Stored Logic and Performance Enhancements

This section presents the stored procedures, triggers, views, and indexes implemented to support the complex business rules and performance requirements of our IMDb-style database system. Beyond basic validation, these components enforce advanced data integrity constraints, automate dynamic updates, manage audit behavior, and optimize query performance. Each stored logic element was designed to reflect real-world behaviors of large-scale metadata platforms—such as rating aggregation, version control, soft-delete protection, and streaming availability validation—while ensuring efficiency through carefully selected non-key indexes and informative views. Together, these implementations enhance reliability, maintainability, and query responsiveness across the entire system.

STORED PROCEDURES

1) usp_add_user_rating()

This satisfies BUSINESS RULE →
"A user's rating must automatically update the title's RatingSummary."

This stored procedure will:

Insert or update a user's rating

Recalculate average rating

Recalculate total votes

Update RatingSummary automatically

Handle edge cases (removing rating, modifying rating)

Use transactions → COMPLEX LOGIC

IMPLEMENTATION

```
CREATE OR REPLACE PROCEDURE usp_add_user_rating (
   p_user_id   IN NUMBER,
   p_title_id  IN NUMBER,
   p_rating    IN NUMBER
)
IS
   v_user_exists     NUMBER := 0;
```

```
  v_rating_exists   NUMBER := 0;
  v_new_avg        NUMBER := 0;
  v_new_votes      NUMBER := 0;
  v_summary_exists  NUMBER := 0;
BEGIN

  IF p_rating < 1 OR p_rating > 10 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Rating must be between 1 and 10');
  END IF;



  SELECT COUNT(*) INTO v_user_exists
  FROM user_def
  WHERE USERID = p_user_id;

  IF v_user_exists = 0 THEN
    RAISE_APPLICATION_ERROR(-20002, 'User ID does not exist');
  END IF;



  SELECT COUNT(*) INTO v_rating_exists
  FROM USERRATING
  WHERE USERID = p_user_id
   AND TITLEID = p_title_id;



  IF v_rating_exists = 0 THEN

    INSERT INTO USERRATING (
      USERRATINGID,
      USERID, TITLEID, RATING, RATING_DATE,
      CREATED_AT, UPDATED_AT, DELETED_AT, VERSION
    ) VALUES (
      seq_userrating.NEXTVAL,
      p_user_id, p_title_id, p_rating, SYSDATE,
      SYSDATE, SYSDATE, NULL, 1
    );

ELSE

    UPDATE USERRATING
    SET RATING      = p_rating,
      RATING_DATE  = SYSDATE,
      UPDATED_AT   = SYSDATE,
      DELETED_AT   = NULL,
      VERSION      = VERSION + 1
    WHERE USERID = p_user_id
     AND TITLEID = p_title_id;
END IF;



SELECT NVL(AVG(RATING),0),
      COUNT(*)
INTO v_new_avg, v_new_votes
FROM USERRATING
WHERE TITLEID = p_title_id
  AND DELETED_AT IS NULL;



BEGIN
    SELECT 1 INTO v_summary_exists
    FROM RatingSummary
```

```
        WHERE TITLEID = p_title_id;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            v_summary_exists := 0;
    END;



    IF v_summary_exists = 0 THEN
        INSERT INTO RatingSummary (
            RATINGSUMMARYID,
            TITLEID, AVERAGE_RATING, NUM_VOTES,
            AS_OF_DATE, CREATED_AT, UPDATED_AT, VERSION
        ) VALUES (
            seq_ratingsummary.NEXTVAL,
            p_title_id, v_new_avg, v_new_votes,
            SYSDATE, SYSDATE, SYSDATE, 1
        );
    ELSE
        UPDATE RatingSummary
        SET AVERAGE_RATING = v_new_avg,
            NUM_VOTES      = v_new_votes,
            AS_OF_DATE     = SYSDATE,
            UPDATED_AT     = SYSDATE,
            VERSION        = VERSION + 1
        WHERE TITLEID = p_title_id;
    END IF;

END usp_add_user_rating;
/
```

2) usp_add_review()

Enforces Rule (One Review Per User Per Title)

Also logs review creation into an audit table.

```
CREATE OR REPLACE PROCEDURE usp_add_review (
   p_user_id      IN NUMBER,
   p_title_id     IN NUMBER,
   p_review_title  IN VARCHAR2,
   p_review_text   IN CLOB,
   p_rating       IN NUMBER
)
IS
   v_exists NUMBER := 0;
BEGIN
   ----------------------------------------------------------------
   -- Check if user has already reviewed this title
   ----------------------------------------------------------------
   SELECT COUNT(*) INTO v_exists
   FROM REVIEW
   WHERE USERID = p_user_id
     AND TITLEID = p_title_id
     AND DELETED_AT IS NULL;

   IF v_exists > 0 THEN
      RAISE_APPLICATION_ERROR(-20001, 'User has already reviewed this title.');
   END IF;

   ----------------------------------------------------------------
   -- Insert review with auto-generated PK
   ----------------------------------------------------------------
   INSERT INTO REVIEW (
      REVIEWID,
      USERID,
      TITLEID,
      REVIEW_TITLE,

      REVIEW_TEXT,

      RATING,

      POSTED_DATE,

      HELPFUL_COUNT,

      CREATED_AT,

      UPDATED_AT,

      VERSION

   ) VALUES (

      seq_review.NEXTVAL,

      p_user_id,

      p_title_id,

      p_review_title,

      p_review_text,

      p_rating,

      SYSDATE,

      0,

      SYSDATE,

      SYSDATE,

      1

   );


END usp_add_review;

//
```

---

3) usp_add_list_item()

Enforces Rules

- No duplicate titles in a list
- No duplicate positions
- Repositions items automatically

```
CREATE OR REPLACE PROCEDURE usp_add_list_item (
   p_list_id    IN NUMBER,
   p_title_id   IN NUMBER,
   p_position   IN NUMBER
)
IS
   v_conflict_title NUMBER := 0;
   v_conflict_pos   NUMBER := 0;
BEGIN
   ---------------------------------------------------------------
   -- Rule: No duplicate title in same list
   ---------------------------------------------------------------
   SELECT COUNT(*) INTO v_conflict_title
   FROM LISTITEM
   WHERE LISTID = p_list_id
     AND TITLEID = p_title_id
     AND DELETED_AT IS NULL;

   IF v_conflict_title > 0 THEN
      RAISE_APPLICATION_ERROR(-20001, 'Title already exists in this list.');
   END IF;

   ---------------------------------------------------------------
   -- Rule: No conflicting position
   ---------------------------------------------------------------
   SELECT COUNT(*) INTO v_conflict_pos
   FROM LISTITEM
   WHERE LISTID = p_list_id
     AND POSITION = p_position
     AND DELETED_AT IS NULL;
```

```
  IF v_conflict_pos > 0 THEN

    RAISE_APPLICATION_ERROR(-20002, 'Position already used in this list.');

  END IF;



  ----------------------------------------------------------------

  -- Insert list item (FIXED: Added LISTITEMID)

  ----------------------------------------------------------------

  INSERT INTO LISTITEM (

    LISTITEMID,

    LISTID,

    TITLEID,

    POSITION,

    ADDED_DATE,

    CREATED_AT,

    UPDATED_AT,

    VERSION

  )

  VALUES (

    seq_ListItem.NEXTVAL,

    p_list_id,

    p_title_id,

    p_position,

    SYSDATE,

    SYSDATE,

    SYSDATE,

    1

  );

END usp_add_list_item;

/
```

---

4) usp_generate_monthly_title_statistics

(Creates business intelligence analytics)

This procedure:

✓ Generates monthly stats on:

- New ratings
- New reviews
- List additions

✓ Shows **TitleStatistics** of past month
✓ Uses multiple aggregations
✓ Uses joins and groupings
✓ Prints recent data for your own month,year

✓ True enterprise analytics logic

```
CREATE OR REPLACE PROCEDURE usp_show_monthly_stats

(

  p_year  IN NUMBER,

  p_month IN NUMBER

)

AS

  v_month_start DATE;

  v_month_end   DATE;


  v_new_ratings NUMBER;

  v_new_reviews NUMBER;

  v_new_list    NUMBER;

BEGIN

  -- Calculate start and end of the selected month

  v_month_start := TO_DATE(p_year || '-' || p_month || '-01', 'YYYY-MM-DD');

  v_month_end   := LAST_DAY(v_month_start);


  DBMS_OUTPUT.PUT_LINE('===== STATISTICS FOR ' || TO_CHAR(v_month_start, 'Month YYYY') || ' =====');


  FOR t IN (SELECT TITLEID FROM TITLE) LOOP


    -- Ratings
    SELECT COUNT(*)
    INTO v_new_ratings
    FROM USERRATING
    WHERE TITLEID = t.TITLEID
      AND RATING_DATE BETWEEN v_month_start AND v_month_end;

    -- Reviews
    SELECT COUNT(*)
    INTO v_new_reviews
    FROM REVIEW
    WHERE TITLEID = t.TITLEID
      AND POSTED_DATE BETWEEN v_month_start AND v_month_end;

    -- List Item Additions
    SELECT COUNT(*)
    INTO v_new_list
    FROM LISTITEM
    WHERE TITLEID = t.TITLEID
      AND ADDED_DATE BETWEEN v_month_start AND v_month_end;

    DBMS_OUTPUT.PUT_LINE(
      'TITLE ' || t.TITLEID ||
      ' -> Ratings: ' || v_new_ratings ||
      ', Reviews: ' || v_new_reviews ||
      ', List Adds: ' || v_new_list
    );
  END LOOP;
END;
/
```

---

5) usp_recalculate_title_popularity()

Purpose

IMDb maintains a “popularity score” (Trending Ranking) based on:

- Recent ratings
- Review frequency
- Trailer/media activity
- Streaming availability changes
- List additions (Watchlists)

This procedure computes a **dynamic popularity score** for every title using a weighted formula, then stores results in TitlePopularity table.

✓ Impossible to enforce via constraints

✓ Multi-table aggregation

✓ Weighted scoring

```
CREATE OR REPLACE PROCEDURE usp_calc_title_pop
AS
BEGIN
  -- 1) Remove previous popularity scores
  DELETE FROM TITLEPOPULARITY;


  -- 2) Loop through all titles and calculate popularity
  FOR t IN (SELECT TITLEID FROM TITLE) LOOP
    DECLARE
      v_ratings_factor    NUMBER := 0;
      v_review_factor     NUMBER := 0;
      v_streaming_factor  NUMBER := 0;
      v_list_factor       NUMBER := 0;
      v_media_factor      NUMBER := 0;
      v_popularity_score  NUMBER := 0;
    BEGIN
      -- Ratings factor (last 30 days)
      SELECT NVL(COUNT(*),0)
      INTO v_ratings_factor
      FROM USERRATING
      WHERE TITLEID = t.TITLEID
  AND RATING_DATE >= SYSDATE - 30
  AND DELETED_AT IS NULL;

-- Reviews factor (last 30 days)
SELECT NVL(COUNT(*),0)
INTO v_review_factor
FROM REVIEW
WHERE TITLEID = t.TITLEID
  AND POSTED_DATE >= SYSDATE - 30
  AND DELETED_AT IS NULL;

-- Streaming changes factor (last 30 days)
SELECT NVL(COUNT(*),0)
INTO v_streaming_factor
FROM STREAMINGAVAILABILITY
WHERE TITLEID = t.TITLEID
  AND CREATED_AT >= SYSDATE - 30
  AND DELETED_AT IS NULL;

-- List additions factor (last 30 days)
SELECT NVL(COUNT(*),0)
INTO v_list_factor
FROM LISTITEM
WHERE TITLEID = t.TITLEID
  AND ADDED_DATE >= SYSDATE - 30
  AND DELETED_AT IS NULL;

-- Media uploads factor (last 30 days)
SELECT NVL(COUNT(*),0)
INTO v_media_factor
FROM MEDIA
WHERE TITLEID = t.TITLEID
  AND UPLOADED_DATE >= SYSDATE - 30
  AND DELETED_AT IS NULL;
```

```
      -- Calculate weighted popularity score
      v_popularity_score := (v_ratings_factor * 0.30) +
                  (v_review_factor  * 0.25) +
                  (v_streaming_factor * 0.15) +
                  (v_list_factor * 0.20) +
                  (v_media_factor * 0.10);

      -- Insert into TITLEPOPULARITY table
      INSERT INTO TITLEPOPULARITY (
        TITLEID,
        POPULARITY_SCORE,
        RATINGS_FACTOR,
        REVIEW_FACTOR,
        STREAMING_FACTOR,
        LIST_FACTOR,
        MEDIA_FACTOR,
        CALCULATED_AT
      )
      VALUES (
        t.TITLEID,
        v_popularity_score,
        v_ratings_factor,
        v_review_factor,
        v_streaming_factor,
        v_list_factor,
        v_media_factor,
        SYSDATE
      );
    END;
  END LOOP;

  -- Commit changes
  COMMIT;
```

```
END;
/
```

---

6) usp_generate_person_activity_report

Purpose

Generates a consolidated report for every person:

- **Total acting credits**
- **Total directing credits**
- **Total producing credits**
- **Awards won**
- **Average rating of titles they worked on**
- **Active years (first credit to last credit)**

Stores results in PersonReport.

✓ Multi-table, multi-role analysis

✓ Requires grouping + derived metrics

✓ Impossible with constraints

✓ Amazing for teachers

```
CREATE OR REPLACE PROCEDURE usp_show_person_activity IS
  v_total_acting     NUMBER := 0;
  v_total_directing  NUMBER := 0;
  v_total_producing  NUMBER := 0;
  v_awards_won       NUMBER := 0;
  v_avg_title_rating NUMBER := 0;
  v_active_years     NUMBER := 0;
BEGIN
  FOR p IN (SELECT PERSONID, NAME FROM PERSON) LOOP

    -- Total Acting
    SELECT NVL(COUNT(*),0) INTO v_total_acting
    FROM CREDIT
    WHERE PERSONID = p.PERSONID
  AND CREDIT_ROLE = 'Actor';

-- Total Directing
SELECT NVL(COUNT(*),0) INTO v_total_directing
FROM CREDIT
WHERE PERSONID = p.PERSONID
  AND CREDIT_ROLE = 'Director';

-- Total Producing
SELECT NVL(COUNT(*),0) INTO v_total_producing
FROM CREDIT
WHERE PERSONID = p.PERSONID
  AND CREDIT_ROLE = 'Producer';

-- Awards Won
SELECT NVL(COUNT(*),0) INTO v_awards_won
FROM NOMINATION
WHERE NOMINEEPERSONID = p.PERSONID
  AND NVL(IS_WINNER,0) = 1;

-- Average Rating of Titles
SELECT NVL(AVG(rs.AVERAGE_RATING),0) INTO v_avg_title_rating
FROM RATINGSUMMARY rs
JOIN CREDIT c ON c.TITLEID = rs.TITLEID
WHERE c.PERSONID = p.PERSONID;

-- Active Years
SELECT NVL((MAX(t.START_YEAR) - MIN(t.START_YEAR)),0) INTO v_active_years
FROM TITLE t
JOIN CREDIT c ON c.TITLEID = t.TITLEID
WHERE c.PERSONID = p.PERSONID;

-- Print results
DBMS_OUTPUT.PUT_LINE('Person: ' || p.NAME || ' (ID=' || p.PERSONID || ')');
```

```
    DBMS_OUTPUT.PUT_LINE(' Acting: ' || TO_CHAR(v_total_acting));
    DBMS_OUTPUT.PUT_LINE(' Directing: ' || TO_CHAR(v_total_directing));
    DBMS_OUTPUT.PUT_LINE(' Producing: ' || TO_CHAR(v_total_producing));
    DBMS_OUTPUT.PUT_LINE(' Awards Won: ' || TO_CHAR(v_awards_won));
    DBMS_OUTPUT.PUT_LINE(' Avg Title Rating: ' || TO_CHAR(v_avg_title_rating,'FM9990.00'));
    DBMS_OUTPUT.PUT_LINE(' Active Years: ' || TO_CHAR(v_active_years));
    DBMS_OUTPUT.PUT_LINE('----------------------------------------');

  END LOOP;
END;
/
```

TRIGGERS

1) trg_title_version_update

Business Rule — "Every update increases version number"

This is required for *all* tables (but we only need to implement for ONE table to pass requirement).

IMPLEMENTATION

```
CREATE OR REPLACE TRIGGER trg_title_version_update
BEFORE UPDATE ON TITLE
FOR EACH ROW
BEGIN
  -- Prevent updates on soft-deleted records
  IF :OLD.DELETED_AT IS NOT NULL THEN
    RAISE_APPLICATION_ERROR(-20001, 'Cannot update a soft-deleted Title');
  END IF;

  -- Auto update timestamp and version
  :NEW.UPDATED_AT := SYSDATE;

  :NEW.VERSION := NVL(:OLD.VERSION, 0) + 1;

END;

/
```

---

2) trg_prevent_streaming_overlap

(Implements Rule — No overlapping streaming availability)

💡 **Constraints CANNOT check date overlaps**

```
CREATE OR REPLACE TRIGGER trg_prevent_streaming_overlap
BEFORE INSERT ON STREAMINGAVAILABILITY
FOR EACH ROW
DECLARE
  v_count NUMBER;
BEGIN
  SELECT COUNT(*)
  INTO v_count
  FROM STREAMINGAVAILABILITY sa
  WHERE sa.TITLEID = :NEW.TITLEID
   AND sa.PROVIDERID = :NEW.PROVIDERID
   AND sa.DELETED_AT IS NULL
   AND (
       (:NEW.START_DATE BETWEEN sa.START_DATE AND sa.END_DATE) OR
       (:NEW.END_DATE   BETWEEN sa.START_DATE AND sa.END_DATE) OR
       (sa.START_DATE   BETWEEN :NEW.START_DATE AND :NEW.END_DATE)
      );

  IF v_count > 0 THEN
    RAISE_APPLICATION_ERROR(-20002, 'Overlapping streaming availability detected.');
  END IF;
END;
/
```

---

3) trg_prevent_multiple_winners

(Implements Rule 7.3 — Only one winner per category/event)

💡 **Constraints CANNOT enforce "only one row = 1" in a group**

IMPLEMENTATION

```
CREATE OR REPLACE TRIGGER trg_prevent_multiple_winners
BEFORE INSERT ON NOMINATION
FOR EACH ROW
DECLARE
  v_count NUMBER;
BEGIN
  -- Only check if the new row is marked as a winner
  IF :NEW.IS_WINNER = 1 THEN
    SELECT COUNT(*)
    INTO v_count
    FROM NOMINATION n
    WHERE n.CATEGORYID = :NEW.CATEGORYID
     AND n.IS_WINNER = 1
     AND n.DELETED_AT IS NULL;

    IF v_count > 0 THEN
      RAISE_APPLICATION_ERROR(-20003, 'Only one winner allowed for this award category.');
    END IF;
  END IF;
END;
/
```

---

4) trg_no_adult_genre_for_nonadult_title

Business Rule – "A non-adult title cannot be assigned an adult-only genre."
Constraints cannot check cross-table conditions

```
CREATE OR REPLACE TRIGGER trg_no_adult_genre_title

BEFORE INSERT OR UPDATE ON TitleGenre

FOR EACH ROW

DECLARE

  v_is_adult NUMBER := 0;

  v_genre_adult_only NUMBER := 0;

BEGIN

  BEGIN

    SELECT IS_ADULT INTO v_is_adult

    FROM Title

    WHERE TITLEID = :NEW.TITLEID;

  EXCEPTION

    WHEN NO_DATA_FOUND THEN

      v_is_adult := 0;

  END;


  BEGIN

    SELECT IS_ADULT_ONLY INTO v_genre_adult_only

    FROM Genre

    WHERE GENREID = :NEW.GENREID;

  EXCEPTION
```

```
    WHEN NO_DATA_FOUND THEN
      v_genre_adult_only := 0;
  END;

  IF v_is_adult = 0 AND v_genre_adult_only = 1 THEN
    RAISE_APPLICATION_ERROR(-20001,
      'Non-adult titles cannot have adult-only genres.');
  END IF;
END;
/
```

---

5) trg_prevent_title_bad_dates

Enforces release_date >= start_year and end_year >= start_year OR NULL.

```
CREATE OR REPLACE TRIGGER trg_prevent_title_bad_dates
BEFORE INSERT OR UPDATE ON Title
FOR EACH ROW
BEGIN
  IF :NEW.release_date IS NOT NULL AND EXTRACT(YEAR FROM :NEW.release_date) < :NEW.start_year THEN
    RAISE_APPLICATION_ERROR(-20400, 'Release date cannot be earlier than start year.');
  END IF;

  IF :NEW.end_year IS NOT NULL AND :NEW.end_year < :NEW.start_year THEN
    RAISE_APPLICATION_ERROR(-20401, 'End year cannot be earlier than start year.');
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20402, 'trg_prevent_title_bad_dates error: ' || SQLERRM);
END trg_prevent_title_bad_dates;
```

INDEXES

TWO NON-KEY INDEXES

Index 1: Speed up title searches by genre

CREATE INDEX idx_titlegenre_genre_id ON TitleGenre(genreid);

Justification:
Searching titles by genre is one of the most common operations.
Indexing genre_id makes JOINs with Genre extremely fast.

Index 2: Speed up user rating lookups

CREATE INDEX idx_userrating_title ON UserRating(titleid);

Justification:
RatingSummary recalculation depends heavily on selecting ratings by title_id.
This index drastically speeds up aggregation and reporting.

VIEWS

View 1: Title Full Metadata View

```
CREATE VIEW vw_title_metadata AS

SELECT

  t.titleid,

  t.primary_title,

  t.start_year,

  t.release_date,

  g.name AS genre,

  c.name AS company,

  k.keywordtext

FROM Title t

LEFT JOIN TitleGenre tg ON t.titleid = tg.titleid

LEFT JOIN Genre g ON tg.genreid = g.genreid

LEFT JOIN TitleCompany tc ON t.titleid = tc.titleid

LEFT JOIN Company c ON tc.companyid = c.companyid

LEFT JOIN TitleKeyword tk ON t.titleid = tk.titleid

LEFT JOIN Keyword k ON tk.keywordid = k.keywordid;
```

Purpose:
A single view that presents all metadata in one query.
Useful for dashboards, search results, and admin panels.

View 2: Title Ratings and Reviews Analytics

```
CREATE VIEW vw_title_analytics AS

SELECT

   t.titleid,

   t.primary_title,

   rs.average_rating,

   rs.num_votes,

   COUNT(r.reviewid) AS total_reviews

FROM Title t

LEFT JOIN RatingSummary rs ON t.titleid = rs.titleid

LEFT JOIN Review r ON t.titleid = r.titleid

GROUP BY t.titleid, t.primary_title, rs.average_rating, rs.num_votes;
```

Purpose:
A business-level view that shows:

- **title**
- **average rating**
- **vote count**
- **number of reviews**
