]

Smart Reader Database Guide

A Comprehensive Schema Reference and Justification

Smart Reader Project
Database Architecture Team

November 2, 2025

Abstract

This guide provides a comprehensive overview of the Smart Reader database architecture, designed for PostgreSQL with Supabase. The document explains the rationale behind schema design decisions, indexes, security policies, and performance optimizations. It serves as both a reference manual and a pedagogical resource for understanding multi-tenant SaaS database architecture patterns, covering topics from basic normalization to advanced partitioning and vector search strategies.

Contents

Introduction

Purpose of This Guide

This guide is designed to be **self-teaching** and **illustrative**. It doesn't just document *what* the database schema contains, but *why* each design decision was made. Whether you're a new developer joining the project, a database administrator reviewing the architecture, or a student learning about production database design, this guide will help you understand the thought process behind each component.

Database Overview

The Smart Reader platform is a **document reading and AI chat application** that allows users to:

- Upload and manage PDF and text documents
- Chat with AI about document contents (RAG - Retrieval-Augmented Generation)
- Take notes and highlight text
- Track reading progress and study time using Pomodoro technique
- Organize documents into collections and tags

Technical Stack:

- **Database:** PostgreSQL 15+ via Supabase
- **Storage:** AWS S3 for large files
- **Vectors:** pgvector extension for semantic search
- **Security:** Row-Level Security (RLS) on all tables
- **Deployment:** Serverless with edge functions

Design Principles

Throughout this database design, we followed several key principles:

1. **Multi-tenancy by design:** Every table is user-scoped with RLS policies
2. **Scalability first:** Ready for 10,000+ users without architectural changes
3. **Security by default:** RLS enabled on all tables, `SET search_path = ''` on functions
4. **Performance optimization:** Strategic indexes, materialized views, and denormalization where appropriate
5. **Data integrity:** Foreign keys, CHECK constraints, and ACID compliance
6. **Maintainability:** Clear naming, documentation, and audit trails

Core Architecture

Schema Organization

One of the most important architectural decisions was **schema separation**. Instead of putting everything in the `public` schema, we split the database into two logical domains:

`public` **schema: Library Management** - Books, notes, highlights, productivity tracking

`chat` **schema: AI/RAG Infrastructure** - Conversations, memories, vector embeddings

Design Decision

Why schema separation?

- **Logical isolation:** Library features and AI chat evolve independently
- **Performance:** Different scaling strategies (library can be cached, chat needs low latency)
- **Security:** Separate RLS policies for different use cases
- **Maintainability:** Clearer for developers to find related tables
- **Database sharding:** Future-proof for splitting to separate databases

Security Model

2.2.1 Row-Level Security (RLS)

Every table in the database has RLS enabled. This means that even if an attacker gains SQL access, they can only see their own data. PostgreSQL automatically filters rows based on policy conditions.

```
-- Users can only read their own books
CREATE POLICY "Users can read own books" ON user_books
  FOR SELECT USING (auth.uid() = user_id);

-- Users can create books for themselves
CREATE POLICY "Users can create own books" ON user_books
  FOR INSERT WITH CHECK (auth.uid() = user_id);
```

Listing 1: Example RLS Policy Pattern

Information

auth.uid() is a Supabase function that returns the authenticated user's UUID. All RLS policies use this to enforce data isolation.

2.2.2 Security Definer Functions

Some database functions need elevated privileges (e.g., creating partitions, refreshing views). We use `SECURITY DEFINER` with `SET search_path = ''` to prevent search path attacks:

```
CREATE OR REPLACE FUNCTION get_user_stats(user_uuid UUID)
RETURNS TABLE (...)
LANGUAGE plpgsql
```

```
SECURITY DEFINER
SET search_path = ''  -- Prevents search path attacks
AS $$
BEGIN
  -- Schema-qualified tables prevent hijacking
  RETURN QUERY
  SELECT * FROM public.user_books
  WHERE user_id = user_uuid;
END;
$$;
```

Listing 2: Secure Function Pattern

Warning

Without `SET search_path = ''`, a malicious user could create a table named `user_books` and the function would query their table instead of the real one! Always use schema-qualified names in SECURITY DEFINER functions.

Core Tables

profiles

The `profiles` table extends Supabase's built-in `auth.users` table. It's the root of our user data model.

```
CREATE TABLE profiles (
  id UUID REFERENCES auth.users(id) PRIMARY KEY,
  email TEXT UNIQUE NOT NULL,
  full_name TEXT,
  tier TEXT DEFAULT 'free' CHECK (tier IN ('free', 'pro', 'premium',
     'enterprise')),
  credits INTEGER DEFAULT 100,
  stripe_customer_id TEXT,
  stripe_subscription_id TEXT,
  subscription_status TEXT,
  ocr_count_monthly INTEGER DEFAULT 0,
  ocr_last_reset TIMESTAMP DEFAULT NOW(),
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Listing 3: profiles Table Schema

Design Decision

Design decisions:

- **Foreign key to auth.users:** Leverages Supabase auth, no password management needed
- **CHECK constraint on tier:** Database-level validation prevents invalid values
- **Denormalized credits:** Fast access without JOINs; updated via triggers/functions
- **Monthly OCR counter:** Tracks usage for billing; resets via scheduled function
- **Stripe integration:** Stores customer and subscription IDs for payment operations

user_books

The **central table** of the application. This evolved from having both a `documents` table and `user_books` table (Migration 026 merged them). It now handles everything related to uploaded documents.

```
CREATE TABLE user_books (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE NOT NULL,

  -- File metadata
  title TEXT NOT NULL,
  file_name TEXT NOT NULL,
  file_type TEXT CHECK (file_type IN ('pdf', 'text')),
  file_size_bytes BIGINT NOT NULL,
  s3_key TEXT,  -- Files stored in S3, not database

  -- Reading support
  total_pages INTEGER,
  page_texts TEXT[],  -- Array of text per page (for TTS)
  text_content TEXT,

  -- AI/RAG support
  content TEXT,  -- Full text for AI chat
  embedding_status TEXT CHECK (embedding_status IN
    ('pending', 'processing', 'completed', 'failed')),

  -- OCR support
  needs_ocr BOOLEAN DEFAULT FALSE,
  ocr_status TEXT CHECK (ocr_status IN
    ('not_needed', 'pending', 'processing', 'completed', 'failed')),
  ocr_metadata JSONB DEFAULT '{}',

  -- Reading progress
  last_read_page INTEGER DEFAULT 1,
  reading_progress DECIMAL(5,2) DEFAULT 0.00,
  last_read_at TIMESTAMPTZ,

  -- Organization
  is_favorite BOOLEAN DEFAULT FALSE,
  custom_metadata JSONB DEFAULT '{}',
```

```
  -- Denormalized counters
  notes_count INTEGER DEFAULT 0,
  pomodoro_sessions_count INTEGER DEFAULT 0,
  total_pomodoro_time_seconds INTEGER DEFAULT 0,

  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Listing 4: user_books Core Schema (Abridged)

Design Decision

Key design decisions:

1. **S3 storage:** Large PDF files aren't stored in the database (Migration 004). This dramatically improves performance and reduces costs.
2. **Array columns (page_texts):** PostgreSQL arrays are perfect for this use case - we extract text from each page and store in an array. Access via subscript notation: `page_texts[0]` for page 1.
3. **Denormalization:** `notes_count` and `pomodoro_sessions_count` are denormalized. We could compute them with `COUNT()`, but that would be slow for dashboard queries. Instead, they're maintained via triggers.
4. **JSONB flexibility:** `custom_metadata` allows extensibility without schema changes. Users could store publication date, ISBN, etc.
5. **CHECK constraints:** Enforce valid values at the database level, preventing invalid data from entering the system.

3.2.1 Critical Indexes

```
-- Most common query: list user's books with pagination
CREATE INDEX idx_user_books_library_view ON user_books
  (user_id, is_favorite, last_read_at DESC, id);

-- Find by file type (common filter)
CREATE INDEX idx_user_books_type_date ON user_books
  (user_id, file_type, created_at DESC);

-- Full-text search (GIN index for fast search)
CREATE INDEX idx_books_search ON user_books
  USING gin(to_tsvector('english', title || ' ' || file_name));

-- Active reading (only books with progress > 0)
CREATE INDEX idx_user_books_active_reading ON user_books
  (user_id, book_id) WHERE reading_progress > 0;
```

Listing 5: Essential user_books Indexes

Design Decision

Index strategy explanation:

- **Composite index order:** Columns ordered by selectivity (user_id most selective, then is_favorite, etc.)
- **Covering index:** Includes `id` to avoid heap access for pagination
- **Partial index:** `WHERE reading_progress > 0` creates smaller, faster index for active books
- **GIN index:** Special index type for full-text search, much faster than regular indexes

user_notes

Annotations on book pages. Supports multiple note types (Cornell, outline, mindmap, etc.) via flexible JSONB.

```
CREATE TABLE user_notes (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE NOT NULL,
  book_id UUID REFERENCES user_books(id) ON DELETE CASCADE NOT NULL,
  page_number INTEGER NOT NULL,
  content TEXT NOT NULL,
  position_x DECIMAL(10,2),
  position_y DECIMAL(10,2),
  note_type TEXT CHECK (note_type IN
    ('cornell', 'outline', 'mindmap', 'chart', 'boxing', 'freeform')),
  note_metadata JSONB DEFAULT '{}',
  is_ai_generated BOOLEAN DEFAULT FALSE,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Listing 6: user_notes Schema

Design Decision

Why position columns? Notes can be positioned on a PDF canvas (like sticky notes). X,Y coordinates enable:

- Visual rendering of notes on PDF viewer
- Dragging notes to reposition
- Grouping nearby notes
- Exporting with spatial information

user_highlights

Text highlights with context for robust text matching.

```
CREATE TABLE user_highlights (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
```

```
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE NOT NULL,
  book_id UUID REFERENCES user_books(id) ON DELETE CASCADE NOT NULL,
  page_number INTEGER NOT NULL,
  highlighted_text TEXT NOT NULL,
  color_id TEXT,
  color_hex TEXT,

  -- Context for matching
  text_start_offset INTEGER,
  text_end_offset INTEGER,
  text_context_before TEXT,   -- 50 chars before
  text_context_after TEXT,    -- 50 chars after

  -- Orphan handling
  is_orphaned BOOLEAN DEFAULT FALSE,
  orphaned_reason TEXT,

  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Listing 7: user$_h$*ighlightsSchema*

Design Decision

Why context columns? If a user edits the original PDF and re-uploads, text positions might change. By storing 50 characters before and after, we can:

1. Find the highlight's new position using fuzzy matching
2. Detect if the text was deleted (mark as orphaned)
3. Warn users when highlights are affected by edits

This is a production-grade solution to a common problem!

Collection and Tag System

3.5.1 user_collections

Folders for organizing books. Supports hierarchical nesting with `parent_id`.

```
CREATE TABLE user_collections (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE NOT NULL,
  name TEXT NOT NULL,
  description TEXT,
  parent_id UUID REFERENCES user_collections(id) ON DELETE CASCADE,
  color TEXT DEFAULT '#3B82F6',
  icon TEXT DEFAULT 'folder',
  is_favorite BOOLEAN DEFAULT FALSE,
  display_order INTEGER DEFAULT 0,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Listing 8: user_collections Schema

3.5.2 book_tags and book_tag_assignments

Flexible tagging system with many-to-many relationships.

```sql
-- Tags themselves
CREATE TABLE book_tags (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE NOT NULL,
  name TEXT NOT NULL,
  color TEXT DEFAULT '#6B7280',
  category TEXT DEFAULT 'general',
  usage_count INTEGER DEFAULT 0,  -- Denormalized
  created_at TIMESTAMPTZ DEFAULT NOW(),
  UNIQUE(user_id, name)  -- No duplicate tags per user
);

  -- Many-to-many: books <-> tags
CREATE TABLE book_tag_assignments (
  book_id UUID REFERENCES user_books(id) ON DELETE CASCADE NOT NULL,
  tag_id UUID REFERENCES book_tags(id) ON DELETE CASCADE NOT NULL,
  assigned_at TIMESTAMPTZ DEFAULT NOW(),
  PRIMARY KEY (book_id, tag_id)
);
```

Listing 9: Tag System Schema

Design Decision

Many-to-many design:

- **Separate junction table:** Allows books to have multiple tags, tags to apply to multiple books
- **Composite PRIMARY KEY:** Prevents duplicate assignments at database level
- **Denormalized usage_count:** Tracks how many books use each tag for "popular tags" features
- **UNIQUE on (user_id, name):** Prevents duplicate tag names per user (e.g., can't have two "work" tags)

AI and RAG Infrastructure (Chat Schema)

chat.conversations and chat.messages

The foundation of the AI chat system. Conversations contain messages.

```sql
-- In chat schema
CREATE TABLE conversations (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE NOT NULL,
  document_id UUID REFERENCES user_books(id) ON DELETE SET NULL,
  title TEXT,
  metadata JSONB DEFAULT '{}',
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

```sql
CREATE TABLE messages (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  conversation_id UUID REFERENCES conversations(id) ON DELETE CASCADE
     NOT NULL,
  role TEXT CHECK (role IN ('user', 'assistant', 'system')),
  content TEXT NOT NULL,
  tokens_used INTEGER,
  model TEXT,
  metadata JSONB DEFAULT '{}',
  created_at TIMESTAMPTZ DEFAULT NOW()
);
```

Listing 10: Conversation Tables in Chat Schema

Design Decision

Why chat schema?

- Separates AI infrastructure from library management
- Different scaling needs (chat needs low latency, library can be cached)
- Enables future sharding (can move chat to separate database)
- Clearer for developers

chat.conversation_memories

Extracted semantic entities from conversations for building knowledge graphs.

```sql
CREATE TABLE conversation_memories (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE,
  conversation_id UUID REFERENCES conversations(id) ON DELETE CASCADE,
  entity_type TEXT NOT NULL,
    -- 'concept', 'question', 'insight', 'reference', 'action',
       'document'
  entity_text TEXT NOT NULL,
  entity_metadata JSONB DEFAULT '{}',
  source_message_id UUID REFERENCES messages(id),
  document_id UUID REFERENCES user_books(id) ON DELETE SET NULL,
  embedding vector(768),  -- pgvector!
  created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Vector similarity search index
CREATE INDEX idx_memories_user_embedding ON conversation_memories
  USING ivfflat (embedding vector_cosine_ops) WITH (lists = 100);
```

Listing 11: Memory System

Design Decision

Vector embeddings explained:

- Each memory gets a 768-dimensional vector embedding (using Gemini's text-embedding-004)
- `ivfflat` is an Inverted File Index - the fastest approximate nearest neighbor algorithm in pgvector
- `vector_cosine_ops` uses cosine similarity (1 - angle between vectors)
- `lists = 100` means 100 clusters for approximate search (trade-off: speed vs. accuracy)

Use case: "What memories are similar to 'quantum entanglement'?" Find related concepts across conversations.

chat.memory_relationships

Connect memories to build a knowledge graph.

```
CREATE TABLE memory_relationships (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE,
  memory_from UUID REFERENCES conversation_memories(id) ON DELETE
     CASCADE,
  memory_to UUID REFERENCES conversation_memories(id) ON DELETE
     CASCADE,
  relationship_type TEXT NOT NULL,
    -- 'relates_to', 'contradicts', 'supports', 'cites', 'explains'
  strength DECIMAL(3,2) DEFAULT 0.5,  -- 0.0 to 1.0
  metadata JSONB DEFAULT '{}',
  created_at TIMESTAMPTZ DEFAULT NOW()
);
```

Listing 12: Memory Graph

Information

This enables **knowledge graph** functionality. Memories become nodes, relationships become edges. Users can see how concepts connect across different conversations.

Productivity Features

Pomodoro Tracking

The Pomodoro Technique is a time management method. We track individual sessions and aggregate statistics.

```
CREATE TABLE pomodoro_sessions (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE NOT NULL,
  book_id UUID REFERENCES user_books(id) ON DELETE CASCADE NOT NULL,
  started_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  ended_at TIMESTAMPTZ,
```

```
  duration_seconds INTEGER,  -- Calculated
  mode TEXT CHECK (mode IN ('work', 'shortBreak', 'longBreak')),
  completed BOOLEAN DEFAULT false,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);

-- Stats aggregated in user_books
total_pomodoro_time_seconds INTEGER DEFAULT 0,
total_pomodoro_sessions INTEGER DEFAULT 0,
last_pomodoro_at TIMESTAMPTZ
```

Listing 13: Pomodoro System

Design Decision

Why track both sessions and aggregates?

- **Sessions:** Detailed history for analytics ("I studied more on weekdays")
- **Aggregates:** Fast dashboard queries without expensive GROUP BY
- Trigger maintains consistency when sessions change

Gamification

Achievements and streaks motivate users to maintain studying habits.

```
CREATE TABLE pomodoro_achievements (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE NOT NULL,
  achievement_key TEXT NOT NULL,  -- 'first_pomodoro', '10_hours', etc.
  title TEXT NOT NULL,
  description TEXT,
  progress_current INTEGER DEFAULT 0,
  progress_target INTEGER,
  unlocked_at TIMESTAMPTZ,
  metadata JSONB DEFAULT '{}',
  UNIQUE(user_id, achievement_key)
);

CREATE TABLE pomodoro_streaks (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE NOT NULL,
  streak_type TEXT NOT NULL,  -- 'daily', 'weekly'
  current_streak INTEGER DEFAULT 0,
  longest_streak INTEGER DEFAULT 0,
  last_activity_date DATE NOT NULL,
  metadata JSONB DEFAULT '{}',
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Listing 14: Gamification Tables

Advanced Features

Materialized Views

Materialized views pre-compute expensive aggregations for dashboard queries.

```sql
CREATE MATERIALIZED VIEW user_books_daily_stats AS
SELECT
  user_id,
  DATE(created_at) as upload_date,
  COUNT(*) as total_books,
  COUNT(*) FILTER (WHERE file_type = 'pdf') as pdf_books,
  SUM(file_size_bytes) as total_size_bytes,
  AVG(reading_progress) as avg_reading_progress
FROM user_books
GROUP BY user_id, DATE(created_at);

-- Unique index for fast lookups
CREATE UNIQUE INDEX ON user_books_daily_stats(user_id, upload_date);
```

Listing 15: Analytics View Example

Design Decision

Why materialized views?

- **Performance:** Pre-aggregated data queries in milliseconds vs. seconds
- **Reduced load:** Dashboard doesn't scan millions of rows
- **Trade-off:** Data is stale until refresh (acceptable for analytics)
- **Refresh strategy:** Nightly via `pg_cron` or on-demand

Note: Cannot use `REFRESH CONCURRENTLY` with views containing `NOW()` - we discovered this during migration fixes!

Table Partitioning

Partitioning enables scaling to thousands of users with per-user data isolation.

```sql
-- Partitioned table (future migration target)
CREATE TABLE user_books_partitioned (
  LIKE user_books INCLUDING DEFAULTS
) PARTITION BY LIST (user_id);

-- Each user gets their own partition
CREATE TABLE user_books_u_{user_id} PARTITION OF user_books_partitioned
  FOR VALUES IN ('{user_id}');

-- Cleanup is now O(1) - drop partition!
DROP TABLE user_books_u_{user_id};
```

Listing 16: Partitioning Infrastructure

Design Decision

Partitioning benefits:

1. **G prune:** PostgreSQL skips irrelevant partitions automatically
2. **Fast cleanup:** GDPR deletion is instant (drop partition)
3. **Index efficiency:** Smaller per-partition indexes
4. **Maintenance:** VACUUM individual partitions
5. **Future sharding:** Can move partitions to different servers

Challenges:

- PRIMARY KEY must include partition key
- Foreign keys require special handling
- Queries crossing partitions are slower

External Vector Store Preparation

```sql
CREATE TABLE embedding_metadata (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE NOT NULL,
  book_id UUID REFERENCES user_books(id) ON DELETE CASCADE NOT NULL,
  provider TEXT CHECK (provider IN
    ('supabase', 'pinecone', 'weaviate', 'qdrant', 'local')),
  namespace TEXT,
  vector_id TEXT NOT NULL,
  collection TEXT,
  embedding_model TEXT DEFAULT 'text-embedding-004',
  vector_dimensions INTEGER DEFAULT 768,
  chunk_index INTEGER,
  chunk_count INTEGER DEFAULT 1,
  generated_at TIMESTAMPTZ DEFAULT NOW(),
  generation_cost_usd DECIMAL(10,6)
);
```

Listing 17: Vector Metadata Table

Design Decision

Why offload vectors?

- pgvector vectors are 768 dimensions $\times$ 4 bytes = 3KB per embedding
- 1000 books $\times$ 100 chunks = 300MB just for vectors
- External stores (Pinecone) are purpose-built and optimized
- Reduces PostgreSQL load
- This table tracks metadata while actual vectors live elsewhere

Migration History

The database has 33 migrations that tell the story of its evolution:

Early Migrations (001-008)

001_initial_schema: Core tables for auth, documents, conversations, messages
002_add_profile_insert_policy: RLS for profile creation
003_add_user_books_table: Separate table for books with TTS support (pdf_data_base64, page_texts array)
004_move_books_to_s3: CRITICAL MIGRATION - Removes large base64 columns. Files now in S3, only metadata in database.
005_add_pomodoro_tracking: Productivity feature
006_add_ocr_support: Scanned PDF OCR capability
007_library_organization: Collections, tags, favorites
008_pomodoro_gamification: Achievements and streaks

Security Hardening (009-015)

010-014: Search path security fixes. Discovered `SET search_path = ''` issue, spent multiple migrations fixing all functions.
012_feature_flags_and_monitoring: Feature toggles and query performance logging

Feature Expansion (016-025)

016_document_relationships: Link related documents
017-021: Various fixes and enhancements
022_structured_rag_memory: Memory system with vector embeddings
023_document_descriptions: AI-generated document summaries
024_add_cleaned_texts: TTS text cleanup
025_fix_remaining_search_path: Final security pass

Major Restructuring (026-033)

These migrations completed in January 2025:
026_consolidate_document_storage:

- Merged `documents` into `user_books`
- Eliminated duplicate models
- Dropped `response_cache`, `document_embeddings`
- **Challenge:** Foreign key constraint ordering

027_standardize_fields:

- Standardized JSONB defaults to `''`
- Unified file size column (`file_size_bytes`)
- Fixed timestamp types
- **Challenge:** Conditional checks for moved tables

028_rationalize_indexes:

- Removed 20+ redundant indexes
- Added covering indexes
- Created partial indexes
- **Challenge:** Partial index with `NOW()` not allowed

029_user_partitioning:

- Created partitioned table infrastructure
- Auto-partition management functions
- GDPR cleanup function
- **Challenge:** PRIMARY KEY must include partition key

030_analytics_views:

- 4 materialized views for analytics
- Refresh functions
- **Challenge:** Can't use `REFRESH CONCURRENTLY` with `NOW()`

031_separate_chat_schema:

- Created `chat` schema
- Moved 6 tables to chat schema
- Updated all RLS policies
- **Challenge:** Cross-schema foreign key references

032_vector_metadata:

- Prepared for external vector stores
- **Challenge:** CTE aggregation for complex stats query

033_cleanup_unused_functions:

- Audited and cleaned functions
- Added proper GRANTs
- Verified security settings

Performance Optimizations

Index Strategy

We carefully crafted indexes based on query patterns:

- **Covering indexes:** Include frequently-selected columns to avoid heap access
- **Composite indexes:** Order by selectivity for multi-column queries
- **Partial indexes:** Smaller indexes for common filters
- **GIN indexes:** Full-text search performance
- **Vector indexes:** IVFFlat for similarity search

Denormalization

Strategic denormalization improves read performance:

- `user_books.notes_count` - Avoids COUNT() on every library load
- `user_books.pomodoro_sessions_count` - Fast dashboard stats
- `book_tags.usage_count` - Popular tags feature

Maintenance: Triggers update denormalized values when source data changes.

Query Optimization Patterns

```
-- Keyset pagination (faster than OFFSET for large datasets)
SELECT * FROM user_books
WHERE user_id = $1
  AND (last_read_at, id) < ($2_last_read, $2_id)
ORDER BY last_read_at DESC, id DESC
LIMIT 20;
```

Listing 18: Pagination Pattern

Information

Why keyset over OFFSET?

- OFFSET 10000 scans 10000 rows before returning results
- Keyset pagination uses index seek - constant time regardless of position
- Essential for apps with millions of rows

Lessons Learned

Common Pitfalls We Encountered

1. **Search path security:** Always use `SET search_path = ''` in `SECURITY DEFINER` functions
2. **Partial indexes:** Can't use volatile functions like `NOW()` in WHERE clauses
3. **Materialized views:** `REFRESH CONCURRENTLY` incompatible with `NOW()`
4. **Partitioned tables:** PRIMARY KEY must include partition key
5. **Foreign keys:** Drop before dropping tables, recreate after
6. **RLS policies:** Must reference correct schema after moves

Best Practices Established

- **Migration numbering:** Sequential, never duplicate
- **Schema qualification:** Always use schema.table in functions
- **RLS by default:** Every table gets RLS policies

- **Check constraints:** Validate at database level, not just application
- **Documentation:** Comment every table and column
- **Testing:** Audit scripts catch issues before production

Conclusion

This database architecture demonstrates several advanced PostgreSQL patterns:

- Multi-tenant security with RLS
- Schema separation for logical organization
- Strategic denormalization for performance
- Materialized views for analytics
- Table partitioning for scalability
- Vector search with pgvector
- Comprehensive migration strategy

The Smart Reader database is designed to scale from proof-of-concept to thousands of users while maintaining excellent query performance, data integrity, and security posture.

Next steps:

1. Monitor query performance with `pg_stat_statements`
2. Schedule materialized view refreshes via `pg_cron`
3. Migrate to external vector store when vectors exceed 1GB
4. Consider TimescaleDB for time-series Pomodoro analytics
5. Implement read replicas for analytics workloads

Migration Reference

Migration	Name	Description
001	initial_schema	Core tables: profiles, documents, conversations, messages
002	add_profile_insert_policy	RLS for profile creation
003	add_user_books_table	User books with TTS support
004	move_books_to_s3	Remove large base64 columns, use S3
005	add_pomodoro_tracking	Pomodoro sessions table
006	add_ocr_support	OCR fields and counters
007	library_organization	Collections, tags, favorites
008	add_pomodoro_gamification	Achievements and streaks
009	search_rpcs	Search function implementations
010-014	fix_search_path_security	Security hardening
012	feature_flags_and_monitoring	Feature toggles, query logging
015	add_highlights_table	Text highlights with context
016	add_document_relationships	Link related documents
017-025	fixes_and_enhancements	Various improvements
022	structured_rag_memory	Memory system with vectors
023	document_descriptions	AI document summaries
026	consolidate_document_storage	Merge documents into user_books
027	standardize_fields	Standardize JSONB, timestamps
028	rationalize_indexes	Remove redundants, add covering
029	user_partitioning	Partitioning infrastructure
030	analytics_views	Materialized views for analytics
031	separate_chat_schema	Move chat tables to new schema
032	vector_metadata	External vector store prep
033	cleanup_unused_functions	Function audit and cleanup

Table Quick Reference

Public Schema

- `profiles` - User accounts and subscriptions
- `user_books` - Documents (PDF/text) uploaded by users
- `user_notes` - Annotations and research notes
- `user_highlights` - Text highlights
- `user_collections` - Folders for organization
- `book_tags` - Flexible tags
- `book_collections` - Books <-> collections (junction)
- `book_tag_assignments` - Books <-> tags (junction)
- `pomodoro_sessions` - Timer sessions
- `pomodoro_achievements` - Unlocked achievements

- `pomodoro_streaks` - Daily/weekly streaks
- `tts_audio_cache` - Cached TTS audio
- `document_descriptions` - AI/user summaries
- `document_relationships` - Links between documents
- `note_relationships` - Links between notes
- `embedding_metadata` - Vector tracking
- `usage_records` - Usage tracking

Chat Schema

- `conversations` - User chat conversations
- `messages` - Conversation messages
- `conversation_memories` - Extracted memories
- `memory_relationships` - Memory graph edges
- `action_cache` - Cached action mappings
- `memory_extraction_jobs` - Background jobs

Important Functions

- `update_updated_at_column()` - Trigger for updated_at
- `handle_new_user()` - Create profile on signup
- `create_user_partition(UUID)` - Create partition for user
- `cleanup_user_data(UUID)` - GDPR deletion
- `refresh_all_analytics_views()` - Refresh materialized views
- `get_similar_memories(..., vector, ...)` - Vector similarity
- `get_related_memories(UUID, ...)` - Memory graph
- `get_book_embeddings(UUID)` - Embedding metadata