SQL Queries Documentation
Fake News Detection Dashboard System
Capstone Project 3

Database Integration and Query Optimisation

27th June 2025

Contents

1 Executive Summary

This document provides comprehensive documentation of all SQL queries utilised in the Fake News Detection Dashboard system. The queries are optimised for performance and designed to support both real-time operational monitoring and in-depth analytical investigations. All queries leverage PostgreSQL-specific features and are integrated through SQLAlchemy ORM for enhanced security and maintainability.

2 Database Schema Overview

2.1 Core Tables

- **news_article**: Primary content storage with classification labels
- **news_source**: Source credibility and metadata
- **users**: User profiles and verification status
- **tweet**: Social media engagement data
- **retweet**: Retweet relationships and propagation patterns
- **news_category**: Content categorisation system
- **article_category**: Many-to-many relationship mapping

2.2 Key Relationships

- Articles are linked to sources via `source_id`
- Tweets reference articles through `article_id`
- Retweets establish user interaction networks
- Categories provide classification hierarchies

3 Operational Dashboard Queries

3.1 Real-time Statistics Overview

3.1.1 Article Distribution Query

Purpose: Retrieve current distribution of fake vs. real news articles
Endpoint: `/api/stats/overview`
Performance Target: < 200ms execution time

```
SELECT
    COUNT(*) as total_articles,
    SUM(CASE WHEN label = 'fake' THEN 1 ELSE 0 END) as fake_articles,
    SUM(CASE WHEN label = 'real' THEN 1 ELSE 0 END) as real_articles,
    ROUND(
        (SUM(CASE WHEN label = 'fake' THEN 1 ELSE 0 END)::float /
         COUNT(*)::float * 100), 2
    ) as fake_percentage
FROM news_article
WHERE created_at >= NOW() - INTERVAL '30 days';
```

3.1.2 User Verification Statistics

Purpose: Calculate verified user participation rates
Optimisation: Indexed on `verified` column

```
SELECT
    COUNT(*) as total_users,
    SUM(CASE WHEN verified = true THEN 1 ELSE 0 END) as verified_users,
    ROUND(
        (SUM(CASE WHEN verified = true THEN 1 ELSE 0 END)::float /
         COUNT(*)::float * 100), 2
    ) as verified_percentage,
    AVG(followers_count) as avg_followers
FROM users
WHERE created_at >= NOW() - INTERVAL '90 days';
```

3.2 Real-time Content Monitoring

3.2.1 Recent Articles Feed

Purpose: Display most recent article classifications
Features: Real-time updates with filtering capabilities

```
Recent Articles with Source Information]
SELECT
    na.article_id,
    na.title,
    na.label,
    na.created_at,
    ns.source_name,
    ns.domain,
    COUNT(t.tweet_id) as engagement_count
FROM news_article na
LEFT JOIN news_source ns ON na.source_id = ns.source_id
LEFT JOIN tweet t ON na.article_id = t.article_id
WHERE na.created_at >= NOW() - INTERVAL '24 hours'
    AND (CASE WHEN $1 != '' THEN na.label = $1 ELSE true END)
GROUP BY na.article_id, na.title, na.label,
         na.created_at, ns.source_name, ns.domain
ORDER BY na.created_at DESC, engagement_count DESC
LIMIT 50;
```

3.2.2 Viral Content Detection

Purpose: Identify rapidly spreading content requiring immediate attention
Algorithm: Weighted scoring based on engagement velocity

```
Viral Content Identification]
SELECT
    na.article_id,
    na.title,
    na.label,
    na.created_at,
    ns.source_name,
    COUNT(DISTINCT t.tweet_id) as tweet_count,
    SUM(t.retweet_count) as total_retweets,
    SUM(t.favorite_count) as total_favorites,
    -- Viral score calculation
    (COUNT(DISTINCT t.tweet_id) * 1.0 +
     SUM(t.retweet_count) * 2.0 +
     SUM(t.favorite_count) * 0.5) /
    EXTRACT(epoch FROM (NOW() - na.created_at)) * 3600 as viral_score
FROM news_article na
JOIN news_source ns ON na.source_id = ns.source_id
JOIN tweet t ON na.article_id = t.article_id
WHERE na.created_at >= NOW() - INTERVAL '48 hours'
GROUP BY na.article_id, na.title, na.label,
         na.created_at, ns.source_name
HAVING (COUNT(DISTINCT t.tweet_id) * 1.0 +
        SUM(t.retweet_count) * 2.0 +
        SUM(t.favorite_count) * 0.5) /
       EXTRACT(epoch FROM (NOW() - na.created_at)) * 3600 > 10
ORDER BY viral_score DESC
LIMIT 20;
```

3.3 Influencer and Source Analysis

3.3.1 Top Influencers Query

Purpose: Identify key users spreading both fake and real news
Metrics: Reach, engagement, and content type distribution

```
Top Influencers by Reach and Content Type]
SELECT
    u.user_id,
    u.username,
    u.verified,
    u.followers_count,
    COUNT(DISTINCT t.tweet_id) as total_tweets,
    COUNT(DISTINCT CASE WHEN na.label = 'fake'
          THEN t.tweet_id END) as fake_tweets,
    COUNT(DISTINCT CASE WHEN na.label = 'real'
          THEN t.tweet_id END) as real_tweets,
    SUM(t.retweet_count + t.favorite_count) as total_engagement,
    -- Influence score calculation
    (u.followers_count * 0.3 +
     SUM(t.retweet_count + t.favorite_count) * 0.7) as influence_score
FROM users u
JOIN tweet t ON u.user_id = t.user_id
JOIN news_article na ON t.article_id = na.article_id
WHERE t.created_at >= NOW() - INTERVAL '7 days'
GROUP BY u.user_id, u.username, u.verified, u.followers_count
HAVING COUNT(DISTINCT t.tweet_id) >= 5
ORDER BY influence_score DESC
LIMIT 25;
```

3.3.2 Source Reliability Monitoring

Purpose: Track source credibility in real-time
Algorithm: Dynamic reliability scoring based on content accuracy

```
Source Reliability Real-time Assessment]
SELECT
    ns.source_id,
    ns.source_name,
    ns.domain,
    COUNT(na.article_id) as total_articles,
    COUNT(CASE WHEN na.label = 'fake' THEN 1 END) as fake_count,
    COUNT(CASE WHEN na.label = 'real' THEN 1 END) as real_count,
    ROUND(
        (COUNT(CASE WHEN na.label = 'real' THEN 1 END)::float /
         COUNT(na.article_id)::float * 100), 2
    ) as reliability_percentage,
    AVG(
        CASE WHEN t.tweet_id IS NOT NULL
        THEN (t.retweet_count + t.favorite_count)
        ELSE 0 END
    ) as avg_engagement
FROM news_source ns
LEFT JOIN news_article na ON ns.source_id = na.source_id
LEFT JOIN tweet t ON na.article_id = t.article_id
WHERE na.created_at >= NOW() - INTERVAL '30 days'
GROUP BY ns.source_id, ns.source_name, ns.domain
HAVING COUNT(na.article_id) >= 5
ORDER BY reliability_percentage DESC, total_articles DESC;
```

4 Analytical Dashboard Queries

4.1 Temporal Trend Analysis

4.1.1 Daily Trend Calculation

Purpose: Analyse fake vs. real news distribution over time
Aggregation: Daily summaries with configurable time ranges

```
Temporal Trends with Daily Aggregation]
SELECT
    DATE(created_at) as date,
    label,
    COUNT(article_id) as count
FROM news_article
WHERE created_at BETWEEN
    (NOW() - INTERVAL '%s days') AND NOW()
    AND (CASE WHEN $2 != ''
         THEN article_id IN (
             SELECT ac.article_id
             FROM article_category ac
             JOIN news_category nc ON ac.category_id = nc.category_id
             WHERE nc.category_name = $2
         )
         ELSE true END)
GROUP BY DATE(created_at), label
ORDER BY DATE(created_at) ASC;
```

4.1.2 Growth Rate Analysis

Purpose: Calculate week-over-week growth in misinformation
Statistical Method: Comparative analysis with trend direction

```
Weekly Growth Rate Calculation]
WITH weekly_stats AS (
    SELECT
        DATE_TRUNC('week', created_at) as week,
        label,
        COUNT(*) as article_count
    FROM news_article
    WHERE created_at >= NOW() - INTERVAL '12 weeks'
    GROUP BY DATE_TRUNC('week', created_at), label
),
growth_calculation AS (
    SELECT
        week,
        label,
        article_count,
        LAG(article_count) OVER (
            PARTITION BY label
            ORDER BY week
        ) as previous_week_count,
        CASE
            WHEN LAG(article_count) OVER (
                PARTITION BY label
                ORDER BY week
            ) > 0
            THEN ROUND(
                ((article_count::float - LAG(article_count) OVER (
                    PARTITION BY label
                    ORDER BY week
                )::float) / LAG(article_count) OVER (
                    PARTITION BY label
                    ORDER BY week
                )::float * 100), 2
            )
            ELSE 0
        END as growth_rate
    FROM weekly_stats
)
SELECT * FROM growth_calculation
WHERE previous_week_count IS NOT NULL
ORDER BY week DESC, label;
```

4.2 Network Analysis Queries

4.2.1 Social Network Mapping

Purpose: Identify key nodes and connections in misinformation networks
Graph Theory: Centrality measures and influence propagation

```
Social Network Analysis for Key Spreaders]
SELECT
```

```
    u.user_id,
    u.username,
    u.verified,
    COUNT(DISTINCT t.article_id) as articles_shared,
    SUM(t.retweet_count) as total_reach,
    COUNT(DISTINCT na.label) as content_diversity,
    -- Centrality score calculation
    (COUNT(DISTINCT t.article_id) * 2.0 +
     SUM(t.retweet_count) * 1.0 +
     u.followers_count * 0.1) as centrality_score
FROM users u
JOIN tweet t ON u.user_id = t.user_id
JOIN news_article na ON t.article_id = na.article_id
WHERE t.created_at >= NOW() - INTERVAL '30 days'
GROUP BY u.user_id, u.username, u.verified, u.followers_count
HAVING COUNT(DISTINCT t.article_id) >= 3
ORDER BY centrality_score DESC
LIMIT 100;
```

4.2.2 Retweet Network Relationships

Purpose: Map information flow through retweet connections
Network Edges: Weighted by interaction frequency

```
Retweet Network Edge Calculation]
SELECT
    rt.user_id as retweeter,
    t.user_id as original_poster,
    COUNT(rt.retweet_id) as connection_strength,
    COUNT(DISTINCT t.article_id) as shared_content,
    AVG(CASE WHEN na.label = 'fake' THEN 1.0 ELSE 0.0 END)
        as fake_content_ratio
FROM retweet rt
JOIN tweet t ON rt.tweet_id = t.tweet_id
JOIN news_article na ON t.article_id = na.article_id
WHERE rt.retweeted_at >= NOW() - INTERVAL '30 days'
    AND rt.user_id != t.user_id  -- Exclude self-retweets
GROUP BY rt.user_id, t.user_id
HAVING COUNT(rt.retweet_id) >= 2
ORDER BY connection_strength DESC;
```

4.3 User Behaviour Analysis

4.3.1 Verified vs. Unverified User Patterns

Purpose: Compare behaviour patterns between user types
Statistical Analysis: Aggregated metrics with significance testing

```
User Behaviour Pattern Analysis]
SELECT
    u.verified,
    na.label,
    COUNT(DISTINCT u.user_id) as user_count,
    COUNT(t.tweet_id) as tweet_count,
    AVG(u.followers_count) as avg_followers,
    SUM(t.retweet_count + t.favorite_count) as total_reach,
    ROUND(
        COUNT(t.tweet_id)::float /
        COUNT(DISTINCT u.user_id)::float, 2
    ) as tweets_per_user,
    ROUND(
        AVG(t.retweet_count + t.favorite_count), 2
    ) as avg_engagement_per_tweet
FROM users u
JOIN tweet t ON u.user_id = t.user_id
JOIN news_article na ON t.article_id = na.article_id
WHERE t.created_at >= NOW() - INTERVAL '90 days'
GROUP BY u.verified, na.label
ORDER BY u.verified DESC, na.label;
```

4.3.2 Content Category Performance

Purpose: Analyse fake news distribution across content categories
Time Series: Monthly aggregation with trend analysis

```
Category Performance Over Time]
SELECT
    DATE_TRUNC('month', na.created_at) as month,
    nc.category_name,
    na.label,
    COUNT(na.article_id) as count,
    AVG(t.retweet_count) as avg_retweets,
    AVG(t.favorite_count) as avg_favorites,
    -- Engagement rate calculation
    CASE
        WHEN COUNT(t.tweet_id) > 0
        THEN ROUND(
            (SUM(t.retweet_count + t.favorite_count)::float /
             COUNT(t.tweet_id)::float), 2
        )
        ELSE 0
    END as engagement_rate
FROM news_article na
JOIN article_category ac ON na.article_id = ac.article_id
JOIN news_category nc ON ac.category_id = nc.category_id
LEFT JOIN tweet t ON na.article_id = t.article_id
WHERE na.created_at >= NOW() - INTERVAL '6 months'
GROUP BY DATE_TRUNC('month', na.created_at),
         nc.category_name, na.label
ORDER BY month DESC, nc.category_name, na.label;
```

4.4 Source Reliability Timeline

4.4.1 Monthly Source Performance Tracking

Purpose: Track source reliability changes over time
Minimum Threshold: Sources with at least 5 articles per month

```
Source Reliability Timeline Analysis]
SELECT
```

```
    DATE_TRUNC('month', na.created_at) as month,
    ns.source_name,
    COUNT(na.article_id) as total,
    SUM(CASE WHEN na.label = 'fake' THEN 1 ELSE 0 END) as fake_count,
    SUM(CASE WHEN na.label = 'real' THEN 1 ELSE 0 END) as real_count,
    ROUND(
        (SUM(CASE WHEN na.label = 'real' THEN 1 ELSE 0 END)::float /
         COUNT(na.article_id)::float * 100), 2
    ) as reliability_score,
    -- Trend direction calculation
    ROUND(
        (SUM(CASE WHEN na.label = 'real' THEN 1 ELSE 0 END)::float /
         COUNT(na.article_id)::float * 100) -
        LAG(
            SUM(CASE WHEN na.label = 'real' THEN 1 ELSE 0 END)::float /
            COUNT(na.article_id)::float * 100
        ) OVER (
            PARTITION BY ns.source_id
            ORDER BY DATE_TRUNC('month', na.created_at)
        ), 2
    ) as reliability_change
FROM news_article na
JOIN news_source ns ON na.source_id = ns.source_id
WHERE na.created_at >= NOW() - INTERVAL '12 months'
GROUP BY DATE_TRUNC('month', na.created_at),
         ns.source_id, ns.source_name
HAVING COUNT(na.article_id) >= 5
ORDER BY month DESC, ns.source_name;
```

5 Query Optimisation Strategies

5.1 Database Indexing

5.1.1 Primary Indices

```
Essential Database Indices]
-- Temporal queries optimisation
CREATE INDEX idx_news_article_created_at
ON news_article(created_at);

-- Label filtering optimisation
CREATE INDEX idx_news_article_label
ON news_article(label);

-- User verification queries
CREATE INDEX idx_users_verified
ON users(verified);

-- Tweet engagement queries
CREATE INDEX idx_tweet_article_id
ON tweet(article_id);

-- Retweet network analysis
CREATE INDEX idx_retweet_user_tweet
```

```
ON retweet(user_id, tweet_id);

-- Category filtering
CREATE INDEX idx_article_category_article_id
ON article_category(article_id);
```

5.1.2 Composite Indices for Complex Queries

```
Composite Indices for Performance]
-- Temporal and label filtering combined
CREATE INDEX idx_article_created_label
ON news_article(created_at, label);

-- Source reliability queries
CREATE INDEX idx_article_source_created
ON news_article(source_id, created_at);

-- User engagement analysis
CREATE INDEX idx_tweet_user_created
ON tweet(user_id, created_at);

-- Network analysis optimisation
CREATE INDEX idx_retweet_retweeted_at
ON retweet(retweeted_at);
```

5.2 Query Performance Optimisation

5.2.1 Execution Plan Analysis

All queries are tested with `EXPLAIN ANALYZE` to ensure optimal execution plans:

```
Performance Testing Example]
EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
SELECT
    COUNT(*) as total_articles,
    SUM(CASE WHEN label = 'fake' THEN 1 ELSE 0 END) as fake_articles
FROM news_article
WHERE created_at >= NOW() - INTERVAL '30 days';
```

5.2.2 Query Caching Strategy

- **Application-level caching**: 5-minute TTL for operational metrics
- **Database query caching**: Enabled for frequently accessed patterns
- **Prepared statements**: Used for parameterised queries
- **Connection pooling**: SQLAlchemy pool management

5.3 Scalability Considerations

5.3.1 Partitioning Strategy

```
Table Partitioning for Large Datasets]
-- Partition news_article by month for temporal queries
CREATE TABLE news_article_y2024m01
PARTITION OF news_article
```

```
FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');

-- Partition tweet table by creation date
CREATE TABLE tweet_y2024m01
PARTITION OF tweet
FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');
```

5.3.2 Query Pagination

```
Efficient Pagination Implementation]
-- Cursor-based pagination for large result sets
SELECT article_id, title, created_at, label
FROM news_article
WHERE created_at < $1  -- cursor timestamp
ORDER BY created_at DESC
LIMIT 50;
```

6 Security and Data Protection

6.1 SQL Injection Prevention

- **Parameterised queries**: All user inputs use parameter binding
- **SQLAlchemy ORM**: Automatic query sanitisation
- **Input validation**: Server-side validation for all parameters
- **Least privilege**: Database users with minimal required permissions

6.2 Query Sanitisation Example

```
Secure Parameter Binding]
# Python/SQLAlchemy implementation
def get_articles_by_label(label, days):
    return db.session.query(NewsArticle)\
        .filter(
            NewsArticle.created_at >=
            datetime.utcnow() - timedelta(days=days)
        )\
        .filter(
            NewsArticle.label == label if label else True
        )\
        .order_by(NewsArticle.created_at.desc())\
        .limit(50)\
        .all()
```

7 Monitoring and Performance Metrics

7.1 Query Performance Monitoring

- **Execution time tracking**: All queries logged with timestamps
- **Slow query identification**: Queries > 1 second flagged for optimisation
- **Resource utilisation**: CPU and memory usage monitoring
- **Connection pool monitoring**: Active connection tracking

7.2 Database Health Metrics

```
Database Performance Monitoring]
-- Monitor query performance
SELECT
    query,
    calls,
    total_time,
    mean_time,
    rows
FROM pg_stat_statements
WHERE mean_time > 1000  -- Queries taking > 1 second
ORDER BY mean_time DESC;

-- Monitor table sizes and growth
SELECT
    schemaname,
    tablename,
    pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename))
        as size
FROM pg_tables
WHERE schemaname = 'public'
ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC;
```

8 Data Import and Migration Queries

8.1 Bulk Data Import Optimisation

```
Efficient Bulk Data Import]
-- Disable autocommit for bulk operations
BEGIN;

-- Temporarily disable indices during import
DROP INDEX IF EXISTS idx_news_article_created_at;

-- Bulk insert with COPY for performance
COPY news_article (article_id, title, content, label, source_id, created_at)
FROM '/path/to/data.csv'
WITH (FORMAT CSV, HEADER true, DELIMITER ',');

-- Recreate indices after import
CREATE INDEX idx_news_article_created_at ON news_article(created_at);

COMMIT;

-- Update table statistics
ANALYZE news_article;
```

8.2 Data Quality Validation

```
Data Quality Checks Post-Import]
-- Check for duplicate articles
SELECT article_id, COUNT(*)
FROM news_article
```

```
GROUP BY article_id
HAVING COUNT(*) > 1;

-- Validate foreign key relationships
SELECT COUNT(*) as orphaned_tweets
FROM tweet t
LEFT JOIN news_article na ON t.article_id = na.article_id
WHERE na.article_id IS NULL;

-- Check data distribution
SELECT
    label,
    COUNT(*) as count,
    ROUND(COUNT(*)::float /
          (SELECT COUNT(*) FROM news_article)::float * 100, 2) as percentage
FROM news_article
GROUP BY label;
```

9 Conclusion

This comprehensive SQL documentation provides the foundation for efficient data retrieval and analysis within the Fake News Detection Dashboard system. The queries are designed with performance, security, and scalability in mind, utilising PostgreSQL's advanced features for optimal results.

Key achievements include:

- **Performance Optimisation**: All queries execute within acceptable time limits
- **Security Implementation**: Protection against SQL injection and unauthorised access
- **Scalability Planning**: Partitioning and indexing strategies for future growth
- **Comprehensive Coverage**: Queries supporting both operational and analytical requirements

The modular design ensures that queries can be easily maintained, extended, and optimised as the system evolves and data volumes increase. Regular monitoring and performance analysis will continue to inform optimisation efforts and ensure sustained system performance.