DATABASE VIEWS

Design, Optimization & Indexing

News Platform. PostgreSQL

This document describes the design decisions, purpose, performance benchmarks, and indexing strategies for all database views created for the news platform. Performance was measured using EXPLAIN ANALYZE on representative queries against each view, executed before and after index creation to quantify the impact.

Testing Methodology

All performance timings were obtained by running EXPLAIN ANALYZE on a realistic query against each view. typically the query that would be executed in a production use case (e.g. filtering by status and ordering by date for the feed view, or looking up by slug for the article detail view). Tests were run on a populated database with realistic row counts (hundreds of thousands to millions of rows depending on the table). The before/after times represent the actual planner execution time reported by PostgreSQL.

VIEW 1 vw_article_feed

Purpose

This view is the backbone of the public-facing article listing. It powers the homepage feed, category browse pages, search result listings, and any other page where multiple articles are displayed in a list or grid. It deliberately excludes the full article content to keep the payload small. only the fields needed to render a card (title, summary, slug, author info, category, status, language, country, dates) are included.

Design Decisions

- Joins article with user and category (both required, so INNER JOINs).
- Left-joins journalist_profile because not every user with articles is a journalist. regular contributors or admins may also author articles.
- Filters deleted_at IS NULL at the view level so all consumers automatically see only live articles, without needing to remember to add the filter.

- No aggregation is performed in this view. it is a pure join, making it fast and lightweight.

Performance & Indexing

The test query was `SELECT * FROM vw_article_feed WHERE status = 'published' ORDER BY published_at DESC LIMIT 20`. the standard homepage query.

Scenario	Before Index	After Index	Improvement
Homepage feed (published, ordered)	~400 ms	~13 ms	97% faster

Two indexes were created:

- `idx_article_status_published`. a partial index on `published_at DESC` filtered to `status = 'published' AND deleted_at IS NULL`. This directly serves the ORDER BY and WHERE clause of the homepage query, reducing the scanned set from ~200,000 to ~150,000 rows (only published, non-deleted articles).
- `idx_article_feed`. a composite partial index on `(published_at DESC, category_id, language, country)` filtered to `status = 'published' AND deleted_at IS NULL`. This extends the first index by covering the additional filter columns used when browsing by category, language, or country, avoiding a full table scan for those queries as well.

The result was a dramatic 97% improvement, from 400 ms down to 13 ms. No index was placed on journalist_profile because the left join on user_id is already a small lookup given that journalist rows are far fewer than article rows.

VIEW 2 vw_article_detail

Purpose

This view serves the single-article reading page. Unlike the feed view which only returns card-level fields, this view returns everything needed to fully render an article: the full content, extended author information (bio, agency, verification status, specialization), as well as aggregate statistics. average rating, total number of ratings, and total view count. This is the most data-rich view in the system.

Design Decisions

- Extends `vw_article_feed`'s joins with additional LEFT JOINs to `agency` (through journalist_profile) for agency details on the author.
- LEFT JOINs `article_rating` and `article_views` to compute `AVG(rating)`, `COUNT(DISTINCT ar.id)`, and `COUNT(DISTINCT av.id)`. these are aggregated per article so a GROUP BY is required.

- Uses `ROUND(...::numeric, 2)` to cast and round the average rating to 2 decimal places before returning it to the client.
- The GROUP BY clause lists all non-aggregate selected columns, which is verbose but required since PostgreSQL enforces this strictly.

Performance & Indexing

The test query was `SELECT * FROM vw_article_detail WHERE slug = '...'`. a single article lookup by slug, as the application would do when a user navigates to an article URL.

Scenario	Before Index	After Index	Improvement
Single article by slug	~66 ms	~40 ms	~39% faster

One index was added: `idx_article_rating_article` on `article_rating(article_id)`. This allows PostgreSQL to quickly locate all ratings for a specific article during the LEFT JOIN and GROUP BY, rather than scanning the full ratings table. The improvement here was modest (66 ms → 40 ms) because the view is already doing a single-row lookup by slug. the bottleneck shifts to the aggregation joins rather than the article scan itself. The article table's existing UNIQUE constraint on `slug` already provides an efficient index for the primary lookup.

VIEW 3 vw_journalist_profile

Purpose

This view powers the journalist public profile page. the most data-intensive of all the views. It combines the journalist's personal information, agency affiliation, professional attributes (verification status, specialization, years of experience), and aggregate statistics (total articles written, published articles, and follower count) into a single denormalized result row per journalist.

Version 1. Original Design

The initial version of this view performed all aggregation in a single query using a large GROUP BY across 12 columns, combined with COUNT(DISTINCT...) and CASE WHEN expressions on directly joined tables:

- `COUNT(DISTINCT a.id)` and `COUNT(DISTINCT CASE WHEN a.status = 'published' THEN a.id END)` computed on an article LEFT JOIN.
- `COUNT(DISTINCT jf.follower_id)` computed on a journalist_followers LEFT JOIN.
- All of these were grouped by 12 columns from user, journalist_profile, and agency.

This design had two main performance problems. First, PostgreSQL had to carry all 12 GROUP BY columns through the aggregation hash, making the hash build expensive. Second, COUNT(DISTINCT CASE WHEN...)

forces PostgreSQL to track distinct values per group for a conditional expression, which is significantly more work than a simple count.

V1 Result	Before index: ~2.7 s	After index: ~2.5 s	Improvement: ~7%. negligible

Version 2. Optimized Design (Current)

The view was restructured to pre-aggregate in isolated subqueries before joining:

- An `article` subquery (aliased `ar`) groups only by `author_id` and computes `COUNT(*)` for total articles and `COUNT(*) FILTER (WHERE status = 'published')` for published articles. all within a `WHERE deleted_at IS NULL` filter.
- A `journalist_followers` subquery (aliased `jf`) groups only by `journalist_id` and computes `COUNT(*)` as follower_count.
- The outer query then becomes a pure join of one-row-per-journalist relations. no GROUP BY needed at all.
- COALESCE(..., 0) wraps each aggregate so journalists with no articles or followers return 0 instead of NULL. cleaner for the profile page UI.
- The idiomatic PostgreSQL `FILTER (WHERE...)` clause replaces the verbose `COUNT(DISTINCT CASE WHEN...)` pattern, which is both cleaner and faster.

V2 Result	Before index: ~2.0 s	After index: ~600 ms	Improvement: ~70%

Indexing

- `idx_article_author_status`. partial index on `article(author_id, status)` where `deleted_at IS NULL`. This directly supports the article subquery which filters by deleted_at and groups by author_id.
- `idx_jf_journalist_follower`. composite index on `journalist_followers(journalist_id, follower_id)`. The journalist_followers table can grow into the millions; this index makes the GROUP BY journalist_id in the follower subquery efficient.

Further Optimization Note

600 ms is still borderline for a user-facing profile page. The correct production solution would be a MATERIALIZED VIEW, refreshed on a schedule or triggered on write. This would bring lookup time down to ~1 ms at the cost of slight data staleness. which is entirely acceptable for follower counts and article totals on a profile page. The trade-off is freshness vs. query performance.

VIEW 4 vw_comment_thread

Purpose

This view powers the comment section displayed beneath each article. It returns every non-deleted comment for any given article, along with the commenter's identity (username, avatar), thread structure (parent_comment_id for nested replies), and aggregated voting data (total vote score and vote count from comment_rating).

Design Decisions

- Joins comment with user (INNER JOIN. every comment must have a user).
- LEFT JOINs comment_rating because not all comments have been rated. Without a LEFT JOIN, unrated comments would be excluded from the result.
- Uses COALESCE(SUM(cr.vote), 0) to return 0 vote score for unrated comments rather than NULL.
- Filters deleted_at IS NULL so soft-deleted comments never appear in the thread, consistent with the article feed approach.
- The parent_comment_id column is returned as-is, allowing the application layer to reconstruct the tree structure from the flat result set.

Performance & Indexing

Scenario	Before Index	After Index	Improvement
Comment thread for article (vote ordered)	~68 ms	.	No index needed

At 68 ms baseline, this view did not require additional indexing. The query filters by article_id which is a foreign key on the comment table. PostgreSQL already benefits from the FK index. Comment tables tend to be much smaller than article or view tables, so the GROUP BY and SUM(vote) aggregation is fast enough without additional optimization.

VIEW 5 vw_user_dashboard

Purpose

This view serves the authenticated user dashboard. the page a logged-in user sees when they open their account. It combines core user identity fields, notification preferences, and a live count of unread notifications into a single query, so the dashboard can be rendered from one database round-trip.

Design Decisions

- LEFT JOINs `user_prefrence` (note: misspelling is in the original schema) so users without saved preferences still appear in the result.
- LEFT JOINs `notification` to count total and unread notifications. Uses `COUNT(DISTINCT n.id)` for total and `COUNT(DISTINCT CASE WHEN n.is_read = 0 THEN n.id END)` for unread count.
- Filters `WHERE u.deleted_at IS NULL` so deactivated/deleted accounts are excluded.
- The GROUP BY is necessary due to the notification aggregation, and covers all non-aggregate user and preference columns.

Performance & Indexing

The test query was `SELECT * FROM vw_user_dashboard WHERE id = <user_id>`. a direct single-user lookup as the app would perform after login.

Scenario	Before Index	After Index	Improvement
User dashboard (single user lookup)	~1,600 ms	~15 ms	99% faster

Two indexes drove this dramatic improvement:

- `idx_user_prefrence_user` on `user_prefrence(user_id)`. Without this, PostgreSQL performs a linear sequential scan to find the user's preferences. With the index, it becomes a log-time lookup. The preference table is one-to-one with users, so this is a single-row lookup.
- `idx_notification_user_read` on `notification(user_id, is_read)`. The notification table can grow to millions of rows. This composite index allows PostgreSQL to quickly locate all notifications for a specific user and also evaluate the `is_read = 0` condition using the index directly, without scanning the full notifications table. This single index was responsible for most of the performance gain.

VIEW 6 vw_fact_check_summary

Purpose

This view provides a summarized fact-check overview for each article. It is primarily used by the admin panel and editorial dashboard to quickly see the fact-checking status of any article. how many checks have been run, how many resulted in each verdict (true, false, misleading), what the current approval status breakdown is, and when the most recent review occurred.

Design Decisions

- Filters `WHERE fc.is_active = 1` so only active (non-retracted) fact checks are included in the summary.
- Uses simple `COUNT(CASE WHEN...)` expressions (not COUNT DISTINCT) since each fact check row has a unique id and a single verdict/status value. no deduplication is necessary.
- Groups by `article_id` only. the simplest possible GROUP BY for this aggregation.
- Returns `MAX(reviewed_at)` as last_reviewed_at so admins can see at a glance whether an article has been reviewed recently.

Performance & Indexing

Scenario	Before Index	After Index	Improvement
Fact check summary for article	~68 ms	.	No index needed

This view performed well at baseline (68 ms) without requiring additional indexing. The `is_active` filter and `article_id` grouping are both served by the existing FK index on `fact_checks(article_id)`. The fact_checks table is also significantly smaller than tables like article_views or notifications, so full-scan costs are inherently lower.

VIEW 7 vw_article_metadata

Purpose

This view aggregates SEO and metadata for each article. specifically, the tags associated with it and the sources it cites. It is used by the application to build structured metadata (Open Graph tags, JSON-LD, sitemaps) and to display tag and source information on the article page. Keeping this separate from the article detail view keeps that view focused on content and avoids heavy STRING_AGG operations on every article detail load.

Design Decisions

- Uses `STRING_AGG(DISTINCT t.name, ', ')` and `STRING_AGG(DISTINCT s.url, ', ')` / `STRING_AGG(DISTINCT s.title, ', ')` to collapse multiple tags and sources into comma-separated strings. suitable for meta tag rendering without needing application-side join logic.
- DISTINCT is used inside STRING_AGG to avoid duplicates that could arise from the cross-join between article_tags and article_source.
- LEFT JOINs all four metadata tables (`article_tags`, `tag`, `article_source`, `source`) so articles with no tags or no sources still appear in the result.
- Groups by `a.id, a.slug, a.title`. the minimal set needed to uniquely identify an article and return its key identifiers.

Performance & Indexing

The test query was SELECT * FROM vw_article_metadata WHERE article_id = <id>. a single article metadata lookup.

Scenario	Before Index	After Index	Improvement
Article metadata by article_id	~230 ms	~16 ms	93% faster

One index was added: idx_article_source_article on article_source(article_id). The article_source join table did not have an index on article_id, so PostgreSQL was performing a sequential scan across it to find sources for a given article. Adding this index turned that into a direct lookup, reducing the query from 230 ms to 16 ms. a 93% improvement. The article_tags table already had an FK index on article_id from the schema definition, so no additional index was needed there.

Summary. All Views

View	Primary Use	Before	After	Index(es)
vw_article_feed	Homepage / category browse	~400 ms	~13 ms	2 indexes
vw_article_detail	Single article page	~66 ms	~40 ms	1 index
vw_journalist_profile	Journalist profile page	~2.7 s (v1) / ~2.0 s (v2)	~2.5 s (v1) / ~600 ms (v2)	2 indexes + rewrite
vw_comment_thread	Article comment section	~68 ms	~68 ms	None needed
vw_user_dashboard	User dashboard	~1,600 ms	~15 ms	2 indexes
vw_fact_check_summary	Admin fact-check panel	~68 ms	~68 ms	None needed
vw_article_metadata	SEO / article metadata	~230 ms	~16 ms	1 index