

HONEYBEE: Efficient Role-based Access Control for Vector Databases via Dynamic Partitioning

Hongbin Zhong
Georgia Institute of Technology
hzhong81@gatech.edu

Matthew Lentz
Duke University
mlentz@cs.duke.edu

Nina Narodytska
VMware Research
n.narodytska@gmail.com

Adriana Szekeres
VMware Research
adriana.szekeres@gmail.com

Kexin Rong
Georgia Institute of Technology
krong@gatech.edu

ABSTRACT

As vector databases gain traction in enterprise applications, robust access control has become critical to safeguard sensitive data. Access control in these systems is often implemented through hybrid vector queries, which combine nearest neighbor search on vector data with relational predicates based on user permissions. However, existing approaches face significant trade-offs: creating dedicated indexes for each user minimizes query latency but introduces excessive storage redundancy, while building a single index and applying access control after vector search reduces storage overhead but suffers from poor recall and increased query latency.

This paper introduces HONEYBEE, a dynamic partitioning framework that bridges the gap between these approaches by leveraging the structure of Role-Based Access Control (RBAC) policies. RBAC, widely adopted in enterprise settings, groups users into roles and assigns permissions to those roles, creating a natural "thin waist" in the permission structure that is ideal for partitioning decisions. Specifically, HONEYBEE produces overlapping partitions where vectors can be strategically replicated across different partitions to reduce query latency while controlling storage overhead. By introducing analytical models for the performance and recall of the vector search, HONEYBEE formulates the partitioning strategy as a constrained optimization problem to dynamically balance storage, query efficiency, and recall. Evaluations on RBAC workloads demonstrate that HONEYBEE reduces storage redundancy compared to role partitioning and achieves up to 6x faster query speeds than row-level security (RLS) with only 1.4x storage increase, offering a practical middle ground for secure and efficient vector search.

1 INTRODUCTION

Vector databases have emerged as a building block in modern applications, powering a wide variety of use cases such as in search engines, recommendation systems, and retrieval-augmented generation (RAG) pipelines driven by large language models (LLMs) [1, 4, 27, 32]. Vector databases provide efficient vector similarity search to retrieve semantically relevant results from high-dimensional vector spaces, often implemented via approximate nearest neighbor (ANN) algorithms [3, 9, 11, 12, 16, 21, 24].

As vector databases gain traction in enterprise settings, particularly in applications like retrieval-augmented generation (RAG) pipelines powered by large language models, enforcing appropriate access controls has become a critical challenge [29]. Role-based

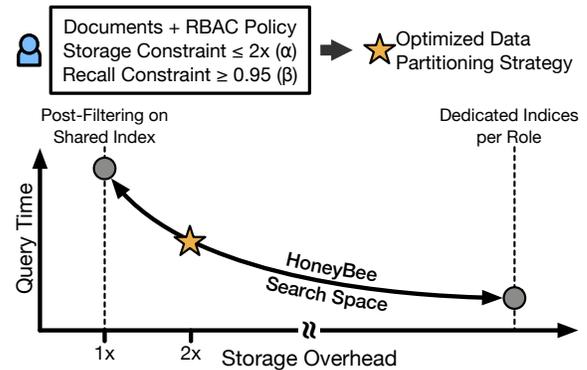


Figure 1: Given RBAC policies, storage and recall constraints, HONEYBEE optimizes for a partitioning of documents to achieve a balanced trade-off between query latency and storage overhead.

access control (RBAC) is a widely adopted framework for managing access to sensitive data [6, 10, 18, 25, 26]. RBAC simplifies permission management by grouping users into roles (e.g., HR, Finance, Engineering) and assigning data access permissions to these roles, rather than managing permissions at the individual user level. While RBAC is well-established for traditional relational databases, its implementation in vector databases introduces unique complexities. In relational databases, RBAC can be enforced through simple query predicates (e.g., "SELECT * FROM documents WHERE user=Alice"). However, vector databases must integrate these relational predicates with vector search on ANN indexes, requiring hybrid queries that simultaneously satisfy both vector similarity and access control constraints.

Access control can be readily integrated with vector databases using one of the two approaches: (1) dedicated indices for each user (or role), or (2) post-filtering on a single, shared index. These approaches represent two ends of a spectrum in the trade-off between storage overhead and query performance, as shown in Figure 1. The dedicated indices approach creates separate indices for each user or role, ensuring that queries only access authorized data. This eliminates the need for runtime filtering and enables fast query execution. However, it incurs significant storage overhead, as vectors accessible to multiple users or roles must be duplicated across partitions. For example, in our experiments, even partitioning by roles (rather than individual users) resulted in storage overheads that were 27x larger than a unified index. In contrast, the post-filtering approach constructs a single unified index and applies

access control filters after performing the ANN search. Although this minimizes storage overhead by avoiding data duplication, it suffers from poor recall and performance when filters are highly selective (*i.e.*, users can access only a small fraction of the data). In such cases, most search results are discarded, requiring the system to expand the search scope to achieve acceptable recall, which increases query latency. Recent works have explored specialized index structures to better support vector similarity search with filtering conditions [13, 23, 33]. However, these methods still operate within the constraints of a single index and do not fully exploit the potential benefits of storing redundant vector copies across indices.

In this work, we propose HONEYBEE, a *dynamic partitioning framework* that efficiently implements RBAC in vector databases. Our key insight is that the structure of RBAC policies can be leveraged to design a hybrid approach that balances the strengths of dedicated indices and post-filtering. Enterprise RBAC deployments typically feature far fewer roles than users (*e.g.*, tens to hundreds of roles versus tens of thousands of users). Moreover, role definitions tend to remain stable over time, even as user assignments change frequently [6, 10, 19, 25, 26, 30]. Empirically, we find that partitioning by user (or unique combinations of roles) significantly increases storage overhead compared to partitioning by roles, while offering diminishing returns in query latency improvements (*e.g.*, $> 50\times$ increase in storage for $< 2\times$ improvement in query latency). We exploit these properties by treating roles as the finest level of partitioning granularity.

Specifically, HONEYBEE introduces analytical models to quantify the expected search performance and recall with respect to key factors, such as partition size, average filter selectivity, and index-specific parameters that control the trade-off between search performance and recall. Based on these models, HONEYBEE formulates the partitioning strategy as a mixed-integer nonlinear programming (MINLP) optimization problem. Since solving this problem is NP-hard, HONEYBEE further introduces a greedy algorithm that generates a spectrum of partitioning strategies under different storage constraints, ranging from post-filtering to dedicated indices per role. Importantly, this partitioning strategy is orthogonal to the choice of index structure. For example, specialized hybrid search indices can be used on individual partitions to further improve post-filtering performance within each partition. This flexibility allows HONEYBEE to create a spectrum of solutions that balance storage overhead and query performance.

In summary, our key contributions are as follows:

- We identify the unique challenges of implementing access control in VectorDBs and analyze the limitations of existing approaches, including post-filtering and partitioning.
- We propose HONEYBEE, which integrates RBAC with optimized data partitioning strategies, striking a balance between storage and query efficiency.
- We demonstrate the adaptability of our framework to various operational scenarios, including its compatibility with hybrid search methods such as ACORN.
- We evaluate our approach on real-world datasets with diverse permission structures, highlighting its practical applicability compared to state-of-the-art baselines. Using

pgvector with the HNSW index, HONEYBEE reduces storage redundancy compared to dedicated indices per role and achieves up to $6\times$ faster query speeds than post-filtering on shared index (implemented via PostgreSQL’s row-level security feature), with only $1.4\times$ storage increase.

2 BACKGROUND

In this section, we provide background on role-based access control (RBAC) (§ 2.1) and vector indexes for ANN search (§ 2.2).

2.1 Role-Based Access Control

The simplest way to enforce access control is via access control lists (ACLs), which specifies for each protected resource (*e.g.*, a patient’s record), a list of users that have access to that object. However, ACLs face several challenges. ACL directly associate users with permissions, which is hard to maintain when dealing with a large number of users and permissions that need constant updating. Furthermore, organizations typically need to specify access policies based on user functions or roles within the enterprise. For example, the information security principle of least privilege, which states that users and applications should only have access to the data and operations necessary for their jobs, is burdensome to implement directly on top of ACLs.

Role-Based Access Control (RBAC) emerged as a solution to address these aforementioned challenges in enterprise settings [10, 19, 26]. By introducing roles as an intermediary layer between users and permissions, RBAC simplifies the management of complex access policies and reduces administrative overhead.

DEFINITION 2.1. $\gamma = \langle U, R, D, \phi_{UA}, \phi_{PA} \rangle$ defines a basic RBAC system, where

- U, R and D are the sets of users, roles, and documents in the systems, respectively.
- $\phi_{UA} : U \rightarrow 2^R$ defines the many-to-many relationship between users and roles.
- $\phi_{PA} : R \rightarrow 2^D$ defines the many-to-many relationship between roles and documents.

In this system, a user u_i ’s authorized permissions are determined by the union of permissions acquired through their assigned roles:

$$acc(u_i) = \bigcup_{r \in \phi_{UA}(u_i)} \phi_{PA}(r) \quad (1)$$

The RBAC system can also incorporate a partial order of roles to support role hierarchies. For example, in a healthcare setting, rather than granting physicians blanket access to all patient record data, RBAC allows administrators to define a hierarchy of roles based on the physician’s specialization, and restrict access to only those fields relevant to a particular type of physician’s practice. RBAC also supports customized constraints on user-role and role-permission assignments. For example, administrators can implement policies to limit the maximum number of users assigned to a particular role, or to establish mutual exclusivity between roles to prevent users from simultaneously holding potentially conflicting positions.

Relational databases such as SQL Server and PostgreSQL provide Row-Level Security (RLS) as a built-in security feature to control access to rows in a database table [5, 22]. RBAC policies can be

easily implemented using a set of tables that mirror the main RBAC components. When a user attempts to access a protected resource, the database system performs a series of JOIN operations across these tables to determine if the access should be granted. For example, Listing 1 checks if a user has permissions to specific rows in the `PatientRecords` table by joining the user-role and role-permission assignment tables.

```
CREATE POLICY access_policy ON PatientRecords
FOR SELECT
USING (
  EXISTS (
    SELECT 1
    FROM PermissionAssignment pa
    JOIN UserRoles ur ON pa.role_id = ur.role_id
    WHERE pa.record_id = PatientRecords.record_id
    AND ur.user_id = current_user::int
  )
);
```

Listing 1: RBAC policy implemented via Row-Level Security.

2.2 Vector Search

Vector Search. Vector indexes for k -ANN search typically fall into two categories: graph-based and partition-based indexes. Graph-based indexes, such as Hierarchical Navigable Small World Graphs (HNSW) and Navigating Spreading-out Graphs (NSG), organize vectors as nodes in a proximity graph where edges connect similar vectors [11, 21, 28, 35]. During search, the algorithm starts from predefined entry points and greedily moves to neighboring nodes that are closer to the query vector. The performance of these indexes can be tuned via parameters like HNSW’s ef_s , which controls the size of the dynamic candidate list during search. A larger ef_s allows the algorithm to explore more paths in the proximity graph, leading to better accuracy but slower search speed.

Partition-based indexes, like the IVF index (Inverted File), divide the vector space into clusters using algorithms such as k -Means [2, 14, 15, 20]. Each cluster is represented by a centroid vector, and vectors are assigned to their nearest centroid during indexing. During search, the system identifies relevant clusters by comparing the query vector with cluster centroids, then performs similarity search within the selected clusters. The index performance is controlled by parameters like IVF’s $nprobes$, which specifies the number of closest clusters to search. Higher $nprobes$ values increase the search space, resulting in better accuracy at the cost of increased search time.

Hybrid Search. Hybrid search refers to vector similarity search integrated with filtering conditions, enabling systems to retrieve semantically relevant results while adhering to specific constraints, such as access control policies. For example, in access-controlled k -ANN search, users expect to retrieve documents relevant to their query (measured by vector distance), among those that they have access to (filtering condition).

Recently, researchers have developed specialized index structures to better support hybrid vector search [13, 17, 23, 31, 33, 34]. For instance, ACORN [23], a state-of-the-art index built on top of HNSW, integrates filtering conditions directly into the index construction process. ACORN considers the selectivity of filtering predicates to improve the connectivity of the HNSW graph, leading

to better query recall and latency compared to treating filtering as a separate post-processing step after vector search.

3 OVERVIEW

In this section, we provide an overview of HONEYBEE, including the problem statement (§ 3.1) and main system components (§ 3.2).

3.1 Problem Statement

Consider a vector database system managing document access with role-based permissions. Let $D = \{d_1, d_2, \dots\}$ be the set of all managed documents. Each document $d_i \in D$ represents an atomic unit for permission assignment and may contain either a single vector (e.g., embeddings for a single image) or multiple vectors (e.g., embeddings for paragraphs within a webpage).

Let $U = \{u_1, u_2, \dots\}$ be the set of users, and $R = \{r_1, r_2, \dots\}$ be the set of roles (e.g., "manager", "HR"). As discussed in Definition 2.1, RBAC policies can be defined by two mappings, $\phi_{UA} : U \rightarrow 2^R$ (assigns users to roles), and $\phi_{PA} : R \rightarrow 2^D$ (grants roles access to documents). $acc(u_i)$ is the set of documents accessible to user u_i .

The system processes queries in the form of $q_i = (u_{q_i}, v_i)$, containing a user $u_{q_i} \in R$ and a query vector v_i . The vector database’s goal is to retrieve the top- k most relevant documents based on vector similarity, subject to the users’ access permissions.

To optimize for search performance, the system can be configured using different partitioning strategies. A configuration is represented as a non-disjoint partitioning $\Pi = \{\pi_1, \pi_2, \dots\}$, where $\cup_i \pi_i = D$ and $\forall i : \pi_i \subseteq D$ and $\pi_i \neq \emptyset$. The configurations can range from creating separate partitions for each user, to using a single shared partition for all users. The set of partition indices that contain documents accessible to role r_i is defined as:

$$AP(r_i, \Pi) = \{j \mid \pi_j \in \Pi, \pi_j \cap \phi_{PA}(r_i) \neq \emptyset\} \quad (2)$$

The set of partition indices that contain documents accessible to user u_i is the union of partitions accessible to each of its roles:

$$AP(u_i, \Pi) = \bigcup_{r \in \phi_{UA}(u_i)} AP(r, \Pi) \quad (3)$$

Configurations are evaluated on three dimensions:

- **Storage overhead:** $\frac{\sum_i |\pi_i|}{|D|}$, the ratio of actual storage to minimum required storage to store the document embeddings. We assume that documents have roughly uniform sizes by default, but the metric can also be extended with appropriate weighting to account for difference in sizes. We omit index size as it also scales linearly with the number of embeddings.
- **Search quality:** $R(\Pi, u_i)$, the search recall for user u_i is defined as the ratio between (1) results from our indexed search filtered by u_i ’s access permissions and (2) the top- k results obtained from first performing an exhaustive search and then applying role-based access control filtering specific to u_i .
- **Search performance:** $C(\Pi, u_i) = \sum_{j \in AP(u_i, \Pi)} c(\pi_j)$, where $c(\pi_j)$ is the cost of retrieving the top- k nearest neighbors for the query in partition π_j . $c(\pi_j)$ depends on factors such as partition size $|\pi_j|$, index types, and index parameters.

In the following sections, we will introduce analytical models to approximate the query performance and recall models.

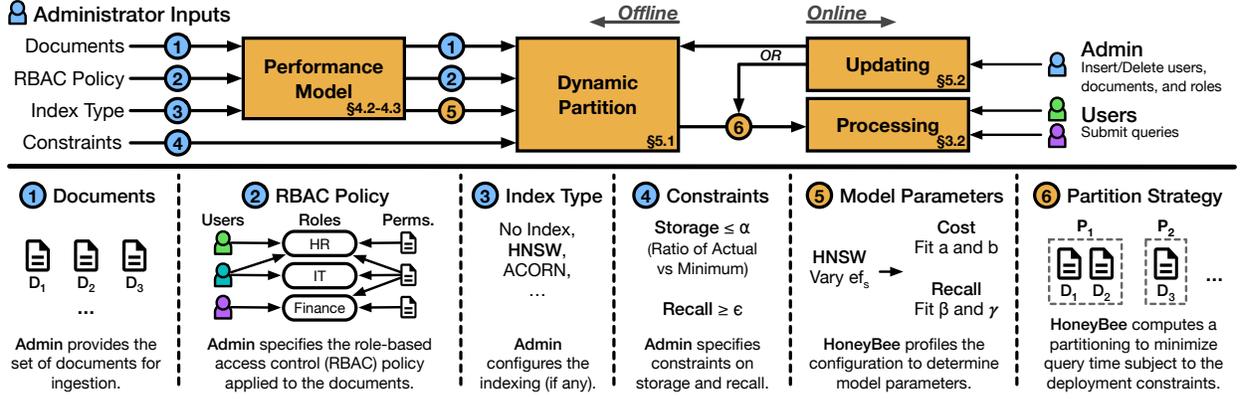


Figure 2: Overview of HONEYBEE’s workflow. Yellow elements represent main components as well as inputs and outputs of HONEYBEE, while blue elements represent configurable components by administrators, such as documents, RBAC policies, index types, and constraints.

PROBLEM 1. Given a set of documents D , RBAC policies, storage constraint $\alpha (\geq 1)$, and recall constraint $\epsilon (< 1)$, find a partitioning $\Pi = \{\pi_1, \pi_2, \dots\}$ that:

$$\begin{aligned} & \underset{\Pi}{\text{minimize}} && \frac{1}{|U|} \sum_{i=1}^{|U|} C(\Pi, u_i) \\ & \text{subject to} && \frac{\sum_i |\pi_i|}{|D|} \leq \alpha, \frac{1}{|U|} \sum_{i=1}^{|U|} R(\Pi, u_i) \geq \epsilon \end{aligned}$$

By default, the objective function optimizes for the average query latency over all users. Alternative objectives are also available, such as the average query latency over a query workload (weighted version of the above), or the average query latency over all roles.

3.2 System Overview

HONEYBEE operates in two phases: an offline phase for data organization and indexing, and an online phase for query processing. Figure 2 provides an overview of the workflow.

Offline. In the offline phase, HONEYBEE optimizes the objectives and satisfies the constraints specified in Problem 1. Depending on the storage constraints, the partitioning strategy can span a spectrum from a single shared partition (post-filter on a shared index approach) to role-specific partitions (dedicated indices approach), or more commonly, utilize a hybrid approach with overlapping partitions based on access patterns.

For each partition $\pi_i \in \Pi$, HONEYBEE builds a separate similarity search index. The type of index is configurable by users, with options ranging from no index (exhaustive search), vector indices for k -ANN search (e.g., HNSW), or specialized indices for hybrid search (e.g., ACORN). Along with partitioning, HONEYBEE determines the index-specific parameter (e.g., ef_s in HNSW) that controls search depth, or how many candidate results are considered during similarity search. When all documents are in one partition, high search depth is required to ensure sufficient results remain after access control filtering. However, when documents are distributed across multiple partitions, each partition contains a higher density of accessible documents for its intended users, allowing HONEYBEE to use a lower search depth while achieving the target recall.

Since partitions may overlap, the set of partitions containing documents accessible to a user ($AP(u_i, \Pi)$ defined in Eq 3) could include redundant partitions. To optimize query processing, HONEYBEE pre-computes and maintains a routing table, $AP_{min}(u_i, \Pi)$, from each unique combination of roles (or each distinct user) to their minimal required set of partitions:

$$AP_{min}(u_i, \Pi) = \arg \min_{S \subseteq AP(u_i, \Pi)} \sum_{j \in S} c(\pi_j), \text{ s.t. } \bigcup_{j \in S} \pi_j \supseteq acc(u_i) \quad (4)$$

where $acc(u_i)$ is the set of documents accessible to user u_i (Eq 1).

Online. When processing a query $q_i = (u_{q_i}, v_i)$, HONEYBEE first identifies the relevant partitions using the precomputed mapping $AP_{min}(u_{q_i}, \Pi)$. For each identified partition, the system performs vector similarity search using the query vector v_i , with search depth controlled by the pre-tuned index parameters. If needed, access control is applied to filter out unauthorized documents. This can be implemented through traditional post-filtering (Listing 1), or via hybrid search indexes that integrate filtering directly into the similarity search process. Finally, HONEYBEE merges the filtered results from all searched partitions, sorts them by similarity score, and returns the global top- k documents as the final result.

4 ANALYTICAL MODELING

In this section, we introduce the analytic models for the search performance and recall using the HNSW index, as well as the optimization problem formulation based on these models. Although we focus on HNSW in this section, we demonstrate in the evaluation that the optimization framework is also applicable to other types of indices (§ 7.2).

4.1 Background: HNSW Index Parameters

HNSW is one of the most widely adopted graph-based indexes for ANN search [21]. Its behavior is governed by three key parameters: M , ef_c ($ef_construction$), and ef_s (ef_search). During the index construction phase, M controls the number of links each node maintains, affecting memory usage and search efficiency, while ef_c determines how many neighbors are considered when inserting a new point, influencing indexing speed and accuracy. In contrast, ef_s is dynamically adjusted at query time, controlling the size of the

priority queue used to manage candidate nodes. Higher ef_s values improve recall at the cost of increased query latency.

In the following study, we use fixed M and ef_c parameters during index construction (e.g., $M = 16$ and $ef_c = 64$ are commonly used [21]), as these parameters primarily impact the structure of the HNSW graph and are typically set during the indexing phase. In contrast, ef_s remains the key tunable parameter at query time, directly influencing recall and query performance.

In the following sections, we establish analytical models to characterize the relationship between ef_s , query recall, and query latency. Recall is primarily influenced by the structure of the HNSW index, including the connectivity parameter M , the search depth parameter ef_s , and the selectivity imposed by user access permissions. Meanwhile, query latency is largely determined by the number of partitions, the size of each partition, and ef_s .

4.2 Model for Search Performance

The query time in HNSW is influenced by two main factors: the traversal of the hierarchical graph and the cost of vector similarity calculations. The traversal complexity is approximately $O(\log n)$, where n is the number of points in the graph [21]. The parameter ef_s directly impacts query time by determining the number of candidate nodes visited during the search.

We model the query time for a partition as $c(\pi_i, ef_s) = \log(|\pi_i|) \cdot f(ef_s)$, where $f(ef_s)$ represents the relationship between search queue size and query time. We use the same ef_s for all partitions. Empirical analysis demonstrates that $f(ef_s)$ exhibits a linear relationship¹ with ef_s which we model as $f(ef_s) = a \cdot ef_s + b$. Here, a and b are system-dependent parameters influenced by hardware capabilities, software optimizations, and dataset characteristics such as intrinsic dimensionality and data distribution.

Overall, the query cost is a function of the partitioning design Π , the index specific parameter ef_s , and the query itself. We consider two types of query costs: user-level (C_u) and role-level (C_r). For a given user u_i , the total query cost $C(\Pi, u_i, ef_s)$ must account for all partitions that contain documents accessible to that user, as defined by $AP_{min}(u_i, \Pi)$ in Eq 4:

$$C_u(\Pi, u_i, ef_s) = \sum_{j \in AP_{min}(u_i, \Pi)} \log(|\pi_j|) \cdot (a \cdot ef_s + b) \quad (5)$$

Similarly, the role-level query cost is defined as

$$C_r(\Pi, r_i, ef_s) = \sum_{j \in AP_{min}(r_i, \Pi)} \log(|\pi_j|) \cdot (a \cdot ef_s + b) \quad (6)$$

Note that since different roles could be mapped to the same partitions via $AP_{min}(r_i, \Pi)$, the user-level cost is not simply a weighted average of the role-level cost.

To fit a and b , we use an RBAC permission generator (§ 6.1) to create a workload where each user maps to one role, and one partition is created per role. Here, $AP_{min}(r_i, \Pi) = \pi_{r_i}$, and $C_u(\Pi, u_i, ef_s) = C_r(\Pi, r_i, ef_s) = \log(|\pi_{r_i}|) \cdot (a \cdot ef_s + b)$. We generate 1000 queries, test multiple ef_s values, and derive an average query time per ef_s . This allows us to compute $\frac{\text{querytime}}{\log(|\pi_{r_i}|)} = a \cdot ef_s + b$ for fitting a and b .

¹This linear relationship is observed under typical settings, though query time can also be influenced by factors such as data distribution and intrinsic dimensionality [7] that we omit in the modeling for simplicity.

4.3 Model for Search Recall

Next, we model the recall behavior of HNSW index with post-filtering access control. Other than the search depth parameter ef_s , the primary factor affecting recall is *selectivity* - the fraction of documents a user can access in their assigned partitions. For a user u_i , selectivity is defined as:

$$s_u(u_i) = \frac{1}{|AP_{min}(u_i, \Pi)|} \sum_{j \in AP_{min}(u_i, \Pi)} \frac{|acc(u_i) \cap \pi_j|}{|\pi_j|} \quad (7)$$

This represents the average fraction of accessible documents across all partitions the user needs to query. We then compute the system-wide average selectivity across all users:

$$\bar{s}_u = \frac{1}{|U|} \sum_{i=1}^{|U|} s(u_i) \quad (8)$$

For simplicity, we ignore additional factors that can impact recall in the modeling, such as the correlation between query vector and the access permission filter [23].

Our analytical model is based on the observation that recall follows a two-phase pattern as ef_s increases: it first grows linearly, then gradually saturates. This leads us to model recall as a piecewise function in Eq 9: a linear function for the initial rapid increase and a sigmoid function to capture the saturation effect.

$$R(\Pi, \bar{s}_u, ef_s) = \begin{cases} \frac{ef_s \cdot \bar{s}_u}{k}, & \text{if } ef_s \leq \gamma \cdot \frac{k}{\bar{s}_u}, \\ \frac{1}{1 + e^{-\beta \cdot \frac{\bar{s}_u}{k} (ef_s - \gamma \cdot \frac{k}{\bar{s}_u})}} + \left(\gamma - \frac{1}{2}\right), & \text{otherwise.} \end{cases} \quad (9)$$

Here, k (from top- k) is the result count, \bar{s}_u is the average selectivity across all users (Eq 8). The recall model uses two scaling relationships with fitted constants γ and β :

- **Transition point** $\gamma \cdot \frac{k}{\bar{s}_u}$: This determines when we transition from linear growth to saturation. With lower selectivity, we need to examine more candidates (higher ef_s) to find k valid results. The offset value $\gamma - \frac{1}{2}$ is chosen to ensure continuity of function value at the transition point.
- **Sigmoid steepness** $\beta \cdot \frac{\bar{s}_u}{k}$: This parameter controls the rate at which recall improves beyond the transition point. Higher selectivity (\bar{s}_u) increases the likelihood of retrieving relevant results, resulting in faster recall gains and a steeper curve.

To estimate β and γ , we use an RBAC generator to create a permission workload with an average selectivity of 0.1. As ef_s increases from 1 to 1000 (a typical upper limit in databases like pgvector), recall transitions from 0 to 1. We execute 1000 randomly generated queries across varying ef_s values (10 to 1000), collecting average recall per setting. Before each query, we compute selectivity and retrieve k to use in Eq 9 to fit β and γ .

4.4 Optimization Problem Formulation

Given the analytical models for query performance and recall, we can formulate Problem 1 as a constraint optimization problem.

Given Constants:

- U, R, D : Sets of users, roles, and documents.
- $acc(u_i)$: User-to-document access mapping from RBAC (Eq 1).
- n_p : Number of partitions in configuration Π .
- α : Storage overhead constraint (≥ 1).

- ϵ : Minimum recall threshold (< 1).
- $R(s, ef_s)$: Recall model with fitted parameters (Eq 9).

Variables:

- $p_{j,k} \in \{0, 1\}^{|D| \times n_p}$: Binary decision variable indicating whether document d_j is assigned to partition π_k .
- $x_{i,k} \in \{0, 1\}^{|U| \times n_p}$: Binary decision variable indicating whether user u_i should access partition π_k for their queries.
- ef_s : HNSW search depth parameter

$$\text{minimize } p_{j,k}, x_{i,k}, ef_s \quad \frac{1}{|U|} \sum_{i=1}^{|U|} \sum_{k=1}^{n_p} x_{i,k} \cdot \log \left(\sum_{j=1}^{|D|} p_{j,k} \right) (a \cdot ef_s + b) \quad (10a)$$

subject to

$$\sum_{k=1}^{n_p} x_{i,k} \geq 1, \quad \forall i \in \{1, 2, \dots, |U|\} \quad (\text{access mapping}), \quad (10b)$$

$$\sum_{k=1}^{|D|} \delta_{i,j,k} \geq 1, \quad \forall i \in \{1, 2, \dots, |U|\}, \forall j \in \text{acc}(u_i), \quad (10c)$$

$$x_{i,k} \geq \delta_{i,j,k}, \delta_{i,j,k} \leq p_{j,k} \quad \forall i, j, k, \quad (10d)$$

$$\sum_{j=1}^{|D|} p_{j,k} \geq 1, \quad \forall k \in \{1, \dots, n_p\} (\text{non-empty partitions}), \quad (10e)$$

$$\sum_{k=1}^{n_p} \sum_{j=1}^{|D|} p_{j,k} \leq \alpha |D| \quad (\text{storage}), \quad (10f)$$

$$R(\bar{s}, ef_s) \geq \epsilon \quad (\text{recall}), \quad (10g)$$

$$\bar{s} = \frac{1}{|U|} \sum_{i=1}^{|U|} \frac{1}{\sum_{k=1}^{n_p} x_{i,k}} \sum_{k=1}^{n_p} x_{i,k} \cdot \frac{\sum_{j \in \text{acc}(u_i)} p_{j,k}}{\sum_j p_{j,k}} (\text{selectivity}) \quad (10h)$$

The objective is to minimize the average query cost across all users, based on the user-level performance model (Eq 5). The constraints ensure that: (1) user access indicators $x_{i,k}$ accurately reflect document assignments via $\text{acc}(u_i)$, using a helper variable $\delta_{i,j,k}$ to link $p_{j,k}$ and $x_{i,k}$ (Eq 10b, 10c, 10d); during optimization, $x_{i,k}$ naturally approaches AP_{\min} defined in Eq 4, (2) no partition is empty (Eq 10e), (3) total storage overhead doesn't exceed α (Eq 10f), (4) recall meets the minimum threshold ϵ using the model from Eq 9 (Eq 10g), and (5) average selectivity is properly computed based on document assignments and access patterns (Eq 10h).

The optimization process involves three steps. First we compute the user selectivity $s(u_i)$ from document assignments $p_{j,k}$, as well as the average selectivity \bar{s} . Second, we determine the minimum ef_s needed to achieve recall threshold ϵ . Note that we apply the same ef_s for all partitions. Finally, we optimize partition assignments $p_{j,k}$ to minimize average query time.

This optimization problem belongs to the class of Mixed-Integer Nonlinear Programming (MINLP), which is NP-hard due to its combinatorial nature and nonlinear constraints. In addition, the problem involves $O((|D| + |R|)n_p)$ binary variables, making it intractable for large-scale datasets using off-the-shelf solvers. In the next section, we introduce a greedy dynamic partitioning algorithm that effectively solves this optimization problem. While n_p (number of partitions) is treated as a constant in the optimization problem, the greedy algorithm treats n_p as a variable to enhance flexibility.



Figure 3: Each iteration of the greedy algorithm, one or more roles are moved from the largest partition to form a new partition (green blocks). The roles are chosen greedily based on the estimate performance improvement using the analytical models.

5 DYNAMIC PARTITIONING STRATEGY

In this section, we introduce HONEYBEE's partitioning strategy that minimizes average query latency while satisfying the given storage and recall constraints.

5.1 Greedy Split Algorithm

To solve the MINLP optimization problem introduced in § 4.4, HONEYBEE makes a key observation. The design space of the partitioning is challenging as it scales with the total number of documents. HONEYBEE constrains this design space by ensuring that all documents accessible by each role are contained in a single partition. This intuition constraint is justified by our performance model, which scales logarithmically with data size. When documents for a specific role are split across two partitions π_i and π_j , the query time becomes $\log(|\pi_i|) + \log(|\pi_j|) = \log(|\pi_i||\pi_j|)$, which is typically larger than the time required for accessing a single merged partition $\log(|\pi_i \cup \pi_j|)$.

Algorithm 1 Greedy Split Algorithm

Input: Storage constraint α
Output: Partitions Π , partition-to-role mapping M

- 1: **function** FINDLARGESTPARTITION(Π, M)
- 2: **return** $\arg \max_{i \in \{1, \dots, |\Pi|\}, |M[i]| > 1} |\Pi[i]|$
- 3: **function** CREATEPARTITION(\mathcal{R}) ▷ All docs needed by roles in \mathcal{R}
- 4: **return** $\bigcup_{r \in \mathcal{R}} \phi_{PA}(r)$
- 5: $\Pi[1] \leftarrow D$ ▷ Initialize partition with all documents and roles
- 6: $M[1] \leftarrow R$
- 7: **while** $\sum_{\pi_i \in \Pi} |\pi_i| \leq \alpha |D|$ **do**
- 8: $i_{src} \leftarrow \text{FINDLARGESTPARTITION}(\Pi, M)$
- 9: $i_{dst} \leftarrow |\Pi| + 1$ ▷ Create new partition
- 10: $\Pi[i_{dst}] \leftarrow \emptyset, M[i_{dst}] \leftarrow \emptyset$
- 11: **while** $\sum_{\pi_i \in \Pi} |\pi_i| < \alpha |D|$ **do**
- 12: $r^* \leftarrow \text{FINDBESTSPLIT}(\Pi, M, i_{src}, i_{dst})$
- 13: $M[i_{src}] \leftarrow M[i_{src}] \setminus \{r^*\}$ ▷ Update mappings
- 14: $M[i_{dst}] \leftarrow M[i_{dst}] \cup \{r^*\}$
- 15: $\Pi[i_{src}] \leftarrow \text{CREATEPARTITION}(M[i_{src}])$ ▷ Update partitions
- 16: $\Pi[i_{dst}] \leftarrow \text{CREATEPARTITION}(M[i_{dst}])$
- 17: **if** $i_{src} \neq \text{FINDLARGESTPARTITION}(\Pi, M)$ **then**
- 18: **break** ▷ Source is no longer the largest
- 19: **return** Π, M

We propose a greedy partitioning algorithm (Algorithm 1) based on this intuition. The algorithm returns the final partitioning design Π , as well as a mapping M which maps each partition to the roles that it contains. M can be used to calculate $AP(u_i, \Pi)$ as defined in Eq 2.

As shown in Figure 3, our algorithm follows an iterative splitting approach that begins with a single partition containing all documents and all roles. In each iteration, we identify the largest partition containing more than one role and attempt to split it by moving selected roles r^* to a new partition. The roles are chosen based on two criteria: (1) all documents accessible by the r^* are grouped into the new partition, and (2) the selection maximizes query performance improvements based on our analytical model. Each split operation introduces a trade-off between storage and query efficiency. Storage increases because documents shared between r^* and roles remaining in the source partition must be duplicated. However, query latency typically improves as queries involving r^* benefit from having a higher fraction of accessible documents (increased selectivity) in the new partition.

For each round of splitting (line 7, outer loop), the algorithms choose $\Pi[i_{src}]$, the largest partition with more than one role as the source partition for splitting. At the end of the round, the partition gets split into two partition with ids $\Pi[i_{src}]$ and $\Pi[i_{dst}]$. In the inner loop (line 11), the algorithm evaluates the benefit of moving each role from partition $\Pi[i_{src}]$ to the new partition $\Pi[i_{dst}]$, and greedily selects the role r^* with the largest improvement of query latency based on the performance model (line 12). We then update $\Pi[i_{dst}]$ by adding documents belonging to r^* , and update $\Pi[i_{src}]$ by removing documents that are unique to r^* , as no other roles in $\Pi[i_{src}]$ would have access to these documents. In each iteration of the inner loop, we add one role to $\Pi[i_{dst}]$, until query latency no longer improves, or the storage constraint is met or $\Pi[i_{src}]$ is no longer the largest splittable partition. Note that at the end of the inner loop, $\Pi[i_{dst}]$ can contain multiple roles.

Algorithm 2 FindBestSplit Algorithm

```

1: function FINDBESTSPLIT( $\Pi, M, i_{src}, i_{dst}$ )
2:   for all  $r \in M[i_{src}]$  do                                ▶ Try each split
3:      $\pi'_{dst} \leftarrow \text{CREATEPARTITION}(M[i_{dst}] \cup \{r\})$ 
4:      $\pi'_{src} \leftarrow \text{CREATEPARTITION}(M[i_{src}] \setminus \{r\})$ 
5:      $\Pi' \leftarrow \Pi \setminus \{\Pi[i_{src}], \Pi[i_{dst}]\} \cup \{\pi'_{src}, \pi'_{dst}\}$ 
6:      $\Delta S \leftarrow |\pi'_{src}| + |\pi'_{dst}| - |\Pi[i_{src}]| - |\Pi[i_{dst}]|$     ▶  $\Delta$ Storage
7:      $\Delta Q_r \leftarrow C_r(\Pi) - C_r(\Pi')$                                 ▶  $\Delta$ Query time
8:      $\Delta Q_u \leftarrow C_u(\Pi) - C_u(\Pi')$ 
9:     if  $\Delta Q_r < 0$  and  $\Delta Q_u < \eta$  then    ▶ Check if move is beneficial
10:      if  $(\Delta Q_r + \Delta Q_u) / \Delta S > \Delta_{max}$  then
11:         $r^* \leftarrow r$ 
12:         $\Delta_{max} \leftarrow (\Delta Q_r + \Delta Q_u) / \Delta S$ 
13:   return  $r^*$ 

```

Algorithm 2 implements FINDBESTSPLIT, evaluating the cost of each candidate split to determine the most effective one. Two performance models are considered: a role-level model C_r (Eq 6) and a user-level model C_u (Eq 6). For $C_r(\Pi)$ or $C_u(\Pi)$, users specify a target recall and input it into the program to compute the corresponding average selectivity \bar{s}_u for a given partition Π . Then the ef_s is derived using Eq. 9. Intuitively, the role-level model reflects the local effect - reduction in per role query time is a good

objective that can guide the partitioning from a single partition to the solution of partitioning by roles (one partition per role). The user-level model reflects the global effect - reduction in user-level model would directly reduce the optimization objective of query time.

A split is considered beneficial if $\Delta Q_r < 0$ and ΔQ_u is below a predefined threshold η , preventing the greedy algorithm from becoming trapped in locally optimal but globally suboptimal configurations. Empirically, if $\Delta Q_r < 0$, it's likely that user-level query time will improve in future splits, even if Q_u slightly increases in the current iteration. ΔS denotes storage cost increase. We normalize the query improvements by the storage cost to identify the split that is most effective per unit storage. In practice, ΔS could also be zero or negative; we then use $(\Delta Q_r + \Delta Q_u) / (\Delta S + \epsilon)$, prioritizing roles r with $\Delta S < 0$ for reduced storage and query latency.

5.2 Handling Updates

In this section, we discuss how HONEYBEE updates the partitioning strategy under changes to the permission workload. We consider three cases: (1) inserting and deleting users, (2) inserting and deleting documents from a role, and (3) inserting and deleting roles. Since HONEYBEE's partitioning strategy is based on roles, these updates can be performed incrementally without requiring a full rebuild of the partitions.

When a new user is added to existing roles, HONEYBEE determines the optimal set of partitions for the user and updates the routing table from users to partitions (AP_{min} , Eq 4) accordingly. Conversely, deleting a user requires no changes to partitions; only the user's access paths in the routing table are removed.

When inserting new documents into an existing role, HONEYBEE locates the corresponding partition and inserts the documents. Similarly, when deleting documents from a role, HONEYBEE removes the specific documents from the partition containing the role, but the user routing remains unchanged.

When inserting a new role, the system evaluates the performance impact ($\Delta C / \Delta \text{Storage}$), assigns the role's documents to either an existing partition or a newly created one, and updates routing for users assigned to the new role. Deleting a role involves identifying all partitions containing the role, removing documents exclusively required by the role, and updating the user-to-role assignment (ϕ_{UA}). In both cases, only the indices for the affected partitions need to be updated.

6 EXPERIMENTAL SETUP

In this section, we describe the setup used for our evaluations.

The experiments are conducted on a server with 64GB of memory and an Intel i5-13600K processor, featuring 14 cores and 20 threads.

All experiments are performed using PostgreSQL 16 with pgvector 0.8.0. The primary evaluations use the HNSW indexing structure, with additional experiments applying HONEYBEE on top of the ACORN [23] index.

6.1 RBAC Benchmark

Following practices in RBAC literature, we use synthetic permission generators to generate permissions with different structures for

evaluation [18, 19, 30]. By default, we use $|U| = 1000$ users and $|R| = 100$ roles for all generators.

Random Generator [30]. This generator creates permission data without imposing any specific structure. It requires two parameters: the maximum number of roles a user can have (m_r), and the maximum number of documents a role can access (m_p). All selections are made uniformly at random.

- *Role-permission assignment* (ϕ_{PA}): For each role r , the generator randomly selects a number of permissions $m(r)$ between 1 and the total number of permissions $|D|$. Then, it randomly picks $m(r)$ permissions from D and assigns them to the role.
- *User-role assignment* (ϕ_{UA}): For each user u , the generator randomly selects a number of roles $m(u)$ the user has between 1 and $|R|$. Then, $m(u)$ roles are randomly chosen from R and assigned to the user.

We evaluate performance using two different sets of parameters. The first set, Random- α , used in § 7.1, is as follows: $m_r = 2$, and $m_p = |D|/|R| \times 5$. The second set, Random- γ , used in § 7.3, includes: $m_r = 1$, and $m_p = |D|/|R| \times 9$.

Tree-based Generator [19]. This generator models hierarchical role structures commonly seen in organizations (e.g., CEO \rightarrow department heads \rightarrow team leads \rightarrow employees). The generator-specific parameters include: the height of the tree (h) and the lower-bound (b_0) and upper-bound (b_1) on the number of children each internal node can have. The generator works as follows:

- *Tree and role construction*: Generate a tree T of height h , where the role hierarchy is constructed recursively from the root, with each internal node assigned a random number of children in the range $[b_0, b_1]$, stopping when the role pool of size $|R|$ is exhausted or height h is reached. Each node in the tree represents a role within the organization. Roles are organized hierarchically. For instance, consider a company with multiple departments, each containing several offices; the root node grants universal permissions, while a leaf node might represent a specialized role in a specific office.
- *Role-permission assignment* (ϕ_{PA}): Divide the set of documents D into $|R|$ subsets, and assign each subset to a corresponding role. Roles inherit all permissions from their ancestor roles, so each role r 's effective document access is computed as the union of documents directly assigned to r and all documents assigned to r 's ancestors².
- *User-role assignment* (ϕ_{UA}): Evenly distribute the set of users across all roles in the tree, excluding the root role. Each user is assigned to one role.

We evaluate performance using two different sets of parameters. The first set, Tree- α , used in 7.1, is as follows: $h = 4$, $b_0 = 3$, $b_1 = 4$. The second set, Tree- γ , has the same basic parameters as Tree- α , but uses the Poisson distribution for ϕ_{PA} . The Poisson distribution parameters are adjusted to vary the selectivity of permissions.

²For example, an employee in the Business Office of Department A would have access to company-wide permissions (from the root), Department A-specific permissions (from an intermediate node), and permissions unique to the Business Office (from a leaf node). Department-wide permissions are exclusive to employees within that department and never shared across departments, while office-specific permissions are restricted to employees of that office and not shared with other offices.

	Tree- α	Random- α	ERBAC- α	ERBAC- β
Avg Selectivity	0.036	0.054	0.128	0.285
Max Roles Per User	1	3	3	9
RP Storage Overhead	3.5 \times	3.8 \times	7.0 \times	7.0 \times
UP Storage Overhead	3.5 \times	74.9 \times	134 \times	408 \times

Table 1: Comparison of Workload configurations

ERBAC Generator [18, 19]. The generator is based on the Enterprise Role-Based Access Control (ERBAC) model, which uses a two-level layered role hierarchy commonly found in real-world organizations. It introduces two types of roles: functional roles, which define specific job functions and hold permissions directly, and business roles, which group functional roles and inherit their permissions. Notably, business roles represent the actual roles (R) assigned to users (U). This generator requires five parameters: the number of functional roles (n_{fr}), the number of business roles (n_{br}), the maximum number of permissions a functional role can have (m_p), the maximum number of functional roles a business role can connect to (m_{fr}), and the maximum number of business roles a user can have (m_{br}). The generator works as follows:

- *Generate functional and business roles*: For each functional role r , randomly select the number of permissions $m(r)$ between 1 and the total number of permissions $|D|$. Assign $m(r)$ randomly chosen permissions from the set D to r . For each business role r , randomly select the number of functional roles $m(r)$ from $\{1, 2, \dots, m_{fr}\}$. Assign $m(r)$ randomly chosen functional roles to r . For each business role the permission set is the union of all the permissions held by its associated functional roles.
- *Assign business roles to users*: For each user u , randomly select the number of business roles $m(u)$ from $\{1, 2, \dots, m_{br}\}$. Assign $m(u)$ randomly chosen business roles to u . For each user the permission is the union of all permissions inherited from the assigned business roles.

We evaluate performance using two different sets of parameters. The first set, ERBAC- α , used in 7.1, is defined as follows: $n_{fr} = 40$, $n_{br} = 100$, $m_{fr} = 3$, $m_{br} = 3$, $n_p = |D|$, and $m_p = |D|/25$. The second set, ERBAC- β , also used in 7.1, is similar to ERBAC- α , except that $m_{br} = 9$. The third set, ERBAC- γ , used in 7.3, shares the same basic parameters as ERBAC- α , with the exception of $m_{br} = 1$.

6.2 Dataset and Query Workload

For our evaluation, we utilize a dataset from the Wikipedia-22-12 collection, accessible via Hugging Face [8]. We work with 1 million rows of Wikipedia articles for this study. Each entry includes attributes such as a unique identifier (id), the article title (title), the content (text), a URL (url), and a unique wiki identifier (wiki_id). In our experiments, we map these fields to our system as follows: the wiki_id serves as a reference for each document, individual paragraphs within the text are treated as distinct units of content, and the text provides the basis for generating embeddings. These embeddings are created using the en_core_web_md model to capture the semantic representation of each paragraph.

Each query $q_i = (u_{q_i}, v_i)$ contains a user u_{q_i} and a query vector v_i . We randomly sample 1000 vectors from the database to serve as query vectors, and randomly select 1,000 user IDs to associate with

the queries. For each query, we want to retrieve top $k = 10$ closest vectors to the query vector under the user’s permission. We record the selectivity for each query, which is defined as the ratio of the number of documents accessible to the user to the total number of documents.

6.3 Baseline Methods and Metrics

We compare HONEYBEE against three baselines:

- **Role Partition:** We create a separate partition for each role. This approach is efficient for single-role users. However, for users assigned multiple roles, its performance may lag behind that of user partitioning due to the need to aggregate documents from multiple partitions during query execution.
- **User Partition:** We create a separate partition for each unique combination of roles. This way, users with the same roles can share the same partition.
- **Row-level Security (RLS):** This method applies PostgreSQL’s row-level security feature [5, 22] to filter query results based on permissions after performing similarity search. This is an example of a post-filter method.

The methods are compared using the following metrics:

- **Storage:** The algorithm leverages PostgreSQL to assess the actual table size, calculating the total storage needed for both data and indexes. We present the normalized storage cost by dividing the storage requirements of HONEYBEE by those of Row-Level Security (RLS).
- **Query Latency:** Average time to execute multiple queries. Each query is run twice: the first execution serves as a warmup to populate the cache, and the second is used for latency measurement.
- **Recall@10:** Proportion of correct documents in the top-10 vector search results, assessing accuracy against ground truth.

7 EVALUATION

We evaluate HONEYBEE using the described permission generators above and real-world document datasets. Our experiments highlight the following key findings:

- HONEYBEE significantly reduces query time while maintaining a reasonable storage overhead, achieving up to 6× speedup compared to RLS with only 1.4× storage increase.
- HONEYBEE performs best in structured datasets with hierarchical role distributions (e.g., Tree and ERBAC), while Random workloads show more limited gains. The trade-off curve is convex for structured workloads but concave in random cases.
- HONEYBEE enhances hybrid search methods like ACORN, achieving up to 3× speedup at 1.2× storage compared to ACORN alone. Further improvements could be achieved by refining the performance model to account for ACORN indexing.
- Lower selectivity and structured sharing degree patterns lead to greater efficiency gains. When selectivity is low (e.g., 0.04), HONEYBEE offers more optimization potential, while at higher selectivity levels, storage demands increase, narrowing the performance gap.

7.1 Trade-off between Storage, Latency, Recall

In this section, we evaluate the trade-off between storage overhead, query latency, and recall under different permission workloads.

Figure 4 shows the trade-off between storage overhead and average query latency across different permission workloads while maintaining a consistent recall threshold of 0.95. RLS (black point) offers minimal storage (1×) but high query latency, while Role Partition (red point) delivers low query latency but at significantly higher storage costs. HONEYBEE generates partitioning strategies that occupy the intermediate space, with partition sizes ranging from 1 to $|R|$ based on the storage constraint (for instance, for Tree- α with 1.4× storage constraint, it produces a partitioning strategy that involves 20 partitions); curves closer to the bottom-left corner reflect a superior balance between storage efficiency and performance. Note that HONEYBEE’s greedy algorithm: 1) *estimates* the space needed by a partitioning strategy as the total number of documents in all its partitions (instead of creating real partitions and measuring their actual size), and 2) it permits the limit to be exceeded (after a last partitioning step). This may lead to a deviation from the space constraint factor (α) used by the algorithm, when calculating this space factor using the actual size of the space occupied by the partitioning strategy (this real space factor is what we report in subsequent experiments). However, in all our experiments this deviation remained within 6%.

Figure 5 illustrates the relationship between query recall and latency under fixed storage constraints. We vary the ef_s parameter to tune this trade-off. User Partition (green curve) consistently delivers the lowest query latency at fixed recall levels, followed by Role Partition (red curve), while RLS delivers the highest latency (black curve). However, as shown in Table 1, this performance comes at substantial storage costs, with User Partition requiring 74-408× the storage of RLS, while Role Partition typically requires up to 10× storage. The minimal latency difference between these approaches suggests limited benefits from extending storage beyond Role Partition. HONEYBEE (yellow curve) produces partition strategies that lie between Role Partition and RLS, and generally shows a slower growth in query latency compared to RLS at high recall levels.

For the Tree- α workload (Figure 4a), HONEYBEE enables 6× improvements in query latency at 1.4× storage overhead compared to RLS, and the performance approaches that of Role Partition which requires 3.5× storage. Figure 5a shows that HONEYBEE with 1.4× storage significantly outperforms RLS and closely matches Role Partition at 3.5× storage at varying recall levels. Moreover, HONEYBEE exhibits a much slower growth in query performance as recall improves compared to RLS. User Partition is omitted from this comparison as it matches Role Partition due to the one role per user relationship in this workload. Overall, HONEYBEE offers a highly effective storage and latency trade-off in this workload.

For the ERBAC- α workload (Figure 4b), HONEYBEE enables a smooth, convex transition between RLS and Role Partition, indicating an effective trade-off. At approximately 3× storage, query latency is more than twice as fast as RLS. Figure 5b further shows that HONEYBEE with 3× storage has query latency closer to Role Partition compared to RLS, and shows a slower growth trend compared to RLS. User Partition shows slightly faster performance but

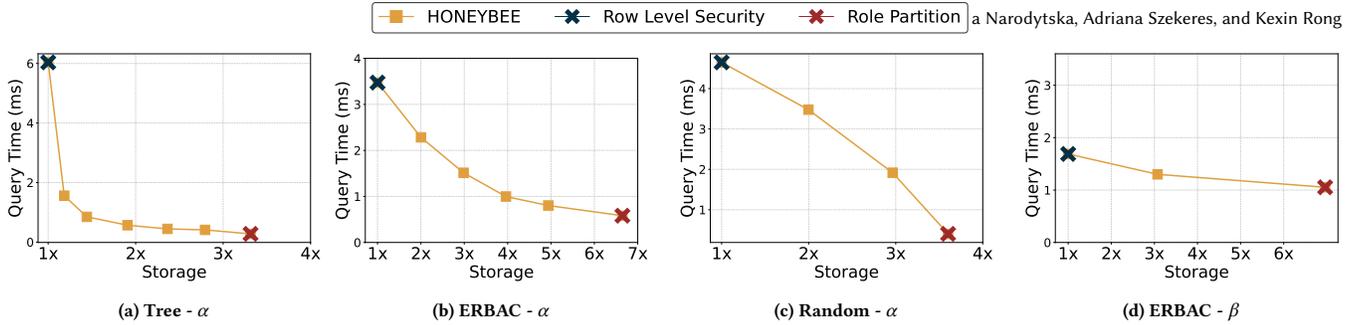


Figure 4: Trade-off between Query Time and Storage across Permission Workloads.

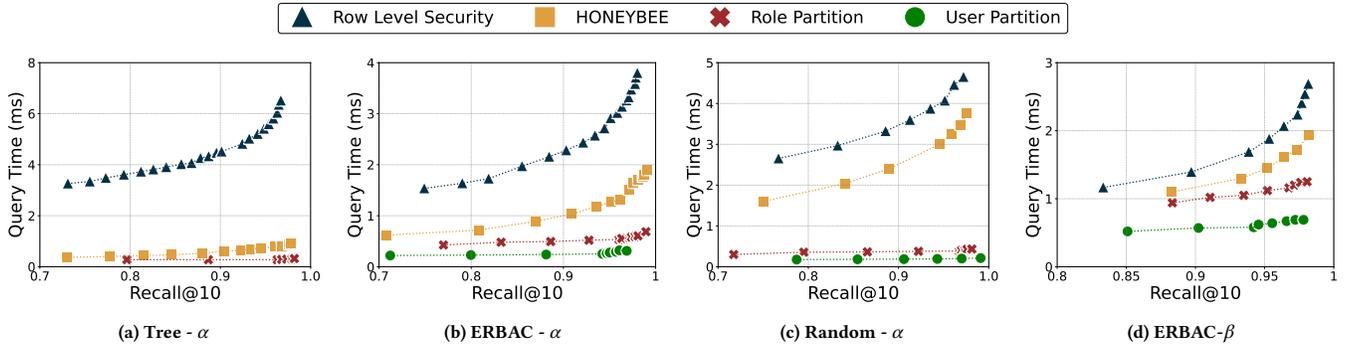


Figure 5: Query Time vs Recall under RLS and Different Partitioning Strategies. For HONEYBEE we used the following storage points for the four workloads: (a) 1.4x; (b) 3.0x; (c) 1.9x; (d) 3.2x.

requires 134x storage compared to Role Partition’s 7x, which suggest diminishing returns in query performance at larger storage budgets.

The ERBAC - β workload (Figure 4d) shows minimal query latency difference between RLS and Role Partition despite a 6x storage gap. This workload assigns up to 9 roles per user (versus 3 in ERBAC- α), resulting in higher user selectivity where post-filtering approaches like RLS perform well. Note that that HONEYBEE would generate the RLS solution if given the 1x storage constraint, and for a 3x storage overhead, HONEYBEE can produce a solution with query latency between RLS and Role Partition (Figure 5d). User Partition, though faster than Role Partition, requires 408x storage.

For the Random workload (Figure 4c), HONEYBEE shows less benefit in trading off storage or query latency compared to the previous workloads, as seen by the concave shape on the intermediate points. At 1.9x storage, query latency improves only 1.3x over RLS, which is still far from the Role Partition performance at around 3.5x storage. Additionally, Figure 5c shows that HONEYBEE at 1.9x storage shows a slight improvement over RLS and has a similar growth pattern as recall increases. The random role-permission assignments make it challenging to identify cleanly separable permission subsets, limiting HONEYBEE’s effectiveness. We provide additional analysis on the impact of the permission workload structure as well as selectivity in § 7.3.

7.2 Evaluation with Alternative Index

To demonstrate that HONEYBEE is compatible with different types of indices, we replace the default HNSW index with ACORN [23], a

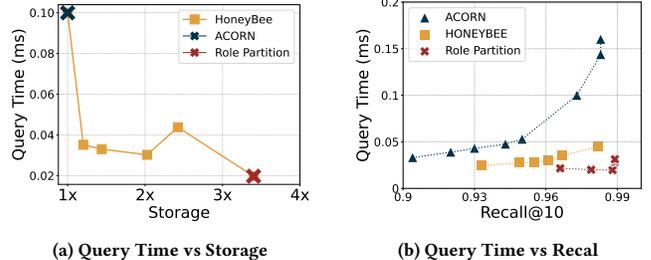


Figure 6: Performance of HONEYBEE using the ACORN index using the Tree- α workload.

specialized index built on top of HNSW designed to support hybrid vector search.

HONEYBEE uses the performance and recall model for the HNSW index to guide partitioning decisions. For each partition, we apply the ACORN index if the partition requires permission filtering, and the HNSW index if the partition does not (e.g., partitions containing only one user). We use a Tree- α generator to produce permissions, resulting in a low-selectivity setting (approximately 0.03). In such settings, ACORN offers significant performance advantages over post-filtering approaches like RLS [23].

The experiment uses ACORN’s implementation on the Faiss library with PostgreSQL. First, HONEYBEE runs on PostgreSQL to create partitions. Then, ACORN indexes are built on the document table (the default single table with all documents) and its partitions. We exclude multi-threading and caching optimizations for fair evaluation. Figure 6a shows the trade-off between query time and storage under dynamic partitioning. At 1.2x storage, HONEYBEE is

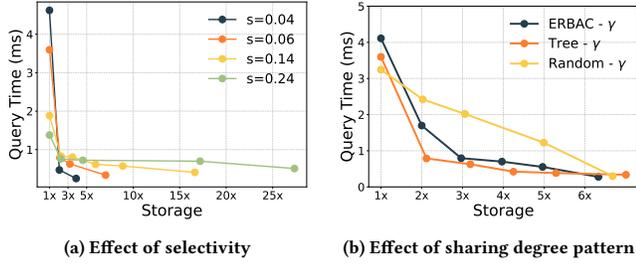


Figure 7: The effect of selectivity and sharing degree distribution on HONEYBEE performance.

roughly 3 \times faster than using a single ACORN index on all documents (1 \times storage). Notably, query time at 2.4 \times storage is slower than at 2 \times , due to the mismatch with performance model based on HNSW indexing. To elaborate, ACORN enhances HNSW by incorporating predicate processing capabilities, altering its original query processing mechanism. This modification particularly benefits recall: ACORN significantly outperforms HNSW when predicates are involved. In contrast, the HNSW model primarily emphasizes vector similarity search and fails to account for these predicate-related improvements. Therefore, replacing this performance model with an ACORN-specific performance model would likely improve results. Figure 6b shows the relationship between query latency and recall, with dynamic partitioning at 1.2 \times storage. The results show that both HONEYBEE and Role Partition maintain stable latency as recall increases, while a single ACORN index experiences a significant increase in latency at higher recall levels.

Overall, these results demonstrate that HONEYBEE are complementary to specialized indexes, such as ACORN. While ACORN improves the performance of post-filtering compared to using generic indices such as HNSW, HONEYBEE replicates vectors across different indices which can further improve the query performance compared to using a single hybrid index on the entire dataset.

7.3 Sensitivity Analysis

In this section, we analyze HONEYBEE sensitivity to *average selectivity* (percentage of documents accessible to the user) and *sharing degree distribution pattern*. The sharing degree distribution shows the percentage of documents by the number of sharing roles (e.g., 50% of the documents are accessible to 1 role each, 20% to 2 roles, 10% to 3, etc.). The sharing degree distribution pattern refers to the pattern of this distribution, such as sharing degree 1 being the highest peak (i.e., the percentage of documents accessible to 1 role each is the largest in the distribution), with other degrees showing lower peaks.

Impact of Selectivity. We use the Tree- γ generator to isolate the pattern across selectivity levels (0.04, 0.06, 0.14, 0.24). In Figure 8, both panels show the pattern where sharing degree 1 is the highest peak, with lower peaks for higher degrees, but the peak is higher at selectivity 0.06 (left) than 0.24 (right), indicating that the pattern is similar, yet the specific sharing degree distribution varies.

Figure 7a shows that at higher selectivity levels, RLS becomes more efficient, so the query latency at 1 \times storage approaches the performance of Role Partition (i.e., Δ Latency is small). Additionally, Role Partition consumes more storage as selectivity increases

(Δ Storage). Therefore, HONEYBEE has the most optimization potential at low selectivity, where the gap between RLS and Role Partition (Δ Latency/ Δ Storage) is largest.

For similar sharing degree distribution patterns, HONEYBEE’s performance curves show similar trends at around 2 \times storage. We observe that the rate of improvement in query latency decreases the fastest at this point, whereas further increases in storage yield diminishing returns.

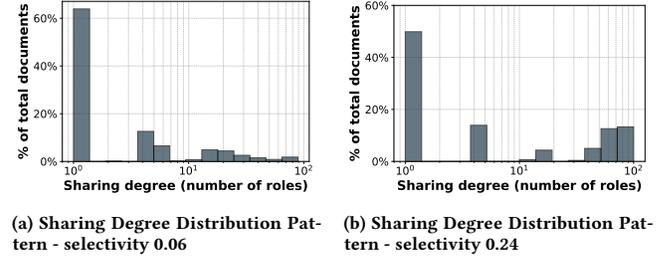


Figure 8: Comparison of sharing degree distribution pattern in different selectivity.

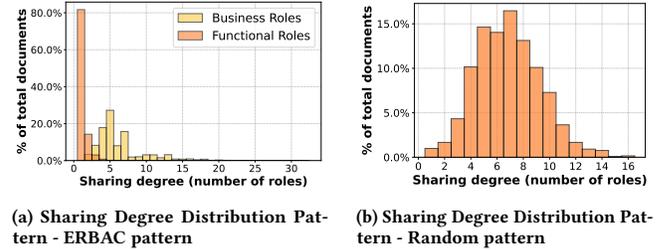


Figure 9: Comparison of sharing degree distribution pattern in different permission workloads.

Impact of Sharing Degree Pattern. Next, we evaluate the performance of HONEYBEE under workloads with different sharing degree patterns and similar selectivity (around 0.06 for all cases).

Three distinct sharing degree patterns are generated to ensure similar selectivity. The Tree pattern (Figure 8a) peaks at small sharing degrees, reflecting a hierarchical structure where documents are primarily accessed by a few roles. The ERBAC pattern (Figure 9a) reflects a two-layer hierarchical structure with functional roles and business roles. The Random pattern (Figure 9b) follows a Poisson distribution with an average sharing degree of 7.

Figure 7b shows that the query latency and storage consumption across patterns are nearly identical when storage is 1 \times (RLS), which is expected given that the workloads have the same average selectivity. However, HONEYBEE shows different trade-offs between storage and query latency, where performance ranking is Tree > ERBAC > Random (seen by the convexity of the trade-off curve). This shows that the structure of the permission workload directly impacts HONEYBEE’s performance.

Comparing Figs. 8a and 9a, we find that functional roles follow Tree pattern, but ERBAC performs worse due to the added

complexity of business roles. Similarly, Fig. 9a vs. 9b shows that Random, lacking hierarchical structure, performs worse, especially as its peak moves away from lower sharing degrees, which makes it more challenging to identify permission subsets that are good candidates for partitioning.

7.4 Update Benchmark

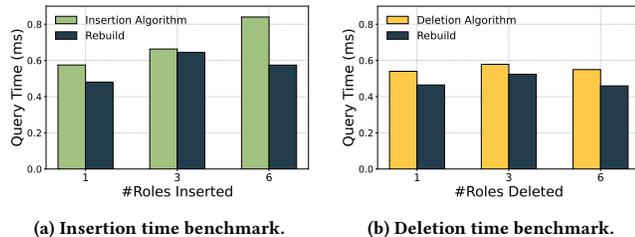


Figure 10: Comparison of incrementally maintaining partitions under workload change versus rebuilding from scratch.

In this section, we evaluate how efficiently HONEYBEE can maintain its partitioning strategy during permission workload changes, specifically insertions and deletions of roles.

Benchmark initialized with Tree- α generator, HONEYBEE at 1.5 \times storage. Insertions fix document sets, creating new roles from existing document subsets. For one insertion, we add users equaling 1% of the original user base. For deletions, removing roles deletes all users tied solely to those roles, potentially may altering document sets. We enforce a one-to-one user-role mapping for both operations to isolate role management effects. We evaluate insertions and deletions, grouped by operation count like (1, 3, or 6), comparing incremental update algorithms (insertion/deletion) to full partitioning rebuilds with HONEYBEE. Different insertion/deletion counts use distinct query workloads (permission change).

Figure 10a shows the insertion performance comparison. For small numbers of insertions (fewer than five), our incremental algorithm performs similarly to the rebuild approach. However, as the number of insertions increases, the incrementally maintained partitioning begins to show higher query latency compared to rebuilt partitions. This suggests that HONEYBEE might benefit from rebuilding when permission workloads change significantly. Figure 10b illustrates the deletion performance. Our incremental deletion algorithm consistently maintains query latency comparable to the complete rebuild approach across all tested scenarios.

8 RELATED WORK

Hybrid Search Indexes. Hybrid search indexes integrate structured filtering conditions into vector similarity search to enforce constraints like access control efficiently [23, 33]. ACORN [23] introduces an innovative approach to performant, predicate-agnostic hybrid search, capable of handling high-cardinality and unbounded predicate sets. It enhances the HNSW index by modifying its design, while integrating seamlessly into existing HNSW libraries, making it straightforward to implement. HQANN [33] offers a simple yet

effective hybrid query processing framework that can be effortlessly embedded into any proximity graph-based ANN search algorithm. It efficiently manages hybrid queries by processing them in a unified manner, enhancing performance and adaptability. Filtered-DiskANN [13] constructs graph-structured indexes by leveraging both vector geometry and label metadata (e.g., date, price). This approach advances beyond traditional methods by incorporating not only the geometric relationships between points but also their associated labels to build a more effective navigational graph structure. Our focus on the partitioning design of the dataset is orthogonal to the type of index used. As described in § 7.2, it complements hybrid search by addressing challenges at the partitioning and access policy management levels, enabling seamless integration of hybrid search techniques, such as ACORN, with our method.

Multi-Tenant Indexing. Curator [17] is a concurrent work addressing access control in vector databases, enhancing vector database indexing for multi-tenant environments. It introduces a hierarchical k-means clustering tree tailored to each tenant’s vector distribution, and embeds permission filters using Bloom filters within a shared clustering tree, enabling efficient search by skipping inaccessible vector clusters with minimal memory overhead. While Curator improves query performance and memory efficiency through its novel index designs, it considers enforcing access via access control list (user to permission mapping), instead of adopting the more general RBAC policies. Moreover, HONEYBEE’s partitioning strategy is orthogonal to the choice of the indices, and Curator can be considered as another specialized index designed for hybrid search. Thus, like ACORN, it is possible to integrate HONEYBEE with Curator to further improve its search performance.

9 CONCLUSION

We introduced HONEYBEE, a dynamic partitioning framework that bridges the gap between storage-efficient but slow post-filtering methods and fast but redundant per-role indexing in access-controlled vector databases. By leveraging the structure of RBAC policies, HONEYBEE partitions the vector space in a way that minimizes query latency while maintaining storage efficiency. Our analytical model predicts search performance and recall, enabling an optimization-driven partitioning strategy that balances query efficiency with access control constraints. Experimental evaluations on RBAC workloads demonstrate that HONEYBEE achieves up to 6 \times faster queries than row-level security while significantly reducing the storage overhead of per-role indexing. The results confirm that HONEYBEE provides a practical, scalable, and effective approach to enforcing access control in vector search, making it well-suited for real-world enterprise applications.

REFERENCES

- [1] [n.d.]. Q&A over Documents - LlamaIndex 0.8.43. <https://gpt-index.readthedocs.io/en/latest/>
- [2] Mohiuddin Ahmed, Raihan Seraj, and Syed Mohammed Shamsul Islam. 2020. The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics* 9, 8 (2020), 1295.
- [3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. *Advances in Neural Information Processing Systems* 28 (2015).
- [4] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 202–216.
- [5] Satori Cyber. n.d.. Postgres Row-Level Security: Comprehensive Guide. <https://satoricyber.com/postgres-security/postgres-row-level-security/>. Accessed: January 5, 2025.
- [6] Ninghui Li, Tiancheng Li, Ian Molloy, Qihua Wang, Elisa Bertino, and Seraphin Calo Jorge Lobo. [n.d.]. Role Mining for Engineering and Optimizing Role Based Access Control Systems. ([n. d.]).
- [7] Owen P Elliott and Jesse Clark. 2024. The Impacts of Data, Ordering, and Intrinsic Dimensionality on Recall in Hierarchical Navigable Small Worlds. In *Proceedings of the 2024 ACM SIGIR International Conference on Theory of Information Retrieval*. 25–33.
- [8] Hugging Face. 2022. Wikipedia 22-12 Dataset. <https://huggingface.co/datasets/Cohere/wikipedia-22-12>. Accessed: January 2025.
- [9] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. 2006. High dimensional nearest neighbor searching. *Information Systems* 31, 6 (2006), 512–540.
- [10] David F Ferraiolo, John A Cugini, and D Richard Kuhn. 1992. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)* (1992).
- [11] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-Out Graph. *Proceedings of the VLDB Endowment* 12, 5 (2019), 461–474.
- [12] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.
- [13] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. 2023. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*. 3406–3416.
- [14] Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. 2020. iDEC: Indexable Distance Estimating Codes for Approximate Nearest Neighbor Search. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1483–1497.
- [15] John A Hartigan, Manchek A Wong, et al. 1979. A k-means clustering algorithm. *Applied statistics* 28, 1 (1979), 100–108.
- [16] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 604–613.
- [17] Yicheng Jin, Yongji Wu, Wenjun Hu, Bruce M Maggs, Xiao Zhang, and Danyang Zhuo. 2024. Curator: Efficient Indexing for Multi-Tenant Vector Databases. *arXiv preprint arXiv:2401.07119* (2024).
- [18] Axel Kern, Andreas Schaad, and Jonathan Moffett. 2003. An administration concept for the enterprise role-based access control model. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (Como, Italy) (SACMAT '03)*. Association for Computing Machinery, New York, NY, USA, 3–11. <https://doi.org/10.1145/775412.775414>
- [19] Ninghui Li, Tiancheng Li, Ian Molloy, Qihua Wang, Elisa Bertino, Seraphin Calo, and Jorge Lobo. 2007. Role mining for engineering and optimizing role based access control systems. *Purdue University, IBM TJ Watson Research Center* (2007).
- [20] Aristidis Likas, Nikos Vlassis, and Jakob J Verbeek. 2003. The global k-means clustering algorithm. *Pattern recognition* 36, 2 (2003), 451–461.
- [21] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, 9 (2018), 2227–2240.
- [22] Microsoft. n.d.. Row-Level Security. <https://learn.microsoft.com/en-us/sql/relational-databases/security/row-level-security?view=sql-server-ver16>. Accessed: January 5, 2025.
- [23] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.
- [24] Yuting Qin et al. 2024. *Understanding Indexing Efficiency for Approximate Nearest Neighbor Search in High-dimensional Vector Databases*. Ph.D. Dissertation. Massachusetts Institute of Technology.
- [25] Ravi S Sandhu. 1998. Role-based access control. In *Advances in computers*. Vol. 46. Elsevier, 237–286.
- [26] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. 1996. Role-based access control models. *IEEE Computer* 29, 2 (1996), 38–47.
- [27] Kunal Sawarkar, Abhilasha Mangal, and Shivam Raj Solanki. 2024. Blended rag: Improving rag (retriever-augmented generation) accuracy with semantic search and hybrid query-based retrievers. In *2024 IEEE 7th International Conference on Multimedia Information Processing and Retrieval (MIPR)*. IEEE, 155–161.
- [28] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc. <https://papers.nips.cc/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Abstract.html>
- [29] Supabase. 2025. RAG with Permissions | Supabase Docs. <https://supabase.com/docs/guides/ai/rag-with-permissions>. Accessed: March 1, 2025.
- [30] Jaideep Vaidya, Vijayalakshmi Atluri, and Janice Warner. 2006. RoleMiner: mining roles using subset enumeration. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (Alexandria, Virginia, USA) (CCS '06)*. Association for Computing Machinery, New York, NY, USA, 144–153. <https://doi.org/10.1145/1180405.1180424>
- [31] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongfeng Ni. 2024. An efficient and robust framework for approximate nearest neighbor search with attribute constraint. *Advances in Neural Information Processing Systems* 36 (2024).
- [32] Brie Wolfson. 2023. Building chat langchain. <https://blog.langchain.dev/building-chat-langchain-2/>
- [33] Wei Wu et al. 2022. HQANN: Efficient and Robust Similarity Search for Hybrid Queries with Structured and Unstructured Constraints. In *Proceedings of the ACM International Conference on Information & Knowledge Management*. 4580–4584.
- [34] Yuxuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 6, Article 239 (Dec. 2024), 26 pages. <https://doi.org/10.1145/3698814>
- [35] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate Nearest Neighbor Search on GPU. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. IEEE, 1033–1044.