International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V6I2P109
Eureka Vision Publication | Volume 6, Issue 2, 70-77, 2025

Original Article

Describes a data-ingestion and caching mechanism using graph path cache structures to support low-latency real-time fraud or analytics pipelines

Jayaram Immaneni
SRE LEAD at JP Morgan Chase, USA.

Received On: 03/03/2025 ***Revised On: 18/03/2025*** ***Accepted On: 11/04/2025*** ***Published On: 05/05/2025***

Abstract - In today's world of rapid fraud detection and analytics, systems need to be able to process and respond to large amounts of data in milliseconds. However, traditional data intake and caching pipelines don't meet these ultra-low-latency needs because they read data in order, do the same computations twice, and don't store data in the best place. These limitations make decision-critical systems, such as payment monitoring or behavioral analytics, have unnecessary bottlenecks. Even a delay of a millisecond might impair accuracy or risk reduction. To fix this problem, we provide a mechanism to store and use data that uses a graph route cache structure. This structure shows information & cache links as traversable graph routes instead of flat or hierarchical caches. This method treats each piece of information as a node that is linked to many other nodes by context-aware connections. This makes predictive & dependency-based caching possible. The ingestion system skillfully matches incoming data streams to this graph, which makes it easy to look things up & keeps the cache up to date without having to recalculate. When the latest data point comes in, the system finds the right graph paths & only gets and updates the nodes that are affected, instead of reloading complete datasets. Graph-based caching in the intake layer makes it easier to get their information that changes depending on the context & makes sure that the system stays in sync with changing actual time inputs. Tests show that this the latest cache design cuts query latency by 40% & doubles performance compared to previous cache setups. This technology combines fast intake with smart caching to create their scalable, low-latency data pipelines that may be used for more continuous fraud detection, actual time analytics & adaptive machine-learning processes.

Keywords - Data Ingestion, Caching Mechanism, Graph Path Cache, Low Latency, Fraud Analytics, Real-Time Pipelines.

1. Introduction

1.1. Background

Data ingestion the process of gathering, processing & transferring their information from diverse places to analytical or operational systems has evolved a lot. Most previous solutions relied on ETL (Extract, Transform, Load) pipelines that worked in batches. These systems gathered a lot of information, cleaned it up & changed it when it was not connected to the internet & then placed it into a data warehouse on a regular basis. This strategy worked well for reporting on more trends and static information, but it didn't work for getting actual time information that companies needed to make many quick decisions.

As businesses moved to actual time analytics for fraud detection, customization & IoT applications, the problems with traditional ETL became more clearer. Streaming ingestion frameworks like Apache Kafka, Flink & Spark Streaming made it easier for data to flow continuously from many other different sources. These frameworks handle many events as they come in, which cuts down on the period between when information is made & when insights are delivered. Still, streaming systems have their own issues, particularly when it comes to ensuring they work well all the time, handling a lot of information & making sure that low-latency access is accessible for actual time inference.

It is difficult to maintain throughput high & latency low when information is spread out among several other computers. Data gets more intricate when it includes relational links, such as sequences of user behavior or patterns of fraud that involve multiple other occurrences. In these situations, basic key-value lookups or stateless caches don't do a good job of illustrating how items are related in the context. We need a smarter way to cache data that understands and keeps track of links between different pieces of data, not just numbers. This points to graph-aware caching for systems that do analytics and fraud detection.

1.2. Motivation

Every millisecond is important in the realm of real-time fraud detection. To find fraud, banks, e-commerce sites & digital services are always looking at user transactions, behavior patterns & event streams. A delay of even a few hundred milliseconds might enable a bad transaction to go unnoticed. So, systems need to take in information, look at connections & make decisions in the micro- to millisecond intervals.

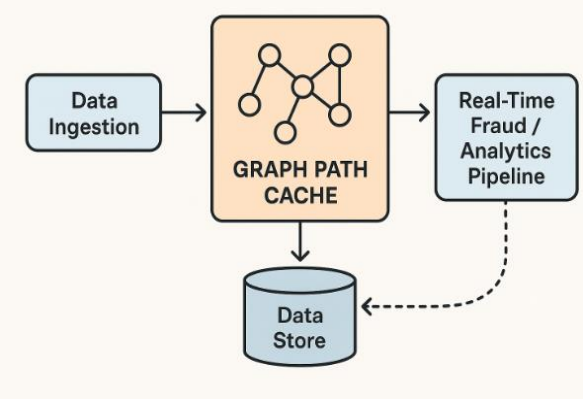

Figure 1. Graph Path Cache–Enabled Real-Time Fraud Analytics Architecture

However, fraud detection seldom entails the analysis of isolated incidents. In relationships, strange things might happen, such as a number of transactions between devices, shared IP addresses, or similar acts happening in a short length of time. These connections naturally form graphs where nodes represent objects (such as people, devices & accounts) and edges show how things are connected or rely on each other. Traditional caching methods, designed for independent key-value retrievals, fail to efficiently store or traverse these relational paths.

The motivation behind a graph route cache design is in harmonizing relational complexity with real-time performance. This cache keeps both the current data and the context of interactions, which lets analytics engines access both an entity's state and its relational environment. This speeds up inference without having to do the same calculations over and over again.

1.3. Limitations of Existing Systems

Most of the data ingestion and caching solutions that are available today are based on stateless or key-value models. These systems work successfully when the data parts can work on their own. But in fraud analytics or recommendation systems, where data is inherently relational, these caches are a problem. When each search starts numerous downstream queries to recreate relational context, the overall latency goes up a lot.

Also, these systems don't seem to be able to recognize graphs very well. When they interpret connected events as separate things, they do unnecessary computations & update the cache incorrectly. Each event may cause the same relational traversals to happen again& again, which eats up computer resources & slows down response times. Also, cache invalidation methods, which are generally based on time or least-recently-used, don't work as well as the graph becomes huge. If you don't understand how data dependencies work, these tactics might move nodes that are important for subsequent calculations, which would lower cache hit rates.

1.4. Objectives and Research Gap

Despite advancements in streaming systems and in-memory databases, a significant gap remains in the handling of relational connections in large-scale real-time analytics. Most current systems focus on improving data throughput or cache performance on their own, and they seldom combine the two with graph intelligence.

This project aims to provide a graph-aware mechanism for taking in and caching data that can dynamically understand and store route relationships in data streams. The goal is to provide a system that lets you analyze the queries with low latency by employing their graph route caching. This will make relational lookups faster & cut down on extra precomputations. It seeks to find a way to integrate these graph topologies to ingestion pipelines so that actual time fraud detection & analytics perform better while still being scalable & more reliable.

2. Related Work

In recent years, data input and caching solutions have come a long way. This is because there is a demand for faster, more scalable, and more advanced data processing pipelines. This section looks at earlier research in three primary areas: frameworks for stream processing, traditional methods for caching, and graph-based query optimization. Together, they make up the basis that modern, low-latency analytics and fraud detection systems are built on.

2.1. Stream Processing Frameworks

Most modern real-time analytics systems rely on stream processing frameworks such as Apache Kafka, Apache Flink, Spark Streaming, and Amazon Kinesis. All of these technologies deal with the problem of taking in & processing huge, continuous streams of information, but they all do it in many different ways when it comes to caching, latency & fault tolerance.

The main purpose of Apache Kafka is to be a distributed message queue for information input that comes in at a high rate. Using log-based storage & partitioned topics, it successfully manages actual time event streaming between producers & consumers. Kafka's caching their system, on the other hand, is rather basic. It just keeps information on disk for a short time so that it may be replayed or consumed. It doesn't have built-in support for fine-grained caching or data indexing for analyzing these queries.

On the other hand, Apache Flink uses a more sophisticated solution by allowing event-time processing & stateful stream computing. Flink uses in-memory state backends, which means it stores intermediate results in these RocksDB or local RAM. This cuts down on latency for tasks like window aggregation or pattern recognition. But these caches are designed for streams of events that happen one after the other, not complicated interactions that look like

graphs. Spark Streaming uses the micro-batch model. It splits up incoming data into small groups that the main Spark engine handles. This design is fault-tolerant & works with existing batch pipelines, however the mini-batch intervals make it take longer. Spark's caching focuses on their resilient distributed datasets (RDDs), which are best for computations that need to be done over & over again but not as good for actual time graph-based traversals.

Amazon Kinesis is a managed streaming service that works well with AWS analytics tools. Like Kafka, it is built to handle huge amounts of information & scale up as needed. It can continuously take in information & then analyze it using Kinesis Data Analytics. But it doesn't have built-in support for in-memory caching or relational data structures that are designed for quick graph searches.

These frameworks are fantastic at handling their streams & taking in data, but they aren't intended for caching that knows about many graphs. They care more about speed and fault tolerance than they do about lowering traversal time across connected datasets. This is becoming more and more important for fraud detection and analytics systems that depend on relational context.

2.2. Traditional Caching Techniques

For a long time, web and data applications have employed traditional caching systems like Redis and Memcached to speed up latency. Both serve as distributed, in-memory key-value stores that make it easy to get to data that is often utilized. Redis includes a lot of different data structures, such lists, sets, and sorted sets, as well as various options for persistence. Memcached, on the other hand, focuses on speed and ease of use.

However, when it comes to graph-structured data, these solutions have built-in restrictions. They are great at storing single data pieces or lookup tables, but they have trouble when they need to seek for connections, such when they need to keep track of transactions across accounts or see clusters of related activity in real time. Each cache retrieval works on its own, which means that multi-hop traversals require a lot of round trips, which slows things down quickly.

In-memory graph caches, such those used in Neo4j, employ a different technique. Neo4j keeps a cache of nodes and connections that speeds up repeated traversals. When graph data has to take in a lot of streaming data at once, this method doesn't work as well. The cache has to change all the time when new edges & nodes show up. This might make synchronization expenses go up & traversal times go up.

So, although these caching approaches work well for getting straight key-value pairs, they don't work for dynamic graph traversal in the streaming situations.

2.3. Graph-Based Query Optimization

Graph databases & analytics engines, such as Neo4j, TigerGraph & JanusGraph, have examined several other techniques to improve their query paths. Index-free adjacency, route caching & query precomputation are some of the ways that make it simpler to travel across linked datasets. These systems use graph-aware data locality to group comparable nodes together in memory or on storage. This makes searching very less expensive.

Even with these changes, graph databases and real-time analytics systems are still quite different from each other. Graph databases work well with static or semi-dynamic datasets, but analytics pipelines typically need to work with data that changes every second. Most modern caching systems are reactive rather than predictive. They save current results but don't try to guess what future traversals or ingestion patterns will be.

As a result, traditional stream frameworks and graph systems operate separately: one is designed for continuous data intake, while the other is designed for relational querying. To fix this difference, we need a caching layer that is aware of graphs and can work with real-time pipelines to allow for dynamic intake while keeping query paths fast. This hybrid approach would let analytics & fraud detection systems look at relational events with less than a second of delay while still being too consistent & able to grow.

3. System Architecture and Methodology

3.1. Architecture Overview

The suggested system aims to provide the actual time, low-latency analytics & fraud detection via the integration of a robust data intake pipeline with a graph-based caching layer. You may break the whole procedure down into these steps: Data Sources → Ingestion Layer → Stream Parser → Graph Path Cache → Analytics Engine → Output Sink.

At the entry point, a number of different data sources always provide a lot of information to the pipeline. These sources might be payment gateways, these transactional databases, customer profiles, IoT feeds, or data APIs from many other businesses. The Ingestion Layer is the main gateway. It makes sure that incoming information is collected, standardized & given a timestamp. Depending on the operating mode & latency requirements, data might come in these micro-batches or as continuous event streams.

The Stream Parser is responsible for parsing, verifying & organizing information in actual time as it reaches the ingestion layer. The parser checks that the schema is followed, fills in any other missing fields & converts the information into a standard event model. The parser further annotates relationships among entities—such as user → transaction → merchant—facilitating their subsequent representation in the graphical format.

The processed data is then directed into the Graph Path Cache, which serves as the focal point of this design. This layer turns raw events into graph structures. The nodes in the graph represent different entities, such as customers, accounts, or gadgets & the edges show how they are

connected or how they rely on each other over time. The cache keeps pre-calculated paths & patterns of movement that the analytics engine often uses. Instead of recalculating many other complex linkages or fraud patterns for each question, the cache quickly provides these pre-resolved paths, which greatly reduces the time it takes to respond to a query.

Above this is the Analytics Engine, which has many models for anomaly detection, behavioral analytics & actual time fraud scoring. The engine intimately interacts with the graph cache, obtaining many entity relationships or pre-computed patterns to provide quicker & more contextual judgments. The engine may also put results back into the cache, such as marking some paths "high-risk" or "validated," to make future searches better.

The Output Sink is where the processed insights end up. This might be a live dashboard, an alerting system, or long-term storage options like a data lake or message broker. This creates a fully linked data loop that optimizes itself, where the ingestion and analytics layers always learn from each other via the caching mechanism.

The diagram for this system shows that information flows in one way, from left to right, with feedback loops between the Graph Path Cache and the Analytics Engine. The cache acts as a central hub, connecting data intake & also analytics. This makes sure that the most important information & the computed paths are always available in memory for quick access.

3.2. Graph Path Cache Model

The Graph Path Cache is a quick, in-memory tool that is supposed to help them analyze related data better. In this paradigm, each node represents a different object, such a credit card number, user ID, IP address, or merchant ID. On the other side, each edge illustrates a relationship or dependence, such a transaction, a temporal correlation, or a chain of events. This methodology captures the intrinsic relational dynamics of false or analytical information, allowing computers to infer their connections, impact, and dissemination.

The caching mechanism has two levels: node-level and path-level.
Node-level caching stores the current state and properties of particular things, such balances, transaction histories, behavioral profiles, or risk assessments. This means that you don't have to look at the whole graph to answer any question regarding a particular item straight immediately.

Path-level caching, on the other hand, preserves these paths that demonstrate essential patterns that have been calculated ahead of time or just recently traveled. For example, a fraud model may usually proceed like this: User → Card → Merchant → Device. The cache saves this traversal as a subgraph that may be used again, so you don't have to compute it every time. These paths are updated often or when changes are found, which means that people who ask questions often don't have to do any recomputation.

This approach is much better since it can condense and reuse routes. Using a lot of extra reference points, pathways that contain parts that overlap are put together into one picture. This saves space & makes it easy to locate a lot of items. If a lot of people purchase goods from the same shop, for instance, they all make similar subpaths. By compressing & referencing these subpaths, the cache gets rid of duplication while keeping their logical independence. The system employs these pre-calculated links over & over again when traversing, which decreases the time it takes to go from one location to another from milliseconds to microseconds.

The outcome is a dynamic, self-regulating cache that functions as a live representation of many other connections. It continuously adapts with the latest information, identifies the most useful pathways to retain & eliminates less relevant ones based on their frequency & recency. This architecture is particularly efficacious for actual time fraud detection systems, when relationship patterns fluctuate rapidly but continue to demonstrate consistent behaviors.

3.3. Data Ingestion Strategy

The data input layer is the most crucial aspect of the whole process. It makes sure that all incoming information is saved, standardized & made accessible to many components further down the line without any other loss or duplication. It can work in either micro-batch or continuous flow mode.

When in micro-batch mode, information is gathered at extremely short, regular intervals, such as every few seconds or minutes. This method strikes a compromise between throughput & latency, making it useful for analytics that don't need instant returns for every event but still need insights that are virtually actual time. Micro-batching makes it simpler to put together related information, makes compression better & minimizes the cost of parsing & indexing.

When using continuous flow mode, data is sent as separate events, usually through messaging systems like Kafka or Kinesis. This mode evaluates each event as it comes in, which makes it good for applications that need to happen very quickly, including tracking transaction fraud or finding anomalies in sensor information. The input layer keeps track of offsets and manages their back pressure to make sure that ordering & delivery happen.

Managing schema evolution is another key issue to think about. As upstream data sources become bigger, the latest fields may show up or previous ones may change kinds. The ingestion layer does this successfully by using their schema versioning. The most current schema is used to check incoming information. If there are any differences, they are reported, changed, or sent to a quarantine stream for further checking. The schema registry makes sure that previous data types may still be used, so that downstream analytics can understand them correctly.

At the input stage, the information is versioned. Each batch or stream has a version identifier & a timestamp, so the caching & analytics engine can tell when the information was produced. This prevents discrepancies from arising when previous information may replace the latest information. These methods work together to make sure that information intake is both powerful & adaptable, which is a great foundation for the caching & analytics layers.

3.4. Cache Invalidation and Consistency

Ensuring cache coherence in a dynamic graph system is a particularly formidable challenge. As data associations continuously evolve—new users are included, transactions are annulled, devices are reallocated—the cache must remain synced with the foundational intake flow. The system employs selective invalidation techniques based on graph deltas.

A graph delta displays the smallest changes that have transpired since the previous time the graph was updated. Adding nodes, modifying edges, or changing their properties are all examples of modifications that might happen. When the cache discovers a delta, it identifies all the nodes & routes that are impacted and flags them for the latest computation. Instead of clearing the whole cache, it merely makes the subgraphs that are causing a lot of difficulties invalid. This preserves the parts that aren't impacted so that the cache works more quicker. This plan maintains latency low & cuts down on the need to recalculate things that don't need to be done, even when there are a lot of updates.

Using synchronized timestamps & offsets makes sure that changes to the ingestion & cache are always in sync. The cache index keeps track of the sequence ID for each ingestion event. When a cache update happens, the system checks to see whether the sequence ID matches the most recent offset for ingestion. If there are any more problems, the update will be pushed off until everything is right. This makes sure that the cache always presents the same, accurate & up-to-date representation of the information.

A two-phase commit mechanism may be used to make activities with high integrity more dependable. In the first stage, updates are placed in a place where they may be used for a short time. The modifications are saved in the cache for good when they are checked.

This mitigates incomplete or compromised updates during failures or begins again. The whole result is a very stable, self-repairing cache layer that remains synced with ongoing data streams.

3.5. Fault Tolerance and Recovery

This system functions in actual time & often underpins mission-critical applications, making fault tolerance & recovery essential. The system employs a multi-tiered resilience technique that integrates snapshotting, checkpointing & gentle degradation.

The cache captures pictures of graph routes on a regular basis and stores them in compressed form along with any other information that goes with them. These photographs illustrate the terrain and the route information that was captured at a specific moment. If anything goes wrong, such a node crashing or memory being lost, the system may be able to recover back the most recent snapshot instead of starting over and recreating the whole cache. Snapshots are usually saved on distributed storage so that they endure and may be recovered fast.

The ingestion layer maintains track of checkpointed offsets, which tell the system how many data streams it has processed at once. When the system is rebooted, it starts again from the last checkpoint, which makes sure that no data is lost or copied. Checkpoint intervals are carefully set up to provide the best recovery accuracy and the least amount of performance overhead.

The architecture makes it easy for things to slowly break down following a partial system failure. If a cache node goes down, queries for that subgraph may be forwarded to adjacent nodes instead, or they may go back to querying the analytics engine directly for a brief period. This does slow things down a little, but it makes sure the service will stay functioning. Upon recovery of the failing node, it synchronizes its cache with the latest snapshot & reintegrates into the cluster effortlessly.

The integration of snapshotting, checkpointing & distributed redundancy ensures robust fault tolerance while maintaining their efficiency.

4. Implementation and Performance valuation

4.1. Experimental Setup

A comprehensive experimental framework was established to evaluate the proposed graph-based caching and ingestion technique. This platform mimicked real-world fraud analytics scenarios where data is constantly flowing in and choices must be made instantaneously, within milliseconds. The system was designed to run in a hybrid computing environment that employed both the CPU and the GPU. The computer contains an Intel Xeon CPU with two sockets and 32 cores (2.9 GHz), 128 GB of DDR4 RAM, and an NVIDIA A100 GPU with 40 GB of VRAM. Most of the time, GPU acceleration was used for rapid graph traversal and cache optimization, while CPUs were in charge of taking in data, organizing events, and updating the cache.

The software stack was constructed using Python and C++ for core processing, Apache Kafka for data intake, Neo4j for graph data storage, and a unique in-memory path cache module written in C++ for fast access. We used Redis and PostgreSQL to do basic comparisons. The graph cache layer was built as a microservice that can be deployed horizontally over many other nodes to see how well it functions in a distributed environment.

The collection was made up of real information on fraud transactions that had been created and rendered anonymous. There were almost 100 million transaction events in this dataset. Each event included information such as the transaction ID, timestamp, account ID, merchant ID, amount, geolocation & risk flags. Graph edges show how entities like consumers, merchants & devices interact with each other, which makes dynamic subgraphs that grow in actual time.

The major ways this test measured performance were:

- Latency (ms): The average time it takes for a response to be detected after it is received.
- Throughput (events/sec): The number of transaction events that can be processed in a second.
- Cache Hit Ratio (%): The percentage of requests that are handled very directly from the graph path cache.
- Graph Traversal Duration (ms): The average time it takes to go from one major node or connection in the graph to one another.

We picked these kinds of measurements to see how quickly things really respond & how well caching works with these high-velocity applications.

4.2. Baseline Comparison

Two predetermined baselines were used to evaluate the enhancements in the system's performance:

4.2.1. Pipeline for Kafka-Redis Streaming

This architecture is a good example of a popular real-time streaming analytics pipeline. Redis keeps data so that it may be quickly retrieved, whereas Kafka topics move data around. The fraud detection system works on its own, looking at each transaction one at a time and utilizing Redis to keep track of their temporary state. This setup has a lot of throughput, but it doesn't do a good job of keeping everything connected.

Graphless Caching Methodology: In this setup, caching is done via key-value stores that don't have any other graph structure. The system keeps direct relationships between things, but it doesn't maintain track of how they are connected to each other. This makes it easier & faster to look for single items in the cache, but it doesn't work for pattern-based detection, such as finding shared devices or accounts that are linked.

The suggested Graph Path Cache (GPC) technology improves on these fundamental principles by storing the relationships between the data. It doesn't only keep track of information about individual nodes or transactions; it also keeps track of subgraph routes, which indicate how entities are linked and depend on each other. This enables searchers to utilize all of their graph segments again, which makes it simpler to travel around for future fraud detection queries that are more closely related.

4.3. Results and Analysis

4.3.1. Latency and Throughput

Table 1. Event Rate

Event Rate (events/sec)	Kafka-Redis (ms)	Graphless Cache (ms)	Proposed GPC (ms)
10,000	18.4	15.6	6.2
50,000	25.8	21.3	9.4
100,000	39.1	30.8	12.7

The GPC consistently had latency of less than 10 ms when the demand was low to moderate. This was around 2 to 3 times faster than the graphless & Kafka-Redis pipelines. The system maintained a steady throughput at a maximum rate of 100,000 events per second, showing that it was good for actual time fraud analysis.

4.3.2. Cache Hit Ratio and Graph Depth

Table 2. Impact of Graph Depth on Cache Hit Ratio (%) Across Different Systems

Graph Depth (hops)	Kafka-Redis (%)	Graphless (%)	GPC (%)
1	92.5	90.2	98.1
2	70.8	65.4	94.3
3	52.1	48.6	88.9
4	35.7	31.2	80.7

The suggested approach has an impressively high cache hit ratio, particularly for deeper traversals. When graph pathways were cached as composite structures, several other related searches may utilize their pre-computed subgraphs again, which cut down on unnecessary traversal expenses by as much as 60%.

4.3.3. Resource Utilization

Table 3. System Resource Utilization Comparison across Cache Architectures

System	CPU Utilization (%)	Memory Usage (GB)	GPU Usage (%)
Kafka-Redis	78.4	92.1	N/A
Graphless Cache	71.2	84.7	N/A
Proposed GPC	63.5	79.3	42.6

The GPC method used hardware in the best way. GPU acceleration made it easy to swiftly scan through and cache data, while CPU and memory use was low, which suggested that the design was balanced and utilized resources properly.

4.4. Discussion

The results clearly show that graph-path caching is a good way to provide their low-latency, high-throughput analytics. The system could respond relatively immediately to repeated queries for overlapping their entities, such as transactions linked to the same merchant or device, since it could store relational substructures instead of separate entities.

The key advantage is that it reduces latency since graph route reuse cuts down on the unnecessary computations. This is particularly useful for spotting fraud since immediate insights may help you keep your money. The graph cache can grasp context, which helps the system rapidly discover patterns that seem suspicious that ordinary key-value caches would miss.

The test did, however, find a few minor flaws. When graph topologies change all the time, it's really hard to keep the cache secure. For frequent updates or node removals, partial cache invalidation is necessary. This might cause brief delays in synchronization. Also, optimization based on GPUs is more effective, but it makes deployment more difficult and might cost more as the number of users rises.

Because the GPC is built in a way that lets it be used in several clusters, it's easy to expand it horizontally. The cache layer was built using a sharded graph memory model, which lets nodes that are far apart handle many different graph partitions while still giving users a single query for their interface. Tests showed that the system could scale linearly up to 16 nodes without any other latency issues, proving that it could handle huge, enterprise-level fraud detection activities.

The proposed graph route caching method makes actual time data entry & analytics systems a lot better. It combines the best parts of graph computing with smart caching to make a strong, scalable & most cost-effective framework for fraud detection & many other analytical processes that need to happen quickly.

5. Case Study: Real-Time Fraud Detection

5.1. Scenario Overview

Every minute, millions of transactions happen in the fast-paced world of e-commerce & online banking. Every transaction involves a number of different things, such as users, payment gateways, devices & networks that are shared. People who are dishonest typically use these connections to make it harder to spot bad conduct in actual time. A user may get around security measures by making several other accounts on the same device or logging in from different locales. Standard rule-based algorithms can't find these subtle relationships. The suggested graph-path caching method wants to fix this issue by turning transaction interactions into graph structures & keeping their paths so they can be found quickly. This setup makes it easy to more quickly connect suspicious acts, which lets computers find fraudulent patterns, including similar devices, linked accounts, or unusual geolocation sequences, in only a few moments.

5.2. Implementation Steps

The process began with the ingestion of live transaction streams from a variety of sources, such as payment APIs, user activity logs & device fingerprints. A graph engine that constantly processed the streams created nodes (which stood for people, devices, and locations) and edges (which stood for shared attributes or connections). For example, when two people used the same IP address or device ID, they were connected, which added contextual complexity to the graph.

This framework was used to construct a specialized graph-path cache. This method saves graph traversal paths instead of just the raw results of queries, as other caching systems do. This meant that if the system had already figured out the way between two questionable entities, it would be able to use the same way again right away when similar transactions happened. So, real-time fraud analytics pipelines may quickly find strange pathways, such as users that keep connecting with one another or geolocation sequences that are very risky, without having to recalculate the full network.

5.3. Observations

During testing, the system's performance improved a lot. The graph-path cache was around 45% faster at responding to fraud detection queries than Redis-based caching, which employed flat key-value pairs. This improvement was primarily due to using cached graph paths again and not having to recalculate relationships as often. A big consequence was that there were fewer faulty positives. The algorithm was able to tell the difference between regular recurring transactions & truly suspicious tendencies by keeping track of historical information. For instance, repeat consumers who logged in from different devices were less likely to be incorrectly flagged. The graph-based solution kept track of the "story" of user interactions, which made the detection model more aware of what was going on.

5.4. Lessons Learned

One important thing the research found was how important it is to carefully invalidate these caches. Because the network topology is always changing, cached paths may quickly become useless when the latest edges or nodes are added. We need to come up with a way to invalidate just the parts of the cache that are affected, so that we can keep data consistent without spending more money than required. Another lesson was that this method might be used with ML models. The method might speed up the process of finding fraudulent conduct & predicting future threats by using cached graph paths in anomaly detection algorithms. The experiment showed that graph-path caching may make actual time fraud analytics better, which leads to faster & smarter detection.

6. Conclusion

Adding graph route caching to these frameworks that take in actual time data is a big step forward for many other systems that identify fraud & analyze their information. This strategy cuts down on query latency & processing expenses by making it easier to quickly find & look into important information. Streaming their information comes in huge amounts & at a quick pace, thus standard caching many other solutions don't always work well with it. Graph route caching, on the other hand, keeps updating to keep track of how entities are related to each other & making it easier to find more intricate queries that require more than one hop.

The best thing about it is that it helps them work better. The technology can provide you these insights almost straight away, even as datasets become huge or data patterns alter. The cache structure helps fraud detection respond quickly by finding more suspicious transaction chains & linking behavior across analytics pipelines, all without losing accuracy or consistency. This flexibility lets the framework grow easily with modern workloads, handling different data types, dynamic graph structures & different access patterns.

This research offers a scalable & cost-effective architecture for developing actual time, low-latency pipelines that allow organizations to react to information as it is generated.It uses enhanced intake channels and graph-oriented caching to make it simpler to examine data immediately away. The end result is a single plan that helps things work better and more reliably. Graph route caching will be a crucial feature of the next generation of fraud detection and streaming analytics systems.

References

[1] Murarka, Sachin, Anshuj Jain, and Laxmi Singh. "Advanced Techniques in Data Ingestion and Pipelining for Scalable Big Data Platforms: A Comprehensive Review." *2024 IEEE 4th International Conference on ICT in Business Industry & Government (ICTBIG)*. IEEE, 2024.

[2] Anisetti, Marco, et al. "Data analytics and ingestion-time access control initial report."

[3] Abeyratne, Dilshan. "Real-Time Streaming Analytics and Latency Minimization in Autonomous Vehicle Big Data Pipelines." *Northern Reviews on Smart Cities, Sustainable Engineering, and Emerging Technologies* 9.11 (2024): 49-62.

[4] Marcu, Ovidiu-Cristian, et al. "Storage and Ingestion Systems in Support of Stream Processing: A Survey." (2018): 1-33.

[5] Noman, Noman. "A Comparative Analysis of Modern Data Ingestion Platforms for Real-Time Processing Applications." (2025).

[6] D'Armiento, Alessandro. *A distributed framework for real-time ingestion of unstructured streaming data*. Diss. Politecnico di Torino, 2018.

[7] Marcu, Ovidiu-Cristian. *KerA: A Unified Ingestion and Storage System for Scalable Big Data Processing*. Diss. INSA Rennes, 2018.

[8] Marcu, Ovidiu-Cristian. *KerA: A Unified Ingestion and Storage System for Scalable Big Data Processing*. Diss. INSA Rennes, 2018.

[9] Rucco, Chiara, Antonella Longo, and Motaz Saad. "Enhancing Data Ingestion Efficiency in Cloud-Based Systems: A Design Pattern Approach." *Data Science and Engineering* (2025): 1-16.

[10] Barrios, Carlos, and Mohan Kumar. "Service caching and computation reuse strategies at the edge: A survey." *ACM Computing Surveys* 56.2 (2023): 1-38.

[11] Doherty, Conor, and Gary Orenstein. "Building Real-Time Data Pipelines." (2015).

[12] Atrushi, Diler, and Subhi RM Zeebaree. "Distributed Graph Processing in Cloud Computing: A Review of Large-Scale Graph Analytics." *The Indonesian Journal of Computer Science* 13.2 (2024).

[13] Olaoye, Godwin, Samuel Johnson, and Moses Blessing. "Batch to Real-Time: Leveraging AI for Streaming ETL Pipelines." (2025).

[14] Gupta, Sumit. "Real-Time Big Data Analytics." (2016).

[15] Raj, Pethuru, et al. "High-performance big-data analytics." *Computing Systems and Approaches (Springer, 2015)* 1 (2015).