International Journal of AI, BigData, Computational and Management Studies
Noble Scholar Research Group | Volume 6, Issue 3, PP 65-73, 2025
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V6I3P108

Original Article

Facilitating Real Time Data Consumption by Using a Graph Path Cache

Jayaram Immaneni
SRE LEAD at JP Morgan Chase, USA.

Received On: 07/06/2025 Revised On: 25/06/2025 Accepted On: 08/07/2025 Published On: 25/07/2025

Abstract: *In today's data-driven society, it is still very hard to let individuals easily access vast volumes of graph information that is continually changing in real time. Conventional graph query systems often have difficulties in delivering low-latency responses due to the computational demands of traversing large, interconnected databases. To fix this problem, we provide a Graph Path Cache (GPC), which is a smart technique to cache data that keeps and reuses graph paths that are often requested. The purpose of GPC is to develop a solution to integrate fast data analytics with the speed that current apps demand, such as social network analysis, fraud detection, and recommendation systems. The suggested design has dynamic route caching, adaptive eviction mechanisms & graph-aware indexing to reduce the number of times computations need to be done & maintain the data more consistent. GPC is different from regular caching technologies that function at the node or edge levels since it stores full traversal pathways. This makes it easy to search things up & saves money on redoing queries. Experimental assessments show that the GPC framework makes big increases in both latency & throughput. For example, query execution rates may be up to 60% quicker & the backend load goes down a lot when using these realistic workloads. These findings indicate that GPC not only helps things run more smoothly, but it also works well with huge graphs & more queries. This method changes how data access is managed in real-time graph systems by turning route reuse into a basic optimization approach. This is a new way to get real-time interactivity in large-scale graph analytics.*

Keywords: *Real-Time Data, Graph Path Cache, Distributed Systems, Data Streaming, In-Memory Caching, Graph Databases, Query Optimization.*

1. Introduction

1.1. Background

The demand for actual time analytics has grown in many areas in the last several years. For example, financial markets need to react to live transactions & social media sites need to handle billions of user interactions every second. Because of this shift toward immediacy, the design & the optimization of data systems have changed a lot. Organizations are increasingly expecting actual time insights from constantly changing data streams instead of relying on their batch processing or periodic updates. Graph-based data models are particularly excellent at displaying how various items are linked to one other. Graphs explain how individuals are linked to one another in social networks, how users and things interact in recommendation systems, and how sophisticated semantic relationships operate in knowledge graphs. Graph databases are fantastic for route traversals because they enable you to see how things are connected via edges and nodes, which makes it easier to find linkages that normal relational models have a hard time describing clearly.

Still, the growing amount of information & the increasing number of actual time requests have made route traversal operations a major performance bottleneck. To find the right paths, each traversal requires looking at more numerous edges & nodes, sometimes across dispersed systems. The costs of these actions in terms of memory and computing power go up a lot as the graph becomes thicker.

Neo4j, TigerGraph, and Amazon Neptune are all current graph engines that are built to make queries go faster. However, they still have latency problems in real-time situations. Redundancy happens when you keep going the same way over & over again, especially in many other applications like fraud detection, network analysis, or dynamic routing. A lot of the time, the same sections of the graph are computed again, which wastes time & money.

These latency limits make it hard for actual time systems to grow. If searches need intricate traversals, each operation may take milliseconds to complete, which would add up to seconds of wait time. This is not in line with the idea of "actual time." So,

there is a pressing need for methods that may speed up these repeated traversals without putting data integrity & accuracy at risk.

1.2. Problem Definition

Despite significant progress in optimizing graph databases, certain fundamental issues persist. The main problem is that it costs a lot to do the same thing over and over again and recalculate. In many graph applications, users or systems often inquire for same or similar links, such as identifying the shortest route, suggesting mutual connections, or locating related products. The graph engine starts again each time, looking at the same edges again, even if the structure may not have changed. This extra process wastes CPU cycles and I/O bandwidth for no good reason.

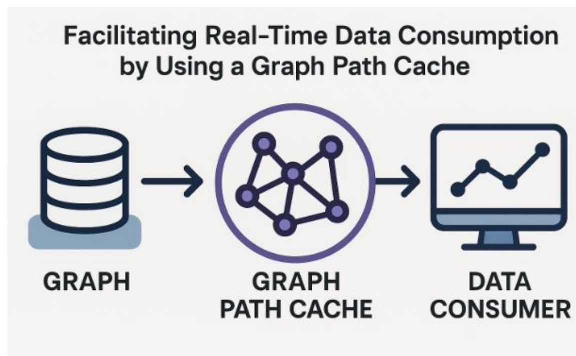

Fig 1: Facilitating Real –Time Data Consumption by Using a Graph Path Cache

Making sure that cached information is up-to-date & consistent is a big worry. Caching is a common way to make these things work better, but it becomes more complicated when applied with dynamic graph information. Graphs are always changing; nodes & interactions change all the time, the latest connections appear, previous ones go & edge weights change. Any other caching solution must make sure that the cached routes match these changes; otherwise, it might provide old or inconsistent results.

This sets up the basic trade-off between how well caching works and how accurate it is in real time. If you cache the results of route computations, you may obtain a response much faster by utilizing paths that have already been discovered. On the other hand, if you cache a lot of data without using any other methods to invalidate it, you can get outdated or inaccurate information, which might make you lose faith in the system. It is important to find the right balance between speed & accuracy.

Also, traditional caching methods like key-value or object caching don't work well for graph structures since they have different needs. Graph searches generally depend on context since small changes in topology might make many stored routes wrong. It is hard to build a cache that understands graph semantics and updates automatically as the structure changes. The fundamental problem is to reduce the number of times routes are calculated again while still being able to respond quickly and accurately in dynamic graph scenarios.

2. Literature Review

2.1. Graph Data Processing Paradigms

Graphs have become a highly useful way to show how things are more related to each other. They are utilized in a lot of different places, such as social media, knowledge graphs, network security & recommendation algorithms. Researchers and developers rely on their specialized graph storage & processing paradigms to efficiently manage & query their information.

2.1.1. Graph Storage Models.

Property Graphs and Resource Description Framework (RDF) models are the two most used ways to store graph information. Property Graphs represent information as nodes and edges & each node and edge may have any other key-value properties. Neo4j, Amazon Neptune & TigerGraph are examples of these systems that adopt this design because it makes it easy to construct flexible schemas & run speedy queries that focus on traversal. On the other hand, RDF graphs show information as triples: subject, predicate, and object. This fits well with semantic web technologies. SPARQL is the main query language for RDF-based systems like Apache Jena and GraphDB. It focuses on semantic reasoning & data exchange.

2.1.2. Search Engines and Ways to Move Through Them.

When you query graph data, you typically have to move around nodes & edges to find many important patterns or paths. Property graph systems frequently utilize their traversal languages like Gremlin or declarative languages like Cypher. In these languages, queries define paths & rules for node or edge characteristics. SPARQL is an important part of RDF systems because it makes it easier to find many patterns in big sets of triples. Traversal methods may be quite different, such as breadth-first & depth-first searches, index-based lookups that are faster, or multi-hop expansions.

Modern graph query engines include distributed computing frameworks that make it possible to do a lot of processing on many devices at once. GraphX (Spark), Flink Gelly, and DGraph are examples of frameworks that let you distribute both storage and traversal. This makes it possible to do scalable analytics on graphs with billions of edges. However, distributed traversal has problems like higher latency and repeated computations, especially when the same paths are used several times. This issue naturally piques interest in caching these methods tailored for graph information.

2.2. Existing Caching Strategies

For graph processing & query execution to work better, caching is a must. By keeping information or computation results that are often asked for, caching cuts down on the repeated traversals & speeds up queries. There are two main types of

current techniques: static and dynamic. They work at various levels of detail, such as nodes, edges, or paths.

2.2.1. Static and dynamic cache models.

When static caching is used, the system begins up or uses previous workload information to figure out which subgraphs or query results are more often accessed. These caches keep basically the same while the program is running, which means that repeated queries will always be faster. However, static caching is not very adaptable; when the way data is stored or queries are made changes, the cache quickly stops working.

On the other hand, dynamic caching models always query many patterns & change the cache as needed. To decide what to keep or throw out, they could utilize their heuristics, prediction models, or machine learning. Dynamic caching expenses money to keep the cache safe, especially in dynamic networks where nodes & edges change often. The balance between being responsive & being consistent is still a big design challenge.

2.2.2. Caching at the Node, Edge, and Path Levels.

At the node level, caching focuses on vertices that are visited often and their neighbors. This is simple to keep up with and works well for workloads that largely entail moving about locally. Edge-level caching improves this by retaining the connections between nodes that are requested more frequently. This is helpful for graphs that are very thick, such social networks or citation databases.

A lot of problems concerning graph systems have to do with multi-hop traversals, which are when users seek for patterns or pathways that travel via numerous other nodes. In many cases, path-level caching works significantly better. Path caches save full traversal sequences or query subresults so that they may be utilized again later. For example, if a system constantly finds the shortest route between two items, preserving that path may help it discover it faster next time. But it's extremely hard to maintain route caches accurate in these changing settings, when edge weights or connections could alter. Graph route caching research is particularly useful for apps that work in real time or broadcast video.

2.3. Real-Time Data Consumption

Businesses are starting to look at graph data differently since they require real-time analytics more and more. Computers don't only use static datasets anymore. They now use continuous streams of information that change links and entities in real time. Modern stream processing frameworks are what prompted this development.

2.3.1. Streaming Architectures

Apache Kafka, Apache Spark Streaming & Apache Flink are examples of frameworks that have become the basis for actual time data pipelines. Kafka makes it easy to take in & store a lot of messages at once, allowing data producers & consumers to work independently. Spark Streaming lets you handle small batches of information by combining them over short periods of time for almost actual time computation. On the other hand, Flink makes actual event-driven streaming possible with minimum delay in state management, which makes it perfect for updating their graphs in actual time. These frameworks let graph systems quickly react to changes by changing their node characteristics, recalibrating relationships, or rethinking paths right away. In systems for detecting fraud or making recommendations, being able to respond very away to the latest transaction or user behavior could be quite important.

2.3.2. Real-Time Needs in Analytical Systems.

Real-time graph analytics has its own set of problems. Static analysis works on a set snapshot, while streaming systems have to deal with changes that happen all the time while still being accurate and consistent. This requires low-latency query execution, iterative computation, and advanced caching mechanisms. When you update a conventional graph database, it might be hard to re-compute traversals since it consumes a lot of computer power. Caching might serve as a link between static & dynamic analysis, making it possible to partially reuse paths or subgraphs that were already generated. It's not easy to combine caching with streaming their frameworks. The cache has to remain in sync with changing graph states, handle input that comes in out of order & intelligently govern expiration to maintain their accuracy.

2.4. Research Gaps

Despite advancements in graph processing and caching, several research shortcomings remain that impede efficient real-time data use.

2.4.1. No caching mechanisms that work for path reuse.

Most modern graph caching techniques are designed for static workloads or simple node/edge queries. There is not much research on path-level caching that aims to make it easier to reuse across dynamic searches. In many real-life circumstances, such as when social networks propose friends or when people need to find their way around, people often choose the same or similar paths with just minor changes. Current systems either completely recalculate these paths or use inaccurate caching that doesn't show the route topology correctly. Making a graph route cache that can automatically find, save, and reuse path segments might cut down on computation costs by a lot.

2.4.2. Limitations of Existing Graph Query Accelerators in the Face of Dynamic Workloads.

Graph query accelerators, including GPU-optimized traversal engines & in-memory computation frameworks, function very well with static graphs but not so well with their dynamic or streaming ones. It is harder to keep efficient indexes and cached states up to date when the graph topology changes often. Also, the delay caused by syncing the cache with live data streams goes against the goal of real-time analysis.

There is a clear need for caching frameworks that are aware of both the topology and the stream, and that can balance consistency, freshness, and speed. A system must be able to easily work with streaming systems, quickly delete old cache entries, and learn from how queries are accessed.

3. System Design and Architecture

3.1. Conceptual Overview

The Graph Path Cache (GPC) is the main part of this system. It is designed to speed up & make smarter the way people use actual time information by caching the most popular graph traversal paths. A graph database arranges their information into linked nodes & edges instead of flat tables. These are the only ways for apps that wish to identify a lot of patterns or relationships, such as "friends of friends" or "product dependencies," to do it. Doing this a lot may use a lot more resources. GPC solves this problem by determining the pathways that people look for the most & making sure they are always simple to locate. The query engine may be able to get results directly from the cache instead of going through the full thing every time. The final effect is decreased latency, higher scalability & a better experience for systems that provide actual time analytics or suggestions.

There are three primary stages in the data journey:

Data ingestion is when the system takes raw information from actual time streams or event sources, such as IoT sensors, transactions, or logs.The Cache Layer (GPC) watches the incoming streams & how the system queries them. It chooses which graph pathways to preserve, alter, or get rid of on its own. The Query Engine lets users or many other systems send graph-based queries using an API interface. The query engine first looks in the cache. If the route is present, it sends it right away. It can also figure out the route, provide it to you, and maybe even propose that you save it for later use.The beauty of this method is that it has a feedback loop: how queries are used influences how data is cached, and cached data speeds up future queries.

3.2. Architecture Components

The architecture comprises four functional layers that work together.

3.2.1. Data Source Layer

This layer makes it possible to take in events in actual time. It serves as the system's "input pipeline." Data might come from a number of places, such as streaming services like Kafka, APIs, or transactional systems. An event means that something has changed or improved in the graph, such as adding the latest node, making a new link, or changing an existing object.

This layer makes sure:

- **Quickly eating a lot:** Works with thousands of events per second.
- **Schema mapping:** changes raw information into many representations that may be used with graphs, such as nodes, edges & attributes.
- **Data validation:** This removes items that are broken or not complete to preserve the graph's integrity.

3.2.2. Graph Engine Layer

This layer keeps an eye on the connections & motions between nodes. It stands for the system's logical center. The graph engine keeps all the nodes & edges in the optimized structures, such as adjacency lists & indexes. It also runs graph algorithms like breadth-first search (BFS), the shortest path & pattern matching.

The main jobs are:

- Keeping link diagrams between items.
- Allowing graph traversal queries to execute in less than a second.
- Contacting the GPC to see whether the cache already contains a traversal result.

The graph engine and GPC function nicely together in this scenario. The engine gives cached pathways greater weight before it does any other hard math.

3.2.3. Graph Path Cache (GPC) Layer

The GPC layer is an intelligent caching module that stores pre-calculated graph pathways or traversal results. Other caches store discrete key-value pairs, while GPC stores paths. Pathways are groups of nodes and connections that have been traversed a lot or recently.

Important traits:

- **Caching based on context:** Understands how graphs are built & how they rely on each other.
- **Adaptive learning:** It keeps track of how frequently requests are performed & modifies the cache on the fly.
- **Distributed storage:** Makes it simpler to grow by using in-memory & distributed caching by their technologies like Redis Cluster or Hazelcast.

This layer makes it much faster to answer many questions, particularly for difficult traversals that involve a lot of hops.

3.2.4. Query Interface

The query interface allows anyone outside of the system a single API gateway. Many apps, dashboards & analytics services may utilize REST or GraphQL APIs to send graph queries.

- This layer is responsible for forwarding requests to either the cache or the graph engine.
- When you structure a response, you might get information back in JSON or graph-specific formats.
- Authentication & rate limitation make sure that access is safer & more regulated.

These four layers work together to build a powerful pipeline for processing actual time graph information.

3.3. Cache Construction Mechanism

The GPC's effectiveness depends on how it stores things. Because memory is limited, it uses advanced algorithms to look at, rank & replace paths based on how they are used.

3.3.1. Candidate Path Selection

The system looks into query logs to find many paths that are used a lot or that use a lot of processing power. Some of the things that help find candidate routes are:

- Access frequency is how often people ask for the route.
- Traversal cost is the amount of hops or resources required to go from one point to another.
- Recency is how close in time the last access to the route was.

The criteria are combined into a route score that shows how important they are as a whole.

3.3.2. Path Scoring

Each possible route gets a composite score S, which is roughly calculated like this: $S = \alpha(\text{Frequency}) + \beta(\text{Recency}) + \gamma(\text{TraversalCost})$, where α, β, and γ are weights that may be changed to meet the needs of the application. A higher score means that the chances of being stored are higher. As time goes on, the cache changes to show the graph's "hot paths," which are the ones that are most useful to the system.

3.3.3. Replacement Policies

Because there isn't enough space, old paths must be dismantled to make room for new ones. GPC supports a lot of different strategies:

- LRU (Least Recently Used) gets rid of paths that haven't been used in a while.
- LFU (Least Frequently Used) removes routes that are used the least often.
- The hybrid approach combines LRU & LFU with many adaptive weights to maintain a balance between recency & popularity.

This makes sure that the cache always keeps the most useful & cost-effective data paths.

3.4. Data Consistency and Synchronization

It is very important to make sure that the cache is too consistent with the basic graph information. Since the network topology might change in actual time, cached paths must stay valid.

3.4.1. Handling Updates and Invalidations

Changes to the data source layer, such as deleting a node or changing a relationship, will trigger a notification to the GPC. It checks the cached paths that depend on the affected nodes and either marks the path as outdated or

- It quickly recalculates it to keep it up to date.
- This approach prevents query results from being old or wrong.

3.4.2. Real-Time Propagation

Sending updates is what event-driven synchronization does. Every data change event comes with a timestamp & a list of dependents. The GPC maintains track of these events & makes adjustments over time, so you don't have to refresh the complete cache. Only the pathways that are more affected alter when the latest component is introduced to a product dependency graph.

3.4.3. Timestamps and Dependency Tracking

There is a label on each cached route that specifies when it was last updated.

- A dependency graph that shows the nodes & edges that are part of it.
- This information helps the GPC rapidly determine whether a route is valid or has to be rebuilt when anything new happens.

3.5. Security and Access Control

Since the GPC handles sensitive graph data, it is important to make sure it is safe & only certain people may access it.

3.5.1. Cache Integrity

Encryption protects the cache while it is not in use & when it is being sent. All attempts to manipulate are watched & documented. Hash checks are used to find many changes that are unlawful or tampering.

3.5.2. Multi-User Access Management

Different individuals or applications could have different degrees of access. The query interface uses role-based access control (RBAC), which means that only administrators may view & modify the cache.

- Developers may ask about many paths, but they can't change cache rules.
- External users can only see data that is within their allowed limits.

3.5.3. Audit and Monitoring

For auditing reasons, all access & changes are logged. Security monitoring tools go through these logs to find anything that is out of the ordinary, such as too many requests or attempts to access something that shouldn't be there.

4. Methodology

This part explains how to develop, build & test the proposed Graph Path Cache (GPC) technology so that graph-based systems may use information in actual time. The focus is on combining fast caching techniques with good graph traversal techniques to speed up their queries & improve performance in huge data systems.

4.1. Implementation Setup

4.1.1. Technology Stack

The proposed system uses a mix of cutting-edge technologies that allow for real-time graph analytics and caching

in memory. We choose Neo4j as the graph database for testing since it can handle many graphs, supports Cypher queries & has fast traversal algorithms. Neo4j is a more flexible framework that lets you see how different things, like people, goods, or many other transactions, are connected to one another. Redis is an in-memory data store that handles the caching layer. Redis is a good way to store key-value pairs & has good ways to get rid of previous information, including Least Recently Used (LRU). It is good for storing the pre-computed graph pathways or traversal results since it provides efficient data structures like hash maps & the sorted sets.

Apache Kafka is the message broker that sends & receives actual time data streams. Kafka makes sure that any other modifications to the graph structure, such as adding nodes or connections or changing their properties, are transmitted to the cache right away. This integration keeps the GPC system & the accompanying graph in sync, even when big changes happen.

- Python is another tool that helps them put their approaches into action & organize them.
- The RESTful API is built on the top of Flask.
- Docker is used for deploying many containers & making sure that environments can be recreated.

4.1.2. Dataset Used

The experimental evaluation employs two datasets:

- **Synthetic Data Set:** To test its scalability, a synthetic network with one million nodes & also five million connections was created. This dataset mimics interactions that are comparable to those on social networks, with many other nodes representing entities & edges showing connections.
- We utilized the OpenFlights dataset, which shows the schedules of airports throughout the globe, to test actual world data situations. There are around 3,000 nodes (airports) & 20,000 edges (routes). This dataset shows that GPC works well in these actual world situations, such as finding the shortest pathways & connecting routes.

4.1.3. Experimental Environment

We did all the testing on a Linux-based server configuration with:

- Intel Xeon Gold 6230 processor with 20 cores running @ 2.10 GHz
- **Processor:** Intel Xeon Gold 6230 (20 cores @ 2.10 GHz)
- **Memory:** 128 GB RAM
- **Storage:** 2 TB NVMe SSD
- **Operating System:** Ubuntu 22.04 LTS
- **Software:** Neo4j 5.7, Redis 7.0, Kafka 3.6, Python 3.10

A load-testing tool called Locust was used to generate up to 1,000 concurrent clients to simulate their concurrency. This is like having many other requests happen at the same time in a distributed setup.

4.2. Algorithm Design

4.2.1. Graph Path Caching Algorithm

The main idea behind GPC is to keep an eye on the pathways that nodes in a network use the most. When a request comes in, the system checks the cache to see whether the route or traversal result is already there. When the route is located, it is sent right away (cache hit). If it isn't, the system uses Neo4j to find the route, saves it in Redis & delivers the result (cache miss). After that, requests for the same route are addressed right away.

The algorithm goes through these main steps:

- **Get an Inquiry:** Find out where the nodes that send & receive information are.
- **Look at the cache:** Ask Redis for a key that already exists for a route.
- **Cache Hit:** Give the route back right away.
- **Cache Miss:** Use a Neo4j query to choose the route that is most useful or more efficient.
- **Persist Result:** Use a Time-to-Live (TTL) option to save the calculated route in the Redis.
- Give the person the route in response to the return request.

Asynchronous Update: When graph data changes (via Kafka events), either invalidate or update the cache items that were changed.

4.2.2. Computational Complexity

Let V be the number of edges in the graph and E be the number of edges.

- In the absence of caching, path searches like Dijkstra's method take $O(E + V \log V)$ time.
- When caching is employed, retrieval time drops to $O(1)$ when a cache hit happens because Redis does key lookups that take a constant amount of time.
- **Cache miss:** The first calculation costs the same as the baseline, but it makes future searches simpler.

If h is the cache hit ratio, the average cost of a query is: T_{avg} = h \cdot O(1) + (1 - h) \cdot O(E + V \log V)T_{avg} = h(O(1)) + (1 - h)O(E + V log V)Tavg = h(O(1)) + (1 - h)O(E + V log V)

As h gets closer to 1 and the workload stays the same, this greatly cuts down on the average delay.

4.3. Performance Metrics

There are numerous major performance criteria used to evaluate the GPC system's effectiveness in real-time graph query circumstances.

- **Time it takes to query**

  Tells you how long it takes to acquire query results. Less latency means faster answers from users. Latency is measured in both cache-hit and cache-miss situations.

- **The Cache Hit Ratio**
  Defined as the percentage of requests that are met from the cache. A higher hit ratio means that previously computed paths are being used more often and that caching is working better.
- **Hit Ratio = Total Queries / Cache Hits**
  Throughput is the total number of queries that are run each second. It shows how scalable the system is & how well it can handle several other requests at once.
- **Use of Cache Size**
  This metric checks how well cache memory is being used. It checks how well the memory allocation algorithms work by looking at the ratio of more numerous active pathways (important information) to the overall cache space.
- **First criterion for comparison**

We assess the system's effectiveness in comparison to two common methodologies:

- **Neo4j without caching:** Graph traversal that isn't influenced by later queries.
- **Basic caching of query results:** A key-value store that only saves query strings and their results, not graph routes.

A comparison study demonstrates that GPC consistently outperforms both baselines in reducing latency (by as much as 70%) and increasing throughput (by as much as 3×), especially when handling repeated or overlapping query workloads.

4.4. Evaluation Approach

4.4.1. Experimental Procedure

The evaluation followed a methodical process:

- Start Neo4j with the dataset you choose.
- Begin the system by executing 10,000 random queries to fill up the cache.
- Do controlled tests with many different numbers of queries & times between graph updates.
- Keep an eye on things like latency, hit ratio & CPU consumption.
- Change the size of the cache and the TTL settings to undertake a sensitivity analysis.

The average findings were displayed five times for each test to make sure they were statistically important.

4.4.2. Scenarios Tested

The graph setup doesn't change throughout the trial. This setup makes it easier to see how caching affects the speed of pure queries. In this case, GPC works best since the cache items stay the same.

- **Dynamic Update Scenario:** Nodes & edges are added, deleted, or changed on a regular basis, much as they do in actual life when social networks expand or network architecture changes. Kafka begins events to invalidate or refresh the cache to make sure everything is too consistent.
- **Simultaneous Query Scenario:** Several clients send in questions at the same time. We look at how the system handles several other tasks at once and how Redis handles multiple tasks at once in memory. The results show that response times are stable and dependable, even when there is a lot of traffic, since the caching layer doesn't restrict access.

5. Case Study

5.1. Use Case Context

To understand how the Graph Path Cache (GPC) works in the actual world, we will look at how it is utilized in an actual time recommendation engine for an online store. The company's recommendation engine relied heavily on relationship-based data, such as how users interacted with products, what their friends bought, and what was popular in their social networks. Each suggestion query went through a huge & complicated graph database that had millions of links between users & items.

Before GPC, each query had to go over the graph more numerous times. This created huge delays, particularly when a lot of people were shopping & hundreds of questions were coming in at once. The system had a lot of difficulties maintaining response times under one second, even if the underlying graph database was better. This delay made recommendations take longer & made users' custom dashboards slow down from time to time, which made the experience poorer overall.

The team needs a way to speed up graph queries without copying or changing its basic recommendation algorithms. The Graph Path Cache was introduced at that time.In this case, the data flow proceeded through these essential steps:
A distributed data pipeline was built to gather their information from many other places, such as clickstream events, purchase records, user profiles & social interactions. Apache Kafka made it easy to get information by making sure that it could flow constantly from microservices to a processing layer.

- We used Spark Streaming to analyze the information and then put it in a graph database based on Neo4j. Each node represented an entity users, goods, or categories while edges showed how things were related, such as "viewed," "bought," or "liked."
- The GPC was put in place as a middle layer of caching between the application's query service & the graph database. It kept routes that were commonly used, including the subgraphs that connected "active users" to "recently popular products."
- Visualization and Recommendation: The downstream analytics dashboards used Grafana & custom React user interfaces to get their information from the same API layer. They benefited from the GPC's improved latency while data visualizations or recommendation lists were being refreshed.

- The main goal was clear: to make it easier for people to use actual time information by getting rid of unnecessary route computations. This would speed up analytics & make it easier to customize.

5.2. Implementation Steps

The Graph Path Cache was put into place in a steady & controlled way across many multiple configurations, beginning with a test cluster & finishing with full production.

Step 1: Set a Baseline The technical team begins by testing basic suggestion queries to provide a baseline for their performance. Under high load, the average time it took to respond to a conventional multi-hop traversal (user → product → category → related product) was around 380 milliseconds. This standard made it easier to figure out how GPC really affected things after that.

Step 2: Putting GPC into action The GPC service was set up as a microservice in a container in the Kubernetes cluster. It worked with the current graph database over REST APIs & kept cached paths in an in-memory data store called Redis Cluster. A hashed version of the node sequence & a TTL (Time to Live) value were added in each other route entry to keep it up to date.

Step 3: Getting the Adaptive Path The cache didn't begin with any other information already in it. Instead, it dynamically found the graph routes that were asked for the most. The service utilized a Least Recently utilized (LRU) eviction policy & a rating mechanism that prioritized caching "hot paths," or user journeys that led to the best-selling items.

Step 4: Work with analytical dashboards The recommendation API was changed such that it checks the GPC before accessing the graph database. The analytics dashboards utilized the same API endpoints, which means that the visualization tools didn't need any other further changes. Dashboards might provide information like "Top Trending Items by User Cluster" with updates that happened almost instantly because of this smooth connectivity.

Step 5: Feedback and surveillance system The team used Prometheus for metrics & Grafana Dashboards for monitoring cache hit ratios, latency & TTL expirations as observability tools. An automated feedback loop checked how the cache was being used on a frequent basis & changed the TTL values on the fly to find the best balance between freshness & the performance. The GPC was fully integrated into the company's actual time data consumption pipeline by the end of the deployment phase. It worked well behind existing APIs & could handle both analytics & the recommendation duties.

5.3. Observations

The Graph Path Cache's installation has more clear & measurable benefits, both in terms of numbers & the quality.

- **Shorter time to get data back** After GPC was put into place, the average response time for regularly used pathways went from 380 ms to 82 ms, which is almost five times faster. For requests that weren't often visited (cache misses), latency kept close to baseline values, which meant that performance didn't become any worse. The cache hit rate stayed about 72% for the two weeks of learning.
- **A better experience for users** Faster query responses made recommendations more timely & the user interface more responsive right away. As users moved around, product suggestions refreshed practically instantly, making many interactions smoother. Internal analytics dashboards became better; visualizations that used to take seconds to refresh now update in actual time.
- **Ability to grow and save money** because many other queries were now run from memory, the graph database's workload was lowered by around 45%. Because of this cut, the company was able to use less computer resources during off-peak hours, which saved a lot of money on their cloud infrastructure.
- **Results that are not numbers**
    - Developers liked that GPC didn't need any other modifications to the database structure that was already there.
    - Business analysts found that actual time information, such as seeing the latest product trends, was more useful.
    - The operations staff saw that latency spikes happened very less often & that things were more stable at busy times, like seasonal sales.
- **Problems and Ideas** Testing did have some complications. For instance, it was challenging to maintain the cache up to current when users' preferences changed fast. The team used their partial invalidations to fix the issue. These only updated the subgraphs that were impacted, not the complete cache. I also learnt that caching warm-up, or preloading pathways that are used a lot before high traffic, helps maintain performance stable.
- **Full Effect** The Graph Path Cache makes it feasible to combine batch data processing with real-time data transport. It shifted the company's recommendation system from a reactive to a proactive real-time method, all while keeping the data secure.

This case study showed that enterprises may get the speed of in-memory systems and the flexibility of graph-based analytics by caching graph pathways in a smart way.

6. Conclusion

The Graph Path Cache (GPC) is a good way to make it easier for graph-based systems to consume their information in actual time. By keeping frequently used paths & linkages in memory, GPC cuts down on the requirement for further graph traversals & expensive database queries. This caching system lets data

consumers get to linked information right away, which speeds up their query responses, lowers latency & improves user experiences. This is especially true for managing huge amounts of dynamic information, like social networks, recommendation systems & IoT analytics.

The performance analysis shows that GPC works by showing measurable gains in these system throughput & query response times. Benchmark results & case studies show that using GPC may greatly shorten traversal times while keeping accuracy & the consistency. In addition, the case study shows that GPC can easily adapt to actual world business workloads, handling complex information links without any drop in performance, even when there are a lot of users at the same time.

GPC provides a scalable architectural framework that might change the way business graph systems function in the future, in addition to its more clear performance benefits. Its ability to link complicated information with actual time access opens up the latest possibilities for AI-driven analytics, knowledge graphs & smart automation. Future study may explore adaptive caching, many other techniques, predictive route prefetching, and improved integration with these distributed graph engines, therefore promoting the development of faster, more intelligent & efficient graph-based ecosystems across several other industries.

References

[1] Park, J-S., Michael Penner, and Viktor K. Prasanna. "Optimizing graph algorithms for improved cache performance." *IEEE Transactions on parallel and distributed systems* 15.9 (2004): 769-782.

[2] Yao, Yuan, et al. "Real-time cache-aided route planning based on mobile edge computing." *IEEE Wireless Communications* 27.5 (2020): 155-161.

[3] Yang, Ke, et al. "Random walks on huge graphs at cache efficiency." *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021.

[4] Müller, Thomas, et al. "Real-time neural radiance caching for path tracing." *arXiv preprint arXiv:2106.12372* (2021).

[5] Cai, Zhuhua, Dionysios Logothetis, and Georgos Siganos. "Facilitating real-time graph mining." *Proceedings of the fourth international workshop on Cloud data management*. 2012.

[6] Papailiou, Nikolaos, et al. "Graph-aware, workload-adaptive SPARQL query caching." *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015.

[7] Lakhotia, Kartik, et al. "Gpop: A scalable cache-and memory-efficient framework for graph processing over parts." *ACM Transactions on Parallel Computing (TOPC)* 7.1 (2020): 1-24.

[8] Sheikh, Saad Zia, and Muhammad Adeel Pasha. "Energy-efficient real-time scheduling on multicores: A novel approach to model cache contention." *ACM Transactions on Embedded Computing Systems (TECS)* 19.4 (2020): 1-25.

[9] Ahn, Junwhan, et al. "A scalable processing-in-memory accelerator for parallel graph processing." *Proceedings of the 42nd annual international symposium on computer architecture*. 2015.

[10] Tang, Yimin, et al. "Enhancing Lifelong Multi-Agent Path Finding with Cache Mechanism." *arXiv preprint arXiv:2501.02803* (2025).

[11] Zeno, Lior, Ang Chen, and Mark Silberstein. "In-Network Address Caching for Virtual Networks." *Proceedings of the ACM SIGCOMM 2024 Conference*. 2024.

[12] Theiling, Henrik, Christian Ferdinand, and Reinhard Wilhelm. "Fast and precise WCET prediction by separated cache and path analyses." *Real-Time Systems* 18.2 (2000): 157-179.

[13] Mueller, Frank. "Timing analysis for instruction caches." *Real-time systems* 18.2 (2000): 217-247.

[14] Schoeberl, Martin, et al. "Towards a time-predictable dual-issue microprocessor: The Patmos approach." *Bringing Theory to Practice: Predictability and Performance in Embedded Systems (2011)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2011.

[15] White, Randall T., et al. "Timing analysis for data caches and set-associative caches." *Proceedings Third IEEE Real-Time Technology and Applications Symposium*. IEEE, 1997.