Top 10 Oracle DB Automation Scripts for Enhanced Performance and Security

Using SQL and PL/SQl Scripts

Asfaw Gedamu

Top 10 Oracle Database Automation Scripts for Enhanced Performance and Security

While achieving 100% performance is an ideal, optimization is an ongoing process. These scripts can significantly improve database health and streamline administrative tasks.

1. **Health Check Script:** This script monitors critical metrics like space usage, performance statistics (waits, latches), and error logs, providing a quick overview of database health.
2. **Backup and Recovery Scripts:** Automating regular backups ensures data protection in case of failures. This script can handle full and incremental backups, and potentially automate recovery processes.
3. **User Management Scripts:** Automate user creation, deletion, and privilege assignment based on predefined roles. This improves efficiency and reduces the risk of human error.
4. **Table Maintenance Scripts:** Schedule regular table maintenance tasks like reorganization, shrinking, and data archiving. This optimizes storage utilization and query performance.
5. **Database Tuning Scripts:** Identify potential performance bottlenecks by analyzing execution plans and wait events. These scripts can suggest adjustments to improve query efficiency.
6. **Reporting Scripts:** Generate reports on various aspects like user activity, resource consumption, and performance trends. This data helps identify areas for further optimization.
7. **Data Archiving and Purging Scripts:** Automate archiving historical data to separate tables or external storage based on defined criteria. This frees up space in the primary database for frequently accessed data.
8. **Index Management Scripts:** Automate index creation, maintenance (analyze), and rebuilding based on fragmentation or usage patterns. This ensures indexes remain effective for optimized querying.
9. **Security Auditing Scripts:** These scripts identify potential security vulnerabilities by checking user privileges, password policies, and audit settings. This helps maintain a secure database environment.

10. **Tablespace and Resource Utilization Scripts:** This approach utilizes PL/SQL procedures and a Bash script to monitor tablespace utilization and resource usage within the Oracle database.

Benefits of Automation:

- Improved Efficiency: Scripts automate repetitive tasks, freeing up DBA time for more strategic activities.
- Reduced Errors: Automating tasks minimizes human error and ensures consistent outcomes.
- Enhanced Performance: Proactive maintenance and optimization scripts contribute to a well-performing database.
- Increased Security: Automated security checks help identify and address potential vulnerabilities.

Remember: These scripts are a starting point. Customize them to your specific database environment and security requirements.

1. **Health Check Script (PL/SQL + Bash)**

This script combines a PL/SQL procedure to gather database metrics and a Bash script to process the output and generate a report.

PL/SQL Procedure (health_check.sql):

```
CREATE OR REPLACE PROCEDURE health_check (
  report_out OUT VARCHAR2
) AS
  v_space_used NUMBER;
  v_space_free NUMBER;
  v_[illegible]_waits NUMBER;
  v_hard_parses NUMBER;
  v_alert_log_errors NUMBER;
  CURSOR c_spaces IS
    SELECT tablespace_name,
```

```
            ROUND (used_bytes / 1024 / 1024 / 1024, 2) USED_GB,
            ROUND (MAXBYTES - used_bytes / 1024 / 1024 / 1024, 2)
FREE_GB
    FROM dba_tablespace_usage_metrics;
  CURSOR c_waits IS
    SELECT value AS "Waits"
    FROM v$system_event_attribute
    WHERE name = 'latch: latch free';
  CURSOR c_stats IS
    SELECT value AS "Hard Parses"
    FROM v$sgastat
    WHERE name = 'parse cursor hard';
  CURSOR c_errors IS
    SELECT COUNT(*)
    FROM dba_errors
    WHERE sysdate - creation_date < 1;  -- Check for last day
errors
BEGIN
  DBMS_OUTPUT.PUT_LINE('**Database Health Check Report**');

  -- Space Usage
  OPEN c_spaces;
  LOOP
    FETCH c_spaces INTO v_space_used, v_space_free;
    EXIT WHEN c_spaces%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_space_used || ' GB used, ' ||
v_space_free || ' GB free in ' || c_spaces.tablespace_name);
  END LOOP;
  CLOSE c_spaces;

  -- Performance Metrics
  OPEN c_waits;
  FETCH c_waits INTO v_waits;
  CLOSE c_waits;

  OPEN c_stats;
  FETCH c_stats INTO v_hard_parses;
  CLOSE c_stats;

  OPEN c_errors;
  FETCH c_errors INTO v_alert_log_errors;
  CLOSE c_errors;
```

```
  DBMS_OUTPUT.PUT_LINE('Latch Waits: ' || v_latch_waits);
  DBMS_OUTPUT.PUT_LINE('Hard Parses: ' || v_hard_parses);
  DBMS_OUTPUT.PUT_LONG(report_out, DBMS_OUTPUT.GET_LINE);
END;
/
```

Bash Script (health_check.sh):

```
#!/bin/bash

sqlplus -s /nolog << EOF
@health_check.sql
SET VERIFY OFF;
WHENEVER SQLERROR EXIT;
/
EOF

report_data=$(cat health_check.sql.out)

echo -e "\n**Alert Log Errors (Last Day)**"
sqlplus -s /nolog << EOF
SET LINESIZE 100;
SELECT message FROM dba_errors
WHERE sysdate - creation_date < 1;
WHENEVER SQLERROR EXIT;
/
EOF >> health_check.sql.out

echo "$report_data" > health_check_report.txt

# Optional: Send report via email
#mail -s "Database Health Check Report" your_email@example.com <
health_check_report.txt
echo "Database health check report generated:
health_check_report.txt"
```

How it works:

1. The PL/SQL procedure health_check gathers various metrics:
    - Tablespace usage (used and free space)
    - Latch waits

    - Hard parses
    - Alert log errors (from the last day)
2. The Bash script:
    - Executes the PL/SQL procedure and captures the output.
    - Queries for recent alert log errors and appends them to the report.
    - Writes the report to a file (health_check_report.txt).
    - Optionally, you can uncomment the mail command to send the report via email.

Benefits:

- Script automates database health checks.
- Reports on critical metrics for space usage and performance.
- Highlights potential issues with latch waits, hard parses, and recent errors.

Note:

- Modify the /nolog connection string in the Bash script to include your username

2. Backup and Recovery Scripts (RMAN + Bash)

This approach utilizes Oracle Recovery Manager (RMAN) for backups and a Bash script for automation.

Preparation:

1. Configure RMAN with backup settings (retention policies, channels, etc.) as per your organization's needs. Refer to the official documentation for details https://docs.oracle.com/cd/E11882_01/backup.112/e10642/rcmquick.htm.

Bash Script (backup_and_recovery.sh):

```
#!/bin/bash

# Set environment variables (modify as needed)
BACKUP_SET="full_db_backup"  # RMAN backup set name
BACKUP_DEST="/u01/backups"    # Backup destination directory
```

```
RECOVERY_TARGET="sid_db"        # Target database SID for
recovery

# Backup the database
echo "**Starting database backup..."
rman target / << EOF
RUN {
  ALLOCATE CHANNEL c1 TYPE DISK
  FORMAT '${BACKUP_DEST}/${BACKUP_SET}_%d_%T.bak';
  BACKUP AS COPY OF DATABASE;
}
EOF

if [ $? -eq 0 ]; then
  echo "**Database backup completed successfully."
else
  echo "**Error during database backup. Please check RMAN logs."
  exit 1
fi

# Optional: Schedule daily backups using cron
# 0 0 * * * /path/to/backup_and_recovery.sh > /dev/null 2>&1

echo "**Daily backup script execution complete."
```

Recovery Steps (using RMAN):

1. **In case of instance failure:**
    - Restore the control file from a recent backup using the RECOVER CONTROLFILE command.
    - Start the instance in NOMOUNT mode.
    - Use RECOVER DATABASE with the USING BACKUP SET clause to restore the database.
2. **In case of data loss (e.g., schema object corruption):**
    - Open the database in READ ONLY mode.
    - Use RECOVER TABLESPACE or RECOVER DATAFILE commands to restore specific datafiles or tablespaces.
    - Switch the database to READ WRITE mode.

Benefits:

- Script automates daily full database backups using RMAN.
- Provides a clear reference for senior DBAs to initiate manual recovery steps in case of failures.
- Uses environment variables for easy customization of backup paths and names.

Note:

- This is a basic example. You can extend it to include differential backups, incremental backups, and archive log backups based on your RMAN configuration.
- Ensure proper access permissions for RMAN and the backup directory.
- Consider implementing a post-backup verification script to ensure data integrity.

3. User Management Scripts (SQL + Bash)

This approach combines a PL/SQL procedure for user creation and a Bash script for user deletion and privilege assignment.

Preparation:

1. Create a table (e.g., user_provisioning) to store user information (username, password, roles, etc.).

SQL Script (create_user.sql):

```
CREATE OR REPLACE PROCEDURE create_user (
  username IN VARCHAR2,
  password IN VARCHAR2,
  roles IN VARCHAR2
)
AS
BEGIN
  EXECUTE IMMEDIATE 'CREATE USER ' || username ||' IDENTIFIED BY
VALUES('''' || password || ''')';
```

```
  FOR r IN SPLIT (roles, ',') LOOP
    EXECUTE IMMEDIATE 'GRANT ' || r ||' TO ' || username;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('User ' || username || ' created
successfully.');
END;
/
```

Bash Script (manage_users.sh):

```
#!/bin/bash

# Set script arguments (modify as needed)
username="$1"
action="$2"
roles="$3"  # Optional, comma-separated list of roles (for
grant)

# Validate arguments
if [ -z "$username" ] || [ -z "$action" ]; then
  echo "Usage: $0 <username> <action> [<roles>]"
  echo "  action: create | delete | grant"
  exit 1
fi

sqlplus -s /nolog << EOF
WHENEVER SQLERROR EXIT;
BEGIN
  IF action = 'create' THEN
    create_user('$username', '$password', '$roles');  -- Replace
password with a secure generation method
  ELSIF action = 'delete' THEN
    EXECUTE IMMEDIATE 'DROP USER ' || username;
    DBMS_OUTPUT.PUT_LINE('User ' || username || ' deleted
successfully.');
  ELSIF action = 'grant' THEN
    FOR r IN SPLIT ('$roles', ',') LOOP
      EXECUTE IMMEDIATE 'GRANT ' || r ||' TO ' || username;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Roles granted to user ' || username);
  ELSE
```

```
    DBMS_OUTPUT.PUT_LINE('Invalid action specified.');
  END IF;
END;
/
EOF

if [ $? -eq 0 ]; then
  echo "**User management operation successful."
else
  echo "**Error during user management. Please check the
database logs."
fi
```

How it works:

1. **User Creation:**
    - Populate the user_provisioning table with user details (username, password [generated securely], roles).
    - Run the script: ./manage_users.sh <username> create
2. **User Deletion:**
    - Run the script: ./manage_users.sh <username> delete
3. **Granting Privileges:**
    - Update the user_provisioning table with the desired roles.
    - Run the script: ./manage_users.sh <username> grant <comma_separated_roles>

Benefits:

- Automates user creation and deletion tasks.
- Provides a secure way to grant roles by referencing a separate table for user information.
- Script arguments allow flexibility for different user management actions.

Note:

- This is a basic example. You can enhance it with error handling, password complexity checks, and logging.

- The script uses a placeholder for password generation. Implement a secure method for generating strong passwords (e.g., DBMS_CRYPTO package).
- Consider integrating this script with a user provisioning tool for a more automated workflow.

4. **Table Maintenance Scripts (PL/SQL + Bash)**

This approach combines PL/SQL procedures and a Bash script for automating table maintenance tasks.

Preparation:

1. Identify tables that benefit from reorganization (fragmented tables) or shrinking (tables with significant unused space).
2. Define a retention policy for data archiving.

PL/SQL Procedures:

1. Reorganize_Table.sql:

```
CREATE OR REPLACE PROCEDURE reorganize_table (
  table_owner IN VARCHAR2,
  table_name IN VARCHAR2
)
AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Reorganizing table ' || table_owner ||
'.' || table_name || '...');
  EXECUTE IMMEDIATE 'ALTER TABLE ' || table_owner || '.' ||
table_name ||' REORGANIZE';
END;
/
```

2. Archive_Data.sql:

```
CREATE OR REPLACE PROCEDURE archive_data (
  table_name IN VARCHAR2,
  retention_days IN NUMBER
)
AS
  archive_cursor CURSOR IS
    SELECT *
    FROM (
      SELECT *
      FROM $table_name
      WHERE last_updated < SYSDATE - retention_days
    );
  archive_record archive_table%ROWTYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Archiving data from table ' ||
table_name || '...');

  OPEN archive_cursor;
  LOOP
    FETCH archive_cursor INTO archive_record;
    EXIT WHEN archive_cursor%NOTFOUND;

    -- Insert data into archive table (implement your logic
here)

  END LOOP;
  CLOSE archive_cursor;

  EXECUTE IMMEDIATE 'DELETE FROM ' || table_name ||' WHERE
last_updated < SYSDATE - ' || retention_days;
  DBMS_OUTPUT.PUT_LINE('Archived data and deleted old
records.');
END;
/
```

Bash Script (table_maintenance.sh):

```
#!/bin/bash

# Set script arguments (modify as needed)
```

```
action="$1"
table_owner="$2"
table_name="$3"
retention_days="$4"  # Optional, for archive action

# Validate arguments
if [ -z "$action" ] || [ -z "$table_owner" ] || [ -z
"$table_name" ]; then
  echo "Usage: $0 <action> <table_owner> <table_name>
[<retention_days>]"
  echo "  action: reorganize | archive"
  exit 1
fi

sqlplus -s /nolog << EOF
WHENEVER SQLERROR EXIT;
BEGIN
  IF action = 'reorganize' THEN
    reorganize_table('$table_owner', '$table_name');
  ELSIF action = 'archive' THEN
    archive_data('$table_name', $retention_days);  # Replace
with actual value if provided
  ELSE
    DBMS_OUTPUT.PUT_LINE('Invalid action specified.');
  END IF;
END;
/
EOF

if [ $? -eq 0 ]; then
  echo "**Table maintenance operation successful."
else
  echo "**Error during table maintenance. Please check the
database logs."
fi
```

How it works:

1. **Table Reorganization:**
    - Run the script: ./table_maintenance.sh reorganize <table_owner> <table_name>
2. **Data Archiving:**
    - Define the retention period (days) for data in the retention_days argument.

- Run the script: ./table_maintenance.sh archive <table_name> <retention_days> (replace <retention_days> with the actual value)

Benefits:

- Automates table reorganization and data archiving tasks.
- Provides separate procedures for each task, promoting modularity.
- Script arguments allow for easy execution with specific table details.

Note:

- Modify the archive_data procedure to implement your specific logic for inserting data into the archive table.
- Consider scheduling these scripts as background jobs to run periodically.
- Analyze table fragmentation and space usage before initiating reorganization or shrinking.

5. **Database Tuning Scripts (PL/SQL + Bash)**

This approach utilizes PL/SQL procedures and a Bash script to gather performance data and identify potential bottlenecks.

Preparation:

- Identify critical queries or slow-performing areas in the database.

PL/SQL Procedures:

1. Get_Top_Executions.sql:

```
CREATE OR REPLACE PROCEDURE get_top_executions (
  num_executions IN NUMBER DEFAULT 10,
  capture_date IN DATE DEFAULT SYSDATE
)
AS
  CURSOR c_executions IS
```

```
    SELECT sql_id, executions, elapsed_time/executions AS
avg_elapsed_time
    FROM v$sql
    WHERE capture_date = capture_date
    ORDER BY elapsed_time DESC
    FETCH FIRST num_executions ROWS ONLY;
  e_record c_executions%ROWTYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('**Top ' || num_executions || ' SQL
Executions (Elapsed Time):**');
  OPEN c_executions;
  LOOP
    FETCH c_executions INTO e_record;
    EXIT WHEN c_executions%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(e_record.sql_id || ': ' ||
e_record.executions || ' executions, Avg. Elapsed Time: ' ||
e_record.avg_elapsed_time || 's');
  END LOOP;
  CLOSE c_executions;
END;
/
```

2. Analyze_Execution_Plan.sql:

```
CREATE OR REPLACE PROCEDURE analyze_plan (
  sql_id IN VARCHAR2
)
AS
  plan_table DBMS_OUTPUT.PUT_LINE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('**Execution Plan for SQL ID: ' ||
sql_id);
  EXECUTE DBMS_XPLAN.DISPLAY_CURSOR (CURSOR => sql_id);
END;
/
```

Bash Script (tuning_script.sh):

```
#!/bin/bash
```

```
# Set script arguments (modify as needed)
num_executions="$1"  # Optional, number of top executions to
display

sqlplus -s /nolog << EOF
WHENEVER SQLERROR EXIT;

-- Get top SQL executions
get_top_executions($num_executions);

-- Analyze plan for the first execution (modify as needed)
analyze_plan((SELECT sql_id FROM v$sql WHERE
capture_date=SYSDATE ORDER BY elapsed_time DESC FETCH FIRST 1
ROWS ONLY).sql_id);

EXIT;
/
EOF

if [ $? -eq 0 ]; then
  echo "**Database tuning script execution complete."
else
  echo "**Error during script execution. Please check the
database logs."
fi
```

How it works:

1. **Identify Top Executions:**
    - Run the script: ./tuning_script.sh <num_executions> (optional, number of top executions to display)
    - This identifies frequently executed queries with high elapsed time.
2. **Analyze Execution Plan:**
    - The script automatically analyzes the execution plan for the first SQL ID in the top executions list.
    - You can modify the script to analyze a specific SQL ID of interest.

Benefits:

- Helps pinpoint potential performance bottlenecks by identifying frequently executed slow queries.
- Provides the execution plan for further analysis and optimization.
- Script arguments allow customization for the number of top executions displayed.

Note:

- This is a basic example. You can extend it to analyze wait events, explain plans for specific queries, and suggest potential tuning actions.
- Consider integrating this script with Automatic Workload Repository (AWR) reports for more comprehensive performance analysis.

6. **Reporting Scripts (SQL + Bash)**

This approach utilizes SQL queries and a Bash script to generate reports on various aspects of the database.

Preparation:

- Define the specific reports you need (e.g., user activity, space usage, schema statistics).

SQL Queries:

1. User_Activity.sql:

```
SELECT username,
       SUM(executions) AS total_executions,
       SUM(elapsed_time) AS total_elapsed_time
FROM v$sql
GROUP BY username
ORDER BY total_elapsed_time DESC;
```

2. Space_Usage.sql:

```
SELECT tablespace_name,
       ROUND (used_bytes / 1024 / 1024 / 1024, 2) USED_GB,
       ROUND (MAXBYTES - used_bytes / 1024 / 1024 / 1024, 2)
```

```
FREE_GB
FROM dba_tablespace_usage_metrics;
```

3. Schema_Statistics.sql:

```
SELECT owner,
       object_name,
       object_type,
       SUM(rows_modded) AS total_modifications
FROM dba_objects o
JOIN dba_hist_tablespace_space_usage h
ON o.owner = h.owner
GROUP BY owner, object_name, object_type
ORDER BY total_modifications DESC;
```

Bash Script (generate_reports.sh):

```
#!/bin/bash

# Set report type (modify as needed)
report_type="$1"

# Validate argument
if [ -z "$report_type" ]; then
  echo "Usage: $0 <report_type>"
  echo "  report_type: user_activity | space_usage |
schema_statistics"
  exit 1
fi

sqlplus -s /nolog << EOF
WHENEVER SQLERROR EXIT;
SET LINESIZE 100;

BEGIN
  IF report_type = 'user_activity' THEN
    SELECT * FROM user_activity;
  ELSIF report_type = 'space_usage' THEN
    SELECT * FROM space_usage;
  ELSIF report_type = 'schema_statistics' THEN
```

```
    SELECT * FROM schema_statistics;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Invalid report type specified.');
  END IF;
END;
/
EOF

> report_$report_type.txt

if [ $? -eq 0 ]; then
  echo "**Report generation successful: report_$report_type.txt"
else
  echo "**Error during report generation. Please check the
database logs."
fi
```

How it works:

1. **Generate Specific Report:**
    - Run the script: ./generate_reports.sh <report_type> (replace <report_type> with the desired report: user_activity, space_usage, or schema_statistics)
    - The script generates a text file named report_<report_type>.txt.
2. **Customize Reports:**
    - Modify the provided SQL queries to suit your specific reporting needs.
    - Add additional reports by creating new SQL queries and referencing them in the Bash script.

Benefits:

- Automates generation of reports on various database aspects.
- Provides a flexible framework for creating custom reports using SQL queries.
- Script arguments allow easy selection of the desired report type.

Note:

- This is a basic example. You can enhance it with formatting, data filtering, and integration with scheduling tools.

- Consider using tools like Oracle Data Miner or third-party reporting solutions for more advanced reporting needs.

7. **Data Archiving and Purging Scripts (PL/SQL + Bash)**

This approach utilizes PL/SQL procedures and a Bash script to automate data archiving and purging based on configurable criteria.

Preparation:

1. Define the tables and data you want to archive or purge.
2. Determine the archiving strategy (separate table, external storage) and purging criteria (age, specific conditions).

PL/SQL Procedures:

1. Archive_Data.sql:

```
CREATE OR REPLACE PROCEDURE archive_data (
  table_name IN VARCHAR2,
  archive_table IN VARCHAR2,
  archive_where_clause IN VARCHAR2 DEFAULT NULL
)
AS
  archive_cursor CURSOR IS
    SELECT *
    FROM $table_name
    WHERE $archive_where_clause;
  archive_record archive_table%ROWTYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Archiving data from ' || table_name || '
to ' || archive_table || '...');

  OPEN archive_cursor;
  LOOP
    FETCH archive_cursor INTO archive_record;
    EXIT WHEN archive_cursor%NOTFOUND;

    -- Insert data into archive table (implement your logic
here)
```

```
  END LOOP;
  CLOSE archive_cursor;

  DBMS_OUTPUT.PUT_LINE('Archived data successfully.');
END;
/
```

2. Purge_Data.sql:

```
CREATE OR REPLACE PROCEDURE purge_data (
  table_name IN VARCHAR2,
  purge_where_clause IN VARCHAR2
)
AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Purging data from ' || table_name ||
'...');
  EXECUTE IMMEDIATE 'DELETE FROM ' || table_name || ' WHERE ' ||
purge_where_clause;
  DBMS_OUTPUT.PUT_LINE('Purged data successfully.');
END;
/
```

Bash Script (archive_purge.sh):

```
#!/bin/bash

# Set script arguments (modify as needed)
action="$1"
table_name="$2"
archive_table="$3"  # Optional, for archive action
archive_where="$4"  # Optional, where clause for archive
purge_where="$5"     # Optional, where clause for purge

# Validate arguments
if [ -z "$action" ] || [ -z "$table_name" ]; then then
  echo "Usage: $0 <action> <table_name> [<archive_table>
<archive_where> <purge_where>]"
  echo "  action: archive | purge"
  exit 1
fi
```

```
sqlplus -s /nolog << EOF
WHENEVER SQLERROR EXIT;
BEGIN
  IF action = 'archive' THEN
    archive_data('$table_name', '$archive_table',
'$archive_where');
  ELSIF action = 'purge' THEN
    purge_data('$table_name', '$purge_where');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Invalid action specified.');
  END IF;
END;
/
EOF

if [ $? -eq 0 ]; then
  echo "**Data archiving/purging operation successful."
else
  echo "**Error during data archiving/purging. Please check the
database logs."
fi
```

How it Works:

1. **Archive Data:**
    - Run the script:

```
./archive_purge.sh archive <table_name> <archive_table>
"<archive_where_clause>"
```

    - Replace placeholders with actual values:
        - <archive_table>: Name of the archive table
        - <archive_where_clause>: Optional WHERE clause to filter data for archiving (e.g., 'last_updated < SYSDATE - 365')

 Purge Data:

- Run the script:

```
./archive_purge.sh purge <table_name> "<purge_where_clause>"
```

2.
    - Replace <purge_where_clause> with the WHERE clause specifying data to purge (e.g., 'last_updated < SYSDATE - 730')

Benefits:

- Automates data archiving and purging based on configurable criteria.
- Separates procedures for archiving and purging, promoting modularity.
- Script arguments allow customization for specific tables, archive locations, and filtering conditions.

Note:

- Modify the archive_data procedure to implement your specific logic for inserting data into the archive table (e.g., consider data transformation or compression).
- Ensure proper backup strategy exists before purging data.
- Schedule

8. Index Management Scripts (PL/SQL + Bash)

This approach utilizes PL/SQL procedures and a Bash script to automate index creation, maintenance, and rebuilding based on specific criteria.

Preparation:

- Identify tables that could benefit from indexing based on query patterns and access needs.

PL/SQL Procedures:

1. Create_Index.sql:

```
CREATE OR REPLACE PROCEDURE create_index (
  table_owner IN VARCHAR2,
  table_name IN VARCHAR2,
  index_name IN VARCHAR2,
  column_list IN VARCHAR2
)
AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Creating index ' || index_name || ' on '
|| table_owner || '.' || table_name || '...');
  EXECUTE IMMEDIATE 'CREATE INDEX ' || index_name || ' ON ' ||
table_owner || '.' || table_name || '(' || column_list || ')';
END;
/
```

2. Analyze_Index.sql:

```
CREATE OR REPLACE PROCEDURE analyze_index (
  owner IN VARCHAR2,
  table_name IN VARCHAR2,
  index_name IN VARCHAR2
)
AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Analyzing index ' || index_name || ' on
' || owner || '.' || table_name || '...');
  EXECUTE IMMEDIATE 'ANALYZE INDEX ' || owner || '.' ||
index_name || ' COMPUTE STATISTICS';
END;
/
```

3. Rebuild_Index.sql (Rebuild based on fragmentation):

```
CREATE OR REPLACE PROCEDURE rebuild_index (
  owner IN VARCHAR2,
  table_name IN VARCHAR2,
  index_name IN VARCHAR2
)
AS
  v_avg_leaf_blocks NUMBER;
BEGIN
  SELECT AVG (LEAF_BLOCKS) AS avg_leaf_blocks
```

```
  INTO v_avg_leaf_blocks
  FROM DBA_INDEXES
  WHERE OWNER = owner AND TABLE_NAME = table_name AND INDEX_NAME
= index_name;

  IF v_avg_leaf_blocks < 0.7 THEN  -- Rebuild threshold (adjust
as needed)
    DBMS_OUTPUT.PUT_LINE('Rebuilding fragmented index ' ||
index_name || '...');
    EXECUTE IMMEDIATE 'ALTER INDEX ' || owner || '.' ||
index_name || ' REBUILD';
  ELSE
    DBMS_OUTPUT.PUT_LINE('Index ' || index_name || ' is not
fragmented.');
  END IF;
END;
/
```

Bash Script (manage_indexes.sh):

```
#!/bin/bash

# Set script arguments (modify as needed)
action="$1"
table_owner="$2"
table_name="$3"
index_name="$4"  # Optional, for create and rebuild actions
column_list="$5"  # Optional, comma-separated list of columns
(for create)

# Validate arguments
if [ -z "$action" ] || [ -z "$table_owner" ] || [ -z
"$table_name" ]; then then
  echo "Usage: $0 <action> <table_owner> <table_name>
[<index_name> <column_list>]"
  echo "  action: create | analyze | rebuild"
  exit 1
fi

sqlplus -s /nolog << EOF
WHENEVER SQLERROR EXIT;
BEGIN
```

```
  IF action = 'create' THEN
    create_index('$table_owner', '$table_name', '$index_name',
'$column_list');
  ELSIF action = 'analyze' THEN
    analyze_index('$table_owner', '$table_name', '$index_name');
  ELSIF action = 'rebuild' THEN
    rebuild_index('$table_owner', '$table_name', '$index_name');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Invalid action specified.');
  END IF;
END;
/
EOF

if [ $? -eq 0 ]; then
  echo "**Index management operation successful."
else
  echo "**Error during index management. Please check the
database logs."
fi
```

How it Works:

1. **Create Index:**
    - Run the script:

```
./manage_indexes.sh create <table_owner> <table_name>
<index_name> "<column_list>"
```

    - Replace placeholders with actual values:
        - <index_name>: Name of the index to create
        - <column_list>: Comma-separated list of columns for the index

 Analyze Index:

- Run the script:

```
./manage_indexes.sh analyze <table_owner> <table_name>
<index_name>
```

- Replace placeholders with actual values:
  - <index_name>: Name of the index to analyze

Rebuild Index (based on fragmentation):

- Run the script:

```
./manage_indexes.sh rebuild <table_owner> <table_name>
<index_name>
```

  - Replace placeholders with actual values:
    - <index_name>: Name of the index to rebuild
- This script checks for fragmentation by analyzing the average number of leaf blocks in the index. You can adjust the rebuild threshold (v_avg_leaf_blocks < 0.7 in the script) based on your specific needs.

Benefits:

- Automates index creation, maintenance (analysis), and rebuilding based on pre-defined criteria.
- Provides separate procedures for each action, promoting modularity.
- Script arguments allow customization for specific tables, index names, and column selections.

Note:

- Consider scheduling these scripts as background jobs to run periodically.
- Analyze index usage and query patterns before creating new indexes.
- Explore alternative rebuilding criteria beyond fragmentation (e.g., index size, usage statistics).

9. **Security Auditing Scripts (PL/SQL + Bash)**

This approach combines PL/SQL procedures and a Bash script to automate basic security checks within the Oracle database.

Preparation:

- Define the scope of your security audit (e.g., user privileges, password policies, auditing settings).

PL/SQL Procedures:

1. List_User_Privileges.sql:

```
CREATE OR REPLACE PROCEDURE list_user_privileges (
  username IN VARCHAR2
)
AS
  CURSOR c_privileges IS
    SELECT privilege, granted_role, admin_option
    FROM user_tab_privs
    WHERE grantee = username
    UNION ALL
    SELECT privilege, granted_role, admin_option
    FROM user_role_privs
    WHERE grantee = username;
  p_record c_privileges%ROWTYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('**Privileges for user: ' || username);
  OPEN c_privileges;
  LOOP
    FETCH c_privileges INTO p_record;
    EXIT WHEN c_privileges%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(p_record.privilege || ' on ' ||
p_record.granted_role || ' (Admin Option: ' ||
p_record.admin_option || ')');
  END LOOP;
  CLOSE c_privileges;
  DBMS_OUTPUT.PUT_LINE('');  -- Add an empty line for better
readability
```

```
END;
/
```

2. Check_Password_Policy.sql:

```
CREATE OR REPLACE PROCEDURE check_password_policy
AS
  v_policies DBMS_OUTPUT.PUT_LINE_TYPE;
  v_min_len NUMBER;
  v_history_days NUMBER;
  v_special_chars BOOLEAN;
BEGIN
  SELECT password_life_days, history_days,
require_special_characters
  INTO v_policies
  FROM dba_profiles
  WHERE profile_name = 'DEFAULT';

  DBMS_OUTPUT.PUT_LINE('**Current Password Policy:**');
  DBMS_OUTPUT.PUT_LINE('  Minimum Password Length: ' ||
v_policies.password_life_days);
  DBMS_OUTPUT.PUT_LINE('  Password History: ' || v_history_days
|| ' days');
  DBMS_OUTPUT.PUT_LINE('  Require Special Characters: ' || CASE
v_policies.require_special_characters WHEN 'YES' THEN 'Yes' ELSE
'No' END);

  -- Add logic here to compare policy settings against your
security requirements
END;
/
```

3. Check_Audit_Settings.sql:

```
CREATE OR REPLACE PROCEDURE check_audit_settings
AS
  v_audit_trail VARCHAR2;
  v_audit_sysactions BOOLEAN;
  v_audit_logins BOOLEAN;
BEGIN
  SELECT audit_trail, audit_sysactions, audit_logins
  FROM v$audit_trail;
```

```
  DBMS_OUTPUT.PUT_LINE('**Current Audit Settings:**');
  DBMS_OUTPUT.PUT_LINE('  Audit Trail: ' || v_audit_trail);
  DBMS_OUTPUT.PUT_LINE('  Audit System Actions: ' || CASE
v_audit_sysactions WHEN TRUE THEN 'Yes' ELSE 'No' END);
  DBMS_OUTPUT.PUT_LINE('  Audit Logins: ' || CASE v_audit_logins
WHEN TRUE THEN 'Yes' ELSE 'No' END);

  -- Add logic here to compare settings against your desired
audit trail coverage
END;
/
```

Bash Script (security_audit.sh):

```
#!/bin/bash

# Set script arguments (modify as needed)
username="$1"  # Optional, username for privilege check

sqlplus -s /nolog << EOF
WHENEVER SQLERROR EXIT;

-- Check user privileges (if username provided)
BEGIN
  IF :username IS NOT NULL THEN
    list_user_privileges(:username);
  END IF;
END;
/

-- Check password policy
check_password_policy;

-- Check audit settings
check_audit_settings;

EXIT;
/
EOF

if [ $? -eq 0 ]; then
  echo "**Basic security audit completed."
else
```

```
  echo "**Error during security audit. Please check the database
logs."
fi
```

How it Works:

1. **List User Privileges:**
    - Run the script: ./security_audit.sh <username> (replace <username> with a specific user)
    - This lists all granted privileges for the specified user.

2. **Check Password Policy:**
    - Run the script: ./security_audit.sh (without arguments)
    - This displays the current password policy settings like minimum password length, password history duration, and special character requirement. You can add logic within the script to compare these settings against your security requirements.
3. **Check Audit Settings:**
    - Run the script: ./security_audit.sh (without arguments)
    - This displays the current audit trail status, whether system actions and logins are audited. You can add logic within the script to compare these settings against your desired audit trail coverage.

Benefits:

- Automates basic security checks for user privileges, password policy, and audit settings.
- Provides a starting point for a more comprehensive security audit.
- Script arguments allow customization for specific user privilege checks.

Note:

- This is a basic example. You can extend it to check for more security aspects like user roles, dba_users settings, and auditing details from dba_audit_trail views.

- Consider using Oracle Database Security assessment tools for a more in-depth security analysis.

10. Tablespace and Resource Utilization Scripts (PL/SQL + Bash)

This approach utilizes PL/SQL procedures and a Bash script to monitor tablespace utilization and resource usage within the Oracle database.

Preparation:

- Define a threshold for tablespace usage (e.g., 80%) to trigger alerts.

PL/SQL Procedures:

1. Get_Tablespace_Usage.sql:

```
CREATE OR REPLACE FUNCTION get_tablespace_usage (
  tablespace_name IN VARCHAR2
)
RETURN NUMBER
IS
  v_used_bytes NUMBER;
  v_total_bytes NUMBER;
  v_usage_pct NUMBER;
BEGIN
  SELECT SUM (bytes) AS used_bytes,
         MAX (bytes) AS total_bytes
  FROM dba_data_files
  WHERE tablespace_name = tablespace_name;

  IF v_total_bytes IS NOT NULL THEN
    v_usage_pct := ROUND ((v_used_bytes / v_total_bytes) * 100,
2);
  ELSE
    v_usage_pct := 0;
  END IF;

  RETURN v_usage_pct;
END;
/
```

2. Monitor_Resource_Usage.sql:

```
CREATE OR REPLACE PROCEDURE monitor_resource_usage
AS
  v_tablespace_name VARCHAR2;
  v_usage_pct NUMBER;
  CURSOR c_tablespaces IS
    SELECT tablespace_name
    FROM dba_tablespaces;
  p_tablespace c_tablespaces%ROWTYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('**Tablespace Usage Report (as of ' ||
SYSDATE || ')');
  OPEN c_tablespaces;
  LOOP
    FETCH c_tablespaces INTO p_tablespace;
    EXIT WHEN c_tablespaces%NOTFOUND;

    v_usage_pct :=
get_tablespace_usage(p_tablespace.tablespace_name);
    DBMS_OUTPUT.PUT_LINE(p_tablespace.tablespace_name || ': ' ||
v_usage_pct || '% used');

    IF v_usage_pct >= 80 THEN  -- Modify threshold as needed
      DBMS_OUTPUT.PUT_LINE('**WARNING: Tablespace ' ||
p_tablespace.tablespace_name || ' approaching capacity limit.');
    END IF;
  END LOOP;
  CLOSE c_tablespaces;
END;
/
```

Bash Script (monitor_resources.sh):

```
#!/bin/bash

sqlplus -s /nolog << EOF
WHENEVER SQLERROR EXIT;
BEGIN
  monitor_resource_usage;
END;
/
```

```
EOF

if [ $? -eq 0 ]; then
  echo "**Tablespace and resource usage monitored successfully."
else
  echo "**Error during monitoring. Please check the database
logs."
fi
```

How it Works:

1. **Monitor Tablespace Usage:**
    - Run the script: ./monitor_resources.sh
    - This script calls the get_tablespace_usage function to calculate usage percentages for all tablespaces. It then displays a report with usage details and alerts for tablespaces exceeding the defined threshold.
2. **Customize Threshold:**
    - Modify the 80 in the IF statement within monitor_resource_usage.sql to adjust the tablespace usage threshold that triggers a warning.
3. **Extend Resource Monitoring:**
    - Modify the monitor_resource_usage procedure to include additional resource checks (e.g., CPU, memory utilization) using relevant views like v$resource_pool_wait_stat or v$session.

Benefits:

- Automates monitoring of tablespace usage and resource consumption within the database.
- Provides alerts for approaching tablespace capacity limits.
- Script allows customization of the monitoring scope and thresholds.

Note:

- Consider scheduling this script to run periodically.

- Explore integrating these scripts with monitoring tools for centralized alerting and visualization.