Université Ferhat Abbas Setif 1

Faculy Of Sciences

Compter Science Department

DATA WAREHOUSES

2nd Year Master Data Engineering and Web Technologies

By Dr. Lyazid TOUMI

Contents

1 Data Warehouse Architecture and Design 9
- 1 Introduction to Data Warehousing . . . 9
- 2 Data Warehoue architecture . . . 10
- 3 Core Architecture Principles . . . 10
  - 3.1 Subject Orientation . . . 10
  - 3.2 Data Integration . . . 10
  - 3.3 Time Variance . . . 11
  - 3.4 Non-Volatility . . . 11
- 4 Multidimensional Data Modeling . . . 12
  - 4.1 OLAP Implementation Approaches . . . 12
- 5 Physical Design Considerations . . . 12
  - 5.1 Redundant Structures . . . 13
  - 5.2 Non-Redundant Structures . . . 13
- 6 Performance Tuning Strategies . . . 13
  - 6.1 Hardware Level . . . 13
  - 6.2 DBMS Level . . . 13
  - 6.3 Logical Level . . . 14
- 7 Commercial Implementation Tools . . . 14
- 8 Conclusion . . . 14

2 Data Warehouse Reporting 15
- 1 Introduction to DWH Reporting . . . 15
- 2 Reporting Architecture . . . 15
- 3 Oracle Reporting Tools . . . 15
  - 3.1 Oracle Analytics Server . . . 15
  - 3.2 Oracle APEX for Reporting . . . 17
- 4 Optimizing Reports with Materialized Views . . . 18
  - 4.1 Report-Specific MVs . . . 18
- 5 Parameterized Reporting . . . 20
  - 5.1 SQL Query Parameters . . . 20
- 6 Scheduled Report Delivery . . . 22
  - 6.1 Automated Email Reports . . . 22

7 Performance Considerations 23
7.1 Report Query Optimization 23
7.2 Query Design Patterns 23
8 Security and Access Control 24
8.1 Row-Level Security 24
9 Advanced Visualization Techniques 25
9.1 Time-Series Analysis 25
10 Best Practices 25
10.1 Report Development Guidelines 25
10.2 Performance Checklist 25
11 Emerging Trends 25
11.1 Modern Reporting Features 25

3 ETL Processes in Data Warehousing 27
1 Introduction to ETL 27
2 Oracle ETL Tools Overview 27
2.1 Tool Comparison 27
3 Extraction Patterns 29
3.1 Change Data Capture (CDC) 29
4 Transformation Techniques 30
4.1 Data Cleansing 30
5 Loading Strategies 31
5.1 Bulk Loading with SQL*Loader 31
5.2 External Tables 32
6 Incremental Loading 34
6.1 SCD Type 2 Implementation 34
7 High-Performance ETL Techniques 35
7.1 Parallel DML 35
7.2 Partition Exchange Loading 36
8 Error Handling and Recovery 37
8.1 ETL Auditing Framework 37
9 Optimizing ETL Performance 38
9.1 Performance Tuning Checklist 38
9.2 Memory Configuration 38
10 Best Practices 38
10.1 ETL Design Principles 38
10.2 Oracle-Specific Recommendations 39
11 Emerging Trends 39
11.1 Modern ETL Approaches 39

4 Data Warehouses Indexation Strategies 41
1 Introduction to Data Warehouse Indexing 41
2 Oracle-Specific Indexing Techniques 41
2.1 B-Tree Indexes in Oracle 41
2.2 Bitmap Indexes in Oracle 42
2.3 Bitmap Join Indexes in Oracle 42
3 Index Selection Methodology 42
3.1 Cost-Based Approach in Oracle 42
3.2 Implementation Example 43
4 Performance Comparison 44
5 Maintenance Strategies 44
6 Conclusion 44

5 Horizontal Partitioning in Data Warehouses 47
1 Introduction 47
2 Partitioning Strategies 47
2.1 Partitioning Key Selection 47
2.2 Partition Granularity 48
3 Oracle Partitioning Fundamentals 48
3.1 Partitioning Types 48
3.2 Creating Partitioned Tables 49
4 Horizontal partitioning modes 49
4.1 Range partitioning mode 49
4.2 Hash partitioning mode 51
4.3 List partitioning mode 52
4.4 Composite partitioning mode 53
4.5 Multicolumn partitioning mode 55
4.6 Reference partitioning mode 56
4.7 Virtual column partitioning 57
5 Partition Pruning Optimization 58
6 Partitioned Index Strategies 59
6.1 Local vs Global Indexes 59
6.2 Index Creation Examples 59
7 Partition Maintenance Automation 60
8 Performance Considerations 60
8.1 Partitioning Overhead 60
8.2 Monitoring Partition Usage 61
9 Real-World Implementation Case Study 61
10 Conclusion and Best Practices 62

6 Materialized Views in Data Warehouses 65
1 Materialized View Selection Problem 65
1.1 Problem Formulation 65
1.2 Selection Algorithms 66
2 Pruning Techniques 66
2.1 Dominance Pruning 67
2.2 Constraint-Based Pruning 67
2.3 Multi-Query Optimization Pruning 67
3 Oracle Materialized View Fundamentals 68
4 Refresh Mechanisms 68
5 Query Rewrite 69
6 Partitioned Materialized Views 70
7 MV Maintenance Best Practices 70
8 Advanced MV Selection in Oracle 71
9 Real-World Example 71
10 Materialized View vs. Indexes 73
11 Advanced Features 73
12 Conclusion 73

7 Data Warehouse Administration and Tuning 75
1 Introduction to DWH Administration 75
2 Oracle Data Warehouse Architecture 77
3 Storage Management 77
3.1 Tablespace Strategy 77
3.2 Compression Techniques 77
4 Performance Tuning Methodology 78
4.1 Tuning Approach 78
4.2 Key Performance Metrics 78
5 Query Optimization Techniques 79
5.1 Optimizer Statistics 79
5.2 SQL Plan Management 79
6 Parallel Execution Tuning 80
6.1 Configuration Parameters 80
6.2 Monitoring Parallel Queries 80
7 ETL Process Optimization 80
7.1 ETL Tuning Techniques 80
8 Resource Management 81
8.1 Database Resource Manager 81

9 Monitoring and Maintenance 82
9.1 Automated Maintenance Tasks 82
10 Troubleshooting Common Issues 82
10.1 Performance Problem Resolution 82
11 Conclusion and Best Practices 83
11.1 Administration Checklist 83
11.2 Ongoing Tuning Process 83

Reference Books

- The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling (3rd Edition), Ralph Kimball and Margy Ross, Wiley , 2013.
- Building the Data Warehouse, W. H. Inmon, John Wiley Sons, 2005
- Oracle database performance tuning: a checklist approach with simple and comprehensive guide to diagnose, optimize, and deliver, SANJAY MISHRA, kindle edition, 2025

Chapter 1

Data Warehouse Architecture and Design

1 Introduction to Data Warehousing

Modern enterprises rely on data warehouses as centralized repositories for analytical processing. Unlike traditional databases focused on day-to-day operations, data warehouses specialize in storing historical business data optimized for complex analysis and decision support.

These systems typically employ specialized schema designs:

- Star Schema: A simple structure with one central fact table linked to dimension tables
- Snowflake Schema: A normalized version of star schema where dimensions are further broken down

Query performance challenges primarily stem from:

- Large-scale join operations between fact and dimension tables
- Increasing data volumes, especially in scientific applications
- Complex analytical queries requiring aggregated results

2 Data Warehoue architecture

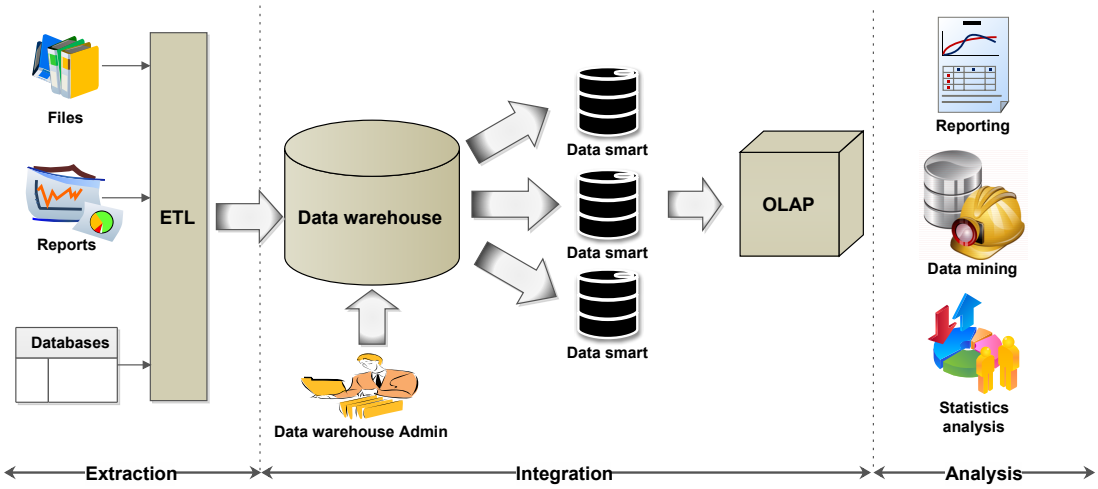

Figure 1: Data warehouses building process.

3 Core Architecture Principles

The fundamental architecture of data warehouses revolves around four key characteristics:

3.1 Subject Orientation

Data warehouses organize information around specific business subjects rather than operational functions. For instance:

- Retail: Sales performance analysis
- Telecom: Call pattern examination
- Healthcare: Patient treatment outcomes

3.2 Data Integration

A data warehouse consolidates information from multiple source systems through:

- Standardized naming conventions
- Consistent measurement units
- Unified data formats
- Conflict resolution mechanisms

3.3 Time Variance

Unlike operational systems that focus on current data, warehouses maintain historical records to enable:

- Trend analysis
- Year-over-year comparisons
- Pattern recognition across time periods

3.4 Non-Volatility

Once data enters the warehouse:

- It remains unchanged for analysis consistency
- Updates occur through periodic refreshes
- Historical snapshots are preserved

Table 1: Comparison of OLTP and OLAP Systems

Characteristic	OLTP Systems	OLAP Systems
Primary Purpose	Transaction processing	Analytical processing
Data Structure	Highly normalized	Denormalized
Query Patterns	Simple, predictable	Complex, ad-hoc
Performance Focus	Fast writes	Fast reads
Data Freshness	Real-time	Periodic updates
Storage Approach	Many small tables	Few large tables

4 Multidimensional Data Modeling

The data cube serves as the foundation for analytical processing, consisting of:

- Dimensions: Business perspectives (e.g., time, location, product)
- Measures: Quantitative values (e.g., sales amount, quantity)
- Hierarchies: Drill-down paths (e.g., year quarter month)

For a cube with n dimensions, there are 2^n possible aggregation levels (cuboids). Figure 2 illustrates a three-dimensional data cube.

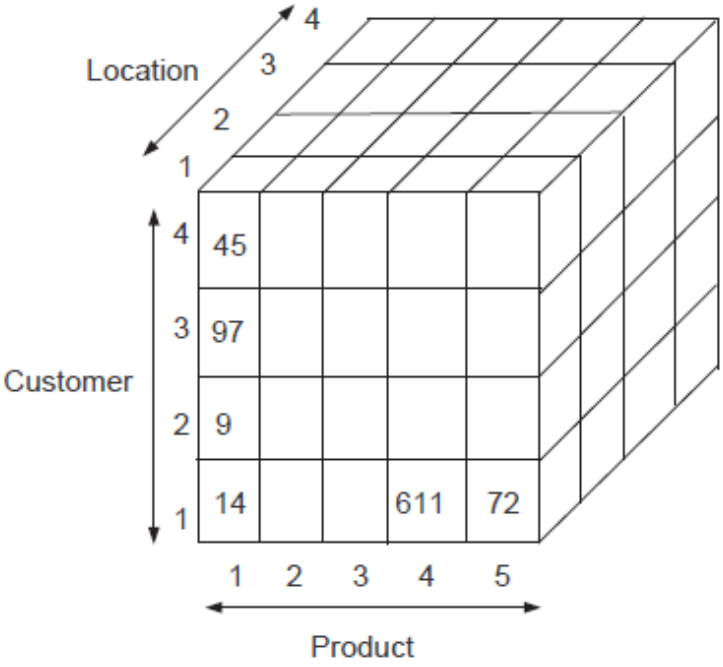

Figure 2: Three-dimensional data cube example

4.1 OLAP Implementation Approaches

5 Physical Design Considerations

Effective physical design significantly impacts query performance. Optimization techniques fall into two categories:

Table 2: OLAP Implementation Comparison

Feature	ROLAP	MOLAP	HOLAP
Storage Medium	Relational DB	Multidimensional array	Hybrid
Query Speed	Moderate	Fast	Balanced
Storage Efficiency	High	Low for sparse data	Medium
Implementation	Star/snowflake	Proprietary format	Combined

5.1 Redundant Structures

- Indexes: Accelerate data retrieval
- Materialized Views: Pre-computed query results
- Vertical Partitioning: Column-wise table splitting

5.2 Non-Redundant Structures

- Horizontal Partitioning: Row-wise table division
- Parallel Processing: Distributed query execution
- Query Scheduling: Workload prioritization

6 Performance Tuning Strategies

Data warehouse tuning operates at three levels:

6.1 Hardware Level

- RAID storage configurations
- Query Processing Units (QPUs)
- Memory allocation optimization

6.2 DBMS Level

- Buffer pool sizing
- Checkpoint frequency adjustment
- Concurrency control settings

6.3 Logical Level

- Query rewriting
- Optimal index selection
- Partitioning strategy refinement

7 Commercial Implementation Tools

Major database vendors provide specialized tuning advisors:

- Microsoft SQL Server: Database Tuning Advisor (DTA)
- IBM DB2: Design Advisor with workload management
- Oracle: Automatic Workload Repository (AWR)-based tuning

8 Conclusion

This chapter covered essential data warehouse concepts including:

- Foundational architecture principles
- Multidimensional modeling approaches
- Physical design optimization techniques
- Performance tuning methodologies

These fundamentals provide the basis for understanding advanced data warehouse optimization techniques covered in subsequent chapters.

Chapter 2

Data Warehouse Reporting

1 Introduction to DWH Reporting

Data warehouse reporting transforms raw data into actionable business intelligence through:

- Standardized operational reports
- Interactive dashboards
- Ad-hoc analytical queries
- Self-service BI tools
- Scheduled report distribution

2 Reporting Architecture

3 Oracle Reporting Tools

3.1 Oracle Analytics Server

```
-- Create dedicated reporting user
CREATE USER report_owner IDENTIFIED BY "R3port$2023"
DEFAULT TABLESPACE reporting
TEMPORARY TABLESPACE temp
QUOTA UNLIMITED ON reporting;

-- Grant necessary privileges
GRANT CREATE SESSION, CREATE VIEW,
      CREATE MATERIALIZED VIEW TO report_owner;
GRANT SELECT ON dwh.sales TO report_owner;
```

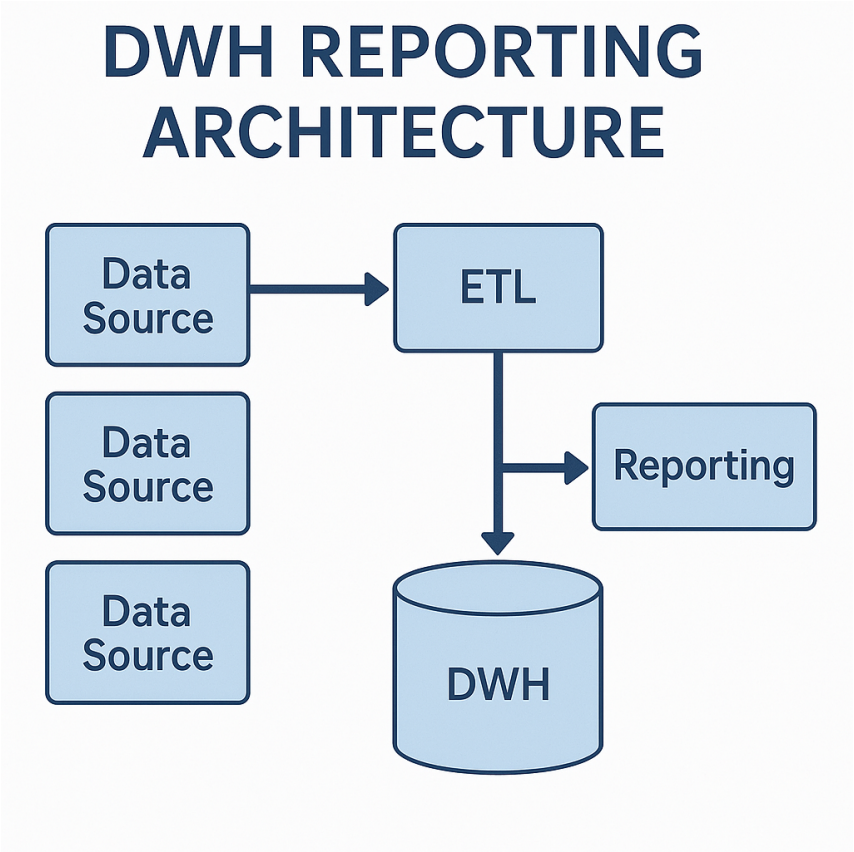

Figure 3: Data Warehouse Reporting Architecture

3.2 Oracle APEX for Reporting

```
-- Create APEX report region
BEGIN
  APEX_APPLICATION_PAGE.CREATE_PAGE(
    application_id => 100,
    page_id => 10,
    page_name => 'Sales Dashboard');

  APEX_APPLICATION_PAGE.CREATE_REGION(
    page_id => 10,
    region_name => 'Monthly Sales',
    source_type => 'SQL',
    source => 'SELECT TO_CHAR(sale_date,''YYYY-MM'') AS month,
                      SUM(amount) AS total_sales
              FROM sales
              GROUP BY TO_CHAR(sale_date,''YYYY-MM'')
              ORDER BY 1 DESC');
END;
```

4 Optimizing Reports with Materialized Views

4.1 Report-Specific MVs

```
CREATE MATERIALIZED VIEW mv_monthly_sales
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE
AS
SELECT
    TO_CHAR(s.sale_date,'YYYY-MM') AS month,
    r.region_name,
    p.product_category,
    COUNT(*) AS transaction_count,
    SUM(s.amount) AS total_sales,
    SUM(s.quantity) AS total_units
FROM
    sales s
    JOIN products p ON s.product_id = p.product_id
    JOIN regions r ON s.region_id = r.region_id
GROUP BY
    TO_CHAR(s.sale_date,'YYYY-MM'),
    r.region_name,
    p.product_category;
```

5 Parameterized Reporting

5.1 SQL Query Parameters

```
-- PL/SQL function for dynamic reporting
CREATE OR REPLACE FUNCTION get_sales_report(
    p_start_date DATE,
    p_end_date DATE,
    p_region_id NUMBER DEFAULT NULL,
    p_category_id NUMBER DEFAULT NULL)
RETURN SYS_REFCURSOR
IS
    v_cursor SYS_REFCURSOR;
    v_sql VARCHAR2(4000);
BEGIN
    v_sql := 'SELECT s.sale_date, c.customer_name,
                        p.product_name, s.amount
              FROM sales s
              JOIN customers c ON s.customer_id = c.customer_id
              JOIN products p ON s.product_id = p.product_id
              WHERE s.sale_date BETWEEN :1 AND :2';

    IF p_region_id IS NOT NULL THEN
        v_sql := v_sql || ' AND c.region_id = :3';
    END IF;

    IF p_category_id IS NOT NULL THEN
        v_sql := v_sql || ' AND p.category_id = :4';
    END IF;

    v_sql := v_sql || ' ORDER BY s.sale_date DESC';

    IF p_region_id IS NULL AND p_category_id IS NULL THEN
        OPEN v_cursor FOR v_sql USING p_start_date, p_end_date;
    ELSIF p_region_id IS NOT NULL AND p_category_id IS NULL THEN
        OPEN v_cursor FOR v_sql USING p_start_date, p_end_date,
         ↪  p_region_id;
    ELSE
        OPEN v_cursor FOR v_sql USING p_start_date, p_end_date,
                                      p_region_id, p_category_id;
    END IF;

    RETURN v_cursor;
END;
```

6 Scheduled Report Delivery

6.1 Automated Email Reports

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB(
    job_name        => 'SEND_DAILY_SALES_REPORT',
    job_type        => 'PLSQL_BLOCK',
    job_action      => 'BEGIN
        -- Generate report as CSV
       DECLARE
         v_file UTL_FILE.FILE_TYPE;
         v_csv CLOB;
             BEGIN
                v_file :=
                 ↪  UTL_FILE.FOPEN(''REPORT_DIR'',''sales.csv'',''W'');

                            FOR r IN (
                              SELECT * FROM sales
                              WHERE sale_date = TRUNC(SYSDATE)-1
                              ORDER BY sale_id
                            ) LOOP
                             v_csv := v_csv || r.sale_id || '','' ||
                                      r.sale_date || '','' ||
                                      r.amount || CHR(10);
                            END LOOP;

                            UTL_FILE.PUT_LINE(v_file, v_csv);
                            UTL_FILE.FCLOSE(v_file);

                            -- Email report
                            APEX_MAIL.SEND(
                             p_to        => ''managers@company.com'',
                              p_from      => ''reports@company.com'',
                               p_subj      => ''Daily Sales Report'',
                               p_body      => ''Attached is
                                ↪  yesterday''''s sales report'',
                           p_attachment => ''REPORT_DIR/sales.csv'');
                          END;
                       END;',
    start_date      => SYSDATE,
    repeat_interval => 'FREQ=DAILY; BYHOUR=8',
    enabled         => TRUE);
END;
```

7 Performance Considerations

7.1 Report Query Optimization

Table 3: Report Query Optimization Techniques

Problem	Solution
Slow-running reports	Create summary materialized views
High concurrency	Implement result caching
Large data volumes	Use pagination (LIMIT/OFFSET)
Complex calculations	Pre-compute in ETL

7.2 Query Design Patterns

```
-- Paginated report with analytic functions
SELECT * FROM (
  SELECT
    s.sale_id,
    s.sale_date,
    c.customer_name,
    p.product_name,
    s.amount,
    SUM(s.amount) OVER (PARTITION BY c.customer_id) AS cust_total,
    ROW_NUMBER() OVER (ORDER BY s.sale_date DESC) AS rn
  FROM
    sales s
    JOIN customers c ON s.customer_id = c.customer_id
    JOIN products p ON s.product_id = p.product_id
  WHERE
    s.sale_date BETWEEN :start_date AND :end_date
)
WHERE rn BETWEEN :page_start AND :page_end
ORDER BY sale_date DESC;
```

8 Security and Access Control

8.1 Row-Level Security

```
-- Create security policy
BEGIN
  DBMS_RLS.ADD_POLICY(
    object_schema   => 'DWH',
    object_name     => 'SALES',
    policy_name     => 'REGION_ACCESS_POLICY',
    function_schema => 'SECURITY',
    policy_function => 'AUTHORIZE_BY_REGION',
    statement_types => 'SELECT',
    update_check   => TRUE);
END;
-- Policy function example
CREATE OR REPLACE FUNCTION security.authorize_by_region(
  p_schema IN VARCHAR2,
  p_object IN VARCHAR2)
RETURN VARCHAR2
IS
  v_predicate VARCHAR2(200);
BEGIN
  IF SYS_CONTEXT('USERENV','SESSION_USER') = 'REPORT_USER' THEN
    v_predicate := 'region_id IN (
      SELECT region_id FROM user_regions
      WHERE username = SYS_CONTEXT(''USERENV'',''SESSION_USER''))';
  END IF;

  RETURN v_predicate;
END;
```

9 Advanced Visualization Techniques

9.1 Time-Series Analysis

```
-- MATCH_RECOGNIZE for trend analysis
SELECT *
FROM daily_sales
MATCH_RECOGNIZE (
  PARTITION BY product_id
  ORDER BY sale_date
  MEASURES
    STRT.sale_date AS start_date,
    LAST(DOWN.sale_date) AS bottom_date,
    LAST(UP.sale_date) AS recovery_date
  ONE ROW PER MATCH
  PATTERN (STRT DOWN+ UP+)
  DEFINE
    DOWN AS amount < PREV(amount),
    UP AS amount > PREV(amount)
) mr
ORDER BY product_id, start_date;
```

10 Best Practices

10.1 Report Development Guidelines

- Modular Design: Build reusable report components
- Parameter Validation: Sanitize all user inputs
- Performance Testing: Validate with production data volumes
- Documentation: Maintain data dictionaries and lineage
- Version Control: Track report changes systematically

10.2 Performance Checklist

11 Emerging Trends

11.1 Modern Reporting Features

- Natural Language Processing: Voice-activated reporting

Table 4: Report Performance Checklist

Area	Verification
Query Design	Proper indexing and partitioning
Execution Plan	Optimal join methods and access paths
Result Size	Appropriate pagination/filtering
Caching	Effective use of result cache
Concurrency	Tested under expected user load

- Augmented Analytics: Automated insights generation
- Embedded Analytics: Reports within operational apps
- Real-time Dashboards: Streaming data visualization
- Mobile Optimization: Responsive report design

```
-- Oracle Continuous Query Notification
DECLARE
  l_regid NUMBER;
  l_cursor SYS_REFCURSOR;
BEGIN
  DBMS_CHANGE_NOTIFICATION.ENABLE_REG(
    regid => l_regid,
    callback => 'reporting.refresh_dashboard',
    qosflags => DBMS_CHANGE_NOTIFICATION.QOS_QUERY);

  OPEN l_cursor FOR
    SELECT product_id, SUM(amount)
    FROM sales
    GROUP BY product_id;

  DBMS_CHANGE_NOTIFICATION.REGISTER(
    regid => l_regid,
    cursor => l_cursor,
    operations => DBMS_CHANGE_NOTIFICATION.ALL_OPERATIONS);
END;
```

Chapter 3

ETL Processes in Data Warehousing

1 Introduction to ETL

ETL (Extract, Transform, Load) forms the backbone of data warehouse population, involving:

- Extraction: Data collection from source systems
- Transformation: Data cleansing and conversion
- Loading: Populating target data warehouse structures

2 Oracle ETL Tools Overview

2.1 Tool Comparison

Table 5: Oracle ETL Tool Comparison

Tool	Best For	Complexity
Oracle Data Integrator (ODI)	Enterprise ETL	High
SQL*Loader	Flat file loading	Low
External Tables	File processing	Medium
PL/SQL	Custom transformations	Medium
APEX Data Loading	Ad-hoc loads	Low

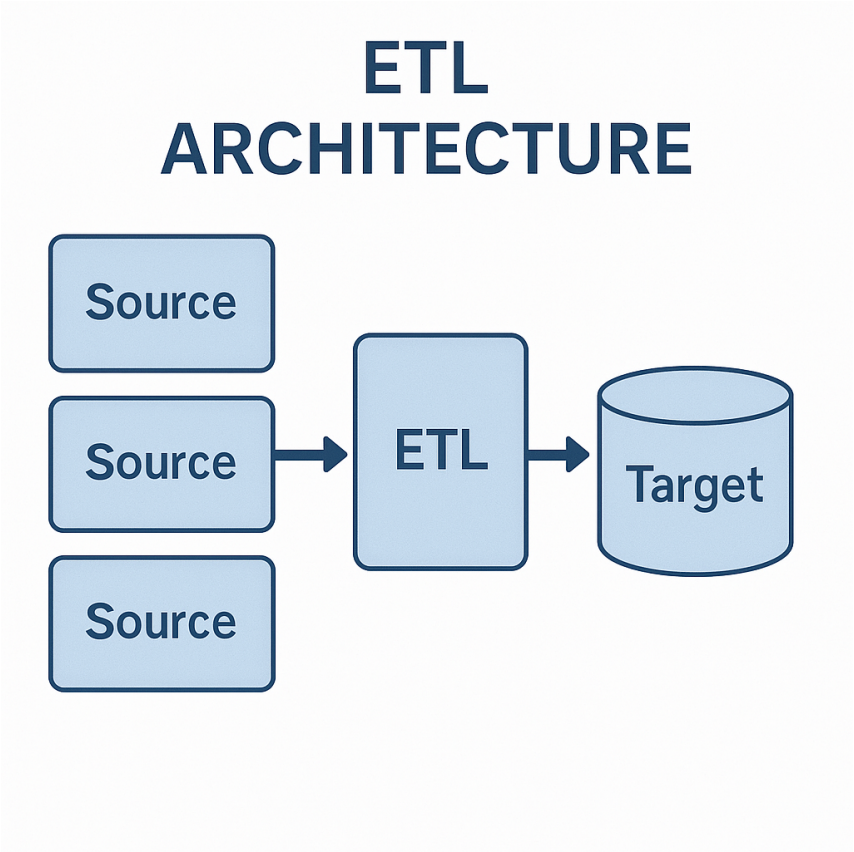

Figure 4: Typical ETL Architecture in Data Warehousing

3 Extraction Patterns

3.1 Change Data Capture (CDC)

```
-- Create change table
BEGIN
  DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE(
    owner             => 'SRC_OWNER',
    change_table_name => 'CUSTOMERS_CT',
    change_set_name   => 'DWH_CHANGE_SET',
    source_schema     => 'SRC_OWNER',
    source_table      => 'CUSTOMERS',
    column_type_list  => 'CUSTOMER_ID NUMBER, NAME VARCHAR2(100)',
    capture_values    => 'both',
    rs_id             => 'y',
    row_id            => 'y',
    user_id           => 'y',
    timestamp         => 'y',
    object_id         => 'n');
END;
-- Subscribe to changes
BEGIN
  DBMS_CDC_SUBSCRIBE.CREATE_SUBSCRIPTION(
    change_set_name   => 'DWH_CHANGE_SET',
    description       => 'Customer changes',
    subscription_name => 'CUSTOMER_SUB');

  DBMS_CDC_SUBSCRIBE.SUBSCRIBE(
    subscription_name => 'CUSTOMER_SUB',
    source_schema     => 'SRC_OWNER',
    source_table      => 'CUSTOMERS',
    column_list       => 'CUSTOMER_ID, NAME',
    subscriber_view   => 'CUSTOMERS_CHANGES');
END;
```

4 Transformation Techniques

4.1 Data Cleansing

```
-- Standardization and cleansing
CREATE OR REPLACE PROCEDURE clean_customer_data AS
BEGIN
  -- Fix phone formats
  UPDATE stage_customers
  SET phone = REGEXP_REPLACE(phone, '[^0-9]', '')
  WHERE REGEXP_LIKE(phone, '[^0-9]');

  -- Standardize addresses
  UPDATE stage_customers
  SET address = INITCAP(TRIM(address)),
      city = UPPER(city),
      state = UPPER(state);

  -- Validate emails
  UPDATE stage_customers
  SET is_valid = CASE
    WHEN REGEXP_LIKE(email,
     ↪  '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$')
    THEN 1 ELSE 0 END;

  -- Deduplicate records
  FOR dup_rec IN (
    SELECT MIN(rowid) keep_rowid, customer_id
    FROM stage_customers
    GROUP BY customer_id
    HAVING COUNT(*) > 1
  ) LOOP
    DELETE FROM stage_customers
    WHERE customer_id = dup_rec.customer_id
    AND rowid != dup_rec.keep_rowid;
  END LOOP;
END;
```

5 Loading Strategies

5.1 Bulk Loading with SQL*Loader

```
# load_customers.ctl
LOAD DATA
INFILE '/data/customers.csv'
BADFILE '/data/customers.bad'
DISCARDFILE '/data/customers.dsc'
APPEND
INTO TABLE stage_customers
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
TRAILING NULLCOLS
(
  customer_id,
  name,
  email,
  phone,
  address,
  city,
  state,
  zip,
  load_date "SYSDATE",
  load_source CONSTANT "CSV_IMPORT"
)
```

5.2 External Tables

```
CREATE OR REPLACE DIRECTORY ext_tab_dir AS '/data/external';

CREATE TABLE ext_sales (
    sale_id      NUMBER,
    sale_date    DATE,
    product_id   NUMBER,
    customer_id  NUMBER,
    amount       NUMBER(10,2)
)
ORGANIZATION EXTERNAL (
    TYPE ORACLE_LOADER
    DEFAULT DIRECTORY ext_tab_dir
    ACCESS PARAMETERS (
        RECORDS DELIMITED BY NEWLINE
        BADFILE 'sales.bad'
        LOGFILE 'sales.log'
        FIELDS TERMINATED BY '|'
        MISSING FIELD VALUES ARE NULL
    )
    LOCATION ('sales_2023.dat')
)
REJECT LIMIT UNLIMITED;

-- Query external data directly
SELECT * FROM ext_sales WHERE sale_date > SYSDATE-30;
```

6 Incremental Loading

6.1 SCD Type 2 Implementation

```
CREATE OR REPLACE PROCEDURE load_dim_customers AS
BEGIN
  -- Insert new records and changed records
  INSERT INTO dim_customers (
    customer_key,
    customer_id,
    name,
    email,
    effective_date,
    expiry_date,
    current_flag
  )
  SELECT
    dim_cust_seq.NEXTVAL,
    s.customer_id,
    s.name,
    s.email,
    SYSDATE,
    TO_DATE('31-DEC-9999','DD-MON-YYYY'),
    'Y'
  FROM
    stage_customers s
    LEFT JOIN dim_customers d ON s.customer_id = d.customer_id
                              AND d.current_flag = 'Y'
  WHERE
    d.customer_key IS NULL OR
    (s.name != d.name OR s.email != d.email);

  -- Expire changed records
  UPDATE dim_customers d
  SET current_flag = 'N',
      expiry_date = SYSDATE-1
  WHERE current_flag = 'Y'
  AND EXISTS (
    SELECT 1 FROM stage_customers s
    WHERE s.customer_id = d.customer_id
    AND (s.name != d.name OR s.email != d.email)
  );

  COMMIT;
END;
```

7 High-Performance ETL Techniques

7.1 Parallel DML

```
-- Enable parallel DML
ALTER SESSION ENABLE PARALLEL DML;

-- Parallel direct-path insert
INSERT /*+ APPEND PARALLEL(8) */ INTO sales_fact
SELECT /*+ PARALLEL(8) FULL(s) */
       s.sale_id,
       s.sale_date,
       c.customer_key,
       p.product_key,
       s.amount
FROM
    stage_sales s
    JOIN dim_customers c ON s.customer_id = c.customer_id
    JOIN dim_products p ON s.product_id = p.product_id
WHERE
    s.sale_date BETWEEN :start_date AND :end_date;

COMMIT;
```

7.2 Partition Exchange Loading

```
-- Load data into staging table
INSERT /*+ APPEND */ INTO sales_stage
SELECT * FROM external_sales_source;

-- Create constraints/indexes on stage table
ALTER TABLE sales_stage ADD CONSTRAINT pk_stage
PRIMARY KEY (sale_id);

-- Exchange partition
ALTER TABLE sales_fact
EXCHANGE PARTITION sales_2023_05
WITH TABLE sales_stage
INCLUDING INDEXES
WITHOUT VALIDATION;

-- Update global indexes
ALTER TABLE sales_fact
UPDATE GLOBAL INDEXES;
```

8 Error Handling and Recovery

8.1 ETL Auditing Framework

```
CREATE TABLE etl_audit (
    audit_id        NUMBER GENERATED ALWAYS AS IDENTITY,
    process_name    VARCHAR2(100),
    start_time      TIMESTAMP,
    end_time        TIMESTAMP,
    rows_processed  NUMBER,
    status          VARCHAR2(20),
    error_message   VARCHAR2(4000),
    CONSTRAINT pk_etl_audit PRIMARY KEY (audit_id)
);

CREATE OR REPLACE PROCEDURE log_etl_event (
    p_process     IN VARCHAR2,
    p_status      IN VARCHAR2,
    p_rows        IN NUMBER DEFAULT NULL,
    p_error       IN VARCHAR2 DEFAULT NULL
) AS
    v_audit_id NUMBER;
BEGIN
    IF p_status = 'START' THEN
        INSERT INTO etl_audit (process_name, start_time, status)
        VALUES (p_process, SYSTIMESTAMP, p_status)
        RETURNING audit_id INTO v_audit_id;
    ELSE
        UPDATE etl_audit
        SET end_time = SYSTIMESTAMP,
            status = p_status,
            rows_processed = p_rows,
            error_message = p_error
        WHERE process_name = p_process
        AND status = 'START'
        AND end_time IS NULL;
    END IF;

    COMMIT;
END;
```

9 Optimizing ETL Performance

9.1 Performance Tuning Checklist

Table 6: ETL Performance Optimization Techniques

Area	Optimization
Extraction	Use change data capture
Transformation	Push processing to database
Loading	Direct-path inserts
Parallelism	Configure appropriate DOP
Memory	Optimize PGA allocation
Partitioning	Implement partition exchange

9.2 Memory Configuration

```
-- Configure memory for ETL operations
ALTER SYSTEM SET pga_aggregate_target=8G;
ALTER SYSTEM SET memory_target=16G SCOPE=SPFILE;

-- Session-level memory settings
ALTER SESSION SET sort_area_size=256M;
ALTER SESSION SET hash_area_size=512M;
```

10 Best Practices

10.1 ETL Design Principles

- Modularity: Build reusable components
- Recoverability: Implement checkpoint restart
- Monitoring: Comprehensive logging
- Performance: Design for throughput
- Maintainability: Clear documentation

10.2 Oracle-Specific Recommendations

```
CREATE OR REPLACE PROCEDURE run_etl_process AS
    v_rows NUMBER;
    v_start TIMESTAMP := SYSTIMESTAMP;
BEGIN
    -- Log start
    log_etl_event('DAILY_SALES_LOAD', 'START');

    -- Extract phase
    extract_sales_data(v_rows);

    -- Transform phase
    transform_sales_data(v_rows);

    -- Load phase
    load_sales_fact(v_rows);

    -- Log completion
    log_etl_event('DAILY_SALES_LOAD', 'COMPLETE', v_rows);

    -- Handle exceptions
EXCEPTION
    WHEN OTHERS THEN
        log_etl_event('DAILY_SALES_LOAD', 'ERROR', v_rows,
                      SQLERRM);
        RAISE;
END;
```

11 Emerging Trends

11.1 Modern ETL Approaches

- ELT: Transform after loading
- Streaming ETL: Real-time processing
- Cloud ETL: Serverless architectures
- Data Mesh: Distributed ownership
- ML Integration: Embedded transformations

```
-- Oracle Autonomous Data Warehouse ETL
BEGIN
  DBMS_CLOUD.CREATE_CREDENTIAL(
    credential_name => 'OBJ_STORE_CRED',
    username => 'cloud_user',
    password => 'secure_password');

  DBMS_CLOUD.COPY_DATA(
    table_name => 'STAGE_SALES',
    credential_name => 'OBJ_STORE_CRED',
    file_uri_list =>
     ↪  'https://t.oraclecloud.com/n/namespace/b/bucket/o/sales*.csv',
    format => json_object('type' value 'csv', 'delimiter' value
     ↪  ','));
END;
/
```

Chapter 4

Data Warehouses Indexation Strategies

1 Introduction to Data Warehouse Indexing

Indexing in data warehouses serves fundamentally different purposes compared to OLTP systems. Where traditional databases optimize for frequent small writes, data warehouses require specialized indexing strategies for:

- Large-scale analytical queries
- Complex joins across star schemas
- Aggregation operations on fact tables
- Historical data analysis

2 Oracle-Specific Indexing Techniques

2.1 B-Tree Indexes in Oracle

While B-trees remain ubiquitous, Oracle implements several optimizations for data warehousing:

```
-- Oracle B-tree index with storage parameters
CREATE INDEX idx_customer_name ON customers(cust_name)
TABLESPACE dw_indexes
STORAGE (INITIAL 256M NEXT 128M)
COMPRESS ADVANCED LOW;
```

Key considerations for Oracle:

- Use `COMPRESS ADVANCED LOW` for space savings
- Larger block sizes (8K/16K) for warehouse indexes
- Consider `NOLOGGING` for bulk loads

2.2 Bitmap Indexes in Oracle

Oracle's bitmap implementation is particularly suited for data warehouses:

```
-- Oracle bitmap index example
CREATE BITMAP INDEX idx_sales_channel ON sales(sales_channel)
TABLESPACE bitmap_indexes
COMPUTE STATISTICS;
```

Best practices:

- Ideal for low-cardinality columns (<100 distinct values)
- Avoid on frequently updated tables
- Use `BITMAP MERGE` for efficient combination

2.3 Bitmap Join Indexes in Oracle

Oracle's implementation pre-joins dimension and fact tables:

```
CREATE BITMAP INDEX idx_sales_customer_region
ON sales(customers.region)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL NOLOGGING;
```

Performance characteristics:

- 3-10x faster for star schema queries
- 50-75% space savings over materialized views
- Automatic maintenance during ETL

3 Index Selection Methodology

3.1 Cost-Based Approach in Oracle

Oracle's DBMS_ADVISOR package provides index recommendations:

```
-- Generate index recommendations
DECLARE
  task_name VARCHAR2(30);
BEGIN
  task_name := DBMS_ADVISOR.CREATE_TASK('SQL Access Advisor');
  DBMS_ADVISOR.SET_TASK_PARAMETER(task_name, 'ANALYSIS_SCOPE',
   ↪  'INDEXES');
  DBMS_ADVISOR.EXECUTE_TASK(task_name);
END;
/

-- View recommendations
SELECT * FROM user_advisor_recommendations;
```

3.2 Implementation Example

Consider a sales data warehouse with these optimization steps:

```
-- Step 1: Create dimension table indexes
CREATE BITMAP INDEX idx_time_year ON time_dim(calendar_year)
TABLESPACE dw_indexes;

-- Step 2: Create fact table join indexes
CREATE BITMAP INDEX idx_fact_customer
ON sales_fact(customer_dim.cust_segment)
FROM sales_fact, customer_dim
WHERE sales_fact.cust_id = customer_dim.cust_id;

-- Step 3: Add B-tree indexes for high-cardinality columns
CREATE INDEX idx_sales_amount ON sales_fact(amount_sold)
TABLESPACE dw_indexes
COMPRESS;
```

4 Performance Comparison

Table 7: Index Performance Characteristics in Oracle

Index Type	Create Time	Storage Size	Query Speedup
B-Tree	Medium	Large	2-5x
Bitmap	Fast	Small	5-20x
Bitmap Join	Slow	Medium	10-50x

5 Maintenance Strategies

```
-- Rebuild fragmented indexes
ALTER INDEX idx_customer_name REBUILD ONLINE;

-- Monitor index usage
SELECT index_name, used FROM v\$object_usage;

-- Partition large indexes
CREATE INDEX idx_sales_date ON sales(sale_date)
GLOBAL PARTITION BY RANGE (sale_date)
(PARTITION p_2020 VALUES LESS THAN
  ↪  (TO_DATE('2021-01-01','YYYY-MM-DD')),
PARTITION p_2021 VALUES LESS THAN (MAXVALUE));
```

6 Conclusion

Effective data warehouse indexing in Oracle requires:

- Strategic combination of B-tree and bitmap indexes
- Proper use of partitioned indexes for large tables
- Regular monitoring and maintenance
- Workload-aware index selection

```
-- Example complete indexing strategy
BEGIN
  -- Drop unused indexes
  FOR rec IN (SELECT index_name FROM user_indexes WHERE status =
   ↪  'UNUSED') LOOP
    EXECUTE IMMEDIATE 'DROP INDEX ' || rec.index_name;
  END LOOP;

  -- Rebuild fragmented indexes
  DBMS_STATS.GATHER_SCHEMA_STATS('DW_USER');
END;
/
```

Chapter 5

Horizontal Partitioning in Data Warehouses

1 Introduction

Horizontal partitioning (HP) has become an indispensable technique in modern data warehouse design, particularly for Oracle-based systems. This physical database design technique divides table rows across multiple physical structures while maintaining a single logical view. The approach offers significant benefits for large-scale data warehouses:

- Improved Query Performance: Partition pruning eliminates unnecessary partitions from scan operations
- Enhanced Manageability: Maintenance operations can target specific partitions
- Better Availability: Individual partitions can remain available during maintenance
- Efficient Data Lifecycle Management: Aging data can be easily archived

2 Partitioning Strategies

2.1 Partitioning Key Selection

The choice of partitioning key significantly impacts performance. Ideal candidates:

- Frequently used in WHERE clauses for partition pruning
- Exhibit natural data distribution (dates, regions)

- Support common access patterns
- Have sufficient cardinality to prevent skew

2.2 Partition Granularity

Table 8: Partition Granularity Trade-offs

Granularity	Advantages	Disadvantages
Coarse (Yearly)	Fewer partitions	Less pruning opportunity
Medium (Monthly)	Balanced approach	Moderate maintenance
Fine (Daily)	Maximum pruning	High partition count

3 Oracle Partitioning Fundamentals

3.1 Partitioning Types

Oracle supports several partitioning methods:

- Range Partitioning: Ideal for time-series data
- List Partitioning: Suitable for discrete values
- Hash Partitioning: Even data distribution
- Composite Partitioning: Combines methods

3.2 Creating Partitioned Tables

```
CREATE TABLE sales (
    sale_id      NUMBER,
    sale_date    DATE,
    customer_id  NUMBER,
    amount       NUMBER(10,2)
) PARTITION BY RANGE (sale_date)
(
    PARTITION sales_2020 VALUES LESS THAN
     ↪  (TO_DATE('2021-01-01','YYYY-MM-DD')),
    PARTITION sales_2021 VALUES LESS THAN
     ↪  (TO_DATE('2022-01-01','YYYY-MM-DD')),
    PARTITION sales_max VALUES LESS THAN (MAXVALUE)
) TABLESPACE sales_data;
```

4 Horizontal partitioning modes

4.1 Range partitioning mode

The range mode is the first partitioning mode integrated in ORACLE 8. This mode uses the domain D_k of the attribute A_k used as partitioning key of R. Each range has lower and upper bounds (see the example in Fig. 4.2 below)

Example The Fig. 5 illustrates a range partitioning of the Customers

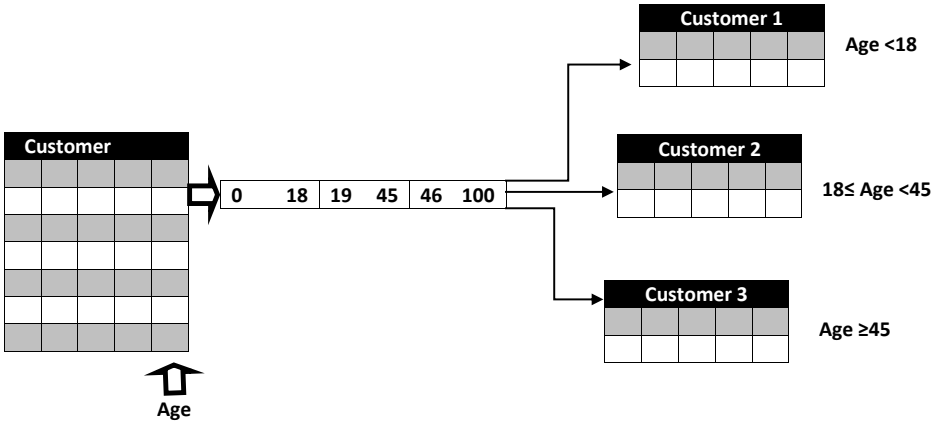

Figure 5: Range Mode.

on Age as partitioning key. The following ORACLE statement allows rang partitioning of Customers:

```
1  CREATE TABLE Customers
2  (CID number(9), Name varchar(25), City varchar(25),
3  Gender char(1), Age number(3)
4  PARTITION BY RANGE(Gender)
5  (PARTITION C-Childs VALUES LESS THAN (18) TABLESPACE TBS-Childs,
6  PARTITION C-Adults VALUES LESS THAN (45) TABLESPACE TBS-Adults,
7  PARTITION C-Olds VALUES LESS THAN (MAXVALUE) TABLESPACE TBS-Olds) ;
```

- The PARTITION BY RANGE clause specifies that range-based partitioning is being used. Each partition is assigned a name, such as $C\_Infants$, which represents the partition containing tuples where $Age < 18$.
- The TABLESPACE clause allows each partition to be stored in a predefined physical space.

When a tuple is inserted into the relation R, it is automatically placed into the appropriate partition based on the value of the 'Age' column. For instance, if a tuple with $Age = 40$ is inserted, the DBMS first compares the 'Age' value with the upper bound of the smallest partition. Finding that

40 > 18, the system proceeds to the next partition. It then checks 40 < 45 and inserts the tuple into the corresponding partition.

This partitioning mode is particularly effective for queries with range-based restriction predicates. For example:

```
1 SELECT Name FROM Customers
2 WHERE Age > 45;
```

In this case, the DBMS only loads the partition stored in the *TBS_Olds* Tablespace to answer the query, optimizing performance.

4.2 Hash partitioning mode

This mode utilizes a hashing algorithm provided by the DBMS. The user is

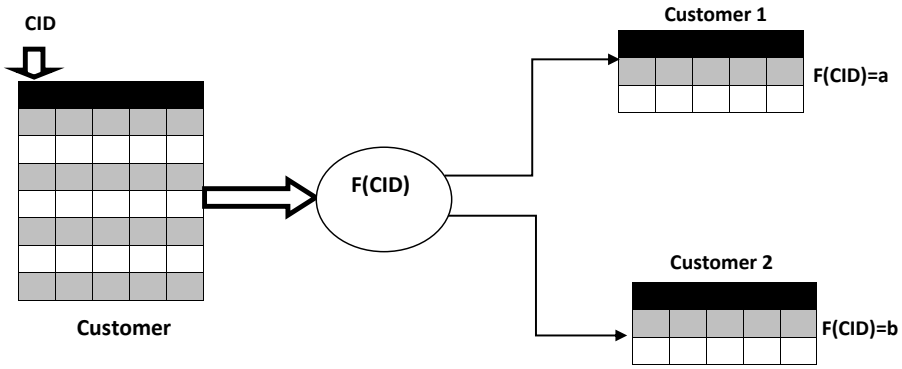

Figure 6: Hach Mode.

required to specify the partitioning key and the desired number of partitions. The hashing algorithm ensures an even distribution of tuples across the partitions, resulting in partitions of approximately equal size (see Fig. 7).

- Example: The following statement demonstrates the partitioning of the ‘Customers‘ table into four partitions using the ‘CID‘ attribute as the partitioning key. Each partition is stored in a separate TABLESPACE (TBS1, TBS2, TBS3, and TBS4).

```
CREATE TABLE CUSTOMER (CID number(9), Name varchar(25),
City varchar(25),Gender char(1), Age number(3))
PARTITION BY HASH (CID)
PARTITION 4 STORE IN (TBS1, TBS2, TBS4, TBS4) ;
```

The partitions names are automatically assigned by DBMS during the partitioning process.

4.3 List partitioning mode

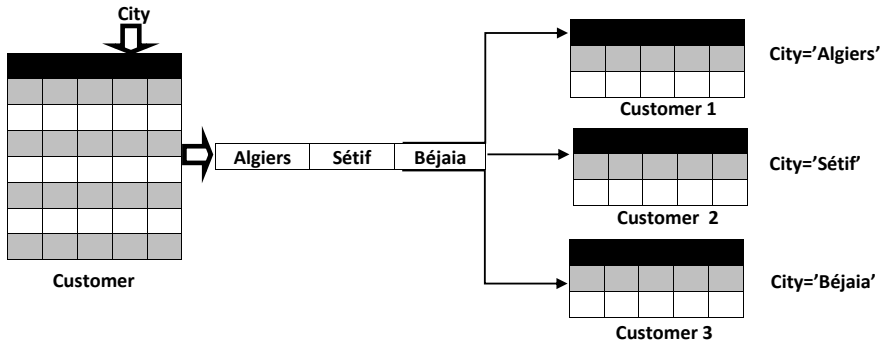

Figure 7: List mode.

List partitioning allows partitions to be defined based on a list of discrete values for the partitioning key. This method enables the grouping and organization of unordered and unrelated sets of data in an intuitive and logical manner.

- Example The following statement demonstrates the partitioning of the 'Customer' relation into four partitions using the list mode, with the 'City' attribute as the partitioning key. The four partitions contain customers from Setif, Bejaia, Algiers, and other cities, respectively (see Fig. 7).

```
1 CREATE TABLE CUSTOMER (CID number(9), Name varchar(25), City
   ↪ varchar(25),
2 Gender char(1), Age number(3))
3 PARTITION BY LIST (City)
4 (PARTITION C-Setif VALUES ('Setif'),
5 PARTITION C-Bejaia VALUES ('Bejaia'),
6 PARTITION C-Algiers VALUES ('Algiers'),
7 PARTITION C-Otherwise VALUES (DEFAULT)) ;
```

4.4 Composite partitioning mode

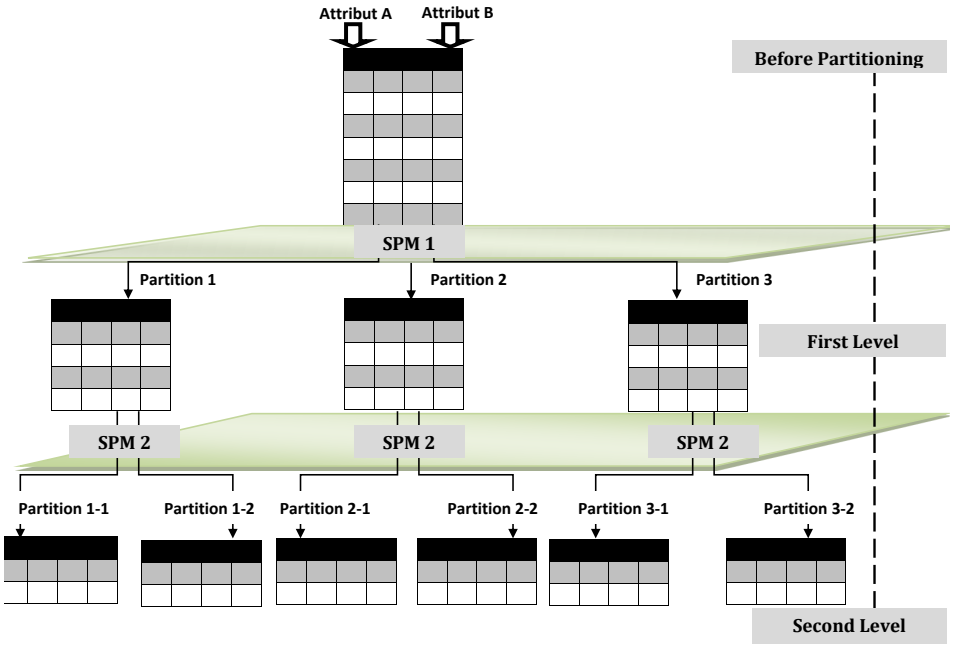

Figure 8: Composite partitioning mode.

Composite partitioning mode (CPO) combines two single partitioning modes, SPM1 and SPM2 (see Fig. ??). In this approach, the relation is first partitioned using SPM1, and then each resulting partition is further subdivided into sub-partitions using SPM2 whiteoracle. Several composite partitioning modes are obtained by combining single partitioning modes.

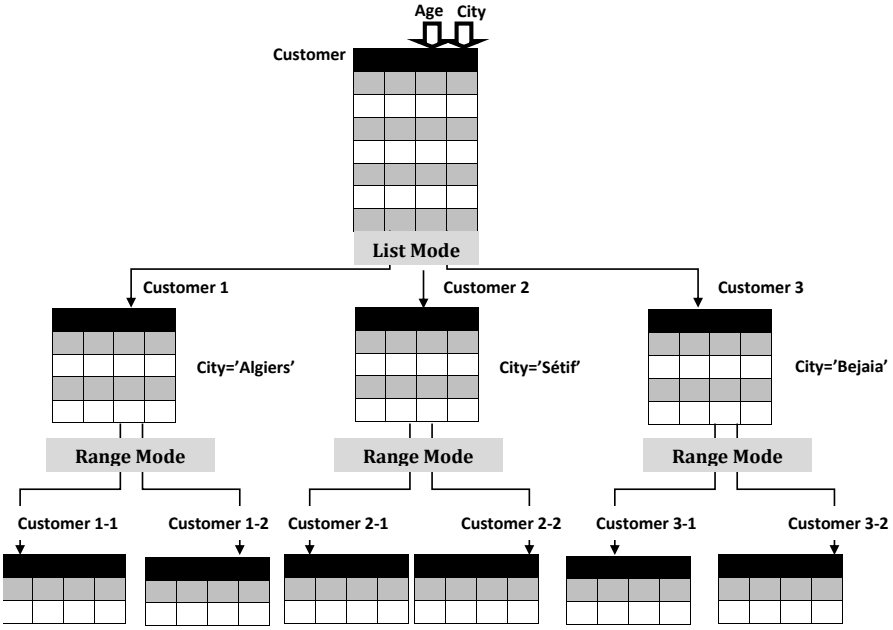

Figure 9: Example of composite partitioning mode.

The 'Customer' relation is first partitioned using 'Gender' as the partitioning key. Each resulting partition is then further subdivided into subpartitions using 'Age' as the partitioning key (see Fig. 9). This is achieved using the following statement:

```
CREATE TABLE CUSTOMER
(CID number(9), Name varchar(25), City varchar(25),
Gender char(1), Age number(3)
PARTITION BY LIST (Gender)
SUBPARTITION BY RANGE (Age)
SUBPARTITION TEMPLATE
(SUBPARTITION C-Childs VALUES LESS THAN (16) TABLESPACE TBS-Childs,
SUBPARTITION C-Adults VALUES LESS THAN (MAXVALUE) TABLESPACE
  ↪ TBS-Adults))
(PARTITION C-Setif VALUES ('Setif'),
PARTITION C-Bejaia VALUES ('Bejaia'),
PARTITION C-Algiers VALUES ('Algiers')
PARTITION C-Otherwise VALUES (DEFAULT));
```

4.5 Multicolumn partitioning mode

The multicolumn partitioning mode combines range and hash partitioning methods, allowing up to 16 partitioning key columns. In this mode, the partitioning key, composed of multiple columns, provides finer granularity compared to single-column partitioning. A common example is a decomposed 'DATE' column, split into separate 'year', 'month', and 'day' columns. In DBMS, the n^{th} partitioning key is evaluated only when the values of the preceding $n-1$ keys exactly match the bounds of the corresponding $n-1$ partitions.

The following example illustrates the range partitioning of the relation Sales using two key partitioning Year and Month:

```
CREATE TABLE sales (
   Year          NUMBER,
   Month         NUMBER,
   Day           NUMBER,
   Amount   NUMBER)
PARTITION BY RANGE (Year,Month)
  (PARTITION before2014 VALUES LESS THAN (2014,1),
   PARTITION q1_2014    VALUES LESS THAN (2014,4),
   PARTITION q2_2014    VALUES LESS THAN (2014,7),
   PARTITION q3_2014    VALUES LESS THAN (2014,10),
   PARTITION q4_2014    VALUES LESS THAN (2014,1),
   PARTITION future     VALUES LESS THAN (MAXVALUE,0));
```

4.6 Reference partitioning mode

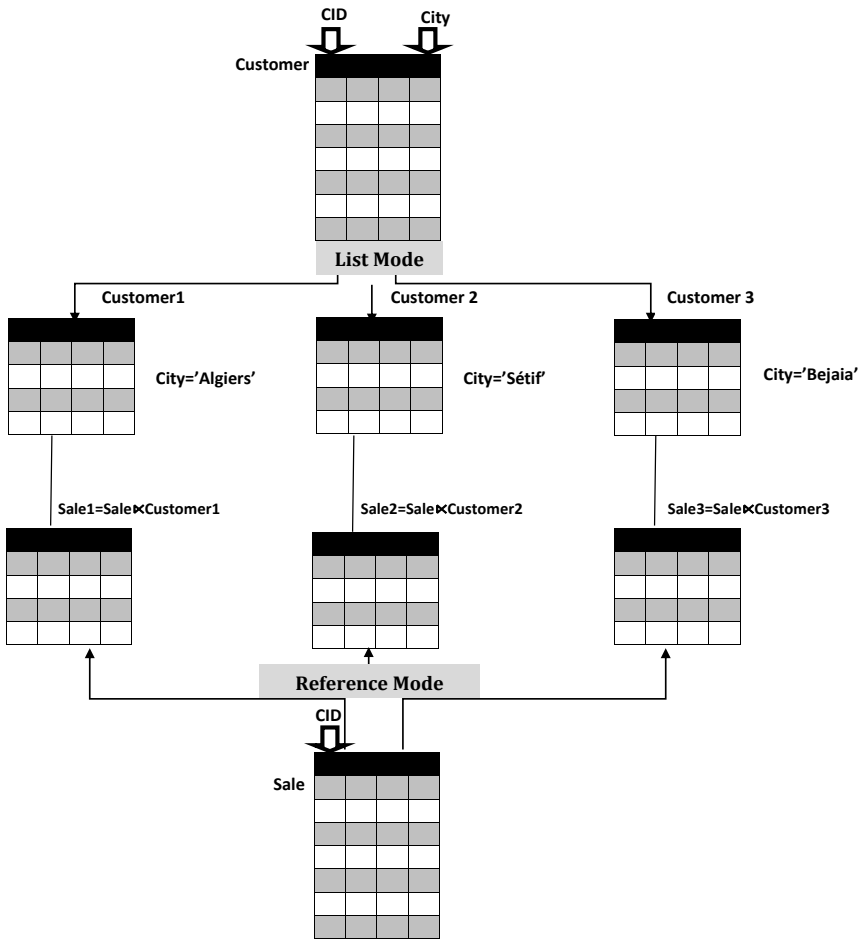

Figure 10: Example of reference partitioning mode.

Previously, we discussed single and composite partitioning methods used for partitioning individual relations. In this section, we introduce the reference partitioning mode, as implemented in the Oracle 11g environment. Reference partitioning enables the partitioning of two related relations, R and S, which are connected through referential constraints. The partitioning key is determined based on the existing parent-child relationship, enforced by active and enabled primary key and foreign key constraints whiteoracle.

First, the relation R is partitioned using either a single or composite partitioning mode. If a single partitioning mode is applied to R, the number of partitions in R will be the same as the number of partitions in S. In contrast, if a composite partitioning mode is used for R, the number of partitions in R will correspond to the number of sub-partitions in S.

- Example
  The ‘Customer‘ relation is divided into three partitions: Customer1, Customer2, and Customer3 (see Fig. 10) using the List partitioning mode. Subsequently, three ‘Sales‘ partitions are created, with each partition corresponding to a specific ‘Customer‘ partition. The following statement demonstrates the partitioning of the ‘Sales‘ relation into three partitions using the reference partitioning mode:

```
1 CREATE TABLE SALES
2 (CID number(9), Date DATE , Amount Number(10,2)
3 CONSTRAINT Customer_Cs FOREIGN KEY (CID) REFERENCES Customer(CID))
4 PARTITION BY REFERENCE(Customer_Cs);
```

4.7 Virtual column partitioning

This partitioning mode utilizes a virtual column in the same way as a regular column. All partitioning modes are supported with virtual columns, including range partitioning and various combinations of composite partitioning modes.

```
CREATE TABLE sales(
Pid NUMBER(6) NOT NULL
, Cid NUMBER NOT NULL
, Tid DATE NOT NULL
, CHid CHAR(1) NOT NULL
, PROMOid      NUMBER(6) NOT NULL
, quantitySold NUMBER(3) NOT NULL
, amountSold   NUMBER(10,2) NOT NULL
, totalAmount AS (quantitySold * amountSold)
)
PARTITION BY RANGE (Tid) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
SUBPARTITION BY RANGE(totalAmount)
SUBPARTITION TEMPLATE
   ( SUBPARTITION Psmall VALUES LESS THAN (1000)
   , SUBPARTITION Pmedium VALUES LESS THAN (5000)
   , SUBPARTITION Plarge VALUES LESS THAN (10000)
   , SUBPARTITION Pextreme VALUES LESS THAN (MAXVALUE)
   )
 (PARTITION sales_before_2007 VALUES LESS THAN
        (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
)
```

5 Partition Pruning Optimization

Partition pruning is Oracle's ability to eliminate partitions from query execution. Effective pruning requires:

- Proper predicate formulation
- Statistics on partitioned tables
- Appropriate partition key selection

```
-- Check execution plan for pruning
EXPLAIN PLAN FOR
SELECT * FROM sales
WHERE sale_date BETWEEN DATE '2021-01-01' AND DATE '2021-03-31';

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);

-- Monitor pruning effectiveness
SELECT * FROM V$SQL_PLAN
WHERE object_name = 'SALES'
AND options LIKE '%PARTITION%';
```

6 Partitioned Index Strategies

6.1 Local vs Global Indexes

Table 9: Index Partitioning Characteristics

Feature	Local Index	Global Index
Alignment	1:1 with partitions	Independent
Availability	Partition-level	Table-level
Maintenance	Automatic	Manual

6.2 Index Creation Examples

```
-- Local partitioned index
CREATE INDEX idx_sales_local ON sales(customer_id) LOCAL;

-- Global partitioned index
CREATE INDEX idx_sales_global ON sales(sale_id) GLOBAL
PARTITION BY RANGE (sale_id)
(
    PARTITION p1 VALUES LESS THAN (1000000),
    PARTITION p2 VALUES LESS THAN (MAXVALUE)
);

-- Global non-partitioned index
CREATE INDEX idx_sales_amount ON sales(amount);
```

7 Partition Maintenance Automation

```
-- Scheduled partition maintenance
BEGIN
  DBMS_SCHEDULER.CREATE_JOB (
    job_name        => 'MAINTAIN_SALES_PARTITIONS',
    job_type        => 'PLSQL_BLOCK',
    job_action      => 'BEGIN
                          maintain_sales_partitions();
                        END;',
    start_date      => SYSTIMESTAMP,
    repeat_interval => 'FREQ=MONTHLY; BYMONTHDAY=1',
    enabled         => TRUE);
END;
/

-- Example maintenance procedure
CREATE OR REPLACE PROCEDURE maintain_sales_partitions AS
BEGIN
  -- Add next month's partition
  EXECUTE IMMEDIATE
    'ALTER TABLE sales ADD PARTITION
     ↪  sales_'||TO_CHAR(ADD_MONTHS(SYSDATE,1),'YYYY_MM')||
    ' VALUES LESS THAN
     ↪  (TO_DATE('''||TO_CHAR(ADD_MONTHS(TRUNC(SYSDATE,'MM'),2),
    'YYYY-MM-DD')||''',''YYYY-MM-DD''))';

  -- Archive old data
  archive_old_sales_data();
END;
/
```

8 Performance Considerations

8.1 Partitioning Overhead

- Increased dictionary complexity
- Additional memory requirements
- Potential for suboptimal execution plans

8.2 Monitoring Partition Usage

```
-- Identify hot partitions
SELECT table_name, partition_name, accesses
FROM (
    SELECT table_name, partition_name,
           SUM(physical_reads) accesses,
           RANK() OVER (ORDER BY SUM(physical_reads) DESC) rnk
    FROM dba_tab_partitions p
    JOIN dba_hist_seg_stat s ON p.partition_name = s.partition_name
    WHERE p.table_name = 'SALES'
    GROUP BY table_name, partition_name
)
WHERE rnk <= 5;

-- Check partition skew
SELECT partition_name, COUNT(*) row_count
FROM sales
GROUP BY partition_name
ORDER BY row_count DESC;
```

9 Real-World Implementation Case Study

Financial services data warehouse with:

- 10TB fact table partitioned by trade date (daily)
- 12 subpartitions by region
- Composite partitioning with range-hash

Performance results:

- ETL processes reduced from 6 hours to 45 minutes
- Month-end reporting queries improved from 3 hours to 12 minutes
- Backup window reduced by 80%

```
CREATE TABLE trade_facts (
    trade_id      NUMBER,
    trade_date    DATE,
    instrument_id NUMBER,
    trader_id     NUMBER,
    counterparty  NUMBER,
    amount        NUMBER(20,2),
    currency      CHAR(3),
    region_code   VARCHAR2(3)
) PARTITION BY RANGE (trade_date)
INTERVAL (NUMTODSINTERVAL(1,'DAY'))
SUBPARTITION BY HASH (instrument_id)
SUBPARTITIONS 12
(
    PARTITION p_initial VALUES LESS THAN (DATE '2000-01-01')
) PARALLEL 8;

-- Local bitmap indexes for low-cardinality columns
CREATE BITMAP INDEX bidx_trade_curr ON trade_facts(currency) LOCAL;
CREATE BITMAP INDEX bidx_trade_region ON trade_facts(region_code)
    LOCAL;
```

10 Conclusion and Best Practices

- Design for Pruning: Structure partitions to match common query patterns
- Monitor Growth: Implement automated partition maintenance
- Balance Granularity: Avoid excessive partition counts
- Consider Storage: Place active partitions on faster storage
- Test Thoroughly: Validate partition strategies with realistic workloads

```
-- Comprehensive partition analysis
SELECT p.table_name, p.partition_name, p.tablespace_name,
       p.high_value, s.bytes/1024/1024 size_mb,
       nvl(s.num_rows,0) row_count
FROM dba_tab_partitions p
LEFT JOIN dba_tab_statistics s
  ON p.table_name = s.table_name
 AND p.partition_name = s.partition_name
WHERE p.table_name = 'TRADE_FACTS'
ORDER BY p.partition_position;
```

Chapter 6

Materialized Views in Data Warehouses

Materialized views (MVs) are one of the most powerful optimization techniques in data warehousing, providing pre-computed results for complex queries. Unlike regular views that execute queries on demand, MVs store the actual result sets physically.

1 Materialized View Selection Problem

The materialized view selection problem is a critical challenge in data warehouse design, involving the identification of the optimal set of views to materialize under resource constraints.

1.1 Problem Formulation

Given:

- A set of queries $Q = \{q_1, q_2, ..., q_n\}$ with frequencies $f(q_i)$
- A set of candidate materialized views $V = \{v_1, v_2, ..., v_m\}$
- Storage space constraint S
- Maintenance cost constraint M

Objective: Select a subset $V' \subseteq V$ that:

- Minimizes total query processing cost $\sum_{q \in Q} f(q) \cdot cost(q, V')$
- Satisfies $\sum_{v \in V'} size(v) \leq S$
- Satisfies $\sum_{v \in V'} maintenance\_cost(v) \leq M$

1.2 Selection Algorithms

Common approaches include:

- Greedy Algorithms: Iteratively select views offering the highest benefit-to-size ratio
- Genetic Algorithms: Use evolutionary techniques to find near-optimal solutions
- Integer Programming: Formulate as optimization problem with constraints

Algorithm 1 Greedy Materialized View Selection

```
Input: Queries Q, Views V, Space S
V' ← ∅                                              ▷ Selected views
remaining_space ← S
while remaining_space > 0 do
    for each v ∈ V \ V' do
        benefit[v] ← query_cost_reduction(v, Q)
        ratio[v] ← benefit[v]/size(v)
    end for
    best ← argmax(ratio)
    if size(best) ≤ remaining_space then
        V' ← V' ∪ {best}
        remaining_space ← remaining_space − size(best)
    else
        Break
    end if
end while
return V'
```

2 Pruning Techniques

Pruning reduces the search space for materialized view selection by eliminating suboptimal candidates.

2.1 Dominance Pruning

A view v_1 dominates v_2 if:

- v_1 can answer all queries that v_2 can answer
- $size(v_1) \leq size(v_2)$
- $maintenance\_cost(v_1) \leq maintenance\_cost(v_2)$

Dominated views can be safely pruned from consideration.

2.2 Constraint-Based Pruning

Eliminate views that:

- Exceed storage constraints ($size(v_i) > S$)
- Have maintenance costs exceeding thresholds
- Provide minimal query performance improvement ($benefit(v_i) < \epsilon$)

2.3 Multi-Query Optimization Pruning

Identify common subexpressions across queries and materialize only the most beneficial shared components.

3 Oracle Materialized View Fundamentals

```
CREATE MATERIALIZED VIEW mv_sales_summary
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE
AS
SELECT
    t.calendar_year,
    p.product_category,
    c.cust_region,
    SUM(s.amount_sold) AS total_sales,
    COUNT(*) AS transaction_count
FROM
    sales s
    JOIN times t ON s.time_id = t.time_id
    JOIN products p ON s.prod_id = p.prod_id
    JOIN customers c ON s.cust_id = c.cust_id
GROUP BY
    t.calendar_year,
    p.product_category,
    c.cust_region;
```

4 Refresh Mechanisms

Oracle provides several refresh options:

```
-- Fast refresh (incremental)
CREATE MATERIALIZED VIEW mv_daily_sales
REFRESH FAST ON COMMIT
AS
SELECT
    TRUNC(sale_date) AS day,
    product_id,
    SUM(amount) AS daily_total
FROM
    sales
GROUP BY
    TRUNC(sale_date),
    product_id;

-- Scheduled complete refresh
CREATE MATERIALIZED VIEW mv_monthly_summary
REFRESH COMPLETE
START WITH SYSDATE NEXT SYSDATE+1
AS
SELECT /* monthly aggregation query */;
```

5 Query Rewrite

Oracle's query rewrite automatically redirects queries to use MVs:

```
-- Enable system-wide query rewrite
ALTER SYSTEM SET query_rewrite_enabled=TRUE;

-- Verify rewrite capability
EXPLAIN PLAN FOR
SELECT
    t.calendar_year,
    SUM(s.amount_sold)
FROM
    sales s
    JOIN times t ON s.time_id = t.time_id
GROUP BY
    t.calendar_year;

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

6 Partitioned Materialized Views

For large datasets, partition MVs by time ranges:

```
CREATE MATERIALIZED VIEW mv_sales_quarterly
PARTITION BY RANGE (quarter_end_date)
(
    PARTITION p_2020_q1 VALUES LESS THAN
     ↪  (TO_DATE('2020-04-01','YYYY-MM-DD')),
    PARTITION p_2020_q2 VALUES LESS THAN
     ↪  (TO_DATE('2020-07-01','YYYY-MM-DD'))
)
REFRESH COMPLETE ON DEMAND
AS
SELECT
    TRUNC(time_id,'Q') AS quarter_end_date,
    product_category,
    AVG(amount_sold) AS avg_sales
FROM
    sales s
    JOIN products p ON s.prod_id = p.prod_id
GROUP BY
    TRUNC(time_id,'Q'),
    product_category;
```

7 MV Maintenance Best Practices

```
-- Monitor MV refresh status
SELECT mview_name, last_refresh_date, staleness
FROM user_mviews;

-- Gather statistics regularly
EXEC DBMS_STATS.GATHER_TABLE_STATS('DW_SCHEMA','MV_SALES_SUMMARY');

-- Parallel refresh for large MVs
ALTER MATERIALIZED VIEW mv_large_sales
REFRESH COMPLETE
PARALLEL 8;
```

8 Advanced MV Selection in Oracle

Oracle provides tools for automated MV selection:

```
-- Use SQL Access Advisor for MV recommendations
DECLARE
  taskname VARCHAR2(30) := 'MV_ADVISOR_TASK';
BEGIN
  DBMS_ADVISOR.CREATE_TASK('SQL Access Advisor', taskname);
  DBMS_ADVISOR.CREATE_SQLWKLD(taskname);
  DBMS_ADVISOR.ADD_SQLWKLD_REF(
    taskname, 'SELECT /* frequent query */');
  DBMS_ADVISOR.SET_TASK_PARAMETER(
    taskname, 'STORAGE_LIMIT', '10GB');
  DBMS_ADVISOR.EXECUTE_TASK(taskname);
END;
/

-- View recommendations
SELECT rec_id, rank, benefit, command
FROM user_advisor_recommendations
WHERE task_name = 'MV_ADVISOR_TASK';
```

9 Real-World Example

Retail data warehouse implementation:

```
CREATE MATERIALIZED VIEW mv_retail_analysis
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE
AS
SELECT
    t.calendar_year,
    t.calendar_quarter_desc,
    c.country_id,
    c.cust_income_level,
    p.prod_category,
    p.prod_subcategory,
    c.channel_desc,
    p.prod_list_price,
    SUM(s.amount_sold) AS sum_amount,
    SUM(s.quantity_sold) AS sum_quantity,
    COUNT(*) AS cnt
FROM
    sales s
    JOIN times t ON s.time_id = t.time_id
    JOIN customers c ON s.cust_id = c.cust_id
    JOIN products p ON s.prod_id = p.prod_id
    JOIN channels c ON s.channel_id = c.channel_id
GROUP BY
    ROLLUP(t.calendar_year, t.calendar_quarter_desc),
    c.country_id,
    c.cust_income_level,
    CUBE(p.prod_category, p.prod_subcategory),
    c.channel_desc,
    p.prod_list_price;
```

Performance impact observed:

- 15x faster for monthly reporting queries
- 80% reduction in CPU usage for dashboard queries
- 40% shorter ETL windows

Table 10: Comparison of Optimization Techniques

Characteristic	Materialized Views	Indexes
Storage Overhead	High	Medium
Maintenance Cost	High	Low-Medium
Query Scope	Complex queries	Single table access
Freshness	Periodic	Real-time
Best For	Aggregations	Predicate filtering

10 Materialized View vs. Indexes

11 Advanced Features

```
-- Nested materialized views
CREATE MATERIALIZED VIEW mv_nested
REFRESH COMPLETE
AS
SELECT
    calendar_year,
    product_category,
    SUM(sum_amount) AS yearly_category_sales
FROM
    mv_sales_summary
GROUP BY
    calendar_year,
    product_category;

-- Real-time materialized views (Oracle 12c+)
CREATE MATERIALIZED VIEW mv_real_time
REFRESH ON STATEMENT
AS
SELECT /* real-time aggregation query */;
```

12 Conclusion

Key recommendations for materialized views:

- Focus on frequently executed complex queries
- Balance refresh frequency with data freshness needs

- Monitor query rewrite effectiveness
- Consider partitioning for large MVs
- Combine with indexes for optimal performance
- Use systematic selection and pruning approaches
- Leverage database advisor tools for recommendations

```
BEGIN
  DBMS_MVIEW.REFRESH('MV_SALES_SUMMARY', method => 'F');
  DBMS_STATS.GATHER_TABLE_STATS('DW_SCHEMA','MV_SALES_SUMMARY');
  DBMS_ADVISOR.TUNE_MVIEW(task_name => 'MV_TUNING');
END;
/
```

Chapter 7

Data Warehouse Administration and Tuning

1 Introduction to DWH Administration

Data warehouse administration requires specialized skills combining database management, performance tuning, and business intelligence expertise. Key responsibilities include:

- Capacity Planning: Forecasting storage and compute requirements
- Performance Management: Ensuring SLAs for query response times
- Data Freshness: Managing ETL schedules and refresh cycles
- Security: Implementing role-based access controls
- Availability: Designing for high availability and disaster recovery

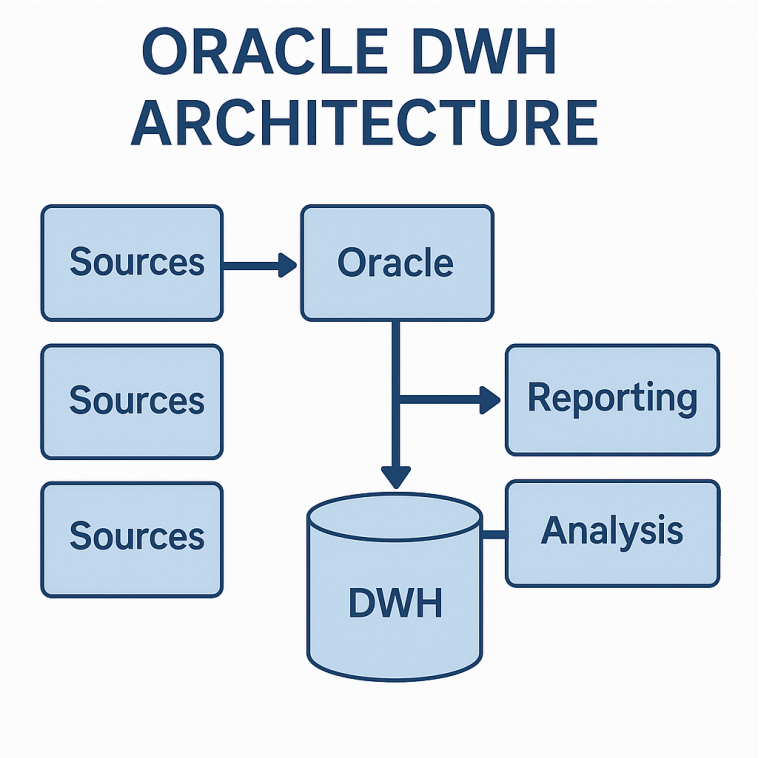

Figure 11: Oracle Data Warehouse Reference Architecture

2 Oracle Data Warehouse Architecture

3 Storage Management

3.1 Tablespace Strategy

```
-- Create dedicated tablespaces for DWH components
CREATE TABLESPACE dwh_data
DATAFILE '+DATA' SIZE 100G AUTOEXTEND ON NEXT 10G
EXTENT MANAGEMENT LOCAL UNIFORM SIZE 128M
SEGMENT SPACE MANAGEMENT AUTO;

CREATE TABLESPACE dwh_index
DATAFILE '+DATA' SIZE 50G AUTOEXTEND ON NEXT 5G
EXTENT MANAGEMENT LOCAL UNIFORM SIZE 64M;

CREATE TABLESPACE dwh_temp
TEMPFILE '+TEMP' SIZE 20G AUTOEXTEND ON NEXT 2G
EXTENT MANAGEMENT LOCAL UNIFORM SIZE 64M;
```

3.2 Compression Techniques

Table 11: Oracle Compression Methods for Data Warehouses

Type	Compression Ratio	Best For
Basic Table Compression	2x-4x	Historical data
Advanced Compression	3x-6x	All data
Hybrid Columnar Compression	10x-15x	Analytic tables

```
-- Enable HCC on partitioned tables
ALTER TABLE sales MOVE PARTITION sales_2020
COMPRESS FOR QUERY HIGH
TABLESPACE dwh_data;

-- Compress indexes
ALTER INDEX sales_pk REBUILD COMPRESS ADVANCED HIGH;
```

4 Performance Tuning Methodology

4.1 Tuning Approach

1. Identify: Pinpoint performance bottlenecks
2. Diagnose: Analyze execution plans and statistics
3. Implement: Apply appropriate tuning techniques
4. Validate: Measure improvement impact
5. Monitor: Establish ongoing performance baselines

4.2 Key Performance Metrics

Table 12: Critical DWH Performance Metrics

Metric	Target	Monitoring Query
Buffer Cache Hit Ratio	>95%	`V$SYSSTAT`
Library Cache Hit Ratio	>98%	`V$LIBRARYCACHE`
Disk I/O per Second	<100	`V$FILESTAT`
Parallel Query Utilization	70-90%	`V$PQ_SYSSTAT`

5 Query Optimization Techniques

5.1 Optimizer Statistics

```
-- Configure automated stats collection
BEGIN
  DBMS_STATS.SET_GLOBAL_PREFS(
    'AUTOSTATS_TARGET', 'ORACLE');

  DBMS_STATS.SET_TABLE_PREFS(
    'DWH_SCHEMA', 'SALES',
    'INCREMENTAL', 'TRUE');
END;
/

-- Gather extended statistics
EXEC DBMS_STATS.GATHER_TABLE_STATS(
  ownname => 'DWH_SCHEMA',
  tabname => 'SALES',
  method_opt => 'FOR ALL COLUMNS SIZE SKEWONLY',
  degree => 8);
```

5.2 SQL Plan Management

```
-- Capture and maintain plan baselines
ALTER SYSTEM SET optimizer_capture_sql_plan_baselines=TRUE;

-- Evolve plans over time
SET SERVEROUTPUT ON
DECLARE
  report CLOB;
BEGIN
  report := DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE(
    sql_handle => 'SYS_SQL_abc123');
  DBMS_OUTPUT.PUT_LINE(report);
END;
/
```

6 Parallel Execution Tuning

6.1 Configuration Parameters

```
-- Configure parallel execution
ALTER SYSTEM SET parallel_degree_policy='AUTO';
ALTER SYSTEM SET parallel_servers_target=64;
ALTER SYSTEM SET parallel_max_servers=128;

-- Table-level parallel settings
ALTER TABLE sales PARALLEL 16;
ALTER INDEX sales_pk PARALLEL 8;
```

6.2 Monitoring Parallel Queries

```
-- Active parallel queries
SELECT sid, serial#, username, sql_id,
       degree, req_degree, px_servers
FROM v$px_session p, v$session s
WHERE p.sid = s.sid;

-- Parallel query performance
SELECT sql_id, plan_hash_value,
       elapsed_time/1000000 secs,
       px_servers, executions
FROM v$sql
WHERE px_servers > 0
ORDER BY elapsed_time DESC;
```

7 ETL Process Optimization

7.1 ETL Tuning Techniques

- Direct Path Loads: Bypass buffer cache for bulk operations
- Partition Exchange Loading: Instant data swaps
- Parallel DML: Accelerate transformations
- Incremental Loading: Process only changed data

```
-- High-performance ETL pattern
INSERT /*+ APPEND PARALLEL(8) */ INTO sales_target
SELECT /*+ PARALLEL(8) FULL(s) */
       s.*
FROM sales_source s
WHERE s.load_date > TRUNC(SYSDATE)-1;

COMMIT;

-- Partition exchange loading
ALTER TABLE sales_staging
EXCHANGE PARTITION p_current
WITH TABLE sales_new
INCLUDING INDEXES;
```

8 Resource Management

8.1 Database Resource Manager

```
-- Create resource plan
BEGIN
  DBMS_RESOURCE_MANAGER.CREATE_PLAN(
    plan => 'DWH_PLAN',
    comment => 'Data Warehouse Workload Management');

  DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP(
    consumer_group => 'ETL_GROUP',
    comment => 'ETL Processing');

  DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    plan => 'DWH_PLAN',
    group_or_subplan => 'ETL_GROUP',
    comment => 'ETL Allocation',
    mgmt_p1 => 70);
END;
/

ALTER SYSTEM SET resource_manager_plan = 'DWH_PLAN';
```

9 Monitoring and Maintenance

9.1 Automated Maintenance Tasks

```
-- Create maintenance window
BEGIN
  DBMS_SCHEDULER.CREATE_WINDOW(
    window_name => 'DWH_MAINTENANCE_WINDOW',
    resource_plan => 'DWH_PLAN',
    start_date => TRUNC(SYSDATE)+22/24, -- 10PM
    duration => INTERVAL '4' HOUR,
    repeat_interval => 'FREQ=DAILY');
END;
/

-- Segment advisor job
BEGIN
  DBMS_AUTO_TASK_ADMIN.ENABLE(
    client_name => 'auto space advisor',
    operation => NULL,
    window_name => NULL);
END;
/
```

10 Troubleshooting Common Issues

10.1 Performance Problem Resolution

Table 13: Common DWH Performance Issues

Symptom	Solution
Slow fact table queries	Verify partition pruning, check statistics
ETL timeouts	Increase PGA, optimize parallel DML
Disk contention	Distribute I/O across multiple devices
Memory pressure	Configure automatic memory management

11 Conclusion and Best Practices

11.1 Administration Checklist

- Implement comprehensive monitoring (AWR, ASH)
- Establish regular maintenance windows
- Document all tuning changes
- Test changes in non-production environments
- Review performance trends weekly

11.2 Ongoing Tuning Process

1. Capture baseline performance metrics
2. Identify top resource-intensive operations
3. Apply targeted tuning techniques
4. Validate improvements
5. Update documentation

```
-- Generate AWR report
SELECT output FROM TABLE(
  DBMS_WORKLOAD_REPOSITORY.awr_report_text(
    l_dbid => (SELECT dbid FROM v$database),
    l_inst_num => (SELECT instance_number FROM v$instance),
    l_bid => NULL,
    l_eid => NULL));

-- Check for system bottlenecks
SELECT * FROM TABLE(DBMS_SQLTUNE.report_system_monitor);
```