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

HONGBIN ZHONG, Georgia Institute of Technology, USA
MATTHEW LENTZ, Duke University, USA
NINA NARODYTSKA, VMware Research by Broadcom, USA
ADRIANA SZEKERES, Microsoft Research, USA
KEXIN RONG, Georgia Institute of Technology, USA

Enterprise deployments of vector databases require access control policies to protect sensitive data. These systems often implement access control through hybrid vector queries that combine nearest-neighbor search with relational predicates based on user permissions. However, existing approaches face a fundamental trade-off: dedicated per-user indexes minimize query latency but incur high memory redundancy, while shared indexes with post-search filtering reduce memory overhead at the cost of increased latency.

This paper introduces Honeybee, a dynamic partitioning framework that leverages the structure of Role-Based Access Control (RBAC) policies to create a smooth trade-off between these extremes. RBAC policies organize users into roles and assign permissions at the role level, 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 memory overhead. To guide these decisions, Honeybee develops analytical models of vector search performance and recall, and formulates partitioning as a constrained optimization problem that balances memory usage, query efficiency, and recall. Evaluations on RBAC workloads demonstrate that Honeybee achieves up to 13.5× lower query latency than row-level security with only a 1.24× increase in memory usage, while achieving comparable query performance to dedicated, per-role indexes with 90.4% reduction in additional memory consumption, offering a practical middle ground for secure and efficient vector search.

CCS Concepts: • **Information systems → Data management systems**; **Information retrieval query processing**; **Nearest-neighbor search**.

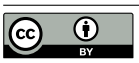
Additional Key Words and Phrases: Role-Based Access Control, Vector Search, Retrieval-augmented Generation

ACM Reference Format:
Hongbin Zhong, Matthew Lentz, Nina Narodytska, Adriana Szekeres, and Kexin Rong. 2026. Honeybee: Efficient Role-based Access Control for Vector Databases via Dynamic Partitioning. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 11 (February 2026), 26 pages. https://doi.org/10.1145/3786625

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) [3, 24, 34, 41]. Vector databases

Authors' Contact Information: Hongbin Zhong, Georgia Institute of Technology, Atlanta, USA, hzhong81@gatech.edu; Matthew Lentz, Duke University, Durham, USA, mlentz@cs.duke.edu; Nina Narodytska, VMware Research by Broadcom, Palo Alto, USA, n.narodytska@gmail.com; Adriana Szekeres, Microsoft Research, Redmond, USA, aszekeres@microsoft.com; Kexin Rong, Georgia Institute of Technology, Atlanta, USA, krong@gatech.edu.



ACM 2836-6573/2026/2-ART11
https://doi.org/10.1145/3786625

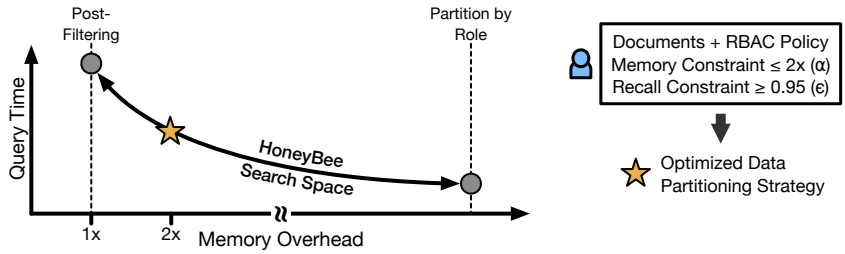


Fig. 1. Given RBAC policies, memory and recall constraints, HONEYBEE optimizes for a data partitioning strategy that achieves a balanced trade-off between query latency and memory overhead.

provide efficient vector similarity search to retrieve semantically relevant results from high-dimensional vector spaces, often implemented via approximate nearest neighbor (ANN) algorithms [2, 12, 14, 15, 18, 25, 30].

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 [37]. Role-based access control (RBAC) is a widely adopted framework for managing access to sensitive data [13, 21, 23, 32, 33]. 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 memory 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 memory 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 memory overheads that were 7× 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 memory 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 [16, 29, 42]. 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 [13, 23, 32, 33, 38]. Empirically, we find that partitioning

by user (or unique combinations of roles) significantly increases memory overhead compared to partitioning by roles, while offering diminishing returns in query latency improvements (*e.g.,* $> 50\times$ increase in memory 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 memory constraints, ranging from post-filtering to dedicated indices per role. The partitioning strategy operates independently of the underlying index structure, making Honeybee applicable across different vector search indices–requiring only the corresponding performance model. 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 memory 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 memory 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 memory redundancy compared to dedicated indices per role and achieves up to 13.5× faster query speeds than post-filtering on shared index (implemented via PostgreSQL's row-level security feature), with only 1.24× memory increase, while achieving comparable query performance to role partition with 90.4% reduction in additional memory consumption.

2 Background

In this section, we provide background on role-based access control (RBAC) (§ 2.1) and hybrid vector 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 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 [13, 23, 33]. 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: $auth(u_i) = \bigcup_{r \in \phi_{UA}(u_i)} \phi_{PA}(r)$.

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.

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 [7, 27]. 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, Figure 2 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
));
```

Fig. 2. RBAC policy implemented via Row-Level Security.

2.2 Relationship with Hybrid Search

Hybrid search refers to queries jointly searching vector data (embedded images and text) and structured data (attributes and keywords). For example, access control policies for vector databases can be implemented as hybrid search queries, where users retrieve documents relevant to their query (measured by vector distance) among those accessible (structured filtering condition).

Existing hybrid search solutions fall into two categories. First, index-based solutions (e.g., ACORN [29], Filtered-DiskANN [16], Curator [20]) modify existing vector indices to handle filtered queries more efficiently. These approaches adapt both index construction and querying processing to improve post-filtering performance *within a single index structure*. For example, ACORN [29] enhances Hierarchical Navigable Small Worlds (HNSW) [25], state-of-the-art graph-based index for approximate nearest neighbor search, by integrating filtering conditions directly into index construction. By expliciting considering filtering predicate selectivity during graph construction, ACORN can improve connectivity and achieve better query recall and latency versus post-processing approaches that apply filters after vector search.

Second, *partition-based solutions* (e.g., HQI [28], our approach) partition datasets according to query predicates and build separate indices for each partition. Our framework falls into this category and is complementary and largely orthogonal to index-based methods–we address how to optimally distribute data across indices based on access policy structure, while index-based

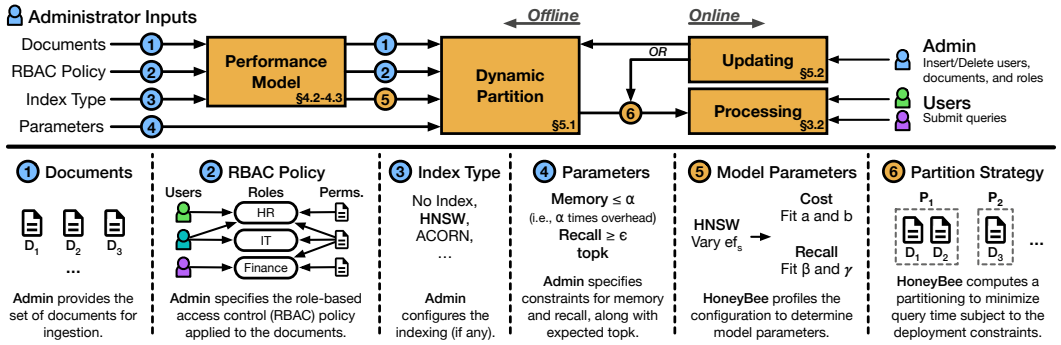


Fig. 3. Overview of HONEYBEE's workflow. Yellow elements represent main components as well as inputs and outputs of HONEYBEE, while blue elements represent configurable components by administrators, such as documents, RBAC policies, index types, and constraints.

methods focus on enhancing individual index structures. In the evaluation (§ 7.2), we show our approach complements index-based methods and integrates with existing hybrid search indexes to further enhance performance for permission-aware vector search.

We provide a more detailed comparison with prior works in both categories in the related work section (§ 8).

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 $D = \{d_1, d_2, ...\}$ 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 $U = \{u_1, u_2, ...\}$ be the set of users, and $R = \{r_1, r_2, ...\}$ 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). $auth(u_i)$ 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, where a separate index is built on each partition. A configuration is represented as a non-disjoint partitioning $\Pi = \{\pi_1, \pi_2, ...\}$, where $\cup_i \pi_i = D$ and $\forall i : \pi_i \subseteq D$ and $\pi_i \neq \emptyset$. We use the term "partitioning" in this context to refer to the creation of physically separate indexes that may have data overlap, rather than a strict mathematical disjoint partitioning. To ensure high query performance, we assume that all indices are stored in memory, whether using a single shared index or multiple indices built on each partition.

The set of partitions that contain documents accessible to role r_i is defined as: $P_r(r_i, \Pi) = \{j \mid \pi_j \in \Pi, \pi_j \cap \phi_{PA}(r_i) \neq \emptyset\}$. The set of partitions that contain documents accessible to user u_i is the union of partitions accessible to each of its roles: $P_u(u_i, \Pi) = \bigcup_{r \in \phi_{UA}(u_i)} P_r(r, \Pi)$.

Configurations are evaluated on three dimensions:

- **Memory overhead**: $\frac{\sum_i |\pi_i|}{|D|}$, the ratio of total required memory of the configuration to the minimum memory requirement of using a single shared index. We use partition size as a proxy for the memory overhead, since index size is proportional to the number of vectors stored. We focus on *physical partitioning*, where each partition maintains its own copy of the vector data. This design follows the implementation of vector indexes such as HNSW, which physically co-locates vector data and index metadata (*e.g.,* graph structure) to enable low latency queries. § 7.4 provides a comparison with logical partitioning.
- **Search quality**: $R(\Pi, u_i)$, the search recall for user u_i is defined as the ratio between (1) results from indexed search filtered by u_i's access permissions and (2) the ground-truth top-k results from exhaustive search followed by access control filtering.
- **Search performance**: $C(\Pi, u_i) = \sum_{j \in P_u(u_i,\Pi)} c(\pi_j)$, 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.

Problem 1. *Given a set of documents D, RBAC policies, memory constraint $\alpha(\geq 1)$, and recall constraint $\epsilon(< 1)$, find a partitioning $\Pi = \{\pi_1, \pi_2, ...\}$ that minimize the average query latency over users:*

$$\underset{\Pi}{minimize} \quad \frac{1}{|U|}\sum_{i=1}^{|U|} C(\Pi, u_i), s.t. \frac{\sum_i |\pi_i|}{|D|} \leq \alpha, \frac{1}{|U|}\sum_{i=1}^{|U|} R(\Pi, u_i) \geq \epsilon$$

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 partitioning and indexing, and an online phase for query processing. Figure 3 provides an overview of the system workflow.

Offline. In the offline phase, Honeybee solves the constraint optimization problem (Problem 1) to determine the optimal partitioning strategy. Depending on the memory constraints, this strategy can range from a single shared partition (equivalent to post-filtering on a shared index) 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 index type is configurable by users, with options including no index (exhaustive search), vector indices for $k-$ANN search (*e.g.,* HNSW, IVF-PQ), or specialized indices for hybrid search (*e.g.,* ACORN). Along with partitioning, Honeybee determines the index-specific parameter that controls search depth (*i.e.*, number of candidate results considered during search) and affects the search latency and recall during query time. The search depth is influenced by the partitioning design. When all documents are in one partition, high search depth is required to ensure sufficient results remain after permission 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. Our analytical model for query recall (Eq 5) provides an initial recommendation for these parameters, which users can further finetune to meet specific requirements if needed.

Since partitions may overlap, the set of partitions containing documents accessible to a user ($P_u(u_i, \Pi)$ defined in § 3.1) could include redundant partitions. To optimize query processing, Honeybee pre-computes and maintains a routing table, $P^*(u_i, \Pi)$, from each unique combination

of roles (or each distinct user) to their minimal required set of partitions:

$$P^*(u_i, \Pi) = \underset{S \subseteq P_u(u_i,\Pi)}{\arg\min} \sum_{j \in S} c(\pi_j), \text{ s.t. } \bigcup_{j \in S} \pi_j \supseteq auth(u_i) \tag{1}$$

where $auth(u_i)$ is the set of documents accessible to user u_i (Def 2.1). For role-level analysis, $P^*(r_i, \Pi)$ is defined analogously by substituting $auth(u_i)$ with $auth(r_i)$ in Eq. 1. Once precomputed, locating the required partitions during query time requires a single key-value lookup ($O(1)$ operation). The routing table itself is extremely lightweight— even with ten thousand entries (each storing a few integer partition IDs), it only occupies a few hundred kilobytes, negligible compared to the gigabyte-scale vector indexes.

To identify the covering set, we iteratively choose the partition that covers the largest number of uncovered documents; when multiple partitions provide equal coverage, we select the smaller partition to reduce scan cost.

Online. When processing a query $q_i = (u_{q_i}, v_i)$, Honeybee first identifies the relevant partitions using the precomputed mapping $P^*(u_{q_i}, \Pi)$. For each identified partition, the system performs vector similarity search using the query vector v_i with a vector index. The search process is controlled by index-specific parameters that determine how many candidates to examine (e.g., *ef_search* for HNSW, *nprobe* for IVF-PQ). After vector search, access control is applied to filter out unauthorized documents through post-filtering (Figure 2). Finally, Honeybee merges the filtered results from all searched partitions, sorts them by similarity score, and returns the global top-k results.

4 Analytical Modeling

In this section, we introduce the analytic models for the vector search performance and recall, using HNSW index with post-filtering as a concrete instantiation of the framework.

4.1 Background: HNSW Index

HNSW is among the most widely adopted graph-based indexes for ANN search [25]. It organizes vectors as nodes in proximity graphs where edges connect similar vectors. During search, the algorithm starts from predefined entry points and greedily moves to neighboring nodes closer to query vectors. HNSW performance is governed by three key parameters: M, $ef\_construction$ (ef_c), and ef_search (ef_s). M and ef_c are construction-time parameters defining the structure and quality of the HNSW graph. M controls link numbers each node maintains, affecting memory usage and search efficiency. ef_c determines neighbors considered when inserting new points, influencing indexing speed and accuracy. In contrast, ef_s is a query-time parameter that can be adjusted without rebuilding the index, making it the primary lever for latency–recall trade-offs. ef_s controls search depth by determining priority queue size used to manage candidate nodes during queries. Higher ef_s values improve recall at increased query latency.

In this study, we use the recommended default settings for M and ef_c following common practices [10, 26]. Our analytical model treats M and ef_c as system-level constants while focusing on jointly optimizing partitioning and the query-time parameter ef_s. Tuning M and ef_c controls the graph's connectivity and construction-time search quality, which affects both index size (and thus memory constraints) and recall (and thus performance models). Incorporating their tuning into Honeybee 's offline optimization could expand the design space and enable partitioning strategies with improved memory–latency tradeoffs. For tractability, we also use a single global ef_s across all partitions. While this simplifies the model, it limits the role of ef_s in the recall–latency tradeoff, as different partitions may benefit from different query-time settings depending on their size and selectivity. Allowing per-partition ef_s could substantially enlarge the online tuning space and lead to

latency gains in highly heterogeneous partitions. We leave such multi-parameter and per-partition tuning to future work.

The following sections introduce analytical models characterizing the relationship between ef_s, query recall, and latency.

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 [25]. 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 $c(\pi_i, ef_s) = \log(|\pi_i|) \cdot f(\mathrm{ef}_s)$, where $f(\mathrm{ef}_s)$ represents the relationship between search queue size and query time. Empirical analysis demonstrates that $f(\mathrm{ef}_s)$ exhibits a linear relationship with ef_s which we model as $f(\mathrm{ef}_s) = a \cdot \mathrm{ef}_s + b$. Here, a and b are system-dependent parameters influenced by hardware capabilities, software optimizations, and dataset characteristics such as intrinsic dimensionality.

Overall, the query cost is a function of the partitioning scheme 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 $C_u(\Pi, u_i, ef_s)$ must account for all partitions that contain documents accessible to that user, as defined by $P^*(u_i, \Pi)$ in Eq 1:

$$C_u(\Pi, u_i, ef_s) = \sum_{j \in P^*(u_i, \Pi)} \log(|\pi_j|) \cdot (a \cdot \mathrm{ef}_s + b) \tag{2}$$

Similarly, the role-level query cost is defined as

$$C_r(\Pi, r_i, ef_s) = \sum_{j \in P^*(r_i, \Pi)} \log(|\pi_j|) \cdot (a \cdot \mathrm{ef}_s + b) \tag{3}$$

Note that since different roles could be mapped to the same partitions via $P^*(r_i, \Pi)$, 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 (§ 6.2) to create a workload where each user maps to one role, and one partition is created per role. Although users have a 1:1 mapping with roles, different role-based partitions may still overlap. Here, $\mathrm{P}^*(r_i, \Pi) = \pi_{r_i}$, 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.

4.3 Model for Search Recall

Next, we model the recall behavior of HNSW index, which uses post-filtering for access control[1]. (see §7.2). Other than the search depth parameter, ef_s, and the k parameter in the top-k query, the primary factor affecting recall is *access selectivity*: the fraction of documents a user can access in their assigned partitions. For a user u_i, access selectivity is defined as the average fraction of accessible documents across all partitions the user needs to query:

$$s_u(u_i) = \frac{1}{|P^*(u_i, \Pi)|} \sum_{j \in P^*(u_i, \Pi)} \frac{|auth(u_i) \cap \pi_j|}{|\pi_j|} \tag{4}$$

[1]Other search techniques (e.g., pre-filtering, single-stage scans) could also be supported in the framework by substituting their respective performance or recall models while preserving the same partitioning logic.

This is averaged across all users to find the system-wide average access selectivity: $\overline{s_u} = \frac{1}{|U|}\sum_{i=1}^{|U|} s(u_i)$. We defined $\overline{s_u}$ using the "average of ratios" rather than a "ratio of sums" to better capture multi-partition search dynamics. Consider a query spanning Partition A (2,000 documents, 90% accessible) and Partition B (100,000 documents, 0.5% accessible). The "ratio of sums" approach yields 2% average selectivity, suggesting both partitions require high ef_s values, which would be overkill for Partition A. In contrast, our "average of ratios" approach yields $\approx$ 45% average selectivity, which better reflects that relatively small ef_s suffices for high recall: Partition A already has high selectivity, while Partition B's sparse accessible documents limit candidates regardless of ef_s.

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 5: a linear function for the initial rapid increase and a sigmoid function to capture the saturation effect.

$$R(\Pi, ef_s, \overline{s_u}, k) = \begin{cases} \frac{ef_s \cdot \overline{s_u}}{k}, & \text{if } ef_s \le \gamma \cdot \frac{k}{\overline{s_u}}, \\ \frac{1}{1+e^{-\beta \cdot \frac{\overline{s_u}}{k}\left(ef_s - \gamma \cdot \frac{k}{\overline{s_u}}\right)}} + \left(\gamma - \frac{1}{2}\right), & \text{otherwise.} \end{cases} \tag{5}$$

This model connects a partitioning scheme (which determines $\overline{s_u}$) to the search efforts (ef_s) required to meet the recall target for a top-k query. ef_s is mainly governed by the ratio $\frac{k}{\overline{s_u}}$, the expected number of search candidates that must be examined to find k results that pass the permission filter. The recall model uses two scaling relationships with fitted constants γ and β with respect to $\frac{k}{\overline{s_u}}$:

- **Transition point** $\gamma \cdot \frac{k}{\overline{s_u}}$: This determines when we transition from linear growth to saturation. With lower access 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 the function value at the transition point.
- **Sigmoid steepness** $\beta \cdot \frac{\overline{s_u}}{k}$: This controls the rate at which recall improves beyond the transition point. Higher access selectivity ($\overline{s_u}$) 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 average access selectivity 0.1. As ef_s increases from 1 to 1000 (typical upper limit in databases like pgvector), recall transitions from 0 to 1. We execute 1000 random queries across varying ef_s values (10 to 1000), collecting average recall per setting. Before each query, we compute access selectivity and retrieve top-k to fit β and γ in Eq 5. Our evaluation shows that these parameters are insensitive to the specific workload used for fitting (§ 7.3).

Modeling search performance and recall for HNSW is a challenging problem that has been extensively studied in prior work [22, 40]. Our goal is not to produce perfectly accurate models, but to capture the key factors that influence performance in order to guide HONEYBEE 's partitioning decisions. For this purpose, we make several simplifying assumptions—such as ignoring the effects of data distribution and intrinsic dimensionality on search latency [10], and the effects of the correlation between query vectors and user access permissions on search recall [29]. Although these simplifications may limit model accuracy when used in isolation, they are sufficient to guide optimization as long as the model can distinguish the relative ranking among candidate partition choices. Our evaluation shows that HONEYBEE 's partitioning algorithm is robust to modeling imperfections (§ 7.3) and works even with asymptotic performance models that require no parameter fitting (§ 7.2).

5 Dynamic Partitioning Strategy

Given the analytical models for query performance and recall, we formulate Problem 1 as a constrained optimization problem (formal specification is provided in the extended version [45]). This

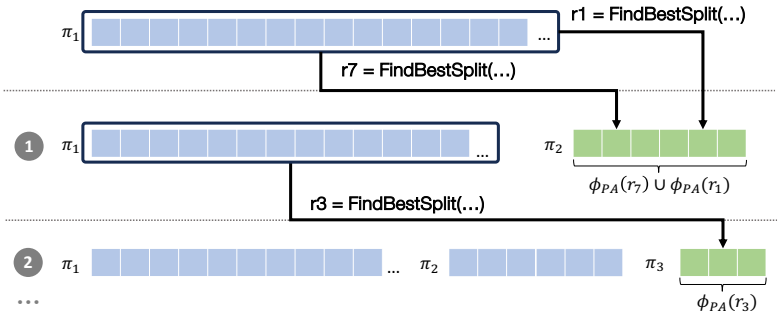


Fig. 4. Each iteration of the greedy algorithm, one or more roles are moved from the largest partition to form a new partition (green blocks). The roles are chosen greedily based on the estimate performance improvement using the analytical models.

problem is a Mixed-Integer Nonlinear Program (MINLP), which is NP-hard due to its combinatorial search space and nonlinear constraints. In this section, we introduce Honeybee 's greedy partitioning strategy that efficiently solves the optimization problem.

5.1 Greedy Split Algorithm

Honeybee makes a key observation to solve the optimization problem. The design space is inherently challenging, as it grows exponentially with the total number of documents. To tackle this scalability challenge, we propose to constrain the search space by ensuring that all documents accessible by a given role are contained in a single partition.

This constraint significantly reduces the search complexity and enables an efficient search algorithm: rather than partitioning over all documents D, we only partition over roles R, where in typical RBAC settings $|R| \ll |D|$. It also provides query latency benefits, since each role's vector search operates over a single partition of size $|\pi_r|$, yielding logarithmic access cost in $|\pi_r|$. In contrast, if a role's documents were split across multiple partitions, the access cost becomes the sum of logarithms across those partitions, effectively growing multiplicatively with the number of partitions (see § 5.2 for details).

We propose a greedy partitioning algorithm (Algorithm 1) based on this intuition. The algorithm returns the final partitioning design Π and mapping M which maps each partition to its contained roles. M can be used to calculate $P_u(u_i, \Pi)$ as defined in § 3.1.

As shown in Figure 4, our algorithm follows an iterative splitting approach beginning with a single partition containing all documents and all roles. In each iteration, we identify the largest partition containing more than one role and split it by moving selected roles r^* to a new partition. The roles are chosen to: (1) group all documents accessible by r^* into the new partition, and (2) maximize query performance improvements based on our analytical model. Each split introduces a trade-off between memory and query efficiency. Memory 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 access selectivity) in the new partition.

Algorithm 1 Greedy Split Algorithm

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

1: **function** CreatePartition($\mathcal{R}$) ▷ All docs needed by roles in $\mathcal{R}$
2: **return** $\bigcup_{r \in \mathcal{R}} \phi_{PA}(r)$
3: $\Pi[1] \leftarrow D$ ▷ Initialize partition with all documents and roles
4: $M[1] \leftarrow R$
5: **while** $\sum_{\pi_i \in \Pi} |\pi_i| \leq \alpha |D|$ **do**
6: $i_{src} \leftarrow$ FindLargestSplittablePartition(Π, M)
7: $i_{dst} \leftarrow |\Pi| + 1$ ▷ Create new partition
8: $\Pi[i_{dst}] \leftarrow \emptyset, M[i_{dst}] \leftarrow \emptyset$
9: **while** $\sum_{\pi_i \in \Pi} |\pi_i| \leq \alpha |D|$ **do**
10: $r^* \leftarrow$ FindBestSplit(Π, M, i_{src}, i_{dst})
11: $M[i_{src}] \leftarrow M[i_{src}] \setminus \{r^*\}$ ▷ Update mappings
12: $M[i_{dst}] \leftarrow M[i_{dst}] \cup \{r^*\}$
13: $\Pi[i_{src}] \leftarrow$ CreatePartition($M[i_{src}]$) ▷ Update partitions
14: $\Pi[i_{dst}] \leftarrow$ CreatePartition($M[i_{dst}]$)
15: **if** $i_{src} \neq$ FindLargestPartition(Π, M) or $r^* = \{\}$ **then**
16: **break**
17: **return** Π, M

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 12), 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 13). Then we 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 (i) query latency no longer improves, (ii) the memory budget would be exceeded, or (iii) $\Pi[i_{\text{src}}]$ is no longer the largest splittable partition. Note that $\Pi[i_{\text{dst}}]$ can contain multiple roles at the end of a round.

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 3) and a user-level model C_u (Eq 2). For $C_r(\Pi)$ or $C_u(\Pi)$, users specify a target recall and input it into the program to compute the corresponding average access selectivity $\overline{s_u}$ for a given partition Π. Then the ef_s is derived using Eq. 5. Intuitively, the role-level model reflects local effects - 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 global effects - reducing user-level cost directly lowers the overall query-time objective.

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 memory cost increase. We normalize the query improvements by the memory cost to identify the split that is most effective per unit memory. In practice, ΔS could also be zero or negative; we then use $(\Delta Q_r + \Delta Q_u)/(\Delta S + \epsilon)$, where a small ϵ is added to avoid division by zero and ensure numerical stability when $\Delta S \approx 0$, prioritizing roles r with $\Delta S < 0$ for reduced memory and query latency.

Algorithm 2 FindBestSplit Algorithm

```
1: function FindBestSplit(Π, M, i_src, i_dst)
2:     r* = {}
3:     for all r ∈ M[i_src] do                                  ▷ Try each split
4:         π'_dst ← CreatePartition(M[i_dst] ∪ {r})
5:         π'_src ← CreatePartition(M[i_src] \ {r})
6:         Π' ← Π \ {Π[i_src], Π[i_dst]} ∪ {π'_src, π'_dst}
7:         ΔS ← |π'_src| + |π'_dst| − |Π[i_src]| − |Π[i_dst]|     ▷ ΔStorage
8:         ΔQ_r ← C_r(Π) − C_r(Π')                              ▷ ΔQuery time
9:         ΔQ_u ← C_u(Π) − C_u(Π')
10:        if ΔQ_r < 0 and ΔQ_u < η then                        ▷ Check if move is beneficial
11:            if (ΔQ_r + ΔQ_u)/ΔS > Δ_max then
12:                r* ← r
13:                Δ_max ← (ΔQ_r + ΔQ_u)/ΔS
14:    return r*
```

5.2 Analysis of the Greedy Algorithm

Greedy Algorithm Complexity. Let $|R|$ be the number of roles and $|D|$ the number of documents. Alg 1 performs at most $|R|$ split operations. Each split calls FindBestSplit (Algorithm 2), which scans up to $|R|$ roles in the source partition. For each role, the dominant cost arises from updating the partitions after a hypothetical move (Lines 3-4), where creating a partition from a set of roles requires taking the union of their accessible document sets, giving a worst-case time complexity of $O(|R||D|)$. Combining these steps yields an overall worst-case time complexity of $O(|R|^3|D|)$.

In practice, the complexity is significantly lower for several reasons. First, the number of split operations, $|P|$, where P is the set of partitions returned by the algorithm, is primarily determined by the storage constraint α rather than $|R|$, and $|P| \ll |R|$ for realistic storage budgets. Second, the average number of documents accessible per role $|D_{avg}|$ is typically much smaller than $|D|$ (*i.e.*, $|D_{avg}| \ll |D|$), reducing the cost of the union to $O(|R||D_{avg}|)$. In our evaluation on the SIFT10M dataset, the greedy algorithm typically completes in under a minute. A tighter bound that formally captures the relationship between α and $|P|$, and considers realistic RBAC workload permissions is left for future work.

Impact of Splitting Roles Across Partitions. For HNSW-based search, the per-partition top-k search cost is $O(ef_s \log|\pi|)$, where ef_s is the candidate expansion for the target recall [25, 29]. Under post-filtering semantics, retrieving k accessible results requires $ef_s \propto \frac{k}{\overline{s_u}(\pi)}$, where $\overline{s_u}(\pi)$ is access selectivity of role u in partition π. Therefore, if a role u must access multiple partitions $P^*(u, \Pi)$, the total query cost is

$$\sum_{j \in P^*(u,\Pi)} O\left(\frac{k}{\overline{s_u}(\pi_j)} \log|\pi_j|\right).$$

Splitting a role's accessible data across multiple partitions introduces overhead, as each partition incurs a separate logarithmic search cost. When summed across partitions, these costs accumulate as $\sum_j \log|\pi_j| = \log(\prod_j |\pi_j|)$, which can exceed the cost of searching a single combined partition, which scales with $\log(|\bigcup_j \pi_j|)$. Therefore, splitting a role across multiple partitions generally increase latency, unless each split significantly improves access selectivity (*i.e.*, $\overline{s_u}(\pi_j)$) to offset the additional $\log|\pi_j|$ factors.

5.3 Handling Updates

We need to consider three cases for how Honeybee updates its partition scheme under permission workload changes: (1) user insertion/deletion, (2) document insertion/deletion from roles, and

(3) role insertion/deletion. Since Honeybee's partitioning strategy relies on roles, updates can be performed incrementally without full partition rebuilds. When adding new users to existing roles, Honeybee identifies the optimal set of partitions for the user and updates user-to-partition routing tables (P^*, Eq 1) accordingly. Conversely, deleting users requires no partition changes; only the user's access paths in the routing table are removed. When inserting new documents into existing roles, Honeybee locates corresponding partitions and inserts documents. When deleting documents from roles, Honeybee removes specific documents from partitions containing the role, but user routing remains unchanged. When inserting new roles, the system evaluates performance impact ($\Delta C/\Delta memory$), assigns role documents to existing or newly created partitions, and updates routing for users assigned to new roles. Role deletion involves identifying all partitions containing the role, removing role-exclusive documents, and updating user-to-role assignment (ϕ_{UA}). In both cases, only affected partition indices require updates.

While the above procedures handle incremental updates efficiently, significant changes in the permission workload may eventually necessitate full repartitioning to restore balance. For instance, if multiple large roles are added over time, or if role ownership patterns shift dramatically, the initial partition scheme may become suboptimal. However, deciding when repartitioning is necessary represents a substantial research challenge in its own right, as it requires carefully balancing the large upfront cost of repartitioning against potential future query time savings. Prior work in different contexts (e.g., partitioning design for OLAP workloads) has explored reorganization strategies using rule-based heuristics [35], reward functions [9] and online algorithms [31]. We leave the design of adaptive repartitioning strategies for RBAC workloads with vector data as a promising direction for future work.

6 Experimental Setup

6.1 Dataset and Query Workload

We evaluate Honeybee on three datasets: (1) Wikipedia-22-12 [11], from which we sample 1 million entries. Each entry includes a unique identifier, article title, full text, and a Wikipedia ID (wiki_id). We use wiki_id as the document identifier, and generate 300-dimensional paragraph embeddings using the spaCy model en_core_web_md. (2) SIFT1M and SIFT10M [19], which are standard vector search benchmarks. SIFT1M contains 1 million 128-dimensional vectors, while SIFT10M contains 10 million. Each vector is associated with a unique identifier and a URL pointing to the original image from which the SIFT features were extracted. We use the unique identifier as the document ID for indexing and retrieval.

Each query $q_i = (u_{q_i}, v_i)$ contains user u_{q_i} and query vector v_i. We randomly sample 1000 database vectors as query vectors, and randomly select 1000 user IDs to associate with the queries. By default, each query retrieves the top-$k = 10$ nearest vectors within the user's permission, but we also experiment with other k values in the evaluation. § 6.2 discusses how we generate RBAC permission workloads with different structures.

We run experiments on a server with Intel i5-13600K processor and 64GB memory. All experiments use PostgreSQL 16 with pgvector 0.8.0. Primary evaluations use the HNSW index, with additional experiments applying Honeybee to ACORN [29] index. Experiments use a warm-up procedure to ensure performance measurements reflect in-memory operations only. We execute each query twice within the same database session: first to warm-up, and second to measure steady-state performance via pure in-memory traversal of the index, eliminating disk I/O and buffer management overhead. We record query selectivity (ratio of user-accessible documents to total documents).

Table 1. Comparison of workload configurations.

	Tree-α	**Uniform-α**	**ERBAC-α**	**ERBAC-β**
Avg Access Selectivity	0.036	0.054	0.128	0.285
Max Roles Per User	1	3	3	9
Role Partition Memory Overhead	3.5×	3.8×	7.0×	7.0×
User Partition Memory Overhead	3.5×	74.9×	134×	408×

6.2 RBAC Benchmark

Following common practices in RBAC literature, we use permission generators to simulate permissions with different structures for evaluation [21, 23, 38]. By default, all generators use $|U| = 1000$ users and $|R| = 100$ roles.

Uniform Generator [38]. This generator produces permission data without specific structure. Two parameters are required: maximum roles per user (m_r) and maximum documents per role (m_p). Both role-permission assignment (ϕ_{PA}) and user-role assignment (ϕ_{UA}) are uniformly generated at random.

We evaluate performance using two different sets of parameters. Performance evaluation uses two parameter sets. Uniform-α, employed in § 7.1: $m_r = 2$, $m_p = |D|/|R| \times 5$. Uniform-s, employed in § 7.5: $m_r = 1$, $m_p = |D|/|R| \times 9$.

Tree-based Generator [23]. This generator models organizational hierarchical role structures (e.g., CEO → department heads → team leads → employees). Parameters include: tree height (h), lower-bound (b_0) and upper-bound (b_1) for children per internal node.

- *Tree and role construction*: Generate tree T with height h, constructing role hierarchy recursively from root, assigning random children in range $[b_0, b_1]$ until role pool $|R|$ exhausted or height h reached. Each tree node represents an organizational role in hierarchical structure. 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}):* Split document set D into $|R|$ subsets, assigning each to its corresponding role. Roles inherit all permissions from ancestor roles, so each role r's effective document access is the union of documents directly assigned to r and its ancestors[2].
- *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 use two parameter sets. Tree-α (used in 7.1): $h = 4$, $b_0 = 3$, $b_1 = 4$. Tree-s shares Tree-α parameters but uses Poisson distribution for ϕ_{PA}, adjusting parameters to vary permission selectivity.

ERBAC Generator [21, 23]. Based on Enterprise Role-Based Access Control (ERBAC) model using two-level layered role hierarchy commonly found in real-world organizations. Introduces functional roles (define job functions, hold direct permissions) and business roles (group functional roles, inherit their permissions). Notably, business roles represent 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}).

[2]For 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 in that department and never shared across departments, while office-specific permissions are restricted to employees of that office and not shared with other offices.

- *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 $\{1, \ldots, m_{fr}\}$ and randomly assign $m(r)$ functional roles to r. For each business role, the permission set is the union of all 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 $\{1, 2, \ldots, m_{br}\}$. 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-s, used in 7.5, shares the same basic parameters as ERBAC-α, with the exception of $m_{br} = 1$.

6.3 Baseline Methods and Metrics

We compare Honeybee with four baselines:

- **Role Partition** creates separate partition per role, which is efficient for single-role users but may underperform user partitioning for multi-role users due to the need to aggregate documents from multiple partitions during query execution.
- **User Partition** creates separate partition per unique role combination, so that users with the same roles share one partition.
- **Row-level Security (RLS)** applies PostgreSQL's row-level security feature to filter query results based on permissions after performing similarity search, serving as an example of post-filter methods.
- **HQI** [28] extends QD-tree to build partitions based on frequent predicates observed in a query workload, serving as an example of partition-based methods for hybrid vector search. However, it only generates non-overlaping partitions.

For all methods, we initialize ef_s from the analytical model's estimate and adjust it by up to 30–40% through a small parameter sweep. We then use binary search to fine-tune the value, which typically converges within 1–2 iterations.

The methods are compared using the following metrics:

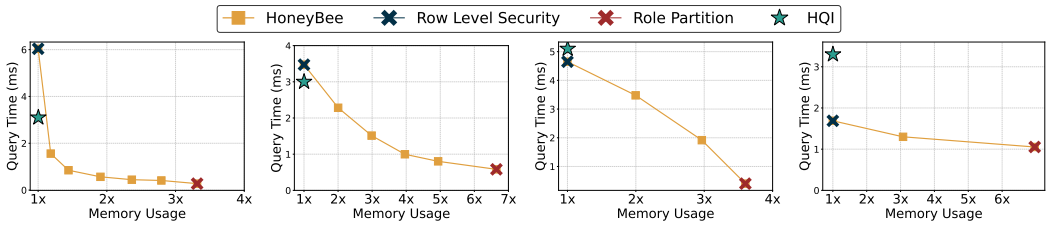
- **Memory**: total index footprint in memory normalized with respect to memory consumption for Row-Level Security (RLS). The index size is calculated using PostgreSQL's pg_indexes_size(), and scales roughly linearly with the number of vectors.
- **Query Latency**: the average execution time per query, measured only on the second run of each query (the first run warms up the cache and is excluded from timing).
- **Recall@k**: the fraction of true top-k nearest neighbors (from ground truth) that appear in the retrieved top-k results, measuring retrieval accuracy.

7 Evaluation

We evaluate Honeybee using the described permission generators and datasets. Our experiments show:

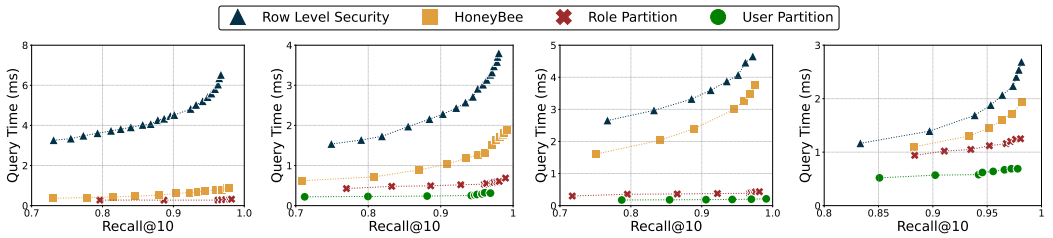
- Honeybee enables efficient trade-off between query latency and memory overhead, achieving up to 6× speedup with 1.4× memory at top-k =10, and up to 13.5× speedup with 1.24× memory at top-k =100 compared to RLS, while achieving comparable query performance to Role Partition with 84% and 90.4% reduction in memory overhead respectively (§ 7.1, § 7.5).
- Honeybee complements hybrid vector search indices such as ACORN, achieving up to 3.5× speedup at 1.2× memory compared to using ACORN alone (§ 7.2).

- HONEYBEE 's partitioning algorithm is robust to small variations in performance model parameters (§ 7.3), and physical partitioning consistently provides a better latency–memory trade-off than logical partitioning (§ 7.4).
- HONEYBEE 's benefits increase with larger top-k values, lower average access selectivity, and hierarchical role structures (§ 7.5), and it can be efficiently maintained under updates to permission workloads (§ 7.6).



(a) Tree - α **(b)** ERBAC - α **(c)** Uniform - α **(d)** ERBAC - β

Fig. 5. Trade-off between Query Time and Memory Usage across Permission Workloads.



(a) Tree - α **(b)** ERBAC - α **(c)** Uniform - α **(d)** ERBAC-β

Fig. 6. Query Time vs Recall under RLS and Different Partitioning Strategies. For HONEYBEE we used the following memory points for the four workloads: (a) 1.4×; (b) 3.0×; (c) 1.9×; (d) 3.2×.

7.1 Trade-off between Memory, Latency, Recall

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

Figure 5 shows the trade-off between memory overhead and average query latency across different permission workloads while maintaining 0.95 recall. RLS (navy cross) offers minimal memory (1×) but high query latency, while Role Partition (red cross) delivers low query latency at significantly higher memory cost. HONEYBEE occupies the intermediate space, producing partitioning strategies ranging from 1 to $|R|$ partitions based on the memory constraint (*e.g.,* HONEYBEE generates 20 partitions for Tree-α with 1.4× memory constraint). Curves closer to the bottom-left corner indicate better latency–memory trade-offs.. Note that HONEYBEE's greedy algorithm *estimates* space needed as total documents in all partitions rather than measuring actual partition sizes, and permits exceeding the limit after the last partitioning step. This may cause minor deviation from the target space factor (α) when measured using actual partition sizes, though the violation remained within 6% in all experiments.

We also compare against HQI [28], another partition-based method that builds separate indices for each partition. It recursively splits data according to query predicates until each partition falls

below a minimum size, which we set to (number of vectors)/(number of roles) to align with Role Partition. Like RLS, HQI produces a single partitioning at 1×-memory, appearing as isolated points in Figure 5. Overall HQI shows unstable performance: it outperforms RLS on some workloads (*e.g.*,, Tree-α: **2.0**× faster, ERBAC-α: **1.09**× faster) but lags on others (*e.g.*,, Uniform-α: **1.1**× slower, ERBAC-β: **1.8**× slower). This behavior is expected because HQI was designed for mostly non-overlapping predicates (*e.g.,* single-valued categorical attributes, numeric range queries), where its splits naturally create disjoint partitions aligned with queries. In contrast, in RBAC workloads, documents can belong to multiple roles, and HQI recursively splits partitions using predicates for different roles. As a result, the documents needed for a single role are spread across multiple small, disjoint partitions, forcing each query to access many partitions and increasing overhead. Without a memory–latency–aware model, these heuristic splits lead to unstable query performance under such many-to-many access patterns.

Figure 6 illustrates the relationship between query recall and latency under fixed memory constraints. We vary the ef_s to tune this trade-off. User Partition (green curve) consistently delivers lowest query latency, followed by Role Partition (red curve), while RLS delivers highest latency (black curve). However, Table 1 shows this performance requires substantial memory costs, with User Partition requiring 74-408× RLS memory, while Role Partition typically requires up to 10× memory. The minimal latency difference between these approaches suggests limited benefits from extending memory beyond Role Partition. Honeybee (yellow curve) produces partition strategies between Role Partition and RLS, showing slower growth in query latency compared to RLS at high recall levels.

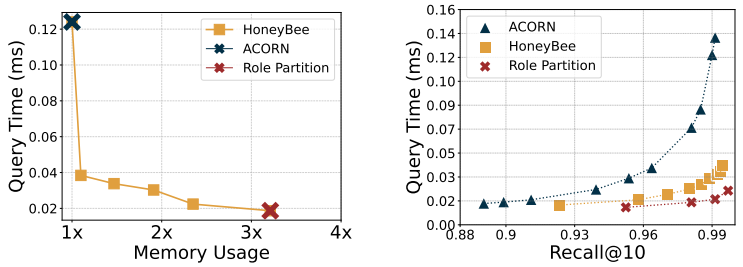
We analyze performance differences across workloads below:

Tree–α. As shown in Figure 5a, Honeybee achieves a 6× latency reduction over RLS at 1.4× memory, approaching Role Partition performance with 84% less memory. Figure 6a further shows that at 1.4× memory, Honeybee significantly outperforms RLS and closely matches Role Partition at 3.5× memory across recall levels. Moreover, Honeybee shows a much slower growth in query performance as recall improves compared to RLS. User User Partition is omitted since it matches Role Partition performance under one-role-per-user mappings. Overall, Honeybee offers a highly effective memory and latency trade-off in this workload.

ERBAC-α. Honeybee achieves a smooth, convex trade-off curve between RLS and Role Partition (Figure 5b), indicating effective trade-off. At approximately 3× memory, query latency exceeds twice RLS speed. Figure 6b shows that User Partition achieves slightly lower latency but at 134× the memory of RLS, compared to 7× for Role Partition, suggesting diminishing returns beyond moderate memory budgets.

ERBAC-β. This workload assigns up to 9 roles per user (versus 3 in ERBAC-α), creating higher user access selectivity where post-filtering approaches like RLS perform well. As a result, RLS performs comparably to Role Partition despite a 6× memory gap (Figure 5d). At 1× memory, Honeybee converges to the RLS configuration, while at 3× memory Honeybee produces solution with query latency between RLS and Role Partition (Figure 6d). User Partition remains the fastest but consumes up to 408× RLS memory.

Uniform. Honeybee shows less benefit trading off memory or query latency versus previous workloads, seen by concave shape on intermediate points in Figure 5c. At 1.9× memory, latency improves only 1.3× over RLS, remaining far from Role Partition performance at around 3.5× memory. Figure 6c shows similar recall–latency growth trends for both Honeybee and RLS. Random role–permission assignments limit the separability of permission subsets, reducing Honeybee 's effectiveness. § 7.5 provides additional analysis on the impact of permission workload structure and access selectivity.



(a) Query Time vs Memory Usage **(b)** Query Time vs Recall

Fig. 7. Performance of HONEYBEE using the ACORN index using the Tree-α workload.

7.2 Evaluation with Hybrid Search Index

To demonstrate HONEYBEE compatibility with hybrid search indices, we replace the default HNSW index with ACORN [29], a state-of-the-art index built on HNSW for hybrid vector search.

HONEYBEE uses an **ACORN-based performance model** to guide partitioning. ACORN provides an analytical expression of search cost: $O\big((d+\gamma)\cdot\log(s\cdot n)+\log(1/s)\big)$, where d is the vector dimension, s is predicate selectivity, n is dataset size, and γ is the neighbor-expansion factor. We use this expression directly as ACORN's search performance model *without any parameter fitting*. Compared to the HNSW-based model, it preserves the logarithmic dependence on n while introducing ACORN-specific terms, especially γ, which controls tradeoff between graph connectivity and filtering efficiency. As a result of ACORN's index design, it can maintain high recall even under low selectivity, so no separate recall model is required.

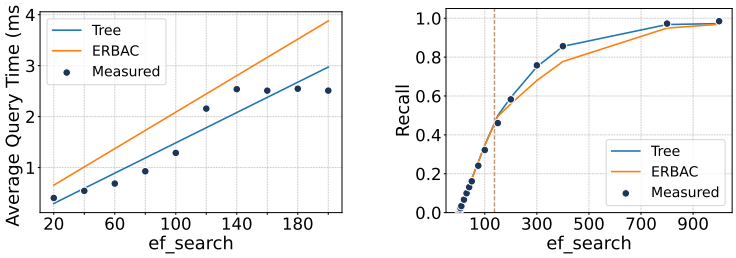
For each partition, we apply ACORN if permission filtering is required and HNSW otherwise (e.g., partition containing a single user). Using Tree-α generator produces low-selectivity settings ($\approx$0.03) where ACORN significantly outperforms post-filtering approaches like RLS. The experiment uses ACORN's implementation on the Faiss library with PostgreSQL. HONEYBEE creates partitions in PostgreSQL. Then, ACORN indexes are built on the document table (the default single table with all documents) and its partitions. We disable multi-threading and caching for fair comparison.

Figure 7a shows at 1.2× memory, HONEYBEE is roughly 3.5× faster than using a single ACORN index on all documents (1× memory). Figure 7b 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 as a partition-based method, HONEYBEE is synergestic with index-based hybrid search methods like ACORN – by strategically replicating vectors across different indices, HONEYBEE can further improve query latency compared to using a single hybrid index on the entire dataset.

7.3 Evaluation of Performance Model

We evaluate the accuracy of HONEYBEE's performance models and their sensitivity to the workload used for fitting. We compare two workloads *Tree-α* (one user–one role) and *ERBAC-α* (one user–many roles). Model parameters are firstly fitted on each workload separately, then validated on *Tree-α* by comparing **estimated** latency and recall with **actual** measurements. As shown in Figures 8a and 8b, parameters trained on *Tree-α* match its measurements closely, while those from *ERBAC-α* show mild deviations.

Second, we evaluate the effect of these parameter differences on HONEYBEE's partitioning algorithm. Running HONEYBEE on *Tree-α* with both parameter sets yields nearly overlapping curves (Figure 9), demonstrating that the partitioning algorithm is robust to slight parameter variations



(a) Query Model **(b)** Recall Model

Fig. 8. Model validation on the Tree workload using performance parameters obtained from Tree (left) and ERBAC (right) models.

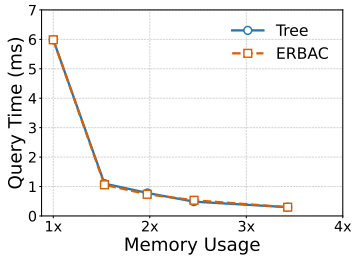


Fig. 9. The effect of performance model on HONEYBEE.

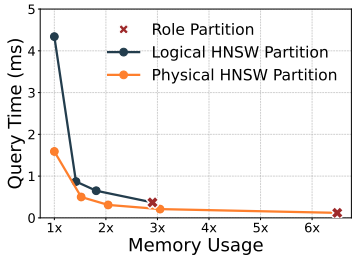


Fig. 10. Comparison of logical vs physical HNSW partition (SIFT10M).

in the performance model. This is because even if the models are not perfectly accurate, they contain enough information to maintain the relative ranking among candidate partitions to guide optimization effectively.

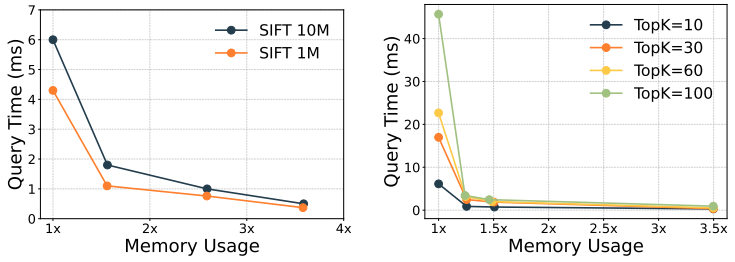
7.4 Physical vs Logical Partition

We compare two HNSW partitioning implementations. In the *physical* version, each partition stores both vectors and graph links. The *logical* version keeps only pointers to a shared vector table, reducing memory usage but introducing pointer indirection that increases query latency.

Logical partitioning still incurs nontrivial memory overhead from graph links and headers, which grows with the number of partitions. Let d be the vector dimension, b_f the number of bytes per scalar (*e.g.*, b_f=4 if vectors and link IDs use float32/int32), and HNSW degree M (each node keeps $\approx 3M$ links in total across levels, since it appears in ≈ 3 layers on average). The logical design uses $b_f\left(|D|d + 3M\sum_{\pi_j\in\Pi}|\pi_j|\right)$ bytes, compared to single-index baseline $b_f(|D|d + 3M|D|)$. Thus the relative overhead is $\frac{|D|d+3M\sum_j|\pi_j|}{|D|d+3M|D|}$.

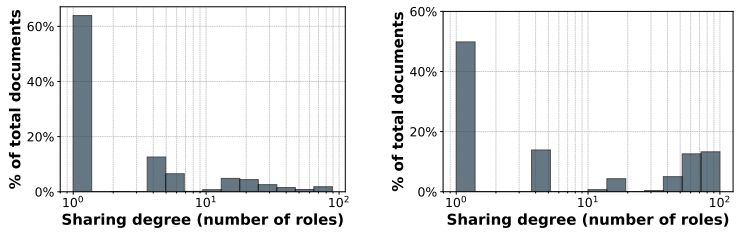
We modified `faiss` to support both implementations and evaluate on the Tree-s workload (0.06 selectivity, SIFT10M dataset) with all optimizations disabled. For this workload, Role Partition with physical partitions uses more than 6× memory, while logical partitions reduce this to around 3× the memory of a single index. As shown in Figure 10, physical partitioning consistently achieves lower latency under the same memory budget, providing a superior latency–memory trade-off. Logical partitioning is roughly 3× slower as each neighbor traversal accesses separate memory regions for graph and vector data, significantly increasing cache misses.

Logical partitioning can be beneficial in limited cases, such as with small datasets (hot caches), very high-dimensional vectors (where vector duplication signifantly increases memory cost), or



(a) Impact of dataset size. **(b)** Impact of topk.

Fig. 11. Sensitivity analysis on Honeybee performance.



(a) Sharing Degree Distribution Pattern (access selectivity 0.06). **(b)** Sharing Degree Distribution Pattern (access selectivity 0.24).

Fig. 12. Comparison of sharing degree distribution pattern in different access selectivity.

extremely large numbers of roles (where one physical partition per role is costly). Even then, Honeybee can optimize logical partitions to balance latency and memory effectively.

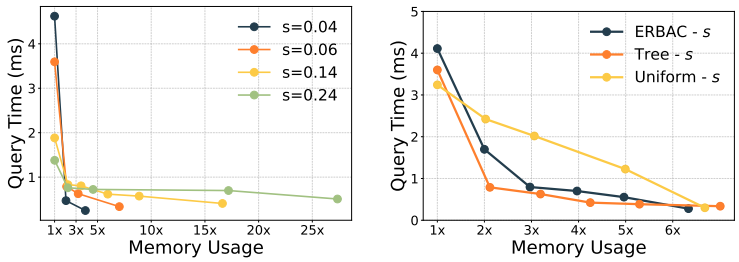
7.5 Sensitivity Analysis

We analyze Honeybee's sensitivity to key parameters: dataset size, permission workload characteristics and top-k values.

Impact of Data Size. We evaluate how dataset size (SIFT1M vs. SIFT10M) affects Honeybee's performance under different memory configurations using the Tree-α workload generator. As shown in Figure 11a, increasing the dataset size degrades overall search performance, as expected, but the shape of the Honeybee trade-off curve remains largely unchanged. This suggests that data size primarily influences absolute latency, but the overall trade-off trend of Honeybee remains stable.

Impact of topk. We evaluated how different topk values (10, 30, 60, 100) affect Honeybee query performance across various memory configurations using Tree-α generator. Figure 11b shows that as k increases, RLS performance degrades significantly, while Role Partition maintains consistent performance. This sharp degradation occurs because RLS requires dramatically larger candidate sets (ef_s > 3000 for k=100) to maintain recall as topk increases. Honeybee demonstrates a middle ground, with performance slowing as topk increases but at a much lower rate than RLS.

Importantly, the performance gap between RLS and Honeybee widens as topk increases, indicating Honeybee provides greater benefits for larger result sets. At higher memory constraints, Honeybee's performance degradation becomes less pronounced. For instance, with topk=100, Honeybee achieves approximately 13.5x faster query performance with only 1.24x memory compared to RLS and 90.4% reduction in additional memory consumption compared to Role Partition (0.24× vs 2.5× additional), demonstrating excellent efficiency for high-topk retrieval scenarios.



(a) Effect of access selectivity **(b)** Effect of sharing degree pattern

Fig. 13. The effect of access selectivity and sharing degree distribution on HONEYBEE performance.

Impact of Access Selectivity. We use the Tree-s generator to examine sharing degree patterns at different access selectivity levels (0.04, 0.06, 0.14, 0.24). Figure 12 shows that while both selectivity levels exhibit similar patterns—sharing degree 1 has the highest peak with decreasing peaks for higher degrees—the magnitude differs: selectivity 0.06 (left) shows higher peaks than 0.24 (right), demonstrating consistent patterns with varying intensity.

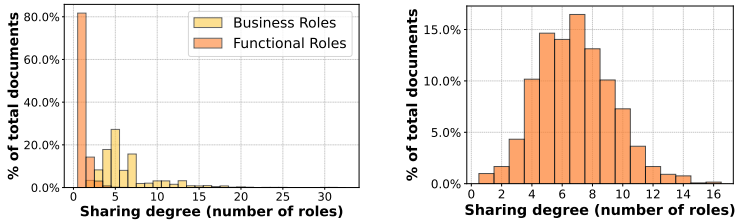
Figure 13a shows that at higher selectivity levels, RLS becomes more efficient, so the query latency at 1× memory approaches the performance of Role Partition (*i.e.*, $\Delta Latency$ is small). Additionally, Role Partition consumes more memory as selectivity increases ($\Delta Memory$). Therefore, HONEYBEE has the most optimization potential at low selectivity, where the gap between RLS and Role Partition ($\Delta Latency/\Delta Memory$) is largest. For similar sharing degree distribution patterns, HONEYBEE's performance curves show similar trends at around 2× memory. We observe that the rate of improvement in query latency decreases the fastest at this point, whereas further increases in memory yield diminishing returns.

Impact of Sharing Degree Pattern. We evaluate HONEYBEE performance under workloads with different sharing degree patterns at similar selectivity (approximately 0.06 across cases). Sharing degree distribution indicates document percentage accessible to varying role numbers (e.g., 50% documents accessible to 1 role each, 20% to 2 roles, etc.). Three distinct patterns are generated ensuring similar selectivity. Tree pattern (Figure 12a) peaks at small sharing degrees, indicating hierarchical structure where documents are primarily accessed by few roles. ERBAC pattern (Figure 14a) indicates two-layer hierarchical structure with functional and business roles. Uniform pattern (Figure 14b), generated by the Uniform Generator, follows Poisson distribution with average sharing degree 7. Figure 13b demonstrates query latency and memory consumption across patterns are nearly identical at 1× memory (RLS), expected given workloads have identical average selectivity.

Overall, HONEYBEE enables different memory-query latency trade-offs, with performance ranking Tree > ERBAC > Uniform (evident from trade-off curve convexity). This demonstrates permission workload structure directly impacts HONEYBEE performance. Comparing Figure 12a and Figure 14a, functional roles follow Tree pattern, but ERBAC performs worse due to the added complexity of business roles. Similarly, Figure 14a vs. Figure 14b demonstrates Uniform, lacking hierarchical structure, performs worse, particularly as its peak shifts from lower sharing degrees, making permission subset identification for partitioning more challenging.

7.6 Adapting to Evolving Permission Workloads

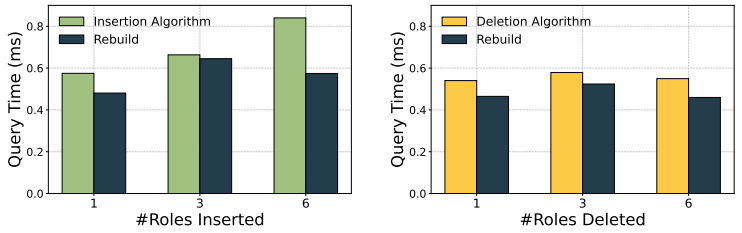
This section evaluates how HONEYBEE's partition design adapts to changes in permission workloads.



(a) Sharing degree distribution pattern (ERBAC pattern).

(b) Sharing degree distribution pattern (Uniform pattern).

Fig. 14. Comparison of sharing degree distribution pattern in different permission workloads.



(a) Insertion time benchmark.

(b) Deletion time benchmark.

Fig. 15. Query latency comparison: incremental partition maintenance vs. full rebuild under permission workload changes.

We first evaluate how efficiently Honeybee maintains its partitioning strategy during permission workload changes, specifically role insertions and deletions. We use Tree-α generator with Honeybee at 1.5× memory. Insertions create new roles from existing document subsets, adding users equal to 1% of the original user base. Deletions remove roles and their exclusive users. We use one-to-one user-role mapping and compare incremental updates to full rebuilds across 1, 3, or 6 operations.

Figure 15a shows that for few insertions (<5), incremental updates perform similarly to rebuilds. However, as insertions increase, incremental partitioning shows higher query latency, suggesting Honeybee benefits from rebuilding when workloads change significantly. Figure 15b shows incremental deletion consistently maintains query latency comparable to complete rebuilds. Role modifications naturally change total memory footprint for all methods. Based on our observation, while absolute memory consumption changes, the relative memory overhead ratios among RLS, Honeybee, and Role Partition remain consistent with those reported in Section 7.1.

8 Related Work

Partitioning in Databases. Traditional horizontal partitioning methods distribute data by key attributes to enable partition pruning, parallel processing, and load balancing [1, 4–6, 8, 46]. For example, QD-tree and followup works [9, 31, 36, 44] optimizes data layouts by partitioning based on query patterns to minimize the number blocks accessed. However, these approaches target traditional query patterns–exact-match or range predicates–rather than similarity search with permission filtering, where vector distance calculations combined with permission checks create unique performance characteristics. HQI [28] extends QD-trees to similarity search queries. It leverages query-workload predicates to build balanced QD-trees and construct separate IVF indexes

Table 2. Overview of related work in hybrid vector search.

	Method type	Index agnostic?	Memory constraint?
ACORN [29]	index	× (HNSW)	×
Filtered-DiskANN [16]	index	× (Vamana)	×
Curator [20]	index	× (IVF)	×
HQI [28]	partition	✓	×
HONEYBEE (ours)	partition	✓	✓

per partition, batching queries by shared attribute predicates. HQI can be considered as a special case where memory overhead is 1×, while HONEYBEE optimizes partitioning for any memory constraint ($\geq 1\times$) through overlapping partitions. As shown in Figure 5, HQI's performance is highly unpredictable: it performs well in certain case (e.g., Tree-α) but degrades sharply in others (e.g., Uniform or ERBAC). This inconsistency arises because HQI relies purely on heuristic partitioning without any performance-guided model. To the best of our knowledge, HONEYBEE is the first system to systematically explore strategic vector replication across partitions achieving efficient memory-performance tradeoffs in vector databases, formulating this as constrained optimization with explicit memory constraints.

Hybrid Search Indexes. Hybrid search indexes integrate structured filtering into vector similarity search for efficient constraint enforcement like access control [16, 29, 39, 42, 43]. ACORN [29] provides performant, predicate-agnostic hybrid search handling high-cardinality and unbounded predicate sets. It improves HNSW by integrating filtering capabilities directly into the graph structure while remaining compatible with existing HNSW implementations. HQANN [42] provides hybrid query processing framework embedded into proximity graph-based ANN algorithms. It manages hybrid queries through unified processing, enhancing performance and adaptability. Filtered-DiskANN [16] constructs graph-structured indexes leveraging vector geometry and label metadata (e.g., date, price). This approach incorporates geometric relationships and associated labels for effective navigational graph structure. Our partitioning design is orthogonal to index type. As described in § 7.2, it complements hybrid search by addressing partitioning and access policy management challenges, enabling integration of hybrid search techniques like ACORN with our method.

Multi-Tenant Vector Databases. Curator [20] represents the most relevant prior work in vector database access control for multi-tenant scenarios. Multi-tenant database implementations generally follow three patterns [17], with different trade-offs in data isolation and resource usage: (1) isolated db instances per tenant, (2) per-tenant tables within a shared db instance (*i.e.*, the specialized index approach), and (3) shared tables storing all tenant data with tenant identification columns (*i.e.*, the post-filtering approach). At a high level, Curator introduces a multi-tenant index to improve approach (3), while HONEYBEE helps users navigate the trade-off between approaches (2) and (3) for a given index.

Specifically, Curator 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. Curator differs from HONEYBEE in two key aspects: (1) Curator uses direct user-permission mappings (ACLs), while HONEYBEE leverages RBAC hierarchies for scalable partitioning; (2) Curator's ACL logic is tightly coupled with its custom k-means tree design, limiting reuse of standard ANN indexes, while HONEYBEE separates partition design from index implementation, enabling seamless integration with HNSW, IVF-Flat, and hybrid techniques. Similar to ACORN [29], Curator could

integrate Honeybee's dynamic partitioning within its k-means clustering tree to gain further performance improvement.

9 Conclusion

We presented Honeybee, a dynamic partitioning framework for access control in vector databases. Honeybee is the first to systematically explore strategic vector replication across partitions to achieve efficient memory–performance tradeoffs, formulating partitioning as a constrained optimization problem with explicit memory limits. Leveraging RBAC structure, Honeybee partitions the vector space to minimize query latency, guided by analytical models predicting search performance and recall. Experiments show up to 13.5× faster queries than row-level security with only 1.24× more memory, and 90.4% less memory than role partitioning at comparable speed. Honeybee offers a practical, scalable solution for enforcing access control in vector databases.

Acknowledgments

This work was supported in part by the National Science Foundation under grant IIS-2335881.

References

[1] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating vertical and horizontal partitioning into automated physical database design. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 359–370.

[2] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[3] 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.

[4] Ladjel Bellatreche, Kamalakar Karlapalem, and Ana Simonet. 2000. Algorithms and support for horizontal class partitioning in object-oriented databases. *Distributed and Parallel Databases* 8, 2 (2000), 155–179.

[5] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. 1982. Horizontal data partitioning in database design. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 128–136.

[6] Krzysztof Cpałka, Marcin Zalasiński, and Leszek Rutkowski. 2014. New method for the on-line signature verification based on horizontal partitioning. *Pattern Recognition* 47, 8 (2014), 2652–2661.

[7] Satori Cyber. n.d.. Postgres Row-Level Security: Comprehensive Guide. https://satoricyber.com/postgres-security/postgres-row-level-security/. Accessed: January 5, 2025.

[8] Aleksandar Dimovski, Goran Velinov, and Dragan Sahpaski. 2010. Horizontal partitioning by predicate abstraction and its application to data warehouse design. In *East European Conference on Advances in Databases and Information Systems*. 164–175.

[9] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 418–431.

[10] 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 International ACM SIGIR Conference on Research and Development in Information Retrieval*. 25–33.

[11] Hugging Face. 2022. Wikipedia 22-12 Dataset. https://huggingface.co/datasets/Cohere/wikipedia-22-12. Accessed: January 2025.

[12] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. 2006. High dimensional nearest neighbor searching. *Information Systems* 31, 6 (2006), 512–540.

[13] 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).

[14] 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 (VLDB)* 12, 5 (2019), 461–474.

[15] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Proceedings of the VLDB Endowment (VLDB)*, Vol. 99. 518–529.

[16] 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 Web Conference (WWW)*. 3406–3416.

[17] Mei Hui, Dawei Jiang, Guoliang Li, and Yuan Zhou. 2009. Supporting database applications as a service. In *IEEE International Conference on Data Engineering*. 832–843.
[18] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *ACM Symposium on Theory of Computing (STOC)*. 604–613.
[19] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2010), 117–128.
[20] 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).
[21] Axel Kern, Andreas Schaad, and Jonathan Moffett. 2003. An administration concept for the enterprise role-based access control model. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*. 3–11.
[22] Conglong Li, Minjia Zhang, David G Andersen, and Yuxiong He. 2020. Improving approximate nearest neighbor search through learned adaptive early termination. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2539–2554.
[23] Ninghui Li, Tiancheng Li, Ian Molloy, Qihua Wang, Elisa Bertino, Seraphin Calo, and Jorge Lobo. 2010. Role Mining for Engineering and Optimizing Role-Based Access Control Systems. *ACM Transactions on Information and System Security (TISSEC)* 13, 4 (2010), 1–29.
[24] LlamaIndex. 2025. Q&A over Documents - LlamaIndex 0.8.43. https://gpt-index.readthedocs.io/en/latest/. Accessed: July 2025.
[25] 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 (TPAMI)* 40, 9 (2018), 2227–2240.
[26] Marqo. 2024. Understanding Recall in HNSW Search. https://www.marqo.ai/blog/understanding-recall-in-hnsw-search. Accessed: October 2025.
[27] 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.
[28] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-throughput vector similarity search in knowledge graphs. *ACM SIGMOD International Conference on Management of Data (SIGMOD)* 1, 2 (2023), 1–25.
[29] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *ACM SIGMOD International Conference on Management of Data (SIGMOD)* 2, 3 (2024), 1–27.
[30] 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.
[31] Kexin Rong, Paul Liu, Sarah Ashok Sonje, and Moses Charikar. 2024. Dynamic Data Layout Optimization with Worst-case Guarantees. 4288–4301.
[32] Ravi S Sandhu. 1998. Role-based access control. In *Advances in computers*. Vol. 46. 237–286.
[33] 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.
[34] 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 *International Conference on Multimedia Information Processing and Retrieval (MIPR)*. 155–161.
[35] Ryan Shelly. 2022. Automatic Clustering at Snowflake. https://medium.com/snowflake/automatic-clustering-at-snowflake-317e0bb45541. Accessed October 2025.
[36] Sivaprasad Sudhir, Wenbo Tao, Nikolay Laptev, Cyrille Habis, Michael Cafarella, and Samuel Madden. 2023. Pando: Enhanced Data Skipping with Logical Data Partitioning. *Proceedings of the VLDB Endowment (VLDB)* 16, 9 (2023), 2316–2329.
[37] Supabase. 2025. RAG with Permissions | Supabase Docs. https://supabase.com/docs/guides/ai/rag-with-permissions Accessed: March 1, 2025.
[38] Jaideep Vaidya, Vijayalakshmi Atluri, and Janice Warner. 2006. RoleMiner: mining roles using subset enumeration. In *ACM Conference on Computer and Communications Security (CCS)*.
[39] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2024. An efficient and robust framework for approximate nearest neighbor search with attribute constraint. In *Advances in Neural Information Processing Systems (NeurIPS)*.
[40] Qitong Wang, Ioana Ileana, and Themis Palpanas. 2025. LeaFi: Data Series Indexes on Steroids with Learned Filters. *ACM SIGMOD International Conference on Management of Data (SIGMOD)* 3, 1 (2025), 1–27.
[41] Brie Wolfson. 2023. Building chat langchain. https://blog.langchain.dev/building-chat-langchain-2/

[42] 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 and Knowledge Management*. 4580–4584.
[43] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. *ACM SIGMOD International Conference on Management of Data (SIGMOD)* 2, 6 (2024).
[44] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 193–208.
[45] Hongbin Zhong, Matthew Lentz, Nina Narodytska, Adriana Szekeres, and Kexin Rong. 2025. HONEYBEE: Efficient Role-based Access Control for Vector Databases via Dynamic Partitioning (Technical Report). *arXiv preprint arXiv:2505.01538* (2025).
[46] Daniel C Zilio. 1998. *Physical database design decision algorithms and concurrent reorganization for parallel database systems*. Ph.D. Dissertation.

Received July 2025; revised October 2025; accepted November 2025