entire process and contribute as follows, starting with a clear definition of the problem setting in Section 2:

- **Utilization of Established Graph Embedding Techniques (Section 4):** The foundation of our work incorporates the use of established node, edge, and subgraph embedding techniques, acknowledged to perform on par with Weisfeiler-Lehman isomorphism testing. These techniques are leveraged to construct high-dimensional vector representations of the subgraphs, which are then indexed and searched within the vector database.

- **Vector Database Integration to Overcome Conventional Limitations (Section 5):** Traditional industry frameworks predominantly use kd-tree structures to cache past results for subsequent reuse, offering certain efficiency benefits. However, they exhibit limitations, such as inefficiencies in high-dimensional spaces and challenges in managing a steady stream of batch tasks. To address these limitations, we propose the integration of a vector database into the framework. This approach leverages the proficiency of vector databases in handling high-dimensional data and effectively transforms the graph isomorphism problem into a vector similarity search issue. This integration not only overcomes the limitations of conventional methods but also significantly enhances the efficiency and scalability of the streaming subgraph framework, marking our first major contribution.

- **Acceleration Techniques for Enhanced Efficiency (Section 5):** Our second major contribution involves the application of techniques like Inverted File (IVF) and Product Quantization (PQ) to expedite the return of k-nearest neighbors. These methods significantly increase the speed of searches within the framework, addressing the need for efficient and timely processing of large-scale, streaming subgraph isomorphism tasks.

The remainder of this paper is structured as follows: Section 2 details the preliminary aspects of our problem setting, discussing its intricacies and challenges. In Section 3, we present related work, examining key literature that has helped shape the direction of our study. In the subsequent Section 4 lays out the existing methods, mainly focusing on current strategies deployed for handling streaming subgraph isomorphism tasks. And the core of our paper, where we introduce our optimized approach designed to enhance the existing methodologies discussed is in Section 5. Then in Section 6, we provide an in-depth experimental evaluation, juxtaposing the performance of our proposed solution with that of the existing work. This comparative study aids in illustrating the usability and advantages of our algorithm. The implications and limitations of our study, along with potential future directions, are discussed more extensively in the concluding sections.

2 Preliminaries

This thesis tackles the intricate problem of managing subgraph isomorphism streaming query subgraph isomorphism requests which are given in batch query format over time, a multifaceted issue that intertwines various aspects of graph theory, database management, and real-time data processing. Central to this task is the challenge of executing subgraph isomorphism tasks, an NP-hard operation that checks for the presence of specific subgraph patterns within larger static, labeled, undirected graphs. In the domain of data processing, one of the emerging challenges lies in the intersection of streaming single queries and batch query requests. Traditionally, these requests have been handled independently; either as single queries in a streaming manner or as batch query requests. However, a paradigm shift is underway, calling for an integrated approach to manage this combination, to cope with the task of massive data processing.

The primary goal of this work is to design an approach that provides an approximate return in scenarios of high demand, maintaining the balance between precision and computational resource expenditure. The proposed system is aimed to ensure low latency and high throughput, striving to process and respond to an extensive volume of requests promptly. Current methodologies tend to be binary in their approach: focusing on precision which incurs high latency, or optimizing for speed resulting in an approximate return. This venture bridge this divide, where the imperative need for an efficient system that can handle high volume requests, while concurrently maintaining low latency, has been significantly under explored.

The notations used throughout this thesis are defined.

In this setting, we consider a data graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each vertex $v \in V$ and edge $e \in E$ is associated with labels l from a finite label set Σ. The label could be k-dimensional real vectors, explained by R^k, so the attributed graph should be refer as (G, l) The graph G is undirected, implying that an edge $e(u, v)$ between two vertices u and v is identical

Notations	Descriptions
B and S	Batch query and streaming query request
$e(u, v)$	An edge between vertex u and vertex v
I	Subgraph isomorphism task
l	Label associated with vertices and edges
M	Mapping between vertex sets
q and G	Query graph and data graph
R^k	k-dimensional real vectors
V, E, and Σ	Vertex set, edge set, and label set
v_c	Vertices in a graph, c is identification label
α	Accuracy of approximate return
Λ	Latency

Table 1.1: Table of Notations

to the edge $e(v, u)$. Furthermore, the graph is static, suggesting that the graph's structure remains invariant over time.

Within this context, we define a subgraph isomorphism problem where the objective is to locate all instances of a smaller query graph, $q = (V_q, E_q)$, within the data graph G as the example show in Figure 1.1. An isomorphism is a bijective mapping $M : V_q \to V_G$ that satisfies the following conditions:

1) For every vertex $v \in V_q$, there is a corresponding vertex $M(v) \in V_G$ such that the labels of v and $M(v)$ are identical, i.e., $l(v) = l(M(v))$.

2) For every edge $e(u, v) \in E_q$, there exists an edge $e(M(u), M(v)) \in E_G$.

These two conditions ensure that the isomorphic subgraph Q is a valid subgraph of G, meaning that it has the same structure and labels as a part of G. The set of all such isomorphisms from q to G is denoted as $M(q, G) = \{M_1, M_2, ..., M_k\}$.

The tasks to be processed are a sequence of queries that arrive in a streaming fashion, and these queries are presented in batches at fixed time intervals. Each batch, represented by B_t, contains multiple query graphs $q_{t,i}$ for $i \in \{1, 2, ..., n\}$ where n is the total number of queries in the batch. Here, t denotes the time instance at which the batch arrives. Thus, we have a continuous stream of batches of queries $B = \{B_1, B_2, ..., B_m\}$ where m is the number of batches, and each B_t consists of multiple queries $q_{t,i}$. The subgraph isomorphism task, in this case, extends to locating the isomorphism of each $q_{t,i}$ within the static graph G for every batch B_t as it arrives. Therefore, for each batch B_t, we need to find
$M(B_t, G) = \{M(q_{t,1}, G), M(q_{t,2}, G), ..., M(q_{t,n}, G)\}$ where $M(q_{t,i}, G)$ is the set of all isomorphisms of the query graph $q_{t,i}$ within G. This problem setting is computationally challenging due to the inherent complexity of the subgraph isomorphism problem and the requirement for real-time processing in response to streaming input.

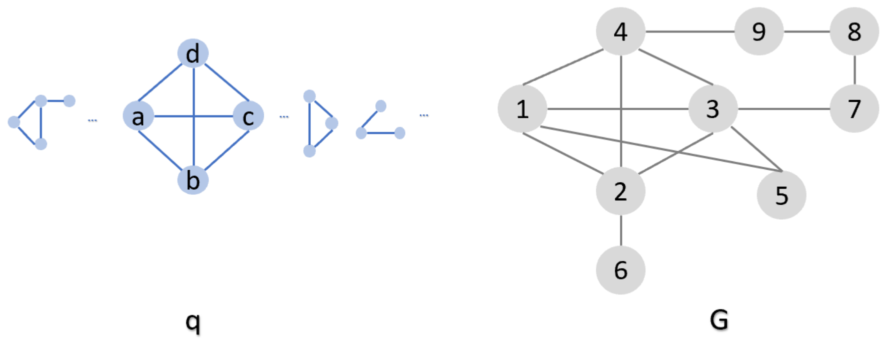


Figure 1.1: Subgraph Isomorphism schema
Use the example of one of the streaming query, called q, which contains four nodes from a to d inside. The left graph G is a simplified version of a static graph, and the task is to find all subgraphs in G that are isomorphic to q.
There are two instance here: G1 = [1, 2, 3, 4], and G2 = [1, 2, 3, 5].

A defining characteristic of streaming subgraph tasks is the frequent recurrence of queries due to inherent common patterns found in isomorphic subgraphs. Simultaneously, the probability of query overlap within a batch is high in this context. This overlap could lead to redundant computations, adding an additional layer of complexity to an already intricate problem. The redundancies emanate primarily from the repeated execution of the same subgraph isomorphism tasks for overlapping queries, underscoring the need for a careful and thoughtful optimization strategy. For instance, in chemical databases, users often query for prevalent molecular structures. As such, tasks within the current stream are likely to encompass identical or similar queries, independently issued by various users.

Such overlapping, coupled with the recurrence of similar queries, leads us to a strategic consideration: the history of past queries is likely to iterate in subsequent lookup tasks. This likelihood justifies the preservation of past queries for future utilization, potentially leading to computational efficiency. The objective of this thesis is to confront them head-on and devise an enhanced solution for the streaming batch subgraph isomorphism problem. The emphasis lies on reducing processing latency and optimizing resource use, all while ensuring the correctness of the results and leveraging the repetitive nature of queries to our advantage.

3 Related work

3.1 Graph Representation

The field of graph representation learning has seen significant advancements in recent years, with a multitude of methodologies and techniques being developed to address various challenges. Graph representation learning aims to map nodes, edges, and entire graphs into a low-dimensional space that preserves graph structural information and node attribute information. This mapping is crucial for many downstream tasks such as node classification, link prediction, and graph classification.

The most Traditional graph representations[16, 17] primarily relied on adjacency matrices or lists to denote the connections between nodes in the graph. While these methods are straightforward and effective for many computational tasks, they often fail to capture the more intricate topological properties of graphs and struggle with scalability issues for large graphs. The rise of deep learning led to a revolution in graph representation, primarily through Graph Neural Networks (GNNs). They have emerged as a powerful tool for graph representation learning. They leverage the graph structure and node features to generate node embeddings, capturing both local and global graph properties. GNNs have been applied in various real-world scenarios, including travel time prediction in navigation services, keyword matching in e-commerce's sponsored search platform, and recommendation systems, among others.

However, GNNs face several limitations [18], including expressive power, over-smoothing, and scalability. The expressive power of a model refers to its ability to distinguish between different graphs. Many GNNs, such as Graph Convolutional Networks (GCNs) and GraphSAGE, are bound by the Weisfeiler-Lehman (WL) test and fail to distinguish certain non-isomorphic substructures. The WL test operates iteratively. In each iteration, it aggregates the labels of a node's neighbours and then hashes them to generate a new label. Nodes are considered equivalent (or indistinguishable) if their labels match after a certain number of iterations. That for a graph $G = (V, E)$, let $L^0(v)$ be the initial label of a node $v \in V$ and $L^t(v)$ denote its label after t iterations. The label update rule in the t-th iteration can be described as: $L^t(v) = \text{hash}\left(L^{t-1}(v), \text{Sort}\left(\{L^{t-1}(u) | (v, u) \in E\}\right)\right)$. The WL test concludes when the labels stabilize, i.e., when they no longer change from one iteration to the next. If at the end of the process, two graphs have the same multi-set of node labels, they are considered indistinguishable by the WL test which limit the performance of GNNs.

To address this, Graph Isomorphism Networks (GINs) were proposed, which use the sum operator instead of the mean or max operators to aggregate the neighbors of a node, thereby increasing their expressive power. Over smoothing is another critical issue with GNNs that need to be considered in algorithm adoption, which occurs when the local information for each node is lost after passing through multiple layers, resulting in indistinguishable node embedding. For the challenge of scalability due to the large size and complexity of real-world graphs, current design principles for GNNs are both homophily and heterophily, including separation of ego and neighbor sampling, higher-order neighbors, and combining intermediate representations. These challenges continue to contribute to the development of the field of graph representation learning, and our understanding and utilization of graph structured data.

3.2 Subgraph Isomorphism

Subgraph isomorphism is a critical task in graph theory and has been the focus of extensive research. Subgraph matching, on the other hand, is a more general concept. It seeks to find all instances of a smaller graph (pattern graph P) within a larger graph G. In other words, subgraph matching extends beyond the requirement for structural identity (isomorphism) and can also involve semantic or attributed correspondences. For example, nodes and edges in P and G could be associated with labels or features, and a match could be based on the compatibility of these labels or features, not just the graph structure. Therefore, while subgraph isomorphism can be seen as a particular type of subgraph matching (where the matching criterion is structural identity), the latter encompasses a broader range of possibilities and can handle more complex scenarios where attributes or semantics are considered.However, subgraph isomorphism and subgraph matching often share a common framework, which include stages like embedding, filtering, and ordering and enumeration, in the practical context of algorithmic implementation.

Similar to graph representation, GNNs could be used to map the nodes of the subgraph into a dimensional space while preserving the structural information of the subgraph during the embedding stage. And the filtering stage is crucial for reducing the complexity of the subgraph isomorphism problem. It involves applying a series of constraints to prune the search space. These constraints can be based on local statistics of a vertex, such as degree (Statistics-based Filtering, SF), or non-local sub-structures, which can be checked by walking on graphs (Walking-based Filtering, WF)[19]. As one type of constraint may not be suitable for all queries and designing such constraints requires knowledge and experience, learning-based filtering models start to be used to handle complex non-local structures.

The traditional subgraph isomorphism technique **VF2** [20] begins its filtering stage by identifying potential candidate structures from the data graph G that could map to the query graph q. The use of node and subgraph similarity measures facilitates this search process. However, due to the exponential nature of the filtering process, which is proportional to the size of the data graph, it becomes computationally expensive and challenging to apply directly to continuous query flow tasks. To address the efficiency concerns of VF2, the VF2++ algorithm incorporates several enhancements. Initially, it selects a root vertex $u_r \in V(q)$ with the least frequent label in G, but with the largest degree, ensuring high uniqueness. VF2++ then constructs a breadth-first search (BFS) tree q_t rooted at u_r. At each depth i in q_t, denoted as $V(q_t)$, VF2++ adds query vertices to M in a depth-by-depth manner, starting from depth 0. In each iteration, the algorithm selects the next vertex u^* with the maximum product of its degree $N(u)$ and label frequency p_l. Mathematically, this is expressed as $u^* = \arg\max_{u \in V(q_t)} N(u) \cdot p_l$. In case of ties, the algorithm prioritizes vertices with the largest degree and the smallest label frequency in G. This systematic approach ensures an ordered and efficient exploration of potential subgraph isomorphisms, striking a balance between exploring the search space and the likelihood of finding correct matches. The algorithm's design, particularly its ordering scheme, significantly enhances the efficiency of continuous query flow tasks, overcoming the limitations of the original VF2 approach. However, it is important to note that the algorithm's performance is still influenced by the characteristics of the data and query graphs.

TurboISO [21], another batch-processing subgraph isomorphism technique, employs more advanced methods to filter candidate nodes, leading to a smaller processing time. It starts by constructing a Maximum Independent Set (MIS) of vertices from the query graph $q = (V_q, E_q)$. A MIS is a set of vertices $M \subseteq V_q$ where no two vertices are adjacent, and the set cannot be further extended by including additional vertices from V_q. That for any two vertices $u, v \in M$, there is no edge between u and v in q. For any vertex $u \in (V_q - M)$, there is at least one vertex $v \in M$ such that there is an edge between u and v in q.The purpose of constructing an MIS is to reduce the search space by identifying a set of representative vertices that can be matched independently.Subsequently, TurboISO picks a starting vertex from the MIS based on two principles. Firstly, the vertex must have a unique label, or a label that appears least frequently in the data graph $G = (V_g, E_g)$. Then vertex must have the highest degree amongst all vertices in the MIS. Formally, the starting vertex v_s from the MIS is selected as: $v_s = \arg\max_{v \in M}(d(v) \times f(L(v)))$ where $d(v)$ is the degree of vertex v, $L(v)$ is the label of vertex v, and $f(L(v))$ is the frequency of label $L(v)$ in G. Next, TurboISO uses a breadth-first search (BFS) strategy to decide the exploration order for the remaining vertices in q. The BFS starts from the starting vertex v_s and explores the vertices layer by layer. The improved candidate filtering of TurboISO is a two-step process. First, it utilizes a simple label checking mechanism for primary candidate pruning. Then it adopts a neighborhood-based filtering strategy for advanced candidate pruning. The comprehensive filtering results in a smaller and more manageable candidate set, significantly reducing the search space and hence the processing time.

The ordering and enumeration stage is an intermediate stage that aims at preparing a satisfactory searching plan for the subsequent enumeration algorithm. The profound effect of this stage on enumeration performance has been justified by previous research. According to the pruned graph, all the query vertices are ordered into a sequence to guide the searching process in the next stage. The goal is to greedily reduce the searching space by picking the most selective query vertex as the starting vertex. In the enumeration stage, starting from candidates for each vertex in the data graph, partial matches are expanded and joined according to the sequence computed in the last stage to form the final matches. This stage can be computationally intensive due to the large size and complexity of real-world graphs.

In conclusion, subgraph isomorphism and matching are complex task that involves several stages, each with its own challenges and solutions. Despite these challenges, significant progress has been made in recent years, with new methodologies and techniques being developed to address these issues.

3.3 High-dimension kNN Search

In the realm of high-dimensional k-nearest neighbors (kNN) search, both exact and approximate methods have been extensively studied. It has been a focal point of research due to its wide-ranging applications in fields such as e-commerce, network security, molecular biology, and industrial applications. These domains often represent their data as high-dimensional feature vectors, necessitating efficient high-dimensional techniques for processing. Approximate kNN search methods have been developed to address the challenges posed by high-dimensional data. The primary objective of these methods is to enhance search efficiency, often at the expense of accuracy. This trade-off between search efficiency and accuracy is a defining characteristic of approximate nearest neighbor (ANN) algorithms. The resultant nearest neighbors from these algorithms may not necessarily be the true k nearest neighbors, but this compromise is often acceptable in many practical applications where speed is a priority over absolute accuracy.

Several strategies have been employed to improve the performance of kNN search in high-dimensional space. One such strategy is parallelization, where the computation is divided into many parts that are processed concurrently on separate processing units. This method does not reduce the number of computations but distributes the task across multiple processing units, thereby speeding up the overall process. Another common strategy is dimensionality reduction (DR), which maps data points from a high-dimensional space to a lower-dimensional space. Searching in a lower-dimensional space is faster and more cost-effective, making DR a popular approach for dealing with high-dimensional data. Principal component analysis (PCA) [22] is a widely used dimensionality reduction technique due to its efficiency and scalability. Moreover, partitioning methods also play a crucial role in improving the performance of kNN search in high-dimensional space. The goal of these methods is to minimize distance computation and speed up the search. These approaches are typically classified into two broad categories: space-based and data-based.

Data-based partitioning strategy involves dividing a collection of data points into a number of groups. It creates k($N \leq k$) divisions of the data, each of which represents a cluster, where N indicates the number of data points. They work better for creating indexes because of their adaptability to data distributions. For example, the performance of techniques such as $\triangle$-tree, $\triangle$+-tree, HDR-tree, EkNNJ, etc., makes retrieval much faster in real-world environments especially for hetergeneous datasets.

On the other hand, space-based partitioning strategy involves dividing the space into two or more subsets or regions in a fixed way or in a way that changes over time. Tree-type data structures are often related to traditional Space Partitioning. Due to the fact that data distributions for a higher feature space are likely to be non-uniform and sparse, techniques based on regular space partitions can suffer from significant overhead costs. Several approaches use standard partitioning schemes for high-dimensional data spaces. For example, R-tree-based approaches represent the space partition via MBRs. One such technique that iteratively splits the original d-dimensional data space into 2d sub-spaces is the quad-tree. Other examples of space-based approaches are iDistance, the R-tree family, Kd-tree, STR, and Quad-tree.

The construction of a **k-d tree**[23] starts by selecting a dimension and a median value in that dimension. The data is then partitioned into two: one set of points that are less than the median in the selected dimension, and another set of points that are greater than or equal to the median. This process is recursively applied to each subset, choosing a new dimension each time. The result is a binary tree where each node partitions the data into two half-spaces. While this structure is efficient for nearest neighbor searches in multi-dimensional spaces, it's important to note that the complexity of creating the tree from an unstructured set of points is O(n log n), where n is the number of points. Furthermore, while kd-trees perform well with static data, they can become unbalanced and less efficient with high-dimensional or frequently changing data.

In order to solve the search task in high-dimensional space more efficiently, one optimized version is the Principal k-d tree (PKD-tree), which involves rotating the data points to align the principal axes with the coordinate axes before building the tree. This rotation preserves the subspace spanned by the k largest principal axes, which is particularly useful when using multiple trees. The original unprojected data is still used in testing the distance between the query point and candidate closest neighbors. Another optimization is Randomized k-d tree (RKD-tree), which introduces randomness into the selection of the partitioning dimension. Instead of always choosing the dimension with the greatest variance, the RKD-tree selects a dimension at random from among those with high variance. This approach retains a high probability that a query can be on either half of the node while maintaining backtracking efficiency. The RKD-tree performs as well as the standard k-d tree while having the same complexity level in storage.

The **Qd-tree**[24], or Query-Driven tree, is an advanced data structure that improves upon the k-d tree for data storage, particularly in the context of big data analytics. The Qd-tree is a data structure specifically engineered to optimize the number of blocks accessed by a given query workload, thereby reducing I/O cost and enhancing performance. It can be formally defined as follows: given a set of tuples V, a workload W of queries, a skipping function S, and a minimal block size b, the objective is to find a partitioning $\mathcal{P}$ that maximizes $C(\mathcal{P})$, subject to the constraint $|P_i| \leq b$ for all $P_i \in \mathcal{P}$. Here, $C(\mathcal{P}) = \sum_{P_i \in \mathcal{P}} C(P_i)$. The function $S(P, q)$ is a binary function that indicates whether a partition P can be skipped when processing a query q. The definition of S depends on the type of metadata maintained at each block. The most common type of metadata is the max-min filters, which are the maximal and minimal values of each dimension over all the tuples in a block. In this case, $S(P, q) = 1$ if the hypercube defined by the max-min filters intersects with the range of query q.

The Qd-tree introduces two key properties: semantic description and completeness. The semantic description provides a precise description of a block's contents as a predicate over table fields. The completeness property guarantees that a block contains all tuples matching the given description. These properties are crucial for accelerating block retrieval and for using blocks as a local cache or partial view over remote data. In terms of structure, the Qd-tree operates similarly to the k-d tree, partitioning data into different nodes based on certain criteria that takes into account the specific queries that will be run on the data. This query-driven approach allows the Qd-tree to optimize data storage for the specific needs of the application, leading to more efficient query performance. Moreover, the use of deep reinforcement learning builds better trees for a given workload through iterative trials, where the construction of tree agents is achieved through proximal policy optimization (PPO). It handles range comparisons by restricting a parent's hypercube description and handles equality comparisons by storing additional metadata. Each node stores a bit vector for each categorical column, representing the distinct values of this column. This approach makes it straightforward to process equality and IN cuts, thereby improving the efficiency of cache management. Qd-tree makes good use of the past query structure to optimize the storage scheme, but it was originally optimized for hard disk storage and is difficult to apply to tasks that require fast determination of similarity by computing distance, which is the claim of this project.

In conclusion, both data-based and space-based partitioning strategies have their unique advantages and are used in different scenarios based on the characteristics of the data and the specific requirements of the application. In the broader field of graph databases, there have been numerous other works focusing on subgraph isomorphism and in-memory storage structures [25, 26]. These works have explored various strategies for node and subgraph similarity computation, candidate filtering, and query processing [19, 27]. However, many of these approaches do not scale well to a dynamic setting of a continuous stream of queries, highlighting the need for further research and innovation in this area. In the following sections, we will present our proposed model and approach, which aims to address these limitations and enhance the efficiency of streaming subgraph isomorphism.

3.4 Vector Database

The advent of large-scale language modelling, generative artificial intelligence, and semantic search has necessitated a paradigm shift in data storage requirements. Traditional databases, which primarily consist of structured data tables containing symbolic information, are ill-equipped to handle the complexity and size of embedding data [28]. This limitation impedes insight extraction and real-time analyses. To address this challenge, vector databases have emerged, offering specialized processing, storage, and querying capabilities for embedding, making them ideally suited for storing and finding similar high-dimensional massive vectors [29].]

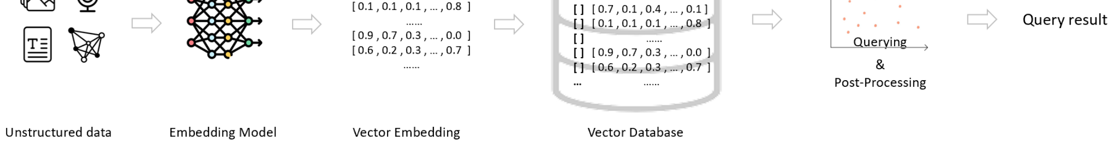


Figure 1.2: vector database structural

The operation of vector databases differs significantly from traditional databases. In the context of streaming query matching request and LLM such as ChatGPT, a user's query is inserted into the embedding model, which creates vector embeddings based on the content to be indexed. These embeddings are then stored in the vector database, which produces an output sent back to the user as a query result. Subsequent queries from the user undergo the same process, with the database queried for similar vector embeddings based on the original content. As shown in Figure 1.2, a vector database query undergoes three main stages: indexing, querying, and post-processing. During indexing, various algorithms map the vector embedding to data structures for faster searching. The querying stage involves comparing the queried vector to indexed vectors and applying the similarity metric to find the nearest neighbor. Post-processing, depending on the specific vector database, involves producing a final output to the query and possibly re-ranking the nearest neighbors for future reference.

One of the key technologies that make vector databases superior is the principle of a single level store. This principle implies that the on-disk representation of data is mirrored as closely as possible in vector format, which enables efficient querying and fast processing of large datasets while minimizing the need for data movement. This synchronization of data storage with the in-memory representation used to process it not only delivers high-speed performance but also opens up new possibilities for data-driven innovation [29].

Unlike traditional databases that store strings and numbers in rows and columns, vector databases work with vectors. They apply a similarity metric to find the vector most similar to the query. Vector databases comprise different algorithms that aid in the Approximate Nearest Neighbor (ANN) search [30], conducted via hashing, graph-based search, or quantization. The results are based on proximity or approximation to the query, with a trade-off between accuracy and speed . making them ideal for handling large indexes. Popular methods for building ANN indexes include techniques such as HNSW, IVF, and PQ, each of which improves performance in different ways, such as reducing memory resources or improving accuracy.

Furthermore, vector databases excel in handling high-speed ingestion of streaming data, which is crucial for applications such as IoT-connected devices [31]. They provide control over trade offs between volume, latency, and algorithmic filtering, enabling developers and administrators to tailor storage to their application's specific requirements. Finally, vector databases employ sharding and GPU support to handle large-scale database problems. When the number of queries or the size of the dataset surpasses a single machine's capabilities, data gets sharded. Sharding can help large queries run quickly across vast amounts of data, making large-scale search tasks more efficient and manageable.

In conclusion, vector databases offer several key advantages, including efficient data storage and retrieval, advanced indexing techniques, powerful filtering capabilities, high speed ingestion of streaming data, and effective handling of large scale database problems through sharding. These features make vector databases an ideal solution for handling large scale, high dimensional data, thereby fostering data driven innovation. Some of the most popular vector databases, as indicated by user preference and Github stars, include Pinecone, Chroma, DeepLake, and Qdrant.

> Pinecone, a cloud-based vector database, is recognized for its real-time indexing and searching, and its ability to handle both sparse and dense vectors. It supports exact and approximate nearest-neighbor search and integrates seamlessly with other machine learning frameworks and libraries, making it a preferred choice for production-grade NLP and computer vision applications.

> Chroma, an open-source vector database, provides a fast and scalable solution for storing and retrieving embeddings. It is user-friendly, lightweight, and supports multiple backends, including RocksDB and Faiss. Chroma's unique features include built-in support for compression and quantization, and the ability to dynamically adjust the database size to accommodate changing workloads, making it a popular choice for research and experimentation.

> DeepLake, another cloud-based vector database, is specifically designed for machine learning applications. It stands out for its built-in support for streaming data, real-time indexing and searching, and the ability to handle both dense and sparse vectors. It also provides a RESTful API and supports multiple programming languages, making it suitable for applications that require real-time indexing and search of large-scale, high-dimensional data.

> Qdrant, an open-source vector database, is designed for real-time analytics and search. It uniquely supports geospatial data and can perform geospatial queries. It also supports exact and approximate nearest-neighbor searches and includes a RESTful API and support for multiple programming languages. This makes Qdrant an excellent choice for applications that require real-time geospatial search and analytics.

Each of these databases offers unique features and capabilities, demonstrating the diversity and adaptability of vector databases to cater to various use cases and requirements.

4 Existing Method

Graph analysis often requires the comparison of various elements including nodes, edges, and subgraphs. Traditional techniques focus on structural aspects by indexing subgraphs such as paths, triangles, and cliques as references for comparison. However, these techniques may disregard the deep-rooted semantic information present in the graph. They can also be computationally demanding, particularly with large graphs or intricate subgraph patterns, as they necessitate exhaustive searches or enumeration of all possible subgraphs, thus elevating the computational cost.

The development of the streaming subgraph isomorphism framework, designed by Chi Thang Duong et al.[6], represents a notable stride in the field of graph databases. This work proposes an alternative - a graph embedding approach. Tailored to handle a continuous influx of subgraph isomorphic queries, this framework ingeniously addresses scalability concerns by facilitating the caching and repurposing of prior results.

Central to the framework's proficiency is its innovative use of a subgraph index founded on graph embeddings. This index significantly expedites traditional subgraph isomorphism algorithms, even in instances of cache misses, by enabling rapid similarity search and kNN search. The embeddings of nodes, edges, and subgraphs generated by the Message Passing Neural Network (MPNN) form the crux of this index, thereby converting intricate graph structures into dense vectors within a high-dimensional space. As a result, the time complexity doesn't scale linearly with the size of the dataset, rendering the method well-suited for large-scale operations. Furthermore, the application of strategic cache management policies that emphasize the reusability of query results further enhances the framework's performance. The value of these policies is evident in the substantial reduction in processing time across multiple datasets, thus highlighting the framework's aptitude for managing a continuous stream of queries.

Evaluations underline the efficacy of the streaming subgraph framework, which consistently outperforms traditional subgraph isomorphism techniques like VF2 and TurboISO. The superior performance of the framework is further demonstrated through comparisons with MQO [32], a batch processing technique. While MQO attempts to reuse results by indexing and conducting searches directly on graph structures, the advantage clearly lies with Duong's framework that leverages graph embeddings for efficiency. In essence, the streaming subgraph isomorphism framework provides a promising avenue for effectively handling large-scale graph database tasks.

This project strategically employs the *Efficient Streaming Isomorphism work* with Graph Neural Networks' as baseline, incorporating a significant enhancement to the cache solution for improved performance and efficiency. In subsequent chapters, I will explain the core technologies in this framework in detail.

4.1 Message Passing Neural Networks

Graph embeddings are a powerful means of representing nodes, edges, and subgraphs within a multidimensional space. In this space, structurally or semantically analogous entities cluster together. This facilitates comparative and inferential analyses that rely on geometric relationships, transcending the limitations of purely symbolic representations. Indeed, a d-dimensional embedding space of domain size k can uniquely represent k^d different concepts. Leveraging the capabilities of Message Passing Neural Networks (MPNNs) for graph indexing leads to the generation of indices for nodes, edges, and subgraphs based on these graph embeddings. In the context of graph indexing:

Node Embeddings: The output of this process is a set of embeddings, one for each node within the subgraph. Each of these embeddings encapsulates the node's local structure or receptive field that is the subgraph around the node. MPNNs are used to construct node embeddings, where both the labels of the nodes and their interconnections within the graph are taken into consideration. The labels capture external knowledge about the nodes while the connections provide crucial structural information. Combining these two types of information enriches the node embeddings, enhancing their expressiveness and usefulness.

The MPNN framework encompasses three crucial steps: sending, receiving, and updating. These steps enable information exchange between nodes, integrating data across the graph and capturing complex node and edge relationships. The 'sending' step involves transmitting a node's information to its neighbours. 'Receiving' then refers to a node integrating incoming messages from neighbouring nodes. Lastly, the 'updating' step pertains to the computation of new node states based on the received information, like shows in Figure 1.3. The process of generating node embeddings is mathematically described by two functions: a message function M and an update function U.

Message Function: M takes as input the states of a node and its neighbor, as well as the edge connecting them, and produces a message. Formally, for each edge (u, v) in the graph, the message receiving function could be defined as:

$$z_N^i(v) = M^i(m_u^i, \forall u \in N(v)) \tag{4.1}$$

where $m_u^i = f_s^i(z_u^i)$ is the sent message from neighbours and z_u^i is the representation of node u.

Update Function: U takes as input the state of a node and the aggregated messages from its neighbors to update the node's state. Formally, the update function is defined as

$$z_V^{i+1} = f_u^i(z_N(v)^i, x_u^i) \tag{4.2}$$

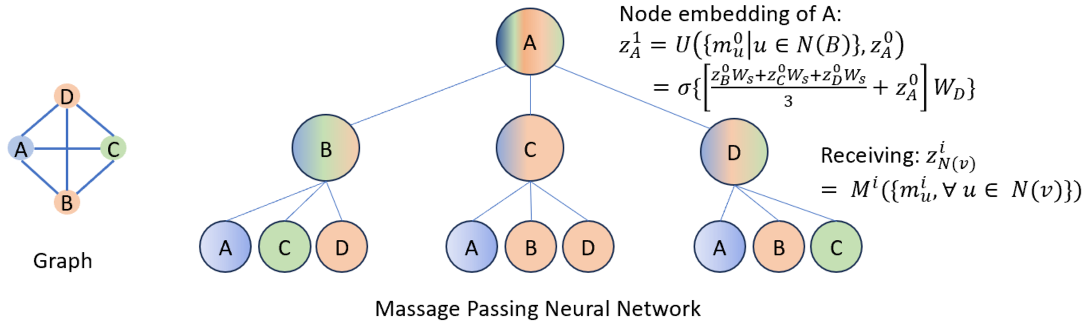

Node embedding of A:
$z_A^1 = U(\{m_u^0 | u \in N(B)\}, z_A^0)$
$= \sigma\{\left[\frac{z_B^0 W_s + z_C^0 W_s + z_D^0 W_s}{3} + z_A^0\right] W_D\}$
Receiving: $z_{N(v)}^i$
$= M^i(\{m_u^i, \forall u \in N(v)\})$
A
B
C
D
Graph
Massage Passing Neural Network

Figure 1.3: MPNN updating structure

The number of these message-passing rounds directly influences the receptive field of a node—the section of the graph that contributes to its final embedding representation. This enables the representation of the node to incorporate both its inherent attributes and the structural specifics of its proximal graph neighbourhood. By iteratively applying these functions, an MPNN can propagate and integrate information across the graph, allowing nodes to capture whole graph information.Once the node embeddings are generated, they serve as the foundation for edge embeddings. Edge embeddings are computed using the node embeddings of the two nodes connected by the edge. The edge embeddings capture the relationship between the two nodes, providing additional information that is not captured by the node embeddings alone.

Edge Embeddings: The construction of edge embeddings entails the averaging of the embeddings of the nodes connected by the edge. By incorporating information from the connected nodes within the edge embedding, this representation becomes a distinguishing feature for graph comparison. The edge embeddings capture both the individual characteristics of the nodes and the relationships between them. This embedding method effectively encapsulates local structural and attribute information, thereby providing a nuanced understanding of the graph's topology.

Subgraph Embeddings: The ultimate goal of this process is to generate meaningful subgraph embeddings which is achieved by projecting the model learned on the whole graph onto the respective nodes and their labels of the subgraph. This projection is akin to truncated message passing, where only the nodes in the subgraph send messages to neighboring nodes that are also in the subgraph. This process yields embeddings for all nodes in a subgraph, and since each embedding summarizes the node's receptive field, which is the subgraph, it is a candidate to represent the whole subgraph.

Then a compositional approach is followed where the edge embeddings are averaged to represent the subgraph in this work. This is also one of the key findings in the *Efficient Streaming Isomorphism work*. The edge embeddings provide a more comprehensive representation of the subgraph, capturing both the individual characteristics of the nodes and the relationships between them. Using averaged edge embeddings to represent the subgraph is equivalent to a degree-weighted combination of node embeddings. Whereas the traditional, direct averaging of node mappings in subgraphs loses information due to weight neglect.

Importantly, the model trained to embed subgraphs from one graph can be transferred to another graph. This permits the learned model for the data graph to also be applied to a query graph. By following this process, subgraph embeddings of the data graph and the embedding of the query graph are created using the same model. Therefore, a subgraph from the data graph that is isomorphic to the query graph will have an equivalent embedding.

The paper substantiates the hypothesis that a correlation exists between the Maximum Common Subgraph (MCS) size, subgraph edit distances, and embedding distances. This verification involves the creation of subgraphs with a particular 'k' edit distance by generating a two-hop ego graph from a random node in the data graph, and subsequently removing between 1 and 7 random edges to maintain connectivity. For assessing the MCS size, 5000 subgraph pairs are selected at random from an array of subgraphs, each of size 15, extracted from various datasets. Post the calculation of their MCS size, their subgraph embeddings are formulated to gauge the corresponding embedding distance. The findings indicate that an increase in the MCS size translates into a proportional increase in the subgraph embedding distance - an observation that holds true across all examined datasets. The strength of this relationship is further confirmed by Pearson's correlation values in Figure 1.4, indicating that subgraph embeddings are indeed reflective of structural similarity.

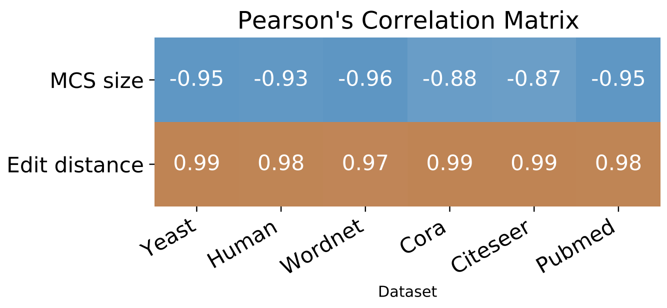


Figure 1.4: Pearson's Correlation Matrix

Parameter Learning: The success of the Message Passing Neural Network (MPNN) model hinges on the precision of its parameters. These parameters are optimized through a defined loss function, which guides the model to allocate similar embeddings to nodes that are proximate within the graph. Consequently, it enables the embeddings to encapsulate the structural and attribute similarities of nodes. Furthermore, the efficacy of the model can be boosted by the incorporation of an additional supervised loss function. This function aids in reconstructing the node labels, thereby improving the model's ability to capture the semantic characteristics of the nodes in addition to their structural features. The proper learning and tuning of these parameters are crucial to ensure that the MPNN model accurately reflects the underlying graph structure and semantics in the produced embeddings.

The Loss function is :

$$L(z_v) = -log(\sigma(z_v^T z_u)) - \mathbb{Q}\mathbb{E}_{u_n} \; P_n(v) log(\sigma(-z_v^T z_{u_n})) \tag{4.3}$$

In this scenario, v is referred to as a positive sample, which is typically a neighbor of u. On the other hand, u_n represents a negative sample that is derived from a negative sampling distribution P_n. $\mathbb{Q}$ is the count of negative samples involved. The function defined above aims to generate similar representations for nodes u and v that are similar, as evidenced by maximizing the dot product $z_v^T z_{u_n}$. Concurrently, it aspires to create dissimilar representations for nodes v and u_n that are not similar, which is achieved by minimizing the dot product $z_v^T z_{u_n}$. This differentiation helps in creating distinct embeddings for dissimilar nodes in the graph.

Proof of Effectiveness of MPNN: The paper of this framework has proved the effectiveness of using MPNN to realize the subgraph isomorphism task, by showing that it can perform comparably or even better than WL test in certain contexts. They share a similar iterative refinement process, but they differ in key areas which can significantly impact their effectiveness depending on the context.

Weisfeiler-Lehman (WL) test is a widely accepted algorithm for graph isomorphism which uses the node's neighborhood information to generate an iterative hashing mechanism. The labels of nodes are updated based on the sorted labels of their neighbors, thus capturing structural information of the graph. However, the WL test is limited to discrete, symbolic representations and does not inherently capture rich attribute information. Let $G = (V, E)$ be a graph where V is a set of nodes and E is a set of edges. The Weisfeiler-Lehman (WL) test can be described as follows:

1. **Initialization:** For each node $n \in V$, assign a unique compressed label $C_{0,n} = 1$.
2. **Iteration:** Repeat the following steps for $i = 1, 2, ..., N$, where N is the number of nodes, or until the partition of nodes by compressed labels does not change:
    (a) **Label Generation:** For each node $n \in V$, assign a tuple $L_{i,n}$, consisting of the node's previous compressed label $C_{i-1,n}$ and the multiset of compressed labels $C_{i-1,m}m \in N(n)$ of all neighboring nodes m of n from the previous iteration $(i - 1)$. Here $N(n)$ denotes the set of neighbors of node n.
    (b) **Compression:** Assign to each node $n \in V$ a new compressed label Ci, n, for example, a hash of $L_{i,n}$. Any two nodes with the same label $L_{i,n}$ must receive the same compressed label $C_{i,n}$.
3. **Partition:** After each iteration, partition the nodes in the graph by their compressed label.

On the other hand, MPNNs provide a more nuanced approach to graph representation. They work by passing messages between nodes, thereby aggregating information from neighboring nodes to update the state of each node in the graph. MPNNs produce continuous embeddings of nodes, edges, and even whole graphs, which encapsulate both structural and attribute information. This embedding approach allows the use of Euclidean geometry to measure distances and angles between points, providing a more detailed analysis.

In terms of performance, MPNNs generally outperform the WL test in tasks that require the learning of complex graph structures and representations. MPNNs, with their continuous and richer representation, are capable of capturing complex patterns and variations that the discrete WL test might miss. This makes MPNNs particularly suitable for large-scale graph analysis tasks where structural and attribute diversity is high. Therefore, while both WL and MPNNs have their unique strengths, MPNNs offer greater flexibility and richness in graph representation and analysis.

Similarity Measuring: Indexing by subgraph embeddings generated by the MPNN process to facilitate fast nearest neighbor searches in numeric spaces. Commonly used methods for such indexing include data structures like R-trees and kd-trees, renowned for their scalability and efficiency. Subsequently, the structural similarity between two subgraphs is determined by calculating the cosine similarity of their embeddings. Cosine similarity, recognized for its locality property, underscores the immediate neighborhood of nodes, irrespective of their global placement in the graph. This attribute of accentuating local proximity makes cosine similarity an exemplary measure for assessing the structural similarity of subgraphs within a larger graph.

The cosine similarity between two non-zero vectors A and B from an inner product space is measured by the cosine of the angle between them. It is defined mathematically as: $cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$, where $A \cdot B$ denotes the dot product of the vectors, and $\|A\|$ and $\|B\|$ represent their magnitudes. The dot product of two vectors A and B is computed as: $A \cdot B = \sum_{i=1}^{n} a_i * b_i$, where a_i and b_i are the respective components of vectors A and B. The implementation of indexing techniques to cosine similarity necessitates the normalization of each embedding to a unit length. Consequently, in this normalized space, the cosine similarity between two embeddings corresponds to their dot product and is inversely related to their Euclidean distance. The Euclidean distance between two vectors A and B is given by:

$$d(A, B) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

Thus, by these measures, smaller Euclidean distances correspond to embeddings with greater similarity, while larger distances suggest less similarity.

4.2 Cache hit

The ability to reuse the cached results of a previous query for the present query is contingent upon their structural overlap. Conventionally, this overlap is ascertained by computing the Maximum Common Subgraph (MCS) between the query graph q and a cached query graph w. A larger MCS implies a greater potential for reutilizing w's results for q. However, as MCS algorithms operate in exponential time complexity relative to the graph size, they impose a significant performance cost.

To expedite this process, the authors suggest confining the MCS calculation to the most promising cached queries w, those with a structural similarity to q. The subgraph indices are used to identify the k nearest neighbours to q, employing the embeddings of the query graph q and the cached query graphs. While the k-Nearest Neighbours (kNN) search is efficiently executed using the established index, the MCS problem necessitates resolution solely for these k pairs of graphs. Following the identification of the promising cached queries through the kNN search, the MCS is calculated for each query pair. Based on the resulting MCS node mapping, four cases can arise:

1. *Exact Match*: In instances of an exact match between q and a cached query w, the cached mapping corresponding to w is returned. This mapping is derived by projecting from q to w, and then from w to g, yielding a mapping from q to g.
2. *Subgraph Isomorphism*: If q is isomorphic to a subgraph of a cached query w, the result is generated through a similar projection of the cached mapping as in the first case.
3. *Partial Overlap*: A cached query w is considered to have some overlap with $q = (V_q, E_q)$ if their MCS is at least of size $|V_q| - \omega$, where ω represents an overlap threshold. Upon sufficiently large overlap, the cached mapping is utilized as a starting point for the search after the projection.
4. *Cache Miss*: If the identification of promising cached queries is unsuccessful, a cache miss is registered and the process described in Section 4.4 is employed, i.e., embedding-based filtering.

The overlap threshold ω is configured to be 10% of $|V_q|$ to maintain a balance. A minor ω yields insignificant differences between a cached query and the current query, whereas a larger ω increases the subgraph isomorphism detection time.

In the course of dealing with streaming subgraph isomorphism tasks, two operational scenarios are typically encountered. The first scenario, often referred to as 'cold start', arises when no previous embeddings are present in the cache. In this case, the cache begins populating only as new tasks arrive, with the decision to store each task's embedding being guided by a pre-established cache management strategy. The second scenario pertains to instances where the cache already contains embeddings from historical tasks. Over time, these embeddings may be updated based on the cache management strategy and the arrival of new tasks. Irrespective of the scenario, the process of identifying cache hits involves two fundamental steps - 'Search' and 'Validation'.

During the 'Search' phase, the cache, which is organized as a kd-tree, is scoured to identify the k-nearest vectors to the embedding of the current task. The kd-tree structure is widely tasked to be conducive to nearest search, with logarithmic search time, thereby significantly expediting this process. However, while the search time is logarithmic in the case of a balanced tree, it is important to note that kd-trees can become unbalanced with high-dimensional data, leading to less efficient search times. This phenomenon is known as the "curse of dimensionality". When the number of dimensions (k) increases, the kd-tree tends to resemble a linear search, diminishing the efficiency of the tree structure.So while kd-trees can indeed expedite nearest neighbor searches, their performance depends on the specific characteristics of the data and the use case. To improve the likelihood of cache hits, a larger number of returned vectors (k=100 in this case) are preferred. This search yields a set of candidate embeddings that may potentially match the current task.

Following the 'Search' phase, the 'Validation' phase is initiated to account for errors arising from iterative division of dimensions in high-dimensional spaces inherent to the kd-tree structure. The purpose of this phase is to ensure that the embeddings returned from the 'Search' phase indeed represent valid cache hits. Given the computational expense associated with the validation process, a time threshold of 0.01 seconds is set for the validation of each returned embedding. This time constraint ensures that the system does not spend an inordinate amount of time validating a single embedding, thus preventing undue delay in the completion of the task at hand. In essence, the 'Validation' phase comprises a series of time-limited checks (each of 0.01 seconds duration), repeated up to a 100 times for each subgraph isomorphism task. This two-stage process of 'Search' and 'Validation' forms the core of the cache hit identification process within the existing method.

4.3 Dealing with cache misses

This section discusses the approach employed by the proposed framework to handle cache misses, with an emphasis on the use of traditional subgraph isomorphism algorithms, the generation of embeddings for filtering candidate structures, and the strategy for selecting subgraphs.

The algorithm begins by initializing an empty set M to keep track of the current mapping, and it enumerates all the structures of the data graph g to create a set Ω. The algorithm then enters into an iterative process, where it continually checks if the size of the current mapping M is less than the size of the query graph q. If it is, the algorithm selects the next structure s from the query graph q. For each selected structure s, the algorithm invokes the FilterCandidates function to identify candidate structures s' from Ω that could potentially map to s based on the existing mapping M. Each candidate structure s' is then combined with the existing mapping M to create an updated mapping. After each combination, the algorithm checks if the updated mapping M is valid. If it is, the mapping M is added to the result set R which accumulates all valid mappings. Regardless of whether the updated mapping is valid or not, the algorithm then backtracks by removing the last added mapping to M, allowing the algorithm to explore other possible mappings. This iterative process continues until the size of the mapping M is equal to the size of the query graph q, ensuring that all potential mappings have been explored. At the end of the process, the algorithm returns the set R, which contains all isomorphic subgraphs of q in g.

Traditional subgraph isomorphism algorithms match a query graph and a data graph based on their nodes. The filtering process employed can vary based on the specific algorithm chosen, with some algorithms leveraging a node's label and degree (as exemplified by Ullmann's algorithm), and others utilizing its connections (such as in the case of VF2 and QuickSI). Yet, these simple strategies that focus on nodes' labels or degrees often deliver limited results. More advanced strategies that operate at the level of subgraphs, despite being highly selective, can lead to considerable computational costs.

Therefore the baseline framework introduces an innovative approach to filter candidate structures through their embeddings. This process involves generating embeddings for the structures of interest in the data graph offline. Subsequently, for each structure identified in the query graph, an equivalent embedding is created. These embeddings then serve to identify candidate structures in the data graph by extracting the k-nearest neighbors. During the search for isomorphic subgraphs, the selection of subgraphs plays a pivotal role. The pruning of candidate structures is based on different types of subgraphs, such as nodes, trees, or paths. However, this framework uses 2-node subgraphs (edges) as the foundation for a pruning strategy, as they are easier to enumerate and are more discriminative than nodes. The use of larger subgraphs, such as three or four node subgraphs, leads to substantial computational challenges due to their exponential growth.

The strategy based on embeddings introduced in this framework works in harmony with other filter mechanisms. Experimental results indicate that incorporating this strategy with any subgraph isomorphism algorithm substantially reduces the number of candidate structures to consider, thereby enhancing the efficiency of the overall process.

Embedding-based pruning: The proposed algorithm employs an embedding-based filtering strategy to identify potential structure matches. This process is divided into two phases: an offline phase for embedding generation and an online phase for candidate identification.

- During the offline phase, each structure of interest within the data graph, represented by Ω, is passed through a trained model f. This model generates an embedding, $z_{s'}$, for each structure s', which is then stored within a cumulative set, Z.

- The online phase starts once a new query graph, q, is received. Each structure s within q is processed to generate its corresponding embedding z_s. Following this, a k-nearest neighbors search is executed on the previously stored set Z to locate structures with embeddings closest to z_s, generating a set of potential matches, C_s. This operation is repeated for every structure in q, resulting in a collective set of candidate structures, C. Consequently, the algorithm delivers a list of matching candidates for each structure in the query graph, effectively pruning potential structures using their embeddings.

Pruning of candidate structures in the search for isomorphic subgraphs can be based on different forms of subgraphs, such as nodes, trees, or paths. However, the framework adopts a pruning strategy rooted in 2-node subgraphs (edges), given their discrimination power and ease of enumeration. It's noteworthy that this embedding-based strategy works in parallel with other filtering mechanisms, and as empirical evidence shows, it substantially curtails the number of candidate structures to be considered by any subgraph isomorphism algorithm.

4.4 Motivation of Cache Setting

The initial strategy employed in the paper of *Efficient Streaming Isomorphism work* leveraged a kd-tree as a cache to store historical embeddings, aiming to circumvent redundant entries into the subgraph isomorphism search phase. By leveraging a kd-tree as a cache, it allows for direct retrieval of past results when similar tasks are encountered. The primary motivation for using a kd-tree as a cache is its proficiency in performing nearest-neighbour searches in high-dimensional spaces, a necessary step to identify potentially reusable results. This approach markedly reduces the combined search and verification time - where verification involves checking the suitability of each of the k nearest embeddings returned by the kd-tree. It proves to be a more time-efficient strategy than resorting directly to traditional subgraph isomorphism search algorithms like the VF2 approach, the advanced TurboISO method, or the batch-processing MQO technique. The efficiency of the proposed method underscores the benefits of reusing historical results, preventing unnecessary computation and saving substantial time.

Despite initial observations showing strong performance by the caching strategy, as depicted in Figure 1.5, subsequent analysis identified limitations in its universal applicability. The key issue stemmed from the strategy's inability to process a high volume of tasks effectively, especially in cases of increased task diversity or reduced task duplication rate. Initially, the caching strategy aimed to capture a broad spectrum of embeddings, continuously refreshing the limited cache space with new or more valuable embeddings. This frequent turnover, though advantageous for diversity, introduced a significant drawback when dealing with large volumes of isomorphism tasks. When identical streaming queries were issued successively, the cache failed to store and recover more than half of the initial embeddings. This inability to maintain embeddings led to a significant destabilization and unpredictability of cache hit probability. When task diversity escalated, cache hit rates fluctuated across different graphs, as cache's effectiveness depended not only on the complexity of graph G but also on the quantity of task nodes. In this light, it becomes evident that while the caching strategy showed promising initial results, its design flaws impede its universal effectiveness, especially in high-volume, diverse task scenarios.

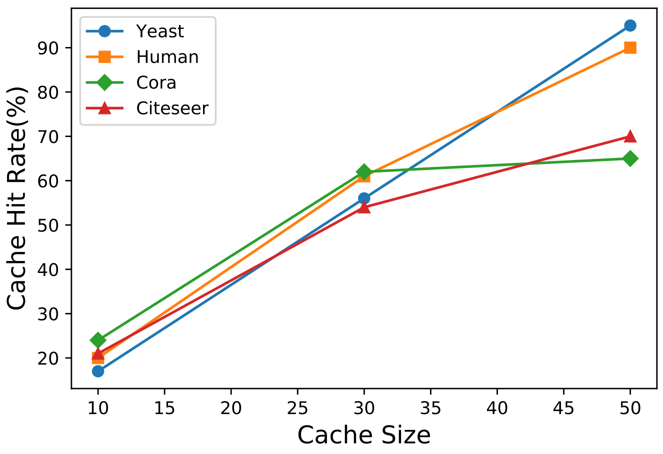


Figure 1.5: Cache Hit Accuracy

To resolve this, we attempted to increase the cache size, following the original paper's indication that a larger cache yields better performance. Nevertheless, increasing the cache size led to unforeseen consequences.

- Firstly, our cache strategy proved ineffective at managing large task volumes. The marginal utility of increasing the cache size gradually diminished as the cache hit rate did not improve proportionally. As the volume of tasks increased, the cache strategy couldn't efficiently handle the load, causing performance to plummet.

- Secondly, the time taken by the kd-tree for scaning in kd-tree and the validation stage began to affect overall efficiency. Despite the growth of scanning time being logarithmic and therefore relatively slower, it is non-negligible and impacted overall performance. And the time taken by the kd-tree for validation began to affect overall cache hit rate. With a strict time limit (0.001ms) in place for low latency, the validation process could time out when handling isomorphism tasks with larger subgraphs (more than 10 nodes).

This situation resulted in an inefficient trade-off between cache size and search time. The performence of adding cache size to kd-tree shown in Figure 1.14. Despite the selection of a larger and more efficient memory size, the application of a kd-tree for cache storage and retrieval may still encounter difficulties. Specifically, the kd-tree might fail to accommodate a sufficient number of unique embeddings. This failure impedes significant advancements in the cache hit rate, thus highlighting the limitations of the kd-tree caching strategy, even when enhanced memory resources are available. In light of these observations, simply expanding the kd-tree is not a practical solution.

Once in practical scenarios, data often arrive faster than they can be processed, accumulating into batch tasks rather than individual ones. This dynamic necessitates a cache system capable of managing large volumes of embeddings while maintaining low latency. Given the importance of user experience, prolonged wait times due to sluggish cache retrieval are unacceptable. Although parallel processing could be a potential solution, it isn't our solution due to its complexities and additional computational demands. It becomes apparent that the existing small cache size forms a bottleneck, struggling to handle large volumes and batch tasks simultaneously. The interplay of cache size, hit probability, and search time calls for a more sophisticated and efficient caching strategy. This understanding marks the beginning of our exploration for an alternative to the kd-tree based cache, which will be outlined in the forthcoming sections of this thesis.

This led us to propose a new solution: replacing the kd-tree with a vector database capable of storing a sufficient number of embeddings efficiently. Our proposed solution is designed to optimize the processing of the query stream, maximizing the cache hit probability, thereby aiming to substantially reduce processing latency. This new approach addresses the limitations of the original caching strategy and seeks to enhance the efficiency of the subgraph isomorphism search process. The rationale behind this decision, as well as a more detailed explanation of the new strategy, will be discussed in the subsequent sections of this thesis.

5 Our Approach

The utilization of an index is pivotal to the efficient handling of vector datasets. The index, a data structure that enhances the speed of data retrieval operations, operates by mapping vector representations of data items to their respective locations within the database. This mechanism allows the database to locate the data associated with a specific vector without the need to scan every item in the database, thereby facilitating efficient search and retrieval.

Our approach to this task involves a two-phase process: an offline phase and an online phase. In the offline phase, the system generates embeddings for a substantial number of historical queries, specifically 20000. These embeddings are then stored in a vector dataset with an index, which enables efficient similarity search and clustering of dense vectors. The data is stored as key-value pairs, with the key being the embedding and the value being the graph. This configuration allows for quick and efficient retrieval of the graph when provided with its corresponding embedding. By storing a large number of diverse historical embeddings, we aim to maximize the probability of cache hits in the online phase, thereby mitigating the dilemma of spending more time on existing subgraph isomorphism algorithms following cache misses.

In the online phase, the system handles incoming queries in real-time. For each new query, the system generates an embedding and searches for it in the index. If a similar embedding is found (a cache hit), the system retrieves the corresponding graph using the embedding as the key, subsequently verifying the graph to ensure its correct match for the query.

The construction of the index begins with the representation of data as vectors, a process completed by the Message Passing Neural Network (MPNN). Following this, a distance metric is defined to quantify the difference between two vectors. While the most common distance metric is Euclidean distance, others such as cosine similarity or Manhattan distance can also be employed. When a query is made to the database, the index is used to find the vectors that are closest to the query vector according to the distance metric. The construction method of the index is directly related to the performance effect of the vector database on this task. We will focused on a few indexes that prioritize search quality, speed, or index memory, and discuss related technologies in detail in the following chapters.

5.1 Exact Search Index

The most fundamental implementation is exact search index which encodes vectors into fixed-size codes. This exact search mechanism facilitates efficient similarity search and clustering of dense vectors. It use flat index operates on the principle of

Euclidean (L2) distance, comparing vectors based on this metric and returning the top-k vectors that bear the most similarity to the query vector.

One of the defining characteristics of flat index is its unaltered handling of vectors. This means that the vectors fed into these indexes retain their original form, without any approximation or clustering. This characteristic ensures the production of highly accurate results, as the integrity of the vectors is preserved. However, this accuracy comes at the expense of longer search times, as each vector in the index must be compared to the query vector. When a new query vector, denoted as q_i, is introduced, the index sequentially decodes all the indexed vectors and compares them to q_i. The comparison yields a distance score, which is the L2 distance between the query vector and each indexed vector. The lower the distance score, the closer the match between the vectors, indicating a higher degree of similarity.

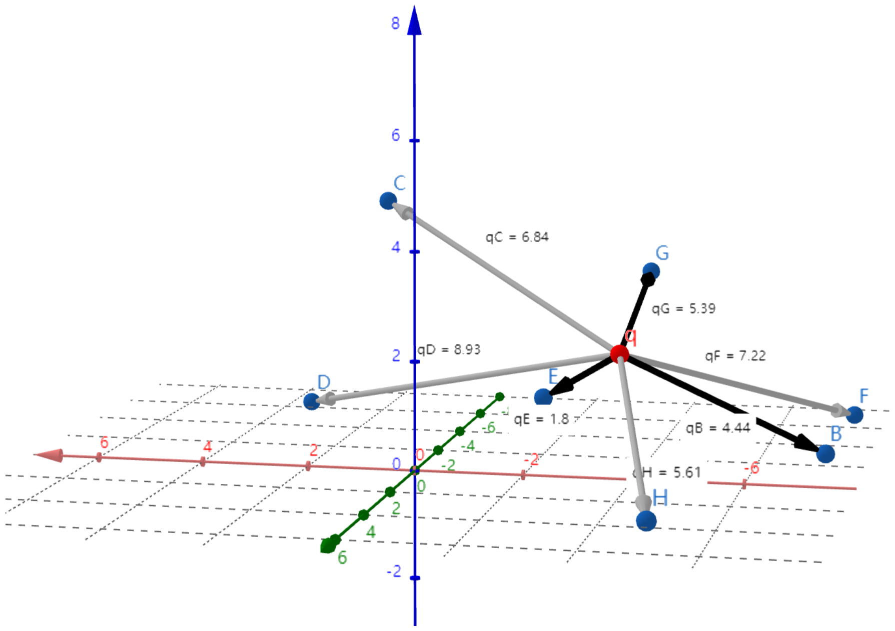


Figure 1.6: Flat Index with L2 distance
This is a simplified version of the 3-dimensional schematic, with the red vector points representing the q for which a similar mapping needs to be found. the goal is to find the 3 vector points that are the closest distance to the q vector,L2 from all the vector points in the multidimensional space.

From this, a kNN search can be implemented based on this distance calculation. Given a query vector M_q, the index calculates the distance to each vector in the index. After all these distances have been calculated, the index returns the 'k' vectors that are nearest to M_q as the most similar matches. This operation is represented as $argmin_{i=1...N}||x_i - x_q||_2$, where $||.||_2$ denotes the L2 distance as shown in figure 1.6. This process is identical to the measure of *Efficient Streaming Isomorphism work*, ensuring consistency in the comparison of vectors.

While the flat index with L2 provides perfect search quality due to its exact matching process, it does so at the cost of increased search times. The performence of this index will be clearly shows in Section 6, named IndexFlatL2. This trade-off makes the IndexFlatL2 suitable for smaller datasets where the precision of results is paramount. However, for larger datasets or tasks where efficiency is a concern, other indexing methods that balance accuracy and speed may be more appropriate. In the context of our task, which is streaming subgraph matching, low latency is a critical requirement. The exhaustive nearest-neighbors search performed by the flat index, while accurate, may not meet the latency requirements of this task due to its unoptimized search speeds. This is particularly true for use-cases where the index is large, or where search-time is a significant factor.

To enhance the speed of our search, we can adopt two main strategies: minimizing the size of the vector and narrowing the search range. The former can be accomplished through techniques such as dimensionality reduction or by decreasing the bit representation of our vectors. The latter involves the organization of vectors into tree structures or clusters based on attributes, similarity, or distance, thereby limiting our search to the most proximate clusters or analogous branches. Adopting these strategies transitions vector database search from an exhaustive nearest-neighbors search to an approximate nearest-neighbors (ANN) search. This shift occurs because our search no longer encompasses the entire dataset in its full resolution, but rather targets a data subset or a lower-resolution version of the data. In the ensuing section, we will delve into the intricacies of ANN indexing. The aim is to illustrate how it can strike a more balanced compromise between search speed and quality, thus fulfilling the requirements of our streaming subgraph matching task.

5.2 Approximate Search Index by Space divide

In our quest to enhance search efficiency, we adopt partition-based indexes that expedite searches by dividing the index into clusters and limiting the search to a select few of these clusters. This method, known as Inverted File Index (IVF) technology [33, 34, 35], operates on the principle of reducing the search scope through clustering. Given a query, the closest codeword or a few closest codewords are identified, and the corresponding lists are concatenated to produce the query response as shown in figure 1.7. It's important to note that this method is approximate, as it does not guarantee that the nearest neighbors will be found within the searched clusters. This approach is based on the concept of the Voronoi diagram[36], or Dirichlet tessellation.

To comprehend Voronoi diagrams, we visualize our high-dimensional vectors in a 2D space. Additional points, acting as centroids for our 'clusters' or Voronoi cells, are introduced into this space. Each centroid expands an equal radius until the boundaries of each cell intersect with another, forming our cell edges. Consequently, every data point is encapsulated within a cell and linked to the corresponding centroid. The introduction of a query vector, denoted as x_q, necessitates its placement within one of these cells, thereby confining the search scope to this particular cell. However, the edge problem arises when our query vector is positioned near the boundary of a

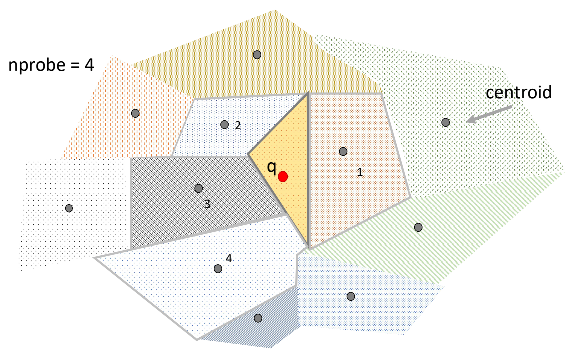


Figure 1.7: Index with Inverted File technology
In a 2D visualization, setting nprobe to 4 means the search operation for a given query vector extends to its own Voronoi cell and the three nearest neighboring cells. This broadens the search scope, improving search quality.

cell, making it highly probable that its nearest datapoint resides within an adjacent cell.

To mitigate this issue and enhance search quality, we can adjust an index parameter, the nprobe value, which stipulates the number of cells to be searched. We can also modify the nlist parameter, which determines the number of cells to be created. A larger nlist value implies that our vector must be compared to a greater number of centroid vectors, but after selecting the cells nearest to the centroid for search, each cell will contain fewer vectors. Therefore, an increase in nlist prioritizes search speed. Conversely, an increase in nprobe broadens the search scope, thus prioritizing search quality. Balancing these two parameters is crucial to achieve the optimal trade-off between search speed and quality.

The search operation with IVF technology is akin to previous indexes. However, we specify the "nprobe" hyperparameter in the Index with IVF to restrict the search to only the defined number of clusters nearest to the query vector. This demonstrates how various indexes can be combined to form a single, more efficient index. We will explore the practical applications and performance of the Index with IVF in greater detail in the subsequent sections.

While IVF provides significant speed-up and memory efficiency over exhaustive search, its efficiency begins to diminish with extremely large datasets. To achieve a finer partition of the search space and return more localized results, the number of codewords in IVF would need to be increased. However, this leads to an increase in query time and index construction time. While approximate nearest neighbor approaches can mitigate this slowdown, they often result in a considerable reduction in the accuracy of the returned candidate lists. Therefore, Product Quantization (PQ) is introduced into our work as an enhancement to the IVF technology, addressing some of the limitations of IVF when dealing with very large datasets [37, 38].

5.3 *Approximate Search Index by PQ*

Product Quantization (PQ)[39, 38] is a well-established technique in the field of Vector Quantization (VQ)[40, 41] that significantly enhances the efficiency of kNN search operations, particularly in the context of vector databases. Unlike dimensionality reduction strategies, which aim to generate a lower-dimensional representation of a high-dimensional vector, PQ concentrates on minimizing the range of potential values that a vector can assume. This is achieved by transforming the vector into a space with a finite number of possible values, where these values serve as symbolic representations of the original vector.

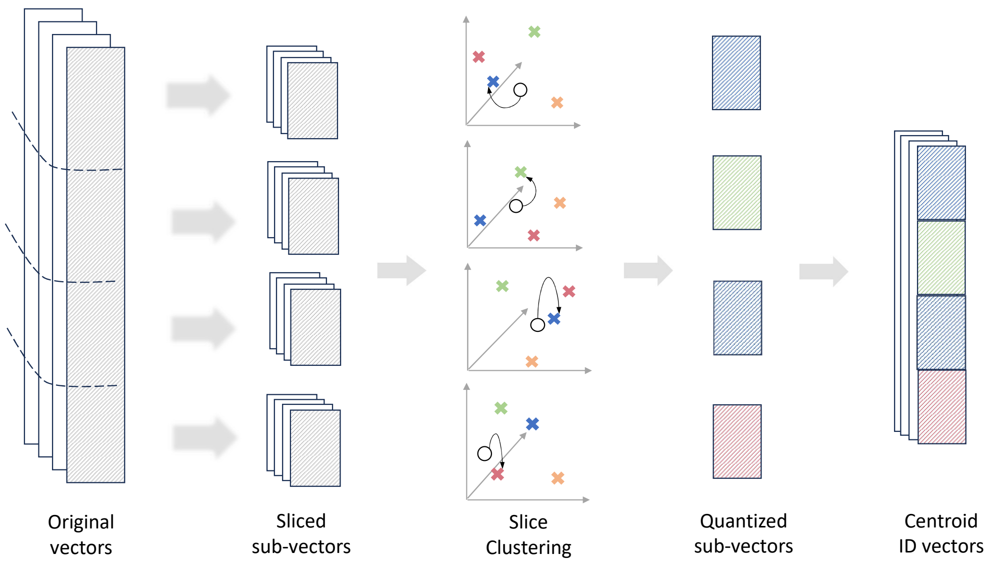


Figure 1.8: The process of Product Quantization

As figure 1.8 shows, the process of PQ involves several steps [42, 36]. First, a high-dimensional vector space into the Cartesian product of subspaces which could also be seen as equally sized chunks, or subvectors. This process is at the heart of PQ and is what allows it to generate a large codebook at a very low memory/time cost. Each of these subspaces is then assigned to its nearest centroid, which serves as a symbolic representation of the subvector. These centroid values are subsequently replaced with unique identifiers (IDs), resulting in a compact vector of IDs that requires significantly less memory than the original vector. The number of centroids (k) and the number of subvectors (m) are key parameters in PQ. The total number of centroids used to represent the vectors and the number of subvectors that the original vector is split into directly impact the memory usage and complexity of the quantization operation. Experiments have proved that the parameter $k2048$

can achieve the best effect. The process of "chunking", reduces the dimensionality of the vectors that need to be compared, which in turn significantly reduces the memory usage and assignment complexity of the quantization operation. This is one of the key advantages of PQ, allowing it to efficiently handle high-dimensional data.

In mathematical terms, this transformation process involves mapping a vector $x \in \mathbb{R}^D$ to a codeword c in a codebook $C = \{c(i)\}$, where i belongs to a finite index set. This mapping, referred to as a quantizer, is denoted by $x \rightarrow c(i(x))$. The function $i(\cdot)$ is known as an encoder, and $c(\cdot)$ is the decoder. The quantization distortion E is defined as:

$$E = \frac{1}{n} \sum_{x} ||x - c(i(x))||^2$$

Here, $|| \cdot ||$ represents the Euclidean distance, and n is the total number of data samples. This definition of distortion applies to any quantizer, regardless of the specific encoder and decoder used.

PQ operates by decomposing the high-dimensional vector space into a Cartesian product of multiple low-dimensional subspaces, which are then quantized independently. Formally, any $x \in \mathbb{R}^D$ is denoted as the concatenation of K subvectors: $x = [x^1, ...x^k, ...x^K]$. For simplicity, we assume the subvectors have an equal number of dimensions D/K. The Cartesian product $C = C^1 \times ... \times C^K$ is the codebook in which any codeword $c \in C$ concatenates K sub-codewords: $c = [c^1, ...c^k, ...c^K]$, with each $c^k \in C^k$.

The objective function of PQ is defined base on the equation of E, essentially:

$$min_{C^1,...C^K} \sum_{x} ||x - c(i(x))||^2$$

It can be shown that x's nearest codeword c in C is the concatenation of K nearest sub-codewords $c = \{c^1, ...c^k, ...c^K\}$ where c^k is the nearest sub-codeword of the subvector x^k. Therefore, this expression can be divided into M separate subproblems, each of which can be solved by k-means. This is precisely the approach taken by PQ. This method ensures that the encoder $i(x)$ maps any x to its nearest codeword in the codebook C, satisfying the first Lloyd's condition and making the quantizer a Voronoi[43] quantizer. This property is valid regardless of the codebook.

The primary motivation behind the use of PQ is its ability to significantly reduce the memory requirements of indexes. This is particularly important when dealing with large arrays of vectors, as these vectors must be loaded into memory for comparison. While there are other quantization methods that can reduce memory size, PQ stands out for its effectiveness in this regard. This is where PQ comes into play [38]. By splitting high-dimensional vectors into dimension groups and approximating each vector as a concatenation of several lower-dimensional codewords, PQ allows for a much finer subdivision of the search space without increasing query time and preprocessing time compared to IVF with moderately-sized codebooks. This results in faster and more accurate retrieval and approximate nearest neighbor search, especially when dealing with very large scale datasets, while retaining the memory efficiency of standard IVF.

In essence, the introduction of PQ into our work results in the creation of an inverted multi-index, which is a multi-dimensional table where each entry corresponds to all possible tuples of codewords from the codebooks corresponding to different dimension groups. This multi-dimensional table replaces the "flat" table of a standard IVF, where entries correspond to codewords. Each entry of a multi-index table corresponds to a part of the original vector space and contains a list of points that fall within that part. A simple and efficient algorithm is proposed to produce a sequence of multi-index entries ordered by the increasing distance between the given query vector and the centroid of the corresponding entry, resulting in a candidate list [44]. This approach maintains the memory efficiency of standard IVF while providing faster and more accurate retrieval, especially for very large scale datasets.

Despite the power of PQ, the optimal decomposition of the vector space remains a largely unaddressed issue. In this task, PQ is formulate as an optimization problem that minimizes the quantization distortion by seeking for optimal codewords and space decompositions by training process. When a query vector is received, the index first determines the Voronoi cell that the vector falls into using the coarse quantizer. It then uses the product quantizer to find the nearest neighbors within that cell. The residuals are calculated as the difference between each vector and its nearest centroid, as determined by the coarse quantizer. And the centroids in these subspaces are determined by running the k-means clustering algorithm on the subvectors.

It is a highly effective method for compressing high-dimensional vectors, achieving up to 97% memory reduction and accelerating nearest-neighbor search speeds by a factor of 5.5 in our evaluations. When combined with an Inverted File (IVF) index, the search speed can be further enhanced by an additional 16.5 times without compromising accuracy. This results in a remarkable total speed increase of 92 times compared to non-quantized indexes.

5.4 Adoption of vector database

To achieve the desired index building technology for our work, we have chosen FAISS as our vector database. Developed by Facebook AI Research, FAISS is an open-source library specifically designed for efficient similarity search and clustering of dense vectors. It is capable of supporting billion-level vector searches, making it the most mature and reliable library for approximate nearest neighbor search currently available. FAISS is equipped with a diverse range of algorithms for searching arbitrary-sized sets of vectors, and it also provides support code for algorithm evaluation and parameter tuning. Its compatibility with Python and Numpy, along with GPU implementations of some core algorithms, makes it highly accessible and efficient for a wide range of applications.

One of the key strengths of FAISS lies in its proficiency in handling k-nearest neighbor (kNN) search, a critical task in the context of vector embeddings. Given a vector x, FAISS can efficiently return the k vectors most similar to x. It also supports the reordering of the nearest neighbors based on different similarity measures. What makes FAISS particularly suitable for our work is its built-in support for Inverted File (IVF) and Product Quantization (PQ) indexing solutions. These indexing techniques are highly effective for large-scale vector databases. IVF provides a significant speed-up and memory efficiency over exhaustive search, while PQ enhances IVF by addressing some of its limitations when dealing with very large datasets. The integration of these indexing solutions within FAISS allows us to leverage their benefits without the need for additional implementation. This makes FAISS a robust and efficient platform for our vector database needs, and an ideal choice for our work.

6 Evaluation

This section is devoted to a stringent evaluation of our proposed batched streaming subgraph isomorphism method. The objective of this evaluation is multifold: first, to determine the optimal parameters that enhance the process's efficiency; second, to validate the robustness of the process under these optimized parameters. The parameters under consideration include the cache size, the most fitting index for the task, and the number of top results (k-top) to return. This evaluation framework is strategically designed to ascertain the best configuration for each module in the process and assure the method's pragmatic utility in real-world scenarios.

The structure of this section is split into two main subsections: the Experimental Setup and the Performance Evaluation of the Batched Streaming Subgraph Isomorphism. The Experimental Setup subsection provides comprehensive details on the dataset, metrics, implementation approach, and the evaluation environment. It will elaborate on our parameter choices, such as cache size, index type, and k-top return, and elucidate the anticipated impact of these choices on the process.

The subsequent section, Performance Evaluation of the Batched Streaming Subgraph Isomorphism, delves into the detailed performance tests conducted. The tests are designed with our primary excellence criteria in mind: shorter processing times and higher precision levels. Attention is given to the impacts of cache size variation, selection of different indices, and variation in the number of top results returned, with the aim of pinpointing the optimal parameter configuration. Additionally, we rigorously evaluate the process's robustness under varying batch sizes. Given the batch size can be seen as a form of pressure or a change in the volume of tasks in the batched streaming subgraph isomorphism task, it is critical to confirm the process's ability to strike a satisfactory balance between speed and cache hit despite these changes.

The results and insights drawn from this systematic evaluation are projected to be instrumental in guiding future process enhancements and adjustments. Through this rigorous testing and evaluation, we aim to exhibit the applicability, robustness, and scalability of our method in real-world subgraph isomorphism tasks.

6.1 Experimental setup

Datasets In order to facilitate the comparison of the optimisation effects of *Efficient Streaming Isomorphism work*, in this study, we follow the six real-world datasets in benchmark for the experiments. These datasets include Wordnet, Human, Yeast, Citeseer, Cora, and Pubmed, each offering unique characteristics and complexities. The Wordnet dataset [45] stands alone in capturing relations between English words. Characterized by a small number of labels and a small node degree, it reflects the simplicity and directness of word relations, providing a unique perspective on linguistic connections. Next, we have two datasets, Yeast and Human[46], representing protein-protein-interaction (PPI) networks. The Yeast dataset, despite its small average degree, encompasses a large number of labels, offering a rich source of diverse protein interactions. The Human dataset, in contrast, is characterized by a large node degree, reflecting the complexity and breadth of human protein interactions. Finally, we have three datasets, Citeseer, Cora, and Pubmed [47], representing scholarly articles in citation networks. Nodes in these datasets are attributed, with edges representing citation relationships. These datasets capture the intricate citation relationships among scholarly articles, providing a rich ground for subgraph isomorphism studies. Each of these datasets offers unique opportunities and challenges for subgraph isomorphism, making them ideal for a comprehensive and robust evaluation of our proposed methods. Further details and analyses of these datasets will be discussed in the subsequent sections of this thesis. The detailed statistical information is listed in the table.

Table 1.2: Statistics of the datasets

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	No.Labels	Avg. Degree
Wordnet	82,670	127,124	5	3.08
Human	4,674	86,282	90	36.92
Yeast	3,101	12,519	71	8.07
Citeseer	3,327	4,600	6	2.77
Cora	2,708	5,278	7	3.90
Pubmed	19,717	44,324	3	4.5

Metric The principal metric in our evaluation framework is the average processing time across the entire query stream. This average processing time serves as a key indicator of our method's efficiency in dealing with batched streaming subgraph matching tasks. We endeavor to maximize the utilization of historical results to expedite future tasks, aiming to attain a high cache hit rate. A cache hit in our system means that we can leverage previously computed subgraph match results, significantly reducing the time compared to employing a traditional subgraph isomorphism algorithm. As a result cache hit rate is a evaluation metric in this task.

Our goal is to ensure that our method not only swiftly executes tasks but also effectively reuses previous computations to yield timely outcomes.

To provide a more holistic assessment, we will also contrast the quality of the results produced by our method with those of the optimal cache management approach in *Efficient Streaming Isomorphism work.* This comparison will primarily revolve around the average hit rate metric, which is a critical parameter in the *Efficient Streaming Isomorphism work.* Further, we will take into account how the Vector Database - used in our project by Faiss - fares against the *Efficient Streaming Isomorphism work* in terms of the average hit rate, providing a comprehensive perspective on the effectiveness of different approaches.

The selected metrics are purposefully designed to facilitate an exhaustive understanding of our method's performance. They provide quantitative measures of efficiency and cache hit rate, which are critical in determining the process's optimal parameter settings and robustness under different batch sizes. This rigorous metric-based evaluation will allow us to validate our proposed method's superiority over existing alternatives and guide future enhancements.

Implementation and environment In *Efficient Streaming Isomorphism work*, the graph indexing is implemented by Python,use PyTorch to for offline training, and C++ for online query evaluation. The experiments were conducted on a Linux platform using the C++ programming language. The machine used for these experiments was equipped with an Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz and 512 GB of RAM. The executable file was compiled using g++ version 11.3.0. The *Faiss* library [48], which is used for efficient similarity search and clustering of dense vectors, was also compiled on this machine.

6.2 Vector Database Performance Analysis

The focus of this subsection is twofold: tuning the parameters for optimal settings and analyzing their impact on the performance of our batched streaming subgraph matching process. The parameters of interest are cache size, k-top return, and the choice of index. These were chosen for their significant influence on the efficiency and precision of the process. This analysis exams all potential combinations of our parameters to identify the optimal configuration. Each combination is evaluated based on our key metrics of processing speed and precision, ensuring we find a robust, optimal setting that maximizes the efficiency and precision of the batched streaming subgraph matching process. This comprehensive approach takes into account all parameter spaces and aims to determine configurations that offer the best balance between speed and cache hit rate, particularly with variable batch sizes.

Training Stage In Figure 1.9, we present the index building time for two distinct indexing methods, *IndexIVFFlat* and *IndexIVFPQ*, across six datasets. The vertical axis represents the index building time in milliseconds, depicted on a logarithmic (base 10) scale. We opted for a logarithmic scale due to the wide range of time values

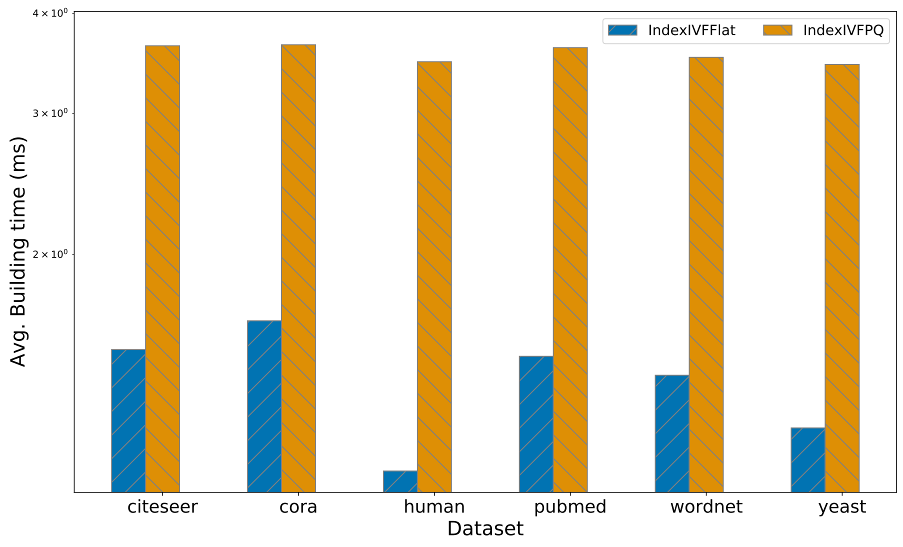


Figure 1.9: Index Building time

The y-axis is on a logarithmic scale. The values represent the exponent to which 10 must be raised to obtain the actual value in the data. For example, a y-axis value of 2 corresponds to 10^2 = 100 ms.

across datasets and indexing methods, which allows us to efficiently represent both small and large values. Moreover, the log scale underscores relative changes, making it easier to interpret differences in terms of ratios or percentage changes rather than absolute values.

We trained these indices on a subset of up to 20,000 vectors per dataset to optimize their initial builds. This approach leverages the latent demand present in the dataset, allowing the indices to reuse past query history, thereby improving their performance. As a result, we expected the more complex *IndexIVFPQ* to take longer to build its index due to its additional training and optimization steps, as it approximately used 10^3 ms, consistently more time than *IndexIVFFlat* which averages around 10^1.2 ms. This observation validates our understanding of the increased complexity associated with *IndexIVFPQ*. Furthermore, the log scale effectively highlights the variation in index building time across different datasets, underscoring the significant impact of dataset characteristics on the efficiency of index construction.

Parameter of k Nearest Return: Figure 1.10 exhibits an intriguing relationship between the number of nearest neighbors returned (k-return) and the resulting average cache hit rate across three different indices: *IndexFlatL2*, *IndexIVFFlat*, and *IndexIVFPQ*. This observed relationship is governed by an inherent bias prevalent in high-dimensional embedding spaces, which often requires the retrieval of more than one nearest neighbor to avoid misses. Consequently, the selection of the k-return value becomes a strategic decision with substantial influence on our task's performance. A larger k-return generally aligns with a higher average cache hit rate, a trend consistent across all indices. However, the incremental increase in the k-return offers diminishing returns. This is evidenced by a gradually decreasing

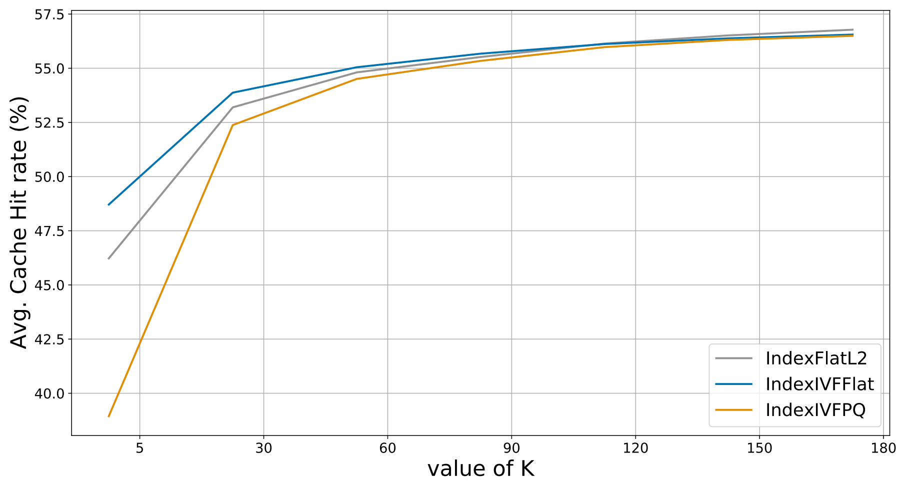


Figure 1.10: Cache Hit rate vs k nearest return
The y-axis represents the mean cache hit rate calculated over all combinations of cache sizes and search batch sizes, where both including value of 5000, 10000, 15000, 20000.

rate of cache hit rate improvements with each additional increase in the k-return. This aspect underscores the importance of fine-tuning the k-return value to balance between achieving high cache hit rates and maintaining efficiency.

A crucial observation is that around k=60, the cache hit rate, also called hit@60, improvement begins to plateau, reaching an approximate value of 0.675 across all indices. We interpret this point as an inflection in the hit@k curve, marking the transition from substantial cache hit rate gains with increased k-return to more negligible improvements. For values of k-return lower than this inflection point (k=5), there are notable discrepancies in cache hit rate among the indices - *Index-IVFPQ* exhibits an hit@5 of 0.525, *IndexFlatL2* shows an hit@5 around 0.615, and *IndexIVFFlat* presents an hit@5 of 0.645. As we increase k to 30, these divergences reduce, and all hit@30 consolidates within a range of 0.650 to 0.675.

Guided by these trends, the choice of setting k=60 as the optimal parameter for future tasks is justified by the combined effects of diminishing cache hit rate returns beyond this point, convergence of all indices at this cache hit rate level, and considerations related to computational load associated with larger k-returns. Therefore, we argue that k=60 provides a judicious trade-off, optimizing the balance between cache hit rate and computational efficiency in the context of batched streaming subgraph isomorphism matching tasks.

Index choice Upon a more in-depth examination of the heatmap in Figure 1.11, we indeed observe certain nuances in the interplay between search batch size, cache size, and search time. While a larger search batch size and cache size generally result in an increased search time, this trend exhibits a greater degree of complexity when we consider the role of the specific index type.

In the case of *IndexFlatL2*, the increase in search time with respect to both the search batch size and cache size seems to follow a more or less linear trajectory.

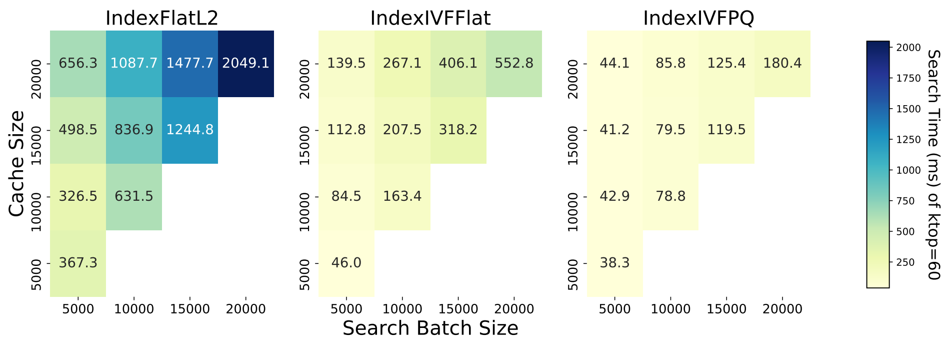


Figure 1.11: Search Time of each Batch and Cache Size
when return k = 60 nearest vector

This might be attributable to the structure of the *IndexFlatL2*, which exhaustively computes distances to every vector in the database, a process that naturally extends in duration with the addition of more data.

However, when we look at *IndexIVFPQ*, the correlation between cache size and search time exhibits a markedly non-linear trend. This behavior could stem from the Product Quantization (PQ) method, which it accomplishes by compressing the vector representation in a way that speeds up distance computations while only slightly compromising the cache hit rates. As such, while the search time does increase as cache size grows, the extent of this increase is not as substantial compared to other index types. This relative robustness to larger cache sizes highlights the efficiency of the PQ technique, especially in scenarios involving larger volumes of data. Further attesting to the efficiency of *IndexIVFPQ* is its consistently lower search times across varying search batch sizes and cache sizes. Even at the highest levels of these parameters, *IndexIVFPQ* still outperforms the other indices, indicating that it copes better with increases in computational demands.

In light of these findings, our analysis substantiates the claim that the choice of index type plays a crucial role in determining the efficiency of the batched streaming subgraph matching process. Moreover, it underscores the potency of techniques like Product Quantization in managing the computational complexities associated with high-dimensional vector databases. In actual tasks, if there is a requirement for response speed, it should be combined with the specific situation, and *IndexIVFPQ* should be given priority.

Parameter of cache Size This heatmap provides a visual encapsulation of the relationships between batch size, cache size, and cache hit rate, thereby offering a more intuitive understanding of these complex correlations and overarching trends. The cache hit rate tends to improve as both the batch size and cache size increase. This trend prevails across all three indices – *IndexFlatL2*, IndexIVFFlat, and *IndexIVFPQ* – underscoring the common principle that larger data volumes provide more data points for similarity comparison, thus enhancing the chances of a successful match and consequent cache hit rate.

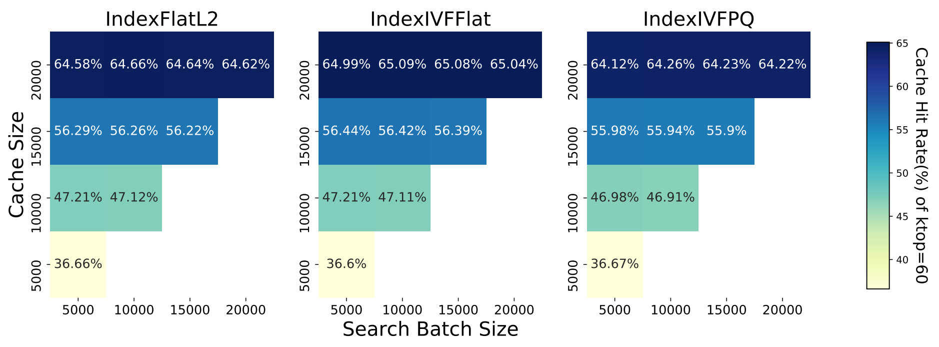


Figure 1.12: Accuracy of each Batch and Cache Size
when return k = 60 nearest vector

The cache hit rate across the three index types exhibits relative consistency across varying batch and cache sizes, with a marginally higher value for *IndexIVFFlat*. This minute variance may stem from *IndexIVFFlat*'s inherent ability to balance cache hit rate and efficiency through its integrated use of an inverted file system and a flat structure. Yet, *IndexIVFPQ*, while recording a slightly lower cache hit rate, is still within an acceptable range. Combined with its excellent performance in speed, it will be the indexing technology of choice for this project.

Turning our attention to the role of batch and cache sizes, the batch size often cannot be manually adjusted, but the cache size does afford us this leverage. The heatmap indicates that a cache size of 20000 tends to yield the best results. A higher cache size typically correlates with better performance as it avails more data for similarity comparisons. Nevertheless, it's crucial to factor in that if all Faiss indexes are stored in RAM, memory availability could limit cache size and should be a point of consideration. Within the constraints of available memory, the RAM limitation guides the precision-speed trade off. This is particularly applicable when exact results are not a stringent requirement. In scenarios requiring larger storage, the index could be stored on disk instead of RAM, although this might involve a performance trade off.

Considering these insights, we suggest setting the cache size to 20000 for this task, keeping in view the optimal performance observed at this level, as well as the need to maintain a balance between cache hit rate and computational efficiency within the boundaries of system resources.

6.3 Robustness Evaluation

The figure 6.3 presents two key facets of the performance of *IndexIVFPQ* - Hit@60 and search time, distributed over two sub-figures. These observations are predicated on the optimal settings identified from our previous evaluations, which determined that the index performs optimally with a cache size of 20000 and k equal to 60.

The first subfigure in this set elucidates the probability of cache hits against varying search batch sizes, ranging from 5000 to 20000. Here, the plot brings to light

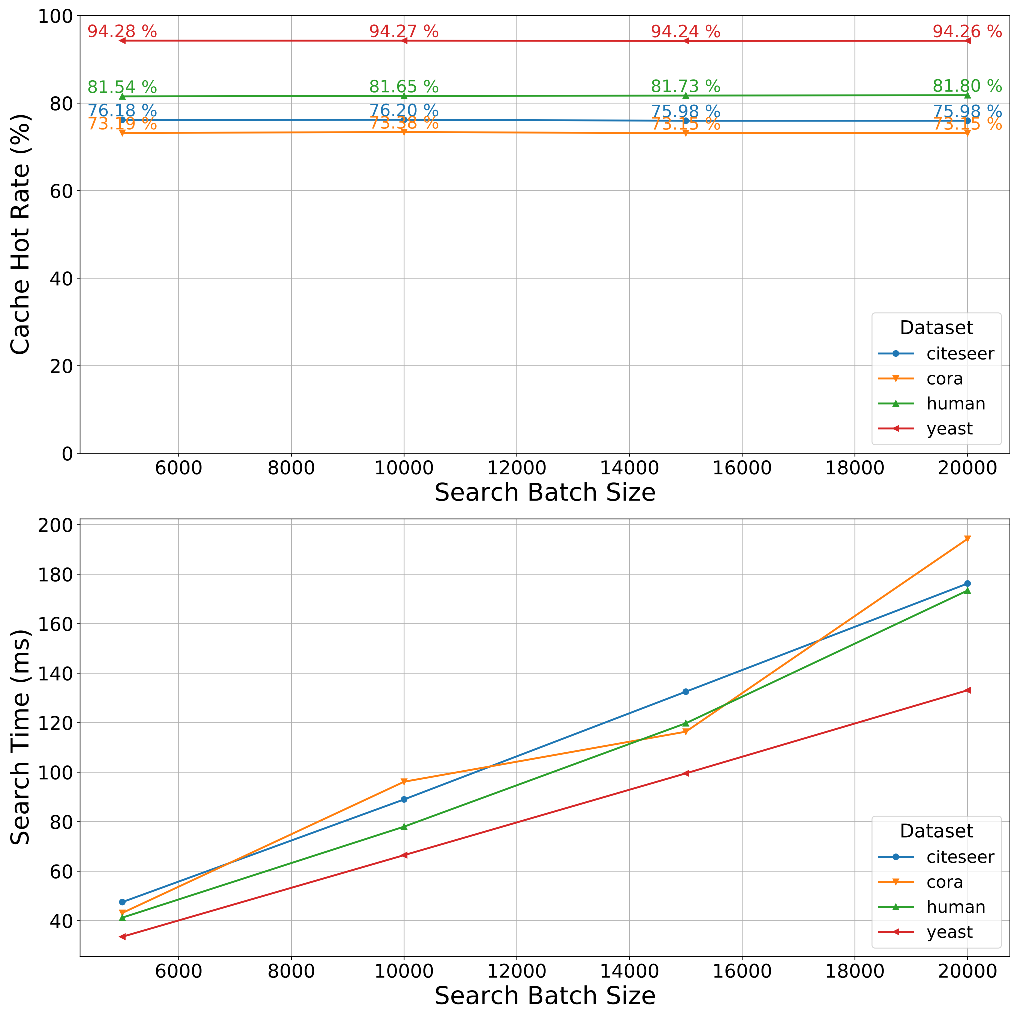


Figure 1.13: Assessing the Robustness of IndexIVFPQ: Hit@60 and Search Time across Batch Sizes
cache size = 20000, k-top nearest return = 60, Index = IndexIVFPQ.

the performance of **IndexIVFPQ** across four distinct datasets: Citeseer, Cora, Human, and Yeast. Notably, the cache hit probability remains impressively steady as the search batch size increases, across all datasets. This evidence underlines the index's ability to maintain high levels of cache utilization, irrespective of batch size. And the variations in the cache hit probabilities between datasets, which likely reflect inherent variations in the data characteristics of the respective datasets. Despite these differences, the average cache hit percentage predominantly lies within the range of 70% to 90%.

The second subfigure transitions the focus to the search time, denoted in milliseconds (ms), and its relationship with the search batch sizes. This depiction reveals a clear linear increase in search times with the growth of batch sizes across all datasets, with times escalating from approximately 40ms for a batch size of 5000 to around 180ms for a batch size of 20000. This pattern indicates that, while *IndexIVFPQ* maintains a consistent level of cache hit probability, the search time remains directly proportional to the batch size.

Together, these subfigures accentuate the robustness of *IndexIVFPQ*, which consistently exhibits high cache hit probabilities irrespective of the search batch size. However, they also spotlight the trade-off that comes with larger batch sizes, necessitating correspondingly extended search times. Such understanding provides a valuable foundation for assessing the scalability and efficiency of *IndexIVFPQ* in managing different batch sizes while maintaining a high likelihood of cache hits.

6.4 Comparison with Existing Methods

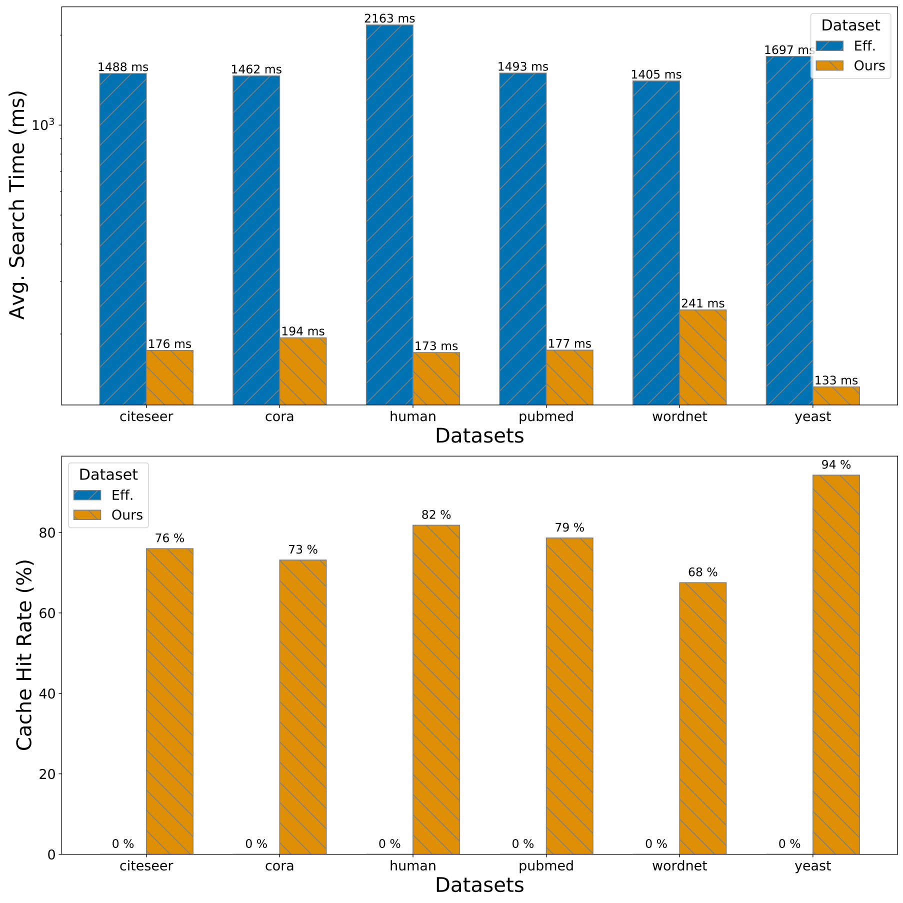


Figure 1.14: Effect Comparison 1
with validation time limit=0.001ms for both, cache size = 20000, finished 20000 searching tasks, k=60 nearest return

Figure 1.14 makes a comparative study of the average search time, represented in a logarithmic scale, between two methods - 'Eff.' (existing) and 'Ours' (proposed). The total search time is defined as the sum of the time taken for scanning the kd-tree and the time dedicated to validating each return. The choice of logarithmic scale serves to highlight the substantial disparities between the two methods. Evidently, our method consistently excels, reducing the search time by approximately six times across all datasets.

This result stems from a crucial experimental condition: A uniform cache size of 20,000 is assigned to both the methods, with the same validation time limit of

0.001ms. That a set validation time limit of 0.001ms for each embedding returned k-nearest search result (k = 60 for both approch) from the kd-tree and vector database. This setting ensures a fair comparison between the two methods. This specific constraint, introduced by the *Efficient Streaming Isomorphism work*, plays a pivotal role in maintaining low latency in the system. The experiment involves running a total of 20,000 queries. While the 'Eff.' method processes these queries sequentially, our method employs a batch processing approach, which significantly contributes to the observed time advantage in the graph. Thus, the systematic arrangement of these experimental parameters, in turn, elucidates the superior performance of our method.

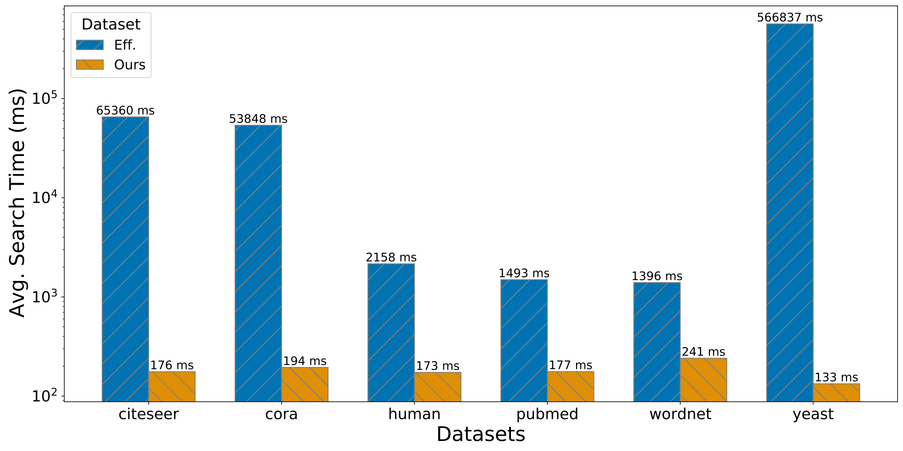
However, it's imperative to underscore that when operating under the strict validation time limit of 0.001ms, 'Eff.' suffers from a drastically zero cache hit rate. This underwhelming performance underscores two pivotal aspects: the cache's inability to manage the current task volume effectively and the diminishing quality of the returns from the kd-tree as its size scales up. The observed performance decline in the 'Eff.' method can be traced back to two critical scenarios.

- The first is the 'original memory case', where despite the cache containing the correct answer, the performance suffers due to the high dimensionality of the embeddings. When the cache size is large, the precision of the 60 nearest neighbor results from the kd-tree is further impeded. Consequently, the returned values that proceed to the verification stage fail to filter out the correct answer. Given the tight validation time limit, the verification phase prematurely terminates, resulting in a missed cache hit.

- In the second scenario, referred to as the 'cold start case', the cache starts off empty. Here, the decision to initiate kd-tree maintenance, i.e., increase cache, hinges on the cache management strategy's judgement, which is invoked after obtaining the subgraph's embedding. However, the judgement phase suffers from a timeout, preventing the addition of the embedding to the kd-tree, hence leaving the cache unpopulated. This results in every query interacting with an empty set, leading to zero comparisons and further highlighting the inefficiency of the 'Eff.' method in this setup.

In summary, the inefficiency of 'Eff.' at larger cache sizes is primarily due to two factors: the high dimensionality impacting the nearest neighbor results' accuracy, and timeouts during critical phases of cache management, which limit the effectiveness of the cache mechanism.

To substantiate the operational efficacy of the 'Eff.' method, we deliberately adjusted our experimental settings. Recognizing that processing time might influence the cache hit rate, we relaxed the low latency requirement, extending the validation time limit from 0.001ms to 0.01ms. This change provided 'Eff.' with an extended processing window. The generous timeframe resulted in 'Eff.' attaining a non-zero cache hit rate, indicating that the method can handle larger cache sizes and indeed affirms the functional capability of 'Eff.', albeit under more generous time



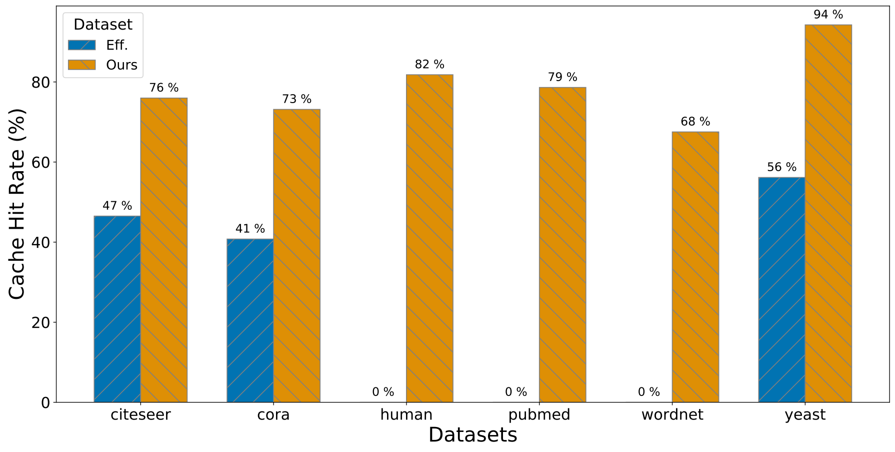


Figure 1.15: Effect Comparison 2
with validation time limit=0.01ms for Eff., and time limit=0.001 for Ours
cache size = 20000, finished 20000 searching tasks, k=60 nearest return

conditions. However, 'Eff.' still demonstrated notable performance shortcomings - specifically, suboptimal execution time and accuracy levels.

As figure 1.15 shows, 'Eff.' employed a logarithmic scale increase in total search time usage to alter the cache hit rate for certain datasets. And it is unstable as still half of the datasets get zero cache hits. These datasets encountered substantial constraints, as they either failed to return the correct k-nearest neighbors to the validation stage or encountered timeouts during this stage. Furthermore, some of these datasets also struggled to successfully execute kd-tree maintenance, highlighting the method's inefficiency in coping with such issues under the current time restrictions. Thus, while the method proved operational, our approach consistently outperformed it, highlighting our method's superior efficiency, even under stringent latency constraints.

7 Conclusion

Our study underlines the pivotal role of the choice of index in governing the efficiency of the batched streaming subgraph matching process. The IndexIVFPQ clearly outperforms its counterparts, boasting superior speed and robustness. A cache size of 20,000 was identified as optimal, striking a balance between system resource constraints and the need for a k-return of 60 to balance accuracy with computational efficiency. And our approach offers significant enhancements. The robustness of IndexIVFPQ is underscored by its ability to maintain high cache hit probabilities, regardless of batch size. This suggests its strong potential for handling real-world subgraph isomorphism tasks, where batch sizes can vary considerably.

Unlike the kd-tree used in the 'Efficient Streaming Isomorphism work,' our approach with the vector database delivers superior performance, particularly in handling larger batch sizes and ensuring the stability of cache hit rates. That our study signifies a substantial progression in the realm of subgraph isomorphism. We have not only addressed the batched streaming condition, but have also innovatively adapted and integrated the emerging vector database technology into existing frameworks. This synergistic amalgamation results in a solution that distinctly surpasses conventional methods. Thus, our research provides valuable insights for practical applications, highlighting the substantial benefits of our integrative approach.

Future Work While we focused on the IndexFlatL2, IndexIVFFlat, and IndexIVFPQ techniques, there exist a plethora of other indexing methods. Exploring these alternatives could potentially lead to the discovery of even more efficient strategies for handling high-dimensional datasets. Our study was concentrated on the optimization of the 'k' parameter and the cache size, keeping in mind their significant impact on search efficiency. However, there exists a multitude of additional parameters, such as the number of probes in IndexIVFPQ, that could also be tuned to fine-tune performance. It's crucial to note that the optimal number of probes may vary based on the specific characteristics of different datasets, and thus, future work could involve an investigation of this parameter, but within the context of the task and dataset at hand.

In this research, conclusions based on four datasets selected in *Efficient Streaming Isomorphism work*. However, the performance of indexing techniques could be influenced by the size and diversity of datasets. Thus, it would be beneficial to further extend this analysis to a more extensive range of larger and more varied datasets. While the performance metrics in this study were primarily centered around search time and accuracy, future research could benefit from considering additional performance metrics. Depending on the application at hand, other metrics like precision, recall, or F1-score could be pivotal and thus, their exploration in the context of high-dimensional vector space search may yield valuable insights.

Lastly, given the ever-increasing size of datasets, optimizing hardware resources presents a promising avenue for enhancing search efficiency. This investigation primarily employed CPU resources; however, the untapped potential of Graphics Processing Units (GPUs) warrants further exploration.

GPUs, due to their architecture designed for parallel processing, serve as robust accelerators for managing large-scale vector databases. This design enables simultaneous execution of thousands of threads, offering significant advantages for the computationally intensive tasks involved in vector database management. These include, but are not limited to, the acceleration of vector search operations through the deployment of parallel computing strategies and efficient indexing methods. Moreover, the transition from single GPU utilization to multi-GPU configurations further enhances the search efficiency. This is achieved by distributing the computational workload across multiple GPUs, thus facilitating the processing of larger datasets. An effective strategy is data sharding, wherein the dataset is divided and allocated across multiple GPUs.

Another advantage offered by GPUs is the flexibility of using mixed precision modes, such as float16 or float32. By toggling the precision, we can strike a balance between performance, memory consumption, and accuracy.

However, the utilization of GPUs also introduces the critical consideration of resource management. The transfer of data from CPU to GPU, and between GPUs, should be minimized to circumvent latency issues, especially in the context of large data volumes where data transfer costs can become a bottleneck. Additionally, the relatively scarce memory resources on GPUs compared to CPUs make memory management a pivotal factor in optimization. Strategies must be formulated for temporary memory allocations for intermediate computations, such as the use of scratch memory, to maintain a balance between memory utilization and computation speed. And the performance gains may vary depending on the GPU architecture, and multiple GPU usage introduces complexities regarding computation distribution and memory management across different devices.

In conclusion, the potential of GPUs to enhance vector database search efficiency is clear. However, leveraging their full capabilities requires a nuanced understanding of their architectural advantages and computational challenges. Future research should thus center on developing advanced GPU-based strategies that maximize the potential of this technology. This endeavor perfectly aligns with the ongoing exploration of the Batched Streaming Subgraph Isomorphism task based on Vector Database. With a continuous refinement of our methods and a deep exploration of the field's complexities, our primary goal is to excel in the efficient management of high-dimensional vector space searches while accommodating a high volume of tasks, whether they arrive in batches or as a steady stream. This targeted approach encapsulates our endeavor to overcome the challenges and improve our understanding of this dynamic domain.

References

[1] Wenfei Fan. Graph pattern matching revised for social network analysis, 2012.

[2] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation, 2019.

[3] John W. Raymond, Eleanor J. Gardiner, and Peter Willett. Heuristics for similarity searching of chemical graphs using a maximum common edge subgraph algorithm. *J. Chem. Inf. Comput. Sci.*, 42(2):305–316, 2002. Publication Date: February 9, 2002.

[4] Yang QingWu and Sing-Hoi Sze. Path matching and graph matching in biological networks. *Journal of computational biology : a journal of computational molecular cell biology*, 14:56–67, 01 2007.

[5] Hanjun Dai, Chengtao Li, Connor Coley, Bo Dai, and Le Song. Retrosynthesis prediction with conditional graph logic network, 2019.

[6] Chi Thang Duong, Trung Dung Hoang, Hongzhi Yin, Matthias Weidlich, Quoc Viet Hung Nguyen, and Karl Aberer. Efficient streaming subgraph isomorphism with graph neural networks, jan 2021.

[7] Steven Wang, Runxin Wu, Jiaqi Lu, Yijia Jiang, Tao Huang, and Yu-Dong Cai. Protein-protein interaction networks as miners of biological discovery. *Proteomics and Systems Biology*, May 2022.

[8] Hirak Kashyap, Hasin Afzal Ahmed, Nazrul Hoque, Swarup Roy, and Dhruba Kumar Bhattacharyya. Big data analytics in bioinformatics: A machine learning perspective, 2015.

[9] Sudip Mittal, Anupam Joshi, and Tim Finin. Cyber-all-intel: An ai for security related threat intelligence, 2019.

[10] S. Noel, E. Harley, K.H. Tam, M. Limiero, and M. Share. Chapter 4 - cygraph: Graph-based analytics and visualization for cybersecurity, 2016.

[11] Kirk Ogaard, Heather Roy, Sue Kase, Rakesh Nagi, and Kedar Sambhoos. Discovering patterns in social networks with graph matching algorithms, 2013.

[12] TingHuai Ma, Siyang Yu, Jie Cao, Yuan Tian, Abdullah Al-Dhelaan, and Mznah Al-Rodhaan. A comparative study of subgraph matching isomorphic methods in social networks. *IEEE Access*, 6:66621–66631, 2018.

[13] Ji Gao, Xiao Huang, and Jundong Li. Unsupervised graph alignment with wasserstein distance discriminator, 2021.

[14] Frank Eichinger and Matthias Huber. On the usefulness of weight-based constraints in frequent subgraph mining, 2011.

[15] Simon Belieres and Nicolas Jozefowiez. A Graph Reduction Heuristic For Supply Chain Transportation Plan Optimization, June 2018.

[16] Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, Sungchul Kim, Anup Rao, and Yasin Abbasi-Yadkori. A structural graph representation learning framework, 2020.

[17] Fenxiao Chen, Yun-Cheng Wang, Bin Wang, and C.-C. Jay Kuo. Graph representation learning: a survey. *APSIPA Transactions on Signal and Information Processing*, 9:e15, 2020.

[18] Xin Zheng, Yixin Liu, Shirui Pan, Miao Zhang, Di Jin, and Philip S. Yu. Graph neural networks for graphs with heterophily: A survey, 2022.

[19] Jiezhong He, Zhouyang Liu, Yixin Chen, Hengyue Pan, Zhen Huang, and Dongsheng Li. Fast: A scalable subgraph matching framework over large graphs, 2022.

[20] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[21] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases, 2013.

[22] Michael Greenacre, Patrick JF Groenen, Trevor Hastie, Alfonso Iodice d'Enza, Angelos Markos, and Elena Tuzhilina. Principal component analysis. *Nature Reviews Methods Primers*, 2(1):100, 2022.

[23] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching, 2008.

[24] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-tree: Learning data layouts for big data analytics, may 2020.

[25] Shixuan Sun and Qiong Luo. In-memory subgraph matching: An in-depth study, 2020.

[26] Shima Khoshraftar and Aijun An. A survey on graph representation learning methods, 2022.

[27] Xun Jian, Zhiyuan Li, and Lei Chen. Suff: Accelerating subgraph matching with historical data, may 2023.

[28] Tariq N. Khasawneh, Mahmoud H. AL-Sahlee, and Ali A. Safia. Sql, newsql, and nosql databases: A comparative survey, 2020.

[29] Samrat Sahoo, Nitin Paul, Agam Shah, Andrew Hornback, and Sudheer Chava. The universal nft vector database: A scaleable vector database for nft similarity matching, 2023.

[30] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance, 2015.

[31] Ivar Koene, Ville Klar, and Raine Viitala. Iot connected device for vibration analysis and measurement, 2020.

[32] Pietro Michiardi, Damiano Carra, and Sara Migliorini. Cache-based multi-query optimization for data-intensive scalable computing frameworks. *Information Systems Frontiers*, 23:35–51, 2021. Published: March 4, 2020.

[33] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[34] Sivic and Zisserman. Video google: a text retrieval approach to object matching in videos, 2003.

[35] Ondrej Chum, James Philbin, Josef Sivic, Michael Isard, and Andrew Zisserman. Total recall: Automatic query expansion with a generative feature model for object retrieval, 2007.

[36] D. Mukherjee and S.K. Mitra. Vector set-partitioning with successive refinement voronoi lattice vq for embedded wavelet image coding. In *Proceedings 1998 International Conference on Image Processing. ICIP98 (Cat. No.98CB36269)*, volume 1, pages 107–111 vol.1, 1998.

[37] James Philbin, Michael Isard, Josef Sivic, and Andrew Zisserman. Descriptor learning for efficient retrieval, 2010.

[38] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, January 2011.

[39] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(4):744–755, 2014.

[40] J. Pan. Extension of two-stage vector quantization-lattice vector quantization. *IEEE Transactions on Communications*, 45(12):1538–1547, 1997.

[41] P. R. Kanawade and S. S. Gundal. Tree structured vector quantization based technique for speech compression. In *2017 International Conference on Data Management, Analytics and Innovation (ICDMAI)*, pages 274–279, 2017.

[42] Nam Ling and Jui-Hua Li. A vector quantizer with increased codebook patterns. In *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 244–248 vol.1, 1995.

[43] Stephan F. Simon. On the sizes of voronoi cells in entropy-constrained vector quantization. In *1996 8th European Signal Processing Conference (EUSIPCO 1996)*, 1996.

[44] Artem Babenko and Victor Lempitsky. The inverted multi-index, 2012.

[45] Xuguang Ren and Junhu Wang. Multi-query optimization for subgraph isomorphism search, 2016.

[46] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases, 2012.

[47] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2016.

[48] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.