

Efficient Indexing for Google Zanzibar Based Authorization Systems Using Graph Dynamic-Transitive Closures

Varuni H K*, Sai Smaran Jawalkar*, Sai Sudhir Sunku*, Samarth Ramesh*, and Dr. Vinodha K

PES University, Bengaluru, Karnataka, India

varunihk7@gmail.com, smaran.jawalkar@gmail.com, saisudhirs@outlook.com, dynaclo@samarthr.com, vinodhak@pes.edu

Abstract—Efficient indexing in Google Zanzibar-like graph authorization systems is critical for achieving scalability and low-latency query performance. These systems play a central role in managing access control, ensuring secure and accurate permissions across users, roles, and resources. However, the current reliance on graph traversal-based approaches, such as Breadth-First Search (BFS), limits scalability in deeply nested permission hierarchies. As authorization checks are in the critical path of user interactions, even slight delays can degrade the overall user experience and system responsiveness. To overcome this limitation, we have proposed an indexing framework that employs dynamic transitive closures using support vertices, allowing for efficient permission checks. Our approach involves maintaining forward and reverse reachability graphs for these vertices to optimize query handling, and using a blue-green write queue for parallelizing writes and updates. This work provides a practical solution for optimizing large-scale graph authorization systems, offering faster permission checks and improved system performance.

Index Terms—Authorization, Authorization requests, Google Zanzibar, Graph Dynamic-Transitive Closures, Indexing, SpiceDB

I. INTRODUCTION

Authorization is the process that determines permissions on resources based on the requester's identity. It decides what actions a user or entity is allowed to perform in a given system. This plays a crucial role in securing data and ensuring that resources are accessed only by those who have the right privileges. [1]. Authorization systems are special systems to which the task of determining the permissions of a principal (often user, but can be any entity) is delegated to. Such systems are critical in the continued success of enterprises across industries which rely on software solutions. They regulate access to critical company resources (e.g., payroll, customer data, contract data, among others). They also play a vital role in physical access control.

An Authorization System (Auth System) which allows for implied permissions (like a principal with write access on a resource, must also have read access on said resource) naturally tend to represent itself as a Directed Acyclic Graph (DAG) [2]. The challenge with authorization in dynamic systems, particularly in large-scale environments, is ensuring fast, efficient, and consistent query responses. Many systems

rely on caching mechanisms to speed up response times, which can introduce some consistency trade-offs, making them prone to eventual consistency. These systems often fall into the "AP" (Availability and Partition Tolerance) category when evaluated through the lens of the CAP theorem [3].

The metrics that affect end users are latency, jitters and tail latency. Latency refers to the time it takes for an authorization request to be processed and for a response to be returned. Jitter measures the variability in latency. The worst-case latencies that affect a small percentage of requests is tail latency [4]. It is necessary to have all of these metrics as low as possible and for latencies to be sub-millisecond for frictionless user experience. Authorization plays a crucial role in the Human-Computer Interaction (HCI) path. First and foremost, a slow authorization process leads to delays in the overall process that the user is executing, negatively impacting user experience [5]. According to a 4-week study conducted by Deloitte Digital, a 0.1 second improvement in site load speed increased retail conversions by 8.4% and average order value increased by 9.2%. Fast authorization reduces the risk of exploitation through session hijacking or interception. When authorization is a part of a user's workflow (e.g., editing documents), any delays in the critical path would interrupt the smooth flow of tasks. In high-traffic environments, such as websites or applications with millions of users, higher latency in authorizations can become a bottleneck, affecting system performance [4]. Fast authorization is also crucial for scaling systems that support a large of concurrent users. Graph data structures are particularly well-suited for handling authorization requests because they naturally represent the relationships between users, roles, and permissions. Google Zanzibar, which is Google's Consistent, Global Authorization system uses a Graph to store all its authorization data. In the graph, entities like users, roles, and resources are modelled as nodes, while the connections (e.g., permissions or access rights) are edges. This structure makes it easy to traverse complex relationships quickly. For example, when a user requests access to a resource, a graph traversal can efficiently determine whether the user has the appropriate permissions, even if the relationships span multiple layers of roles or rules. Graphs are also highly scalable, making them ideal for large systems where the number of users, roles, and permissions can grow significantly. Another benefit is their flexibility in handling such permissions. As rules and

* These authors contributed equally to this work.

roles change over time, these updates can be easily reflected in the graph structure. Many real-world systems—like organizational hierarchies or social networks—already resemble graphs, making this approach not only intuitive but also highly efficient for managing dynamic access control. In our earlier work [6], we have explored the design, implementation, and optimization of authorization systems based on relationship graphs, and examined key challenges and potential solutions. SpiceDB the leading open-source implementation of Google Zanzibar, uses Breadth-First Search (BFS) for graph traversal during authorization checks [7]. Although BFS ensures that all relevant permissions are evaluated, it has a key limitation: it can introduce significant latency with deeply nested queries. This is due to BFS's $O(n)$ [8] time complexity, where n is the number of nodes, making it inefficient for graphs with many layers. To overcome this challenge, we propose an optimization using dynamic transitive closure-based indexing. This technique computes the transitive closures of relationships ahead of time and indexes them, allowing for $O(1)$ time complexity for most authorization checks and falling back to BFS in only few cases (around 1%) as driven by a probabilistic algorithm which shall be discussed in detail in this paper, compared to the $O(n)$ complexity in all cases introduced by BFS.

II. RELATED WORK

A. Zanzibar-based Authorization Systems

Google Zanzibar is a unified authorization system that offers fine-grained access control through an object-relation-user mapping-based Access Control List (ACL) approach [2]. It promotes interoperability, supports shared infrastructure like access-control-aware search indexes, and guarantees consistent semantics and user experience across applications. Zanzibar, which was created to solve issues with scalability and data integrity, has emerged as a standard for contemporary authorization frameworks. SpiceDB is the leading open source implementation of Zanzibar [7]. SpiceDB employs a method of distributed query processing with consistent hashing. The system breaks down access requests (PermChecks) into smaller sub-requests and dispatches them across a network of servers using a consistent hashing mechanism. This approach facilitates efficient caching, as sub-requests are routed to the server most likely to have the relevant data cached based on a hash of the sub-request itself. This distribution and caching strategy enables parallel processing of queries, improving scalability. However, SpiceDB currently lacks an indexing mechanism, potentially impacting query performance for complex authorization graphs. Additionally, the core access control check itself still relies on a BFS or on-disk graph traversal, potentially introducing inefficiencies for certain authorization scenarios. The authors of SpiceDB conducted load tests to showcase the performance capabilities of SpiceDB as a system designed for scalable authorization. [9] They aimed to achieve 1 million requests per second (QPS) while maintaining low latency. The test focused on authorization checks for users seeking to access content, view other users' profiles, and

interact with the system. The results indicate that SpiceDB successfully processed 1 million QPS with a P95 latency of 5.76 milliseconds for access control checks on a distributed setup with a combined capacity of 1376 vCPUs and 2.3TB of memory. However, the blog post acknowledges limitations in the testing methodology. The data and test load generator used are not open source, making it difficult to reproduce the results independently and potentially hindering efforts to benchmark SpiceDB's performance under various conditions.

B. Graph Transitive Closure Algorithms

Single-source reachability is an essential component for transitive closures, and as our graphs can undergo additions and deletions, these must be dynamic as the authorization graph changes in real-time. Italiano [10] investigates dynamic solutions for various graph problems such as connectivity, transitive closure, minimum spanning trees, and shortest paths. The paper formalizes the path-insert problem that involves intermixed ADD and SEARCHPATH operations on directed graphs and develops a new data structure supporting these operations in $O(n)$ amortized time and utilizing $O(n^2)$ space. The paper utilizes the concept of representing the transitive closure of the graph by associating each node with a set containing all its descendants, organized as spanning trees. Path retrieval is facilitated by bottom-up traversal of the spanning trees, allowing efficient determination of paths between nodes. However, the data structure lacks support for deletion operations. Roditty [11] uses strongly connected components to compute the transitive closure and provides two fully dynamic algorithms for answering reachability queries: one with a worst-case query time of $O(\sqrt{n})$ (deterministic) and another with $O(m^{0.43})$ (randomized). However, this is only a ten-order-of-magnitude improvement over static BFS and DFS. Hanauer et al. propose a family of algorithms that leverage support vertices and forward and backward reachability graphs to compute reachability information in $O(1)$ time [12]. They introduce dynamic algorithms like SV(k), SVA(k, c), and SVC for maintaining reachability in large graphs. These algorithms use supportive vertices to efficiently approximate reachability without recomputing the entire transitive closure (R^+ and R^-). The SV algorithms dynamically update reachability information, leveraging single-source/single-sink (SI) trees and incorporating periodic adjustments based on changes in Strongly Connected Components (SCCs), which we think is useful for balancing performance and accuracy in dynamic settings like authorization graphs. [12], also presents two Single Source Reachability (SSR) algorithms in their study [?]: the Simple Incremental (SI) algorithm and Simplified Even-Shiloach trees (SES). The SI algorithm extends a BFS tree incrementally for edge insertions and reconstructs it during deletions if the cost is expected to be high. SES adapts Even-Shiloach trees, maintaining a height-minimal BFS tree and updating it with gradual level adjustments during deletions. Both methods support quick reachability queries in $O(1)$ time but involve $O(n + m)$ time complexity for updates. In our current implementation, we manage R^+ and R^- sets using

a key-value (KV) store, simplifying updates and lookups. Pang et al. [13] introduce efficient algorithms for maintaining consistency in Role-Based Access Control (RBAC) models within database environments. Leveraging directed acyclic graphs (DAGs) and transitive closure relations, the paper simplifies authorization processing and query evaluation by pre-computing reachability information. By integrating group, role, and privilege graphs along with user and authorization assignments into a novel RBAC state, redundant assignments and hierarchical relationships within the RBAC model are identified and eliminated. The paper significantly improves query evaluation efficiency, particularly for queries involving reachability, and reduces redundancy within the RBAC model, leading to easier maintenance. We are thus adopting a similar approach akin to Google Zanzibar, referred to as ReBAC (Relationship-Based Access Control), for our authorization system, albeit with considerations for potential scalability concerns in large and dynamic RBAC models.

III. PRELIMINARIES

A. Authorization Systems and Google Zanzibar

The ever-expanding digital landscape demands effective access control mechanisms to safeguard sensitive information and resources. Traditional access control models like Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC) often fall short in addressing complex, dynamic environments. To address these limitations, Relationship-Based Access Control (ReBAC) has emerged, focusing on the relationships between users, resources, and other entities to determine permissions [14]. ReBAC leverages defined relationships (e.g., hierarchical or team-based), relationship-based policies, and contextual information to enable fine-grained and adaptive access control. This approach enhances security and operational efficiency by dynamically incorporating relational attributes and environmental factors into access decisions. A prominent implementation of ReBAC is Google Zanzibar [2], a distributed access control system designed to meet the complex needs of Google's diverse services and platforms. Originating as a scalable and centralized solution, Zanzibar operationalizes ReBAC principles with components such as a distributed SQL database, an indexing system for efficient policy evaluation, a snapshotting system for state consistency, and periodic jobs for background updates. Leopard is an indexing system implemented by Google to optimise operations on deeply nested sets [2] using techniques such as flattening group-to-group relations for faster access. Our novelty lies in employing a support vertex-based algorithm for dynamic transitive closures to answer reachability queries. Unlike Leopard, our implementation maintains a dynamically updated index that provides strong consistency for authorization checks, avoiding the staleness issues inherent in Leopard's snapshot-based approach.

B. SpiceDB

SpiceDB is the leading open source implementation of Google Zanzibar. While answering authorization requests in

SpiceDB, a Breadth-First Search (BFS) is triggered. This search is often slow, because of the number of queries needed to conclusively answer a check permission request. While a cache can mitigate this problem, long tail-request latencies remain an issue. Classical transitive closures (calculated using Warshall's algorithm) can be an "obvious" solution, but, it is not without its problems. The most apparent drawback with using Warshall's algorithm is that every update involves a quadratic operation (updates have quadratic time complexity in Warshall's Algorithm) [15]. While modern dynamic transitive closures do exist, as mentioned above in the example "Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures", they often have large constant factors, or are not easily parallelizable [16]. Write queues offer a way to amortize the cost of recalculating R+ and R-trees over multiple update operations. While they may impact correctness of the responses, the host system (SpiceDB) does not always guarantee that responses take into account the most recent writes. The cost of recalculating R+ and R-trees can be spread out over several update processes with Write Queues. The host system (SpiceDB) does not always ensure that responses consider the most recent writes, even if they might affect the accuracy of the answers.

C. Overview of the Support Vertices based approach for Dynamic Transitive Closures

In SpiceDB, the authorization data is represented as a Directed Acyclic Graph (DAG). A dynamic graph, unlike static graphs, evolves with real-time updates, additions, and deletions of edges and vertices. Maintaining efficient query performance in such a graph is crucial, especially for nested queries, where performing a Breadth-First Search (BFS) at runtime can be computationally expensive. To address this challenge, we propose leveraging precomputed transitive closures to optimize reachability queries and offer an efficient indexing mechanism. [12] introduces a family of algorithms that relies on identifying a small subset of vertices, termed supportive vertices, that act as representatives for their respective graph regions [12]. These vertices facilitate faster query responses by leveraging key observations about reachability relationships.

1) *Reachability Set Definitions:* For a vertex v :

- $R^+(v)$: The set of all vertices reachable from v (outgoing reachability set or forward graph).
- $R^-(v)$: The set of all vertices that can reach v (incoming reachability set or reverse graph).

2) *Key Observations For Support Vertex Based Systems:*

Let v be an arbitrary vertex in the graph, and let $R^+(v)$ and $R^-(v)$ represent the reachability sets. To answer reachability queries between two vertices a and b , we utilize the following observations:

- **(O1)** If $a \in R^-(v)$ and $b \in R^+(v)$, then a can reach b .
- **(O2)** If $a \in R^+(v)$ and $b \notin R^+(v)$, then a cannot reach b .
- **(O3)** If $b \in R^-(v)$ and $a \notin R^-(v)$, then a cannot reach b .

While the first observation ($O1$) is commonly utilized in several algorithms, observations ($O2$) and ($O3$) are less frequently exploited, despite their potential utility in reachability queries. [12]. Using the precomputed Forward and Backward reachability sets and these axioms, we can check graph reachability information in $O(1)$ time, compared to a BFS which is $O(n)$.

IV. IMPLEMENTATION

A. Dynaclo - The Index

Dynaclo is the indexing mechanism which uses the support vertices based transitive closure algorithm described above, to answer reachability queries efficiently. The design and implementation of Dynaclo is described in this section. For our work, we have created a fork of the SpiceDB repository and Dynaclo is implemented as a module which is directly called by the SpiceDB codebase. Dynaclo is currently used in the hotpath of 2 SpiceDB request types - CheckPermission and WriteRelationship. CheckPermission is the request type that is called by client to check if an Entity e has permission to access Object o (eg: Does user Alice have access to file Reports?) This request type is translated to check if node e can reach node o in the stored graph both by SpiceDB internally and in our Index. WriteRelationship is the request type that is called by the client to add a permission to the DB (eg: Grant Alice access to file Reports) This request type is translated to add an edge in the stored graph by both spicedb and our index. Whenever SpiceDB receives a CheckPermission or WriteRelationship request (from an external client), the Spicedb code passes off the request to the Dynaclo Index for handling via a module function call. The specifics of how Dynaclo handles the aforementioned 2 requests is detailed below

1) *CheckPermission*: SpiceDB converts the CheckPermission request into the format of a Reachability Query (Can Node e reach Node o ?) before calling Dynaclo's CheckReachability function which takes in a src node and $dest$ as its arguments. The reachability query is then handled using the following steps:

- 1) If src is a Support Vertex, the R^+ set of src i.e $R^+(src)$ is used to check if dst is reachable from src . If $dst \in R^+(src)$, dst is reachable from src and the Index returns **True** for the Reachability Query.
- 2) If dst is a Support Vertex, the R^- set of dst i.e $R^-(dst)$ is used to check if dst is reachable from src . If $src \in R^-(dst)$, dst is reachable from src and the Index returns **True** for the Reachability Query.
- 3) For each vertex $v \in L_{sv}$ i.e for every support vertex v stored by Dynaclo, Dynaclo tries to apply **O1**, **O2**, **O3** as follows:
 - a) **O1**: If $src \in R^-(v)$ and $dst \in R^+(v)$, i.e if src can reach v and if v can reach dst , then src can reach dst and the Index returns **True** for the Reachability Query.
 - b) **O2**: If $src \in R^+(v)$ and $dst \notin R^+(v)$, i.e if v can reach src but not dst , then src cannot reach dst

and the Index returns **False** for the Reachability Query.

- c) **O3**: If $dst \in R^-(v)$ and $src \notin R^-(v)$, i.e if dst can reach v but src cannot, then src cannot reach dst and the Index returns **False** for the Reachability Query.

- 4) If the above methods fail, as prescribed by the Support Vertex Algorithm by Hanauer et al. [12] Dynaclo falls back to performing a Breadth First Search on the stored graph. In order to do this, we use Onyx which is an embedded graph library we built which is designed for efficient edge-list scans, which has been described extensively in the Section 4.2. Onyx is used for storing the graph in Dynaclo and the Breadth First Search is performed on this graph during the fallback of the algorithm. The result of this BFS is used to answer the Reachability Query

2) *WriteRelationship*: Dynaclo uses a Write Queue for writes. SpiceDB converts the WriteRelationship request into the format of a Edge Insertion (Insert edge from Node e to Node o) before calling Dynaclo's InsertToQueue function with the source and destination nodes as arguments which inserts the write into the WriteQueue. The design and implementation of the Write Queue has been detailed in Section 4.3

B. Support Vertex Implementation

The R^+ and R^- sets are stored as HashSets in Dynaclo. This allows existence checks to happen in $O(1)$ time. Dynaclo picks and stores a list of support vertices L_{sv} at startup. Additionally new support vertices are picked every time writes are applied from the write queue to the graph. These support vertices are picked randomly from all the nodes that exist in the graph. In our implementation we set the number of support vertices picked to be 5.

C. Backing Store – Onyx

Onyx is an embedded, on-disk, concurrent graph database we built which provides efficient edge-list scans and therefore efficient Breadth First Search (BFS). Onyx uses the Badge [17] Key-Value store as its storage engine. Since we use Badger, Onyx inherits many of its features, including transaction support, ACID compliance, and serializable snapshot isolation.

1) *Storage Format*: The graph is stored in the badger key value store as a series of edge lists for every source node. An edge list is a list of all the destination nodes dst for a particular source node src for which the edge $src \rightarrow dst$ exists in the graph. The edge lists are stored in the Badger key-value store, with each key representing a source node and its value as a serialized edge list implemented as a map for efficient $O(1)$ lookups of destination nodes. Serialization and deserialization happens through the use of gob

2) *Onyx Operations*: The core operations in Onyx are adding edges in the graph, removing edges in the graph and getting all the neighbors of a particular node i.e getting all edges of a given source node. Adding an edge involves loading the edge list, checking if the source and destination nodes

exist, creating them if needed, and appending the destination to the edge list for the source node and storing the edge list in badger. Removing an edge involves loading the edge list, deleting the destination node from it, and updating the stored key-value pair. Getting the neighbors of a source node involves simply querying badger with the source node as the key and getting back the edge list for that node. This design decision of storing the edge list as a single serialized value which is queried at once is what makes BFS extremely fast in Onyx. Edge-list scans are very fast as the whole edge list is stored and queried together.

3) *Transactions*: Onyx operations are exposed as Go module functions. All Onyx Operations run inside of badger transactions. The client is allowed to pass in a transaction to the onyx functions, allowing multiple operations to be performed in one ACID Transaction. Otherwise, each operation runs in its own transaction. This allows Onyx to support concurrent read-write transactions and scale to higher throughputs. This is made possible by leveraging Badger's concurrent ACID transactions

4) *Serialization and BFS efficiency*: The use of a map for the adjacency list allows for $O(1)$ lookups to check when a destination node exists in the list, rather than having to traverse through a list. Onyx handles all the serialization and de-serialization of the data to and from the Badger key-value store, allowing it to provide a higher-level graph database interface on top of the underlying key-value store. This design not only optimizes neighbor lookups but also improves BFS efficiency by enabling batch loading of edge lists, minimizing storage access overhead. The code for onyx is available at <https://github.com/anonymous-repo-maker/onyx>.

D. Write Queues – Supercharging an index

A write queue is an essential mechanism for ensuring that index updates are applied in a thread safe, efficient, and correct manner. By providing a controlled, atomic approach to handling writes, a write queue helps establish a lower bound on write granularity—the minimum time required for a *write* to be fully reflected in the index. While most queries do not specify a maximum-staleness requirement, queries that do request a maximum-staleness beyond the last index update are rejected and handled by SpiceDB's internal resolver. This process ensures that the index remains in a consistent state, and any latency from delayed index updates is managed effectively.

The structure of a write queue is designed to handle the complexities of high-concurrency environments. It consists of two primary components: a list of elements (the actual data entries to be written) and a thread safe method for appending new *writes* to the queue. The system operates with two queues at any given time—one active and one inactive. New *writes* are initially added to the active queue, while the inactive queue holds the changes awaiting the next index re-computation. When an index recompute is triggered, the roles of the two queues are swapped: the active queue becomes inactive, and the inactive queue becomes the active queue. During this recompute phase, the changes in the active queue are applied

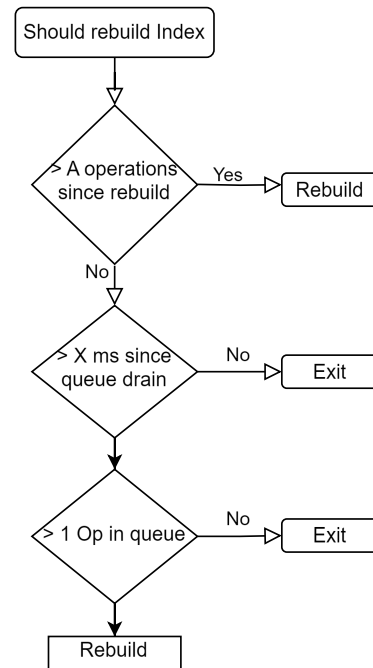


Fig. 1: Decision flow for recomputing index

to the index's underlying data structures, recalculating the associated R+ and R- trees to ensure the index remains up-to-date and accurate.

The write queue mechanism relies on a careful balance between handling incoming writes and the need for index re-computation. When the decision is made to trigger an index recompute, several conditions are taken into account. First, if a sufficient number of writes—specifically more than a predefined threshold, referred to as A—have occurred since the last index update, a recompute is initiated. This ensures that any suboptimal choices for the current state of the index are eventually corrected by better choices over time. Additionally, if a certain amount of time, denoted by X milliseconds, has passed since the last write to the queue, and there are pending operations in the queue, a recompute will also be triggered to prevent the system from becoming stale. The entire decision flow is demonstrated in Fig.1. These conditions help maintain the overall performance and accuracy of the system, especially in high-traffic environments.

To manage index re-computation effectively, a blue-green deployment strategy is employed. This approach ensures that one queue is always available to accept new writes, while the other is dedicated to re-computation tasks. The blue-green system also provides the advantage of reducing the risk of write delays or lost writes during the re-computation process. For example, in a system with a duty cycle of three writes per 100 operations, and an average index rebuild time of 100 milliseconds per operation, the upper bound on write latency will typically be the sum of X milliseconds and the time required to rebuild the index. In practice, this latency should remain under 1 second, even under high load.

In more extreme cases, such as when the index rebuild's time is close to the upper bound of 1 second, the system's performance can still be maintained. Under these conditions, the write queue size can be estimated based on the number of writes per second (RPS) and the expected queue latency. For a warmed-up system, the queue size can be approximated as $(\frac{X}{1000} + 1) \times 3$, ensuring that the system remains responsive and does not become overwhelmed by the volume of write operations. This estimation assumes that the system can efficiently handle the average number of operations without excessive queuing.

To prevent contention and ensure that both writes and index re-computations proceed without interruption, careful application of locks is necessary. Specifically, locks are applied to the current active queue as well as the internal queue managing write operations. This lock management ensures that the system maintains consistency during the index update and recompute phases, avoiding race conditions that could lead to missed or incorrect updates. By carefully coordinating write operations and re-computation tasks, the system can maintain high throughput while keeping index latencies to a minimum.

In summary, the write queue system leverages a combination of thread safe data structures, timed recompute decisions, and blue-green deployment strategies to effectively manage the flow of *writes* and index updates. This architecture ensures that write operations are never missed, that the index is consistently updated, and that the system performs optimally even under high load conditions. With appropriate lock management and careful tuning of parameters such as write latency and queue sizes, a system can maintain low-latency, high-availability performance while ensuring the integrity and correctness of its data indices. The code for our implementation – Dynaclo (forked from SpiceDB) is available at an anonymous repository <https://github.com/anonymous-repo-maker/spicedb>.

V. RESULTS AND DISCUSSION

A. Experimentation

One of the important factors to consider when load testing SpiceDB is the initial data seeded in the SpiceDB datastore. The initial dataset refers to the graph that must be seeded into SpiceDB before the load testing can commence. All subsequent SpiceDB graph operations will be performed against this initial dataset. The properties of the dataset have a disproportionate influence on the load testing results. After a wide search, we realized that, presently, there exist no public datasets that model real world auth graphs. Solutions like Zanzibar and SpiceDB claim to be tested against proprietary data [2]. While we did find an Amazon dataset [18] which had RBAC relations, it was released with some data being obfuscated and incomplete. Apart from the fact that we required a ReBAC compatible dataset, the mentioned dataset was specifically released for an ML based Kaggle competition. Therefore, it was understood that there is a need to generate a synthetic dataset. This synthetic dataset was generated on the broad lines of Erdos-Renyi model, which is a model for generating random graphs. Our choice was reinforced by

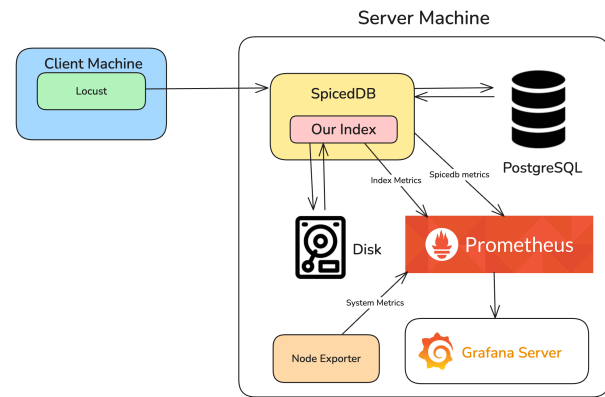


Fig. 2: Testing infrastructure's architecture

the fact that, in [12] this very model was used to dataset generation as well. We ensured hierarchical acyclic graph ordering by generating the graph in layers. We distributed nodes across 2120 layers to simulate nested hierarchies, while incorporating customizations such as layer skipping, node distribution and connections to non-adjacent nodes to simulate realistic scenarios. The generated data set comprises 20 million nodes and 155 million edges, with a mean in-degree of 7.76 and mean out-degree of 11.63.

B. Architecture and Infrastructure

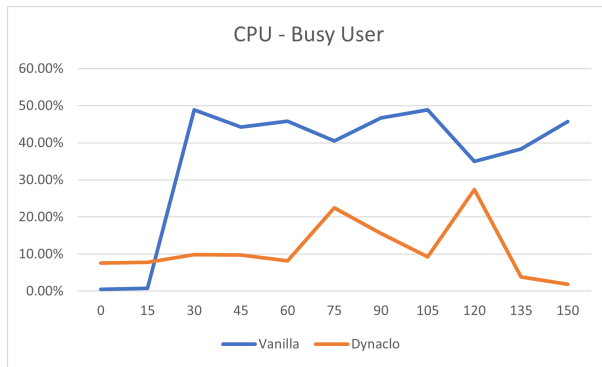
We have benchmarked our “modified” variant of SpiceDB which includes our index against release-1.38 version of SpiceDB which we shall henceforth call “vanilla” SpiceDB.

Our Load Test architecture consists of 2 machines – the server machine and the client machine which are both on the same physical LAN. The server machine hosts SpiceDB, SpiceDB's backing datastore for which the recommended option of Postgres is used, node exporter to expose system metrics, Prometheus and Grafana for monitoring. Our Index stores its data directly on disk (via Onyx). Apart from the system metrics exported by node exporter, custom metrics for SpiceDB and our index are also monitored. The Client Machine is used to generate load. Locust is used as the load test framework. The Server Machine, a bare metal Linux machine running Fedora Server 37 and has 4 CPUs and 12 GB of RAM.

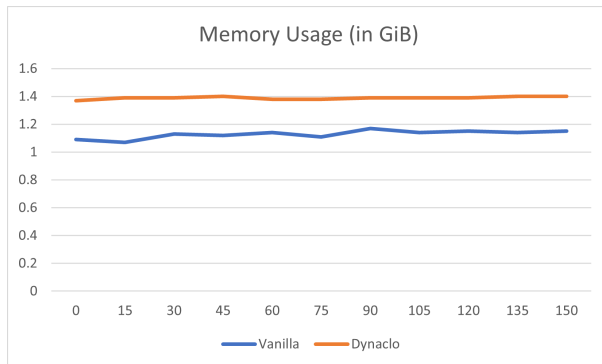
C. Load Test Methodology

Using the architecture described above, Queries of type Check Permission and Insert Edge were sent to SpiceDB in varying ratios (which shall be elaborated further). Queries are generated on the fly by stochastically choosing 2 nodes from the dataset to query permissionship or insert an edge on. The same query generation method is used for both vanilla and modified SpiceDB and the database is cleared and re-seeded with the dataset between load-test runs. The system was load tested using 2 different scenarios – a Pure Read condition with 100% CheckPermission requests, and a mixed scenario with 99% CheckPermission requests and 1% InsertEdge requests.

This particular ratio of 99% read and 1% write was chosen as it is the typical ratio of reads to writes in a real-life Zanzibar workload as observed by Google in their production traffic and also corroborated by SpiceDB [2], [7]. In the Pure Read Case, the test was started with 1 concurrent user and then ramped up to 50 concurrent users at a rate of 5 users per second. The load test was stopped after maintaining a load of 50 concurrent users for enough time for the latency numbers to stabilize. In the mixed workload case, the test was performed using 1 concurrent user, as writes consume non-trivial % of CPU. The above procedures were performed on both the variants of SpiceDB and the results were compared.



(a) CPU usage



(b) Memory usage

Fig. 3: Performance comparisons for Mixed Workload: CPU and Memory usage of Vanilla SpiceDB (blue) vs. Dynaclo (orange)

D. Results and Discussion

In the pure read workload we observed a **13.8x** reduction in median latency from 400ms (vanilla) to 29ms (our implementation) and a **23.8x** reduction in p95 latency (which is the minimum latency of the slowest 5% requests) from 1500ms (vanilla) to 63ms (our implementation) with a **15x** increase in throughput (requests per second - RPS) from 53 RPS to 802 RPS. These results establish the maximum performance limit of our system. However, when write operations are introduced, these numbers are expected to decline considerably. In the more realistic mixed workload with 99% reads and 1% writes, which mirrors a real world scenario [2], we observed the

following. We observed a **30%** reduction in median read latency from 20ms (vanilla) to 14ms (our implementation) and a **72%** reduction in p95 read latency from 180ms (vanilla) to 50ms (our implementation) with a more than **2x** increase in throughput from 23 RPS to 48 RPS for read requests. This establishes that our implementations shows better read performance than the baseline during a mixed workload. In Addition, we observed the latency of write requests decrease slightly but the throughput of write requests also decrease which is an acceptable regression in line with our intended tradeoff. Our implementation has much better (see Fig. 5.a) CPU utilization when compared with baseline SpiceDB (11.22% average vs. 35.93% average). While the memory usage of our system was slightly higher (0.3 GiB higher then vanilla SpiceDB), it is not significantly larger when compared to the size of the data (at 3 GiB).

VI. CONCLUSION AND FUTURE WORK

Projects in the systems engineering domain inherently involve trade-offs to achieve their intended gains, and our project is no exception to this principle. Our work demonstrates that significant performance improvements can be attained with careful tuning and optimization of SpiceDB. On comparing our implementation — Dynaclo against SpiceDB’s baseline implementation, we can conclude that we have achieved a 72% reduction in p95 read latency in a realistic workload. When both implementations were tested on equivalent loads, Dynaclo consistently showed lower CPU utilization (11% sustained vs. 36% sustained) with not significantly different memory consumption. Additionally, Dynaclo manages to also improve on the throughput (requests per second), which results in a 100% improvement on the already high baseline set by SpiceDB. These improvements were possible due to a combination of multiple innovative techniques: using a custom graph wrapper library which also optimizes for BFS, using a custom storage backend, and the SVA algorithm. In the future, we plan to implement the SI tree, replace mutual exclusion locks with atomic references to address race conditions, and enhance update operations to avoid fallback to SpiceDB. Additionally, we aim to integrate Dynaclo with upstream SpiceDB, ensuring full compliance with its distributed nature.

REFERENCES

- [1] J. Lopez, R. Oppliger, and G. Pernul, “Authentication and authorization infrastructures (AAIs): a comparative survey,” *Computers & Security*, vol. 23, no. 7, pp. 578–590, 2004.
- [2] R. Pang, R. Caceres, M. Burrows, Z. Chen, P. Dave, N. Gerner, A. Golynski, K. Graney, N. Kang, L. Kissner *et al.*, “Zanzibar:{Google’s} Consistent, Global Authorization System,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 33–46.
- [3] J. Moshenko, “Jake Moshenko on Zanzibar: Google’s Consistent, Global Authorization System [PWL NYC],” Jul 2021. [Online]. Available: <https://www.youtube.com/watch?v=1nbSbe3kw2U>
- [4] J. Dean and L. A. Barroso, “The tail at scale,” *Commun. ACM*, vol. 56, no. 2, p. 74–80, Feb. 2013. [Online]. Available: <https://doi.org/10.1145/2408776.2408794>
- [5] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994.

- [6] V. H. K, S. S. Jawalkar, S. S. Sunku, S. Ramesh, and V. K, "Exploring caching and indexing mechanisms for zanzibar based authorization systems," in *Proceedings of the 2025 IEEE 11th International Conference on Applied System Innovation (ICASI)*, Tokyo, Japan, 2025, to appear.
- [7] Authzed, "Github - authzed/spicedb: Open source, google zanzibar-inspired database for scalably storing and querying fine-grained authorization data," last accessed: November 26, 2024. [Online]. Available: <https://github.com/authzed/spicedb>
- [8] A. Bundy and L. Wallen, "Breadth-first search," *Catalogue of artificial intelligence tools*, pp. 13–13, 1984.
- [9] V. R. Betancort, "Google-Scale Authorization: Getting to 1 Million QPS on SpiceDB Dedicated with CockroachDB," Sep. 2024. [Online]. Available: <https://authzed.com/blog/google-scale-authorization>
- [10] G. F. Italiano, "Amortized efficiency of a path retrieval data structure," *Theoretical Computer Science*, vol. 48, pp. 273–281, 1986.
- [11] L. Roditty, "A faster and simpler fully dynamic transitive closure," *ACM Transactions on Algorithms (TALG)*, vol. 4, no. 1, pp. 1–16, 2008.
- [12] K. Hanauer, M. Henzinger, and C. Schulz, "Faster fully dynamic transitive closure in practice," in *18th International Symposium on Experimental Algorithms*, 2020, p. 1.
- [13] C. Pang, D. Hansen, and A. Maeder, "Managing rbac states with transitive relations," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, 2007, pp. 139–148.
- [14] P. Hlushchenko and V. Dudykevych, "Exploratory survey of access control paradigms and policy management engines," in *International Workshop on Computer Modeling and Intelligent Systems*, 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:270369934>
- [15] J. L. van de Snepscheut, "A derivation of a distributed implementation of warshall's algorithm," *Science of Computer Programming*, vol. 7, pp. 55–60, 1986.
- [16] C. Demetrescu and G. F. Italiano, "Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures," *Journal of Discrete Algorithms*, vol. 4, no. 3, pp. 353–383, 2006.
- [17] Dgraph-io, "Github - dgraph-io/badger: Fast key-value db in go." last accessed: September 30, 2024. [Online]. Available: <https://github.com/dgraph-io/badger>
- [18] L. Massaron, "Amazon employee access challenge," <https://www.kaggle.com/datasets/lucamassaron/amazon-employee-access-challenge>, 2025, accessed: May 05, 2025.