Database-Native Authorization for Human and Autonomous Principals: The SHRBAC Model

Anonymous Author(s)
Removed for double-blind review

***Abstract*—Enterprise applications must answer two authorization questions: "can this principal act on this resource?" (point check) and "which resources can this principal access, filtered, sorted, and paginated?" (list filtering). Point checks are boolean decisions with well-understood evaluation strategies. List filtering is fundamentally harder: it requires composing authorization decisions with application-defined filtering, sorting, and pagination into a single efficient query—and when authorization logic resides outside the database, an impedance mismatch arises. We present SHRBAC (Scoped Hierarchical Role-Based Access Control), a formal authorization model whose structural constraints—tree-structured resources, flat roles, polymorphic principals including autonomous agents—guarantee that enforcement is a composable database predicate with bounded cost—concretely, a parameterized inline Table-Valued Function (TVF) that the optimizer folds into the execution plan. The model enforces the principle of least privilege through grants scoped by resource subtree, role, and time window, enabling both human users and AI agents to operate with minimum necessary authority. We formalize the model, prove soundness and completeness of the TVF enforcement, and show that under cursor pagination, per-page cost is $O(k \cdot D)$—predictable, linear, and independent of total dataset size. Empirical evaluation on SQL Server 2022 with 1.2M resource nodes (D=5) and 1.5M resource nodes (D=10) confirms N-independence across three orders of magnitude.**

***Index Terms*—access control, RBAC, authorization, query composition, table-valued function, resource hierarchy, agentic AI**

I. Introduction

A. The Problem

Modern B2B SaaS applications require authorization systems that answer two fundamentally different questions:

1) **Point check**: "Can principal p perform action a on resource r?"—a boolean decision.
2) **List filtering**: "What resources of type T can principal p access, given search criteria C, sorted by S, paginated to page k?"—a set-returning query integrated with application data.

The first question is well-understood. Google's Zanzibar [1] and its descendants handle millions of point checks per second through graph traversal and caching.

The second question—the **list filtering problem**—is well-solved for flat access models. A single-tenant predicate (`WHERE tenant_id = @currentTenant`) composes trivially. However, when access is determined by grants at varying levels of a resource hierarchy, resolved through transitive group memberships, and subject to temporal constraints, list filtering becomes substantially more complex.

Increasingly, principals are not only human users but **autonomous AI agents** that issue queries and perform actions over enterprise data. These agents operate in loops, enumerating and filtering large resource sets programmatically. For agentic systems, efficient list filtering is not merely a user-experience optimization—it is a correctness and cost requirement. An agent that makes N external authorization calls per page degrades linearly; an agent whose authorization is a composable database predicate operates at constant cost per page.

Beyond efficient querying, the authorization model must enforce the **principle of least privilege** [2]: principals, whether human or autonomous, should operate with minimum necessary authority scoped to specific resources and time windows. For AI agents, this is not merely good practice but a safety requirement—an agent should receive a narrow grant (specific subtree, limited role, bounded duration) rather than broad access.

When authorization logic resides outside the database, an impedance mismatch arises. Industry bridges this gap through ID lookups, batch post-filtering, partial policy evaluation, or materialized permission views—each trading scalability, consistency, or complexity. Some of these approaches are SQL-implementable, but they require runtime predicate construction that changes shape as the policy set evolves.

SHRBAC takes a different approach: the model's structural constraints are chosen so that per-page enforcement cost is $O(k \cdot D)$—linear in page size and tree depth, independent of total resource count N and policy set size (Fig. 1).

B. Contributions

1) **Constraint-driven model with composability guarantee.** We introduce SHRBAC, whose four structural constraints—tree-structured resources, non-recursive principal groups, flat roles, and scoped grants (principal $\times$ role $\times$ resource $\times$ time)—are deliberately chosen so that enforcement admits a fixed, schema-level relational artifact with bounded cost independent of policy set size.
2) **Formalization with two-dimensional resolution.** Access evaluation performs upward traversal over a bounded-depth resource tree and outward expansion over effective principals (humans, groups, and agents). We prove soundness and completeness and bound per-row cost at $O(D \cdot M \cdot G_{\max})$. Unlike prior hierarchical RBAC models, the cost structure is part of the model definition.

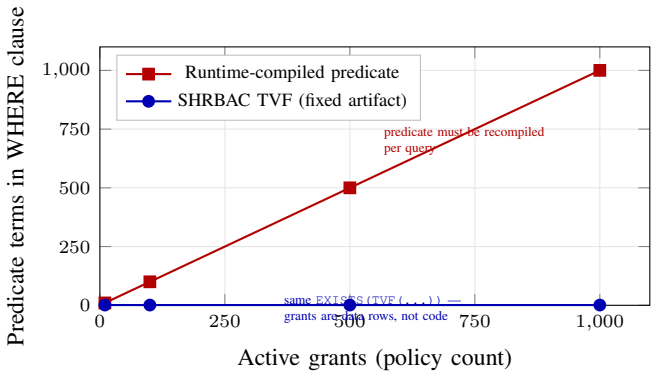

Fig. 1. Enforcement artifact complexity as the policy set grows. Runtime-compiled approaches emit a WHERE clause whose size is proportional to the number of active grants. SHRBAC's TVF is a single, fixed predicate defined at schema time—grants are rows in a table, not terms in a predicate. The enforcement artifact never changes.

3) **Fixed inline TVF enforcement.** In the lineage of Stonebraker [4], we realize enforcement as a parameterized inline Table-Valued Function—a single `EXISTS` predicate defined at schema time whose shape is constant regardless of policy size. Grants are data rows, not predicate terms, requiring no runtime compilation, post-filtering, or external graph traversal.
4) **Complexity theorem for list filtering.** Under cursor pagination, per-page cost is $O(k \cdot D)$ when M and $G_{\max}$ are bounded—independent of resource count N, policy set size, and page depth. This formal bound for the list-filtering problem is absent in prior RBAC, ABAC, and ReBAC models.
5) **Production-scale empirical validation.** At 1.2M resources (D=5) and 1.5M resources (D=10), we confirm N-independence across three orders of magnitude, linear scaling in k and D, constant per-CTE-hop cost ($\sim$0.029ms), and grant density independence.
6) **Polymorphic principal model for agent governance.** SHRBAC includes autonomous agents as first-class grant recipients with least-privilege, time-bounded authority. Agent-driven list queries inherit the $O(k \cdot D)$ guarantee, providing database-native agent governance without external authorization infrastructure.

C. Paper Organization

Section II surveys related work. Section III formalizes the model. Section IV defines the access evaluation algorithm. Section V describes TVF enforcement. Section VI analyzes complexity. Section VII presents empirical evaluation. Section VIII discusses design constraints, agentic applications, and limitations. Section IX concludes.

II. Background and Related Work

A. RBAC Foundations

Sandhu et al. [5] defined the RBAC96 family: RBAC0 (core), RBAC1 (hierarchical roles), RBAC2 (constraints), and RBAC3 (combined). The NIST standard [6] formalized Core and Hierarchical RBAC. **Critical gap:** RBAC96 defines only a role hierarchy; the resource space is flat with no concept of resource hierarchy or scope.

B. Hierarchical Models

Three models extend RBAC with organizational or resource structure:

ROBAC (Zhang et al. [3]) introduced the (user, role, organization) triple with an organization hierarchy where grants cascade from parent to child organizations. SHRBAC's (principal, role, resource) grant is structurally identical, but ROBAC defines a policy semantics without constraining the model to admit a fixed, schema-level enforcement artifact with bounded evaluation cost. SHRBAC's novelty is not the hierarchy alone, but the alignment of model constraints with relational query planning guarantees—extending the subject to polymorphic principals and providing a concrete enforcement mechanism whose per-row cost is analyzable.

RRBAC (Solanki et al. [7]) explicitly formalized resource hierarchies with grant cascade semantics, but did not address polymorphic principals, query composition, or enforcement mechanisms.

OrBAC (Kalam et al. [8]) generalized RBAC with five hierarchies (organizations, roles, activities, views, contexts). SHRBAC is a practical engineering subset—OrBAC's expressiveness exceeds what TVF-based enforcement requires.

C. Query Modification and Enforcement

Stonebraker and Wong [4] introduced query modification for INGRES, transparently rewriting queries to enforce access control. This is the ancestor of Oracle VPD, SQL Server RLS, and PostgreSQL RLS. Rizvi et al. [9] distinguished the *Truman model* (silent filtering) from the *Non-Truman model* (query rejection). SHRBAC implements the Truman model. Pappachan et al. [10] showed that naive predicate injection does not scale with thousands of policies; SHRBAC avoids this because the predicate is always a single TVF invocation regardless of policy count.

D. Zanzibar and ReBAC

Google's Zanzibar [1] uses relationship tuples with a userset rewrite system. Fong [11] formalized ReBAC using modal logic. Zanzibar's `tuple_to_userset` handles resource hierarchies through general graph traversal. SHRBAC constrains resources to a tree (not an arbitrary graph) and uses typed roles rather than arbitrary relation composition—this constraint is what makes TVF enforcement tractable, as tree traversal has bounded depth while general graph traversal does not.

E. The List Filtering Problem

Despite its practical importance, composing authorization into list queries has no canonical academic name. AuthZed calls it "ACL-aware filtering" [12], Oso calls it "list filtering" [13], Cerbos frames it as a "query plan" problem [14]. SpiceDB's documentation reveals the architectural tension: at

scale, they recommend progressively more complex strategies (LookupResources → CheckBulkPermissions → Materialize). SHRBAC is designed so that the integration surface is a relational predicate—composable by construction.

F. Summary of Gaps

Table I summarizes the capability comparison. To our knowledge, no prior model jointly formalizes resource hierarchy with grant cascade, polymorphic principals (humans, groups, and agents as first-class grant recipients), and a concrete query-composable enforcement mechanism with a complexity bound.

III. The SHRBAC Model

A. Basic Sets

- $\mathcal{P}$ — a finite set of *principals*. Each principal has a type $\tau(p) \in \{\text{user}, \text{group}, \text{service\_account}, \text{agent}\}$. An agent principal represents an autonomous process—an AI agent, orchestration pipeline, or background service—that issues queries and mutations on behalf of an organization. Agents participate in the same grant relation as human users, subject to the same scoping and temporal constraints.
- $\mathcal{R}$ — a finite set of *roles*.
- $PERM$ — a finite set of *permissions*. Each permission is an atomic capability (e.g., PROJECT_VIEW, SUBCONTRACTOR_EDIT).
- RES — a finite set of *resources*, forming nodes in a rooted tree.
- RT — a finite set of *resource types* (e.g., `portal_root`, `agency`, `project`). Each resource has a type: $type : RES \to RT$.
- $\mathcal{T}$ — the time domain (UTC timestamps).

B. Resource Hierarchy

The resource hierarchy is a rooted tree $(RES, parent)$ where:

- $parent : RES \to RES \cup \{\bot\}$ maps each resource to its parent. The root r_0 has $parent(r_0) = \bot$.
- $ancestors(r) = \{parent^i(r) \mid i \geq 0 \wedge parent^i(r) \neq \bot\}$. Note that $r \in ancestors(r)$.
- $descendants(r) = \{r' \in RES \mid r \in ancestors(r')\}$
- $depth(r) = |ancestors(r)| - 1$ (root has depth 0).

The tree is bounded: $depth(r) \leq D$ for all r. In practice, $D = 3$–5 for typical SaaS and up to 10–15 for deep enterprise hierarchies.

An **authorized entity** is a row in an application table carrying a `ResourceId` referencing a node in the resource tree.

C. Role-Permission Assignment

$PA \subseteq \mathcal{R} \times PERM$ is the role-permission assignment. Each permission has an associated resource type: $applicable : PERM \to RT$. Roles are **flat**—no role hierarchy. The resource hierarchy provides the inheritance dimension, avoiding the combinatorial complexity of dual hierarchies.

D. Principal Resolution

Groups are themselves principals. $members(g) = \{u \in \mathcal{P} \mid \tau(u) = \text{user} \wedge u \text{ is a member of } g\}$. Membership is not recursive—groups cannot contain other groups.

Principal resolution expands a user to their effective principal identities:

$$resolve(p) = \{p\} \cup \{g \in \mathcal{P} \mid \tau(g) = \text{group} \wedge p \in members(g)\}$$

For non-user principals (groups, service accounts, agents), $resolve(p) = \{p\}$.

E. Grants

$$G \subseteq \mathcal{P} \times \mathcal{R} \times RES \times (\mathcal{T} \cup \{\bot\}) \times (\mathcal{T} \cup \{\bot\})$$

A grant $g = (p, role, res, t_{from}, t_{to})$ assigns role $role$ to principal p at resource res, effective during the closed interval $[t_{from}, t_{to}]$. Both bounds are inclusive. If $t_{from} = \bot$, effective from the beginning of time; if $t_{to} = \bot$, effective indefinitely.

Active grants at time t:

$$active(t) = \{(p, role, res) \mid (p, role, res, t_f, t_t) \in G \wedge (t_f = \bot \vee t_f \leq t) \wedge (t_t = \bot \vee t_t \geq t)\}$$

The grant triple (principal, role, resource) enforces least privilege: authority is scoped to a specific subtree (not global), a specific role (not all permissions), and optionally a specific time window (not permanent).

F. Access Evaluation Function

Definition 1 (Access Decision)**.** *Given principal p, permission $perm$, resource r, and time t:*

$$allowed(p, perm, r, t) = \exists\, p' \in resolve(p), \exists\, r' \in ancestors(r), \exists\, role \in \mathcal{R} : (p', role, r') \in active(t) \wedge perm \in perms(role)$$

Access is granted if *any* effective identity has *any* active grant at *any* ancestor of the target resource with a role including the requested permission. This is **two-dimensional resolution**: simultaneously walking UP the resource tree and expanding OUT the principal set.

Running example. Consider a resource tree: `portal_root` → `agency_7` → `project_42`, with user Alice who belongs to the `engineering` group. A grant exists: (`engineering`, VIEWER, `agency_7`). To evaluate $allowed(\text{alice}, \text{PROJECT\_VIEW}, \texttt{project\_42}, now)$: $resolve(\text{alice}) = \{\text{alice}, \text{engineering}\}$; $ancestors(\texttt{project\_42}) = \{\texttt{project\_42}, \texttt{agency\_7}, \texttt{portal\_root}\}$. The algorithm searches the 2×3 grid of (principal, ancestor) pairs. At (`engineering`, `agency_7`), it finds the VIEWER grant, which includes PROJECT_VIEW. Access is granted—the grant at `agency_7` cascades to its descendant `project_42`.

TABLE I
CAPABILITY COMPARISON ACROSS AUTHORIZATION MODELS. "✓" INDICATES NATIVE SUPPORT.

Capability	RBAC96	ROBAC	RRBAC	OrBAC	ReBAC	SHRBAC
Role hierarchy	RBAC1+	✓	✓	✓	N/A	No (flat)
Resource hierarchy	No	Org hier.	✓	Org hier.	Graph	✓ (tree)
Grant cascade	No	✓	✓	✓	`tuple_to_userset`	✓
Polymorphic principals	No	No	No	Partial	Usersets	✓
Group-as-principal	No	No	No	No	Via tuples	✓
Agent-as-principal	No	No	No	No	No	✓
Temporal grants	No	No	No	Contexts	No	✓
Query composition	No	No	No	No	No	✓
Formal enforcement	No	No	No	No	Graph trav.	Query mod.

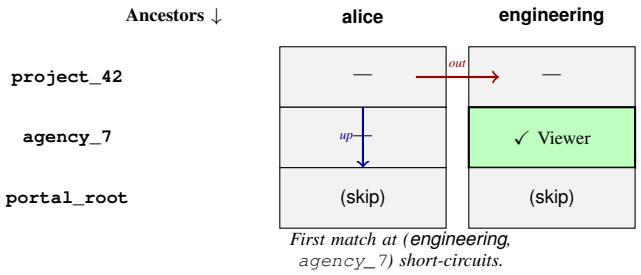

Fig. 2. Two-dimensional resolution for $allowed(\text{alice}, \text{PROJECT\_VIEW}, \texttt{project\_42}, now)$. The principal set is expanded outward (columns) while the ancestor chain is walked upward (rows). The first matching grant short-circuits evaluation.

G. Properties

Property 1 (Monotonicity of Hierarchy)**.** *If* $allowed(p, perm, r, t)$ *and* $r' \in descendants(r)$*, then the same grant provides access at* r'*.*

Proof. If $(p', role, r_{anc}) \in active(t)$ and $r_{anc} \in ancestors(r)$, then since $r' \in descendants(r)$, we have $r \in ancestors(r')$, and by transitivity $r_{anc} \in ancestors(r')$. Therefore the same grant satisfies the access decision for r'. □ □

Property 2 (Monotonicity of Groups)**.** *Adding* p *to group* g *preserves existing access and may grant additional access.*

Proof. Adding p to group g expands $resolve(p)$ to $resolve(p) \cup \{g\}$. Since the access decision uses existential quantification over $resolve(p)$, all previously satisfying assignments remain valid. Any grant held by g additionally becomes effective for p. □ □

Property 3 (Bounded Evaluation)**.** *For any access decision, evaluation examines at most* $|resolve(p)| \times |ancestors(r)| \times G_{\max}$ *grant entries, giving* $O(M \cdot D \cdot G_{\max})$*.*

IV. THE ACCESS EVALUATION ALGORITHM

A. Point Check

```
ALGORITHM PointCheck(p, perm, r, t):
  principals <- resolve(p)
  ancestors <- getAncestors(r)
  FOR EACH r' IN ancestors:
    FOR EACH p' IN principals:
      grants <- getActiveGrants(p', r', t)
      FOR EACH g IN grants:
        IF perm in perms(g.role): RETURN allowed
  RETURN denied
```

B. List Filter

For list filtering, the algorithm is expressed as a SQL predicate:

```
ALGORITHM ListFilter(p, perm, query, t):
  principals <- resolve(p)
  principalIds <- join(principals, ',')
  permId <- lookupPermissionId(perm)
  now <- UTC_NOW()
  RETURN query.WHERE(entity =>
    EXISTS(fn_IsResourceAccessible(
      entity.ResourceId, principalIds,
      permId, now)))
```

The TVF is evaluated per-row by the database engine, composed with user-defined filters, sorting, and pagination into a single execution plan.

V. QUERY-COMPOSABLE ENFORCEMENT VIA TVF

A. The Table-Valued Function

The enforcement mechanism is an inline Table-Valued Function (iTVF):

```
CREATE FUNCTION dbo.fn_IsResourceAccessible(
    @ResourceId NVARCHAR(128),
    @PrincipalIds NVARCHAR(MAX),
    @PermissionId NVARCHAR(128),
    @Now DATETIME2
)
RETURNS TABLE
AS
RETURN
(
    WITH ancestors AS (
        SELECT Id, ParentId, 0 AS Depth
        FROM Resources WHERE Id = @ResourceId
        UNION ALL
        SELECT r.Id, r.ParentId, a.Depth + 1
        FROM Resources r
        INNER JOIN ancestors a ON r.Id = a.ParentId
        WHERE a.Depth < 10
    )
    SELECT TOP 1 a.Id
    FROM ancestors a
    INNER JOIN Grants g ON a.Id = g.ResourceId
    INNER JOIN RolePermissions rp
      ON g.RoleId = rp.RoleId
    WHERE g.PrincipalId IN (
        SELECT LTRIM(RTRIM(value))
        FROM STRING_SPLIT(@PrincipalIds, ',')
    )
      AND rp.PermissionId = @PermissionId
      AND (g.EffectiveFrom IS NULL
           OR g.EffectiveFrom <= @Now)
      AND (g.EffectiveTo IS NULL
           OR g.EffectiveTo >= @Now)
```

```
)
```

The recursive CTE walks UP from the target resource to the root (at most D levels), joining against active grants at each ancestor. `TOP 1` provides existential semantics—one matching grant suffices. The caller-controlled `@Now` parameter ensures deterministic evaluation across all rows in a query. As an inline TVF, the optimizer composes its body into the calling query's execution plan, enabling authorization, user filtering, sorting, and pagination in a single statement. We use SQL Server as an exemplar because its optimizer aggressively inlines iTVFs; however, the enforcement requires only recursive CTEs and predicate inlining, both supported by PostgreSQL and other modern engines.

B. Correctness

Theorem 1 (Soundness)**.** *If the TVF returns a non-empty result for entity e, then $allowed(p, perm, e.ResourceId, now) = true$.*

Proof. The recursive CTE produces exactly $ancestors(e.ResourceId)$: the base case selects the resource itself, and each recursive step follows *parent* pointers until the root. The JOIN against `Grants` on `ResourceId` and `PrincipalId ∈ resolve(p)` implements the existential quantification $\exists p' \in resolve(p)$, $\exists r' \in ancestors(r)$ in Definition 1. The temporal predicates (`EffectiveFrom` $\leq$ `@Now`, `EffectiveTo` $\geq$ `@Now`) correspond to $active(t)$. The JOIN against `RolePermissions` verifies $perm \in perms(role)$. A non-empty result therefore witnesses a satisfying $(p', role, r')$ triple. □ □

Theorem 2 (Completeness)**.** *If $allowed(p, perm, e.ResourceId, now) = true$, the TVF returns a non-empty result, provided $depth \leq 10$.*

Proof. The CTE with guard `Depth < 10` produces all ancestors for resources with depth ≤ 10. For any satisfying witness $(p', role, r')$ in the formal model, the CTE includes r' (since $r' \in ancestors(r)$), the principal filter includes p' (since $p' \in resolve(p)$), and the temporal and role-permission predicates match by construction. The witness therefore produces a matching row. □ □

C. Composition into Application Queries

Because the TVF is an inline function, the query optimizer composes its body into the calling query's execution plan. The application layer resolves the principal's group memberships once per request, then appends the TVF as a predicate alongside application-defined filters, cursor pagination, and sorting:

```
FUNCTION AuthorizedList(principal, permission,
                        filters, cursor, k):
  principalIds <- ResolveGroupMemberships(principal)
  query <- FROM table
           WHERE Accessible(row.ResourceId,
                    principalIds, permission)
             AND ApplyFilters(row, filters)
             AND row.SortKey > cursor
           ORDER BY row.SortKey
           LIMIT k
  RETURN Execute(query)  // single DB operation
```

Authorization, application filtering, pagination, and sorting compose into a single database operation—one round-trip, zero external calls.

VI. Complexity Analysis

A. Per-Row TVF Cost

For a single entity with resource at depth $d \leq D$:

1) **CTE expansion:** $O(D)$ index seeks on `Resources(Id)`.
2) **Grant matching:** Joins D ancestors against `Grants(ResourceId, PrincipalId)` and `RolePermissions(RoleId, PermissionId)`. With $M = |resolve(p)|$ principals, examines at most $D \cdot M \cdot G_{\max}$ candidates.
3) **TOP 1 exit:** Short-circuits on first match.

Per-row cost: $O(D \cdot M \cdot G_{\max})$. With $D = 5$, $M = 10$, $G_{\max} = 3$: $\sim$150 index lookups.

B. Per-Page Cost Under Pagination

Theorem 3 (Per-Page Complexity)**.** *Let N be total entities, k the page size, D the max tree depth, $M = |resolve(p)|$, $G_{\max}$ the max active grants per (principal, resource) pair, and $\sigma \in (0, 1]$ the selectivity. Assume: (1) index coverage on Resources, Grants, RolePermissions, and entity ResourceId; (2) cursor pagination on an indexed ordering key; (3) nested loops plan with index seeks. Then:*

- ***Dense*** *($\sigma \approx 1$): $O(k \cdot D \cdot M \cdot G_{\max})$, simplified to $O(k \cdot D)$ when M and $G_{\max}$ are bounded constants.*
- ***Sparse*** *($\sigma \to 0$): $O(k/\sigma \cdot D \cdot M \cdot G_{\max})$. Degenerate case (no access): $O(N \cdot D \cdot M \cdot G_{\max})$.*

Proof. With cursor pagination, the engine seeks to the cursor position in $O(\log N)$ and evaluates rows sequentially. Each candidate row incurs a TVF evaluation: the CTE performs $O(D)$ index seeks to walk ancestors, each ancestor is joined against Grants (M principal checks, $G_{\max}$ grants each) and RolePermissions (one seek per grant), yielding $O(D \cdot M \cdot G_{\max})$ per row. `TOP 1` short-circuits on the first matching grant, so this is an upper bound. To collect k authorized rows with selectivity σ, the engine examines $\sim k/\sigma$ candidates. Total: $O(k/\sigma \cdot D \cdot M \cdot G_{\max})$. When $\sigma \approx 1$, this simplifies to $O(k \cdot D \cdot M \cdot G_{\max})$. Critically, the cursor start position is resolved by index seek, not by scanning prior pages, so cost is independent of page depth and dataset size N. □ □

The per-page cost is *parameterized and predictable*—independent of N, linear in D.

C. Comparison with Alternatives

D. Known Considerations

`STRING_SPLIT` produces poor cardinality estimates; table-valued parameters are recommended for $M > 5$. CTE traversal is not short-circuitable; a closure table replaces $O(D)$ recursive expansion with $O(1)$ lookup for deep hierarchies. Plan shape assumes nested loops with index seeks (observed under correct indexing); stale statistics may produce hash joins, mitigated by `UPDATE STATISTICS`.

TABLE II
PER-PAGE COST COMPARISON ACROSS AUTHORIZATION APPROACHES.

Approach	Per-page	Deps.	Consist.
SHRBAC (cursor)	$O(k \cdot D)$	Local DB	Strong
LookupResources + IN	$O(\text{graph}) + O(k)$	External	Eventual
Batch post-filter	$O(k/\sigma \cdot \ell)$	External	Per-check
Partial evaluation	$O(k) + O(\text{comp})$	Policy eng.	Varies
Materialized JOIN	$O(k)$	Denorm.	Eventual

VII. EMPIRICAL EVALUATION

We evaluate SHRBAC on SQL Server 2022 (Docker, 4 vCPU, 8 GB RAM) across three tiers: (1) small-scale isolation tests (1K–100K entities) that individually vary each factor, (2) a production-scale workload at D=5 with 1.2M resource nodes, and (3) a deep-hierarchy workload at D=10 with 1.5M resource nodes.

A. Benchmark Methodology

Measurement protocol. Each query: 3 warmup runs (discarded), 20 measured runs. Each run opens a fresh connection and records wall-clock elapsed time via `Stopwatch`. We report median, P95, and IQR. Page size $k = 20$ unless stated otherwise.

Query pattern. All list filtering benchmarks execute the canonical authorized-list query:

```
SELECT TOP (@k) p.Id, p.Name, p.SKU,
       p.Price, p.ResourceId
FROM Products p
WHERE EXISTS (
    SELECT 1 FROM dbo.fn_IsResourceAccessible(
        p.ResourceId, @principalIds,
        'product_view'))
AND p.Id > @cursor
ORDER BY p.Id
```

Point checks use the same TVF against a single ResourceId. Query plans verified to use nested loops with index seeks.

Hierarchies. D=5: root → 15 chains → 150 regions → 15K stores → 1.2M products = **1,215,166 resources**. D=10: root → 5 divisions → 25 regions → 125 districts → 500 areas → 2K zones → 12K stores → 60K departments → 240K sections → 1.2M products = **1,514,656 resources**. Resources and domain rows are bulk-inserted; `UPDATE STATISTICS` is run before benchmarks.

Isolation tests. All benchmarks use the same retail SaaS domain model. Isolation tests (Benchmarks 1–7) seed a hierarchy of real domain tables scaled to the target depth: at D=3, root → Regions (10) → Stores (100) → Products (N); at D=5, root → Divisions (10) → Chains (100) → Regions (1K) → Stores (10K) → Products (N). Each intermediate level has its own domain table with domain-appropriate columns, and each row maps to a unique resource in the tree. The benchmark query always targets the Products table, ensuring that even small-scale isolation tests exercise the same schema structure and query patterns as the production-scale benchmarks.

TABLE III
ISOLATION TEST RESULTS. k=20 THROUGHOUT.

Experiment	Configuration	Med. (ms)	P95 (ms)
N (resource count)	N=1K, D=3	3.57	4.19
	N=5K, D=3	3.51	4.58
	N=10K, D=3	3.39	4.01
	N=50K, D=3	3.63	4.05
	N=100K, D=3	3.32	4.07
Depth	D=1, N=10K	2.45	4.90
	D=2, N=10K	2.94	3.96
	D=3, N=10K	3.17	6.16
	D=4, N=10K	3.55	4.43
	D=5, N=10K	4.21	4.79
Principal set (M)	M=1, N=10K	3.47	5.53
	M=4, N=10K	3.14	3.75
	M=11, N=10K	3.45	4.81
	M=21, N=10K	3.32	4.09
TVF vs. mater.	TVF EXISTS, N=100K	3.42	3.97
	Materialized JOIN, N=100K	1.01	1.28
Pagination	Cursor, N=100K	3.30	4.22
	Offset COUNT, N=100K	2,310	2,473

TABLE IV
PRODUCTION-SCALE RESULTS AT 1.2M RESOURCES, D=5.

Sub-experiment	Configuration	Med. (ms)	P95 (ms)
Page size (k)	k=10	2.28	7.59
	k=20	3.47	4.55
	k=50	6.89	8.65
	k=100	13.71	20.25
Scope level	Company admin (1.2M)	3.47	4.55
	Chain manager (∼80K)	3.20	3.95
	Region manager (∼8K)	3.11	6.34
	Store manager (∼80)	2.02	2.57
Deep cursor	Page 1	3.08	4.04
	∼Page 50	3.25	3.63
	∼Page 500	3.48	4.28

B. Isolation Tests (1K–100K)

Table III presents results from controlled single-factor experiments, re-seeded for each configuration to isolate individual factors.

These isolation results confirm core predictions: N-independence across 1K–100K (3.32–3.63ms at D=3), monotonic depth scaling from 2.45ms at D=1 to 4.21ms at D=5 consistent with $O(k \cdot D)$, negligible principal set size effect (3.14–3.47ms across M=1–21), and constant cursor pagination cost independent of dataset size. The question is whether these properties hold at production resource scale.

C. Production-Scale at D=5 (1.2M Resources)

We constructed a retail SaaS hierarchy: root → 15 chains → 150 regions → 15K stores → 1.2M products, yielding **1,215,166 resource nodes** at D=5. Principals range from company administrators (all 1.2M products accessible) to store managers (∼80 products).

Table IV presents results across three dimensions.

Page size scales linearly: the ratio $13.71/2.28 = 6.01\times$ for a $10\times$ increase in k confirms $O(k)$. Per-row cost at D=5: ∼0.14ms, corresponding to ∼0.034ms per CTE hop.

TABLE V
PAGE SIZE COMPARISON: D=5 VS. D=10.

k	D=5 (ms)	D=10 (ms)	Ratio
10	2.28	3.44	1.51×
20	3.47	5.69	1.64×
50	6.89	11.80	1.71×
100	13.71	22.01	1.61×

TABLE VI
POINT ACCESS CHECK LATENCY. D=5: 1.2M RESOURCES; D=10: 1.5M RESOURCES.

Depth	Level	D=5 (ms)	D=10 (ms)
0	Root	0.86	0.89
1	Chain / Division	0.97	0.95
2	Region	1.06	1.27
3	Store / District	1.04	0.99
4	Product / Area	1.31	1.09
5–9	(D=10 only)	—	1.13–1.46

Scope level: despite a 15,000× difference in accessible product count, latency varies minimally—store-level grants fastest in median because CTE short-circuits.

Cursor depth is flat: page 500 at 3.48ms is indistinguishable from page 1 at 3.08ms.

D. Deep Hierarchy at D=10 (1.5M Resources)

To test whether $O(k \cdot D)$ scaling holds at deeper hierarchies, we constructed a 10-level tree yielding **1,514,656 resource nodes**. Table V compares identical queries across both tree depths.

The D=10/D=5 ratio ranges from 1.51× to 1.71×. Per-row cost at D=10: ∼0.22ms, or ∼0.024ms per CTE hop. The per-hop cost difference (0.034ms at D=5 vs 0.024ms at D=10) suggests fixed per-page overhead is proportionally larger at D=5; at larger k the costs converge.

Scope-level and cursor-depth results at D=10 mirror D=5: scope varies minimally (3.06–5.54ms), and cursor depth is flat (5.11–5.38ms across pages 1–500).

E. Point Access Checks

While SHRBAC's primary contribution is list filtering, the TVF also serves as a point check. Table VI measures point check latency at both D=5 (1.2M resources) and D=10 (1.5M resources).

Even at depth 9 with 1.5M resource nodes, all point checks complete in **under 1.5ms median**. Grant set size (1–20 active grants) produces 1.19–1.38ms—flat. Principal set size (M=1–21) produces 0.98–1.28ms—near-constant. The CTE examines only the target resource's ancestor chain ($\leq D$ nodes), not the grant table at large.

F. Factor Analysis

The multi-dimensional benchmarks at 1.2M resources (D=5) and 1.5M resources (D=10) enable a comprehensive factor analysis (Table VII).

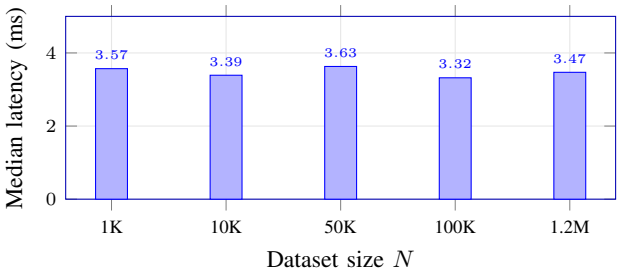

Fig. 3. Per-page median latency (k=20, cursor) across three orders of magnitude. Latency remains flat at ∼3.3–3.6ms. The small increase at 1.2M is attributable to depth (D=5 vs D=3), not resource count.

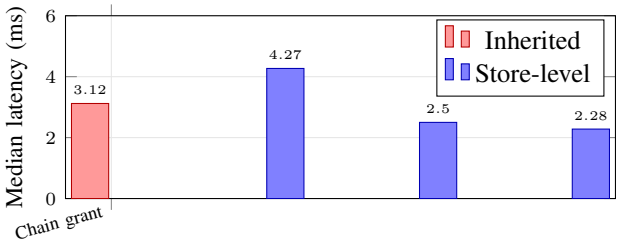

Fig. 4. Store-level grants outperform inherited grants at 1.2M resources. A single store grant (2.28ms) is 27% faster than a chain-level grant (3.12ms). The CTE walks UP from the product; grants closer to the leaf short-circuit sooner.

*The 10K→1.2M increase (3.39→3.47ms) compares D=3 isolation vs D=5 production; the difference is attributable to depth, not resource count.

Table VIII presents the dimensional analysis at D=5.

G. Analysis

N-independence extends to 1.5M resources (Fig. 3). With 1,215,166 unique resource nodes at D=5, list filtering median is 3.47ms (k=20)—compared to 3.39ms at N=10K with D=3 in isolation tests. The small increase is attributable to depth (D=5 vs D=3), not resource count. At D=10 with 1,514,656 resources, per-page cost is 5.69ms—again a depth effect.

Per-hop cost. At D=5, per-row cost is ∼0.14ms (4 CTE hops × 0.034ms/hop). At D=10, per-row cost is ∼0.22ms (9 CTE hops × 0.024ms/hop). The per-hop cost converges at higher page sizes as fixed per-page overhead diminishes, giving $O(k \cdot D)$ a concrete, measurable constant.

Grant density remains irrelevant. At D=5: 3.31 vs. 3.40ms—a 0.09ms difference for a 10× increase. At D=10: 5.00 vs. 5.76ms. The TVF evaluates each row via its ancestor chain; the breadth of authority does not affect per-row cost.

Least-privilege grants outperform inherited grants (Fig. 4). A chain-level grant (3.12ms) is 37% slower than a single store grant (2.28ms). The CTE walks UP from the product toward the root; a store-level grant matches at the first ancestor hop, while a chain-level grant requires 3–4 hops. This validates the short-circuit property and reveals that least-privilege grants are not only more secure—they are faster.

TABLE VII
FACTOR SENSITIVITY ANALYSIS AT PRODUCTION SCALE. RESULTS FROM D=5 (1.2M RESOURCES) AND D=10 (1.5M RESOURCES).

Factor	Range	List filter effect	Point check effect	Predicted	Confirmed?
N (resource count)	1K–1.5M	None (3.39ms → 3.47ms*)	N/A (CTE-local)	$O(1)$	✓
k (page size)	10–100	Linear (2.28–13.71ms D=5; 3.44–22.01ms D=10)	N/A	$O(k)$	✓
D (tree depth)	1–10	Linear (2.45 → 4.21 → 5.69ms)	Sub-1.5ms both	$O(D)$	✓
Per-hop cost	D=5 vs D=10	0.034ms D=5, 0.024ms D=10	—	$O(1)$	✓
M (principals)	1–11	Negligible (3.32–3.44ms D=5; 5.42–5.85ms D=10)	None (0.98–1.28ms)	$O(M)$	Negligible
G (grant density)	1–10	None (3.23–3.40ms D=5; 5.00–5.80ms D=10)	None (1.19–1.38ms)	$O(G_{\max})$	Refuted
Cursor depth	Page 1–500	None (3.08–3.48ms D=5; 5.11–5.38ms D=10)	N/A	$O(1)$	✓

TABLE VIII
DIMENSIONAL ANALYSIS AT 1.2M RESOURCES, D=5, k=20.

Dimension	Configuration	Med. (ms)
Principal set (list)	M=1	3.36
	M=3	3.32
	M=6	3.36
	M=11	3.44
Grant density	1 chain (∼80K accessible)	3.31
	3 chains (∼240K)	3.23
	5 chains (∼400K)	3.41
	10 chains (∼800K)	3.40
Grant depth	Chain grant (inherit all)	3.12
	100 store grants	4.27
	10 store grants	2.50
	1 store grant	2.28

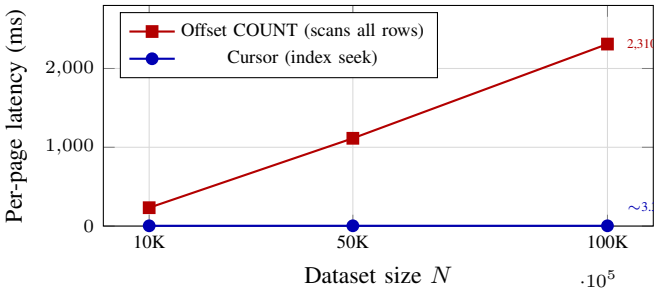

Fig. 5. Offset COUNT per-page cost grows linearly with N: 231ms (N=10K) → 1,114ms (N=50K) → 2,310ms (N=100K). Cursor pagination maintains ∼3.3ms. At N=100K, cursor is 700× faster.

Principal set size is negligible. M=1 vs M=11 produces 3.36 vs. 3.44ms. At D=10: 5.63 vs. 5.85ms. `STRING_SPLIT` overhead is negligible at practical principal counts.

Point checks are sub-1.5ms even at D=10. At depth 9 (the deepest leaf), point checks complete in 1.32ms median. Grant set size (1–20) and principal set size (M=1–21) have no measurable impact.

Page size scales linearly. k=10 → 100 yields 2.28 → 13.71ms (6.01× for 10×). At 1.5M resources (D=10): 3.44 → 22.01ms (6.40× for 10×). Both closely track $O(k)$.

Cursor vs. offset pagination (Fig. 5). Offset COUNT costs grow linearly: 231ms (N=10K) → 1,114ms (N=50K) → 2,310ms (N=100K). Cursor pagination remains at ∼3.3ms. At N=100K, cursor is 700× faster.

TVF vs. materialized. The materialized approach is ∼3.4× faster per-page (1.01ms vs. 3.42ms at N=100K) but requires denormalization infrastructure. Both are N-independent under cursor pagination. SHRBAC's TVF provides strong consistency with zero infrastructure overhead.

VIII. DISCUSSION

The theoretical analysis in Sections III–VI established that SHRBAC's per-page enforcement cost is $O(k \cdot D)$, and the empirical results in Section VII confirmed this bound across three orders of magnitude. This section examines the design trade-offs that make these guarantees possible, positions SHRBAC relative to alternative authorization models, and discusses implications for agentic systems.

A. Constraints as Architectural Choice

SHRBAC's performance guarantees are a direct consequence of four structural constraints. Each constraint eliminates a source of unbounded computation in the enforcement path:

Theorem 4 (Constraint-Composability Tradeoff)**.** *SHRBAC intentionally disallows: (C1) DAG-structured resources—resources form a tree, not a DAG; (C2) recursive group membership—groups cannot contain groups; (C3) role hierarchies—roles are flat; (C4) arbitrary attribute predicates—only role + resource-scope + time are evaluated. In return: (G1) CTE traversal is bounded at $O(D)$; (G2) principal resolution is a single join producing a fixed set; (G3) the TVF body is fixed at schema design time—no runtime policy compilation; (G4) enforcement is a standard SQL `EXISTS` subquery composable with arbitrary `WHERE`, `ORDER BY`, and pagination.*

Each constraint enables its corresponding guarantee. Relaxing (C1) to allow DAGs replaces a unique bounded ancestor chain with potentially unbounded multiple ancestor paths, eliminating deterministic $O(D)$ traversal and breaking (G1). Relaxing (C2) to allow nested groups turns principal resolution into a recursive traversal, breaking (G2). Relaxing (C3) to allow role inheritance requires transitive closure at evaluation time, breaking (G3). Relaxing (C4) to allow arbitrary predicates requires runtime policy compilation, breaking (G4).

Crucially, these constraints are not artificial restrictions imposed to simplify the model—they codify the *de facto* structure of most multi-tenant B2B SaaS systems. Resource containment naturally forms a tree rooted at a tenant boundary (organization → departments → projects → artifacts). Groups are typically flat membership lists. Roles enumerate permissions rather than

inheriting from other roles. SHRBAC formalizes this common but undocumented pattern and demonstrates that when the dominant structural assumptions are embraced explicitly, strong composability and performance guarantees follow.

B. Relationship to ABAC and ReBAC

SHRBAC evaluates two attributes—role and resource-scope—making it a two-attribute constrained ABAC system [15]. The resource tree can be viewed as a constrained ReBAC graph where all relationships are `parent_of` typed and the graph is a tree. The tree constraint (bounded depth, deterministic ancestor chains) is what makes TVF enforcement tractable. For applications requiring arbitrary relationship graphs, ReBAC/Zanzibar is more appropriate; for organizational hierarchies, SHRBAC's constraint matches the domain and provides predictable performance.

C. SHRBAC for Agentic Systems

SHRBAC's polymorphic principal model naturally accommodates autonomous AI agents. An agent is a principal with $\tau(p) =$ agent that participates in the same grant relation as human users, with three properties critical for agent governance:

Least-privilege delegation. SHRBAC's grant triple (principal, role, resource) with temporal bounds directly enforces least-privilege delegation: an agent receives only the role it needs, at only the resource subtree it operates on, for only the duration of its task. A 15-minute grant at a specific project subtree is expressible directly, without special-case logic.

Predictable query cost. Agents that enumerate resources in loops amplify per-query costs. The TVF's $O(k \cdot D)$ per-page bound ensures that agent-driven queries have predictable, bounded database impact regardless of how many resources exist—preventing runaway load from autonomous operations.

Auditability. Every agent action traces to a specific (agent, role, resource) grant with temporal bounds. When an agent's authority expires ($\mathit{EffectiveTo} < now$), access ceases immediately without requiring token revocation infrastructure.

D. Scope and Limitations

- **Tree constraint (C1):** DAG resources (matrix management, multi-parent projects) are not supported. ReBAC is more appropriate for these domains.
- **Non-recursive groups (C2):** Deeply nested group structures require extending $resolve(p)$ with CTE traversal.
- **Flat roles (C3):** Role definitions must explicitly enumerate permissions rather than inheriting.
- **No arbitrary attributes (C4):** IP-based, device-type, or other dynamic attribute conditions are not expressible.
- **Database engine:** Portability requirements discussed in §V. PostgreSQL and MySQL 8.0+ support the necessary primitives (recursive CTEs, predicate inlining).

IX. Conclusion

We presented SHRBAC, a formal authorization model whose structural constraints guarantee that per-page enforcement cost is $O(k \cdot D)$—linear in page size and tree depth, independent of total resource count and policy set size. The model sits at the intersection of hierarchical RBAC (ROBAC/RRBAC), polymorphic principal resolution, and Stonebraker's query modification.

Empirical evaluation at 1.2M resources (D=5) and 1.5M resources (D=10) confirms the predicted complexity: per-page latency is N-independent across three orders of magnitude, scales linearly with k and D, and is unaffected by grant density or cursor depth. The per-CTE-hop cost is constant across both tree depths, giving $O(k \cdot D)$ a measurable constant. A notable finding is that least-privilege grants are not only more secure but faster—the upward tree walk short-circuits sooner at narrower scopes.

As autonomous agents become principals in enterprise systems, the need for scoped, auditable, time-bounded authorization with efficient list filtering will intensify. SHRBAC's grant model—where an agent receives exactly the authority it needs, at the scope it needs, for the duration it needs—provides a foundation for database-native agent governance without requiring external authorization infrastructure.

Acknowledgments

[Removed for double-blind review.]

AI Disclosure: In accordance with IEEE policy, the authors disclose that AI tools (Claude, Anthropic) were used to assist with manuscript preparation and editing. All technical content, formal definitions, proofs, implementation, and experimental design are the work of the authors.

References

[1] R. Pang et al., "Zanzibar: Google's Consistent, Global Authorization System," in *Proc. USENIX ATC*, 2019.

[2] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[3] Z. Zhang, X. Zhang, and R. Sandhu, "ROBAC: Scalable Role and Organization Based Access Control Models," in *Proc. CollaborateCom*, 2006.

[4] M. Stonebraker and E. Wong, "Access Control in a Relational Data Base Management System by Query Modification," in *Proc. ACM National Conf.*, 1974, pp. 180–187.

[5] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control Models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[6] D. F. Ferraiolo et al., "Proposed NIST Standard for Role-Based Access Control," *ACM TISSEC*, vol. 4, no. 3, pp. 224–274, 2001.

[7] N. Solanki et al., "Resource and Role Hierarchy Based Access Control for Resourceful Systems," in *Proc. IEEE COMPSAC*, pp. 396–401, 2018.

[8] A. Abou El Kalam et al., "Organization Based Access Control," in *Proc. IEEE POLICY*, 2003.

[9] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending Query Rewriting Techniques for Fine-Grained Access Control," in *Proc. ACM SIGMOD*, pp. 551–562, 2004.

[10] P. Pappachan, R. Yus, S. Mehrotra, and J.-C. Freytag, "Sieve: A Middleware Approach to Scalable Access Control for Database Management Systems," *PVLDB*, vol. 13, no. 12, 2020.

[11] P. W. L. Fong, "Relationship-Based Access Control: Protection Model and Policy Language," in *Proc. CODASPY*, pp. 191–202, 2011.

[12] AuthZed, "Protecting a List Endpoint," SpiceDB Documentation. [Online].

[13] Oso, "List Filtering," Oso Documentation. [Online].

[14] Cerbos, "Filtering Data Using Authorization Logic," Cerbos Blog. [Online].

[15] V. C. Hu et al., "Guide to Attribute Based Access Control (ABAC) Definition and Considerations," *NIST SP 800-162*, 2014.

[16] Microsoft, “What is Azure role-based access control (Azure RBAC)?” Microsoft Learn. [Online].
[17] E. Bertino, P. Bonatti, and E. Ferrari, “TRBAC: A Temporal Role-Based Access Control Model,” *ACM TISSEC*, vol. 4, no. 3, 2001.
[18] R. Sandhu, V. Bhamidipati, and Q. Munawer, “The ARBAC97 Model for Role-Based Administration of Roles,” *ACM TISSEC*, vol. 2, no. 1, pp. 105–135, 1999.
[19] D. R. Kuhn, E. J. Coyne, and T. R. Weil, “Adding Attributes to Role-Based Access Control,” *IEEE Computer*, vol. 43, no. 6, pp. 79–81, 2010.
[20] Microsoft, “Row-Level Security,” SQL Server Documentation. [Online].