Chapter 10: Schema Objects - Oracle SQL Study Guide

Certification Objective 10.01: Schema Objects

Schema Objects Overview

Schema objects are database objects that are:

- Owned by a user and exist in a collection under a user account
- Examples include: **Tables**, **Views**, **Indexes**, **Sequences**

Tables

- Store all data in the database
- Structure stored in the **data dictionary** (metadata)
- Columns are usually ordered by creation; new columns added via `ALTER` go to the end

Constraints

- Not schema objects themselves, but restrict data in table columns
- Types:
  - `NOT NULL`
  - `UNIQUE`
  - `PRIMARY KEY`
  - `FOREIGN KEY`
  - `CHECK`
- Violations during `INSERT`, `UPDATE`, or `DELETE` cause SQL errors

Views

A view is a **stored SELECT statement** that:

- Behaves like a table but stores no data
- Can sometimes be **updatable** if it meets certain conditions

Types of Views

1. **Simple Views** - built from a single table, straightforward SELECT
2. **Complex Views** - use joins, subqueries, or aggregates

View Usage

- **Security**: mask sensitive data
- **Simplify complex queries**

Creating Views

```sql
CREATE [OR REPLACE] VIEW view_name AS SELECT ...
```

Rules for Creating Views

- Must assign **column aliases** for expressions
- Must have **valid column names**
- `OR REPLACE` overwrites existing view with no warning

Views and Constraints

- Constraints can be defined but **not enforced** unless configured for data warehousing

Indexes

- Improve query performance by creating a **lookup structure** for columns
- Created on frequently queried columns
- Automatically used if beneficial:
  - SQL engine analyzes if index helps
  - If so, redirects query to index to fetch rows directly

Sequences

- Generate unique values (e.g., for `PRIMARY KEY`)
- Independent of tables—must be explicitly tied to logic or application code

Inline Views

- A subquery used **in place of a table** in a `FROM` clause
- Useful with:
  - Complex joins
  - Aggregates
  - Sorting with `ROWNUM` (to avoid ordering issues)

Example:

```sql
SELECT * FROM (SELECT ... FROM ...) WHERE ROWNUM <= 3;
```

Creating and Using Views

Basic View Creation Example

```sql
CREATE VIEW vw_employees AS
SELECT employee_id, last_name, first_name, primary_phone
FROM employees;
```

Aliasing Expressions

```sql
-- INVALID (no alias for expression)
SELECT last_name || ', ' || first_name FROM employees;

-- FIXED with alias
SELECT last_name || ', ' || first_name AS full_name FROM employees;
```

Updatable Views

Conditions for Update/Insert/Delete

- Must reference only **one table**
- Must include all required (e.g., `NOT NULL`) columns
- Must not:
  - Use `GROUP BY`, `DISTINCT`, `SET operators`
  - Omit required columns
  - Reference more than one table (unless key-preserved)

Update Behavior

- `UPDATE` can work if constraints are satisfied
- `INSERT` fails if required column (like `employee_id`) is missing

- `DELETE` deletes entire row

ALTER VIEW

Used to:

- Add/modify/drop constraints
- Recompile invalid views

Invalid Views

- A view becomes invalid if the **underlying table structure changes**
- Recompile with:

```sql
ALTER VIEW view_name COMPILE;
```

- Cannot change a view's `SELECT` with `ALTER`; must `DROP` and `CREATE` again

Visible/Invisible Columns

Invisible Columns in Tables

Created with `INVISIBLE` keyword:

```sql
CREATE TABLE ship_admin (
  ship_admin_id NUMBER PRIMARY KEY,
  ship_id NUMBER,
  construction_cost NUMBER(14,2) INVISIBLE
);
```

Invisible Column Behavior

- `DESC` won't show invisible columns
- INSERT must **explicitly list** invisible columns, or it will fail
- `SET COLINVISIBLE ON` shows them in `DESC` in SQL*Plus

Invisible Columns in Views

- Views **ignore invisible columns** if created with `SELECT *`

- To include them:
  - Must **explicitly name** invisible columns in the SELECT
  - View will then expose them just like visible ones

Oracle SQL Sequences

What is a Sequence?

- A database object that generates unique numbers, typically for **primary keys**
- Can generate values **ascending** or **descending**
- Not tied to any specific table; it's independent

Creating Sequences

Basic syntax:

```sql
CREATE SEQUENCE sequence_name [sequence_options];
```

Options:

- `INCREMENT BY n` — Step size. Defaults to 1. Negative values create descending sequences
- `START WITH n` — First value in the sequence. Defaults:
  - Ascending: `MINVALUE` or 1
  - Descending: `MAXVALUE`
- `MAXVALUE n` / `NOMAXVALUE` — Upper limit / no upper limit (default)
- `MINVALUE n` / `NOMINVALUE` — Lower limit / no lower limit (default)
- `CYCLE` — When limit reached, restarts at opposite end
- `NOCYCLE` — Stops when limit is reached (default)

Examples:

```sql
CREATE SEQUENCE seq_order_id;
CREATE SEQUENCE seq_order_id START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE seq_order_id START WITH 10 INCREMENT BY 5;
```

Dropping a Sequence

```sql
DROP SEQUENCE sequence_name;
```

Using Sequences in SQL

Pseudocolumns:

1. **NEXTVAL**
   - Advances sequence and returns the next value
   - Must be called **first** in a session before `CURRVAL`
   - Advances **even if the statement fails** (not reset by ROLLBACK)
2. **CURRVAL**
   - Returns the current value of the sequence
   - Can only be used **after** `NEXTVAL` has been called in the same session

Example usage:

```sql
INSERT INTO orders (order_id, order_date, customer_id)
VALUES (seq_order_id.NEXTVAL, SYSDATE, 28);
```

Rules & Restrictions for NEXTVAL and CURRVAL

Cannot use NEXTVAL or CURRVAL in:

- `DEFAULT` clause of a table column
- Subqueries of CREATE VIEW or in SELECT/UPDATE/DELETE subqueries
- `WHERE` clauses
- `CHECK` constraints
- With `DISTINCT`, `UNION`, `INTERSECT`, `MINUS`

Can use them:

- Anywhere a valid expression is allowed in a SQL statement (e.g., `SELECT`, `VALUES`, calculations)

Special Notes on Sequences

- Even if a statement using `NEXTVAL` **fails**, the sequence still advances

- `CURRVAL` will **not reset** after a rollback. It holds the last generated value until the session ends
- `CYCLE` causes the sequence to wrap around to the start when limit is hit; `NOCYCLE` prevents this
- Sequences are ideal for maintaining **primary-foreign key integrity** across related tables

Oracle SQL Indexes

What Is an Index?

- A database object that stores sorted subsets of table data to speed up queries
- Created on one or more columns of a single table
- Automatically updated on every **INSERT, UPDATE, DELETE**
- Speeds up **WHERE**, **ORDER BY**, and subquery lookups (e.g., `SELECT`, `UPDATE`, `DELETE`)

Index Basics

- Cannot be created on **LOB** or **RAW** columns
- You can create **many indexes per table**, but too many can hurt performance
- The **optimizer** decides whether to use an index for a query—it's not guaranteed
- Indexes support the optimizer in building execution plans

Oracle Optimizer

- Built-in SQL engine component that determines the best query execution path
- Uses factors like index presence, data distribution, and functions
- Indexes help the optimizer by reducing rows early in processing

Implicit Indexes

- Automatically created when you define **PRIMARY KEY** or **UNIQUE** constraints
- Names generated by Oracle (`SYS_...`)
- View implicit indexes with:

```sql
SELECT * FROM USER_INDEXES WHERE TABLE_NAME = 'YOUR_TABLE';
```

Types of Indexes

Single Column Index

```sql
CREATE INDEX index_name ON table_name(column_name);
```

- Speeds up queries where `WHERE column_name = value`

Composite Index

```sql
CREATE INDEX index_name ON table_name(col1, col2);
```

- Used when `WHERE` references **all or leading columns**
- Index sorts by col1 first, then col2
- If only second column used in WHERE, **skip scanning** may still use index

Skip Scanning

- Allows use of composite index even if the **first column** is not in the `WHERE` clause
- Less efficient than full-match but better than no index

Unique Index

```sql
CREATE UNIQUE INDEX index_name ON table_name(column);
```

- Enforces column uniqueness
- Automatically created with **UNIQUE** or **PRIMARY KEY** constraints

Function-based Index

- Not tested on the exam
- Used when indexing based on the result of a function applied to a column

Index Visibility: Visible vs Invisible

- **Visible** (default): Used by optimizer
- **Invisible**: Ignored by optimizer
- Useful for testing/tuning performance
- Maintained by SQL (still updated with DML)

```sql
CREATE INDEX ix1 ON ports(port_name) INVISIBLE;
ALTER INDEX ix1 VISIBLE;
ALTER INDEX ix1 INVISIBLE;
```

Query visibility:

```sql
SELECT VISIBILITY FROM USER_INDEXES WHERE INDEX_NAME = 'IX1';
```

Multiple Indexes on Same Column Set

You can create multiple indexes if they differ by:

- **Uniqueness** (unique vs. non-unique)
- **Type** (B-tree vs. Bitmap)
- **Partitioning** (e.g., local vs. global, range vs. hash)

Only **one index** on the column set can be visible at a time.

Performance & Tuning Tips

- High **selectivity** = better performance
- Equality (`=`) comparisons use indexes best
- `LIKE 'abc%'` uses index; `LIKE '%abc'` does not
- `<>` (not equal) does **not** use indexes
- Avoid indexes on frequently modified tables with few queries

Index Maintenance

- SQL keeps indexes updated automatically
- DML statements are **slower** on indexed tables
- Avoid unnecessary indexes; periodically review and drop unused ones:

```sql
DROP INDEX index_name;
```

Index Summary for Exam

- Know **how to create**, **when they're used**, and **types of indexes**
- Understand **optimizer behavior**
- Know **when indexes are/aren't used** (e.g., selectivity, comparison types)
- Be familiar with **VISIBLE/INVISIBLE**, **implicit indexes**, and **skip scanning**

Oracle Flashback Operations

Edition Limitations

- Some features require **Oracle Enterprise Edition**
- **Flashback Query** works in **Standard Edition**, but **Flashback Table** requires **Enterprise Edition**

What Flashback Operations Can Do

- Recover dropped tables
- Undo DML changes within tables
- Analyze how data changed over time
- Compare data at different points in time
- Query data "as of" a prior time period

Recover Dropped Tables: FLASHBACK TABLE

Syntax:

```sql
FLASHBACK TABLE table_name TO BEFORE DROP;
```

Recovers table and:

- **All constraints**, except foreign keys
- **All indexes**, except bitmap join indexes
- **Granted privileges**

Optional clause:

```sql
RENAME TO new_table_name
```

Example:

```sql
FLASHBACK TABLE houdini TO BEFORE DROP;
```

Restrictions:

- You **cannot** roll back a FLASHBACK TABLE
- Does not work if table structure changed (e.g., dropped column)
- Not available in Standard Edition

Recycle Bin

- Dropped tables go into **Recycle Bin**
- Tables can be recovered if still in the bin
- View your bin:

```sql
SELECT * FROM USER_RECYCLEBIN;
```

- DBA view:

```sql
SELECT * FROM DBA_RECYCLEBIN;
```

- Enable/disable:

```sql
ALTER SESSION SET recyclebin = ON/OFF;
```

PURGE

Permanently deletes from recycle bin:

```sql
PURGE TABLE table_name;
PURGE RECYCLEBIN;
PURGE DBA_RECYCLEBIN; -- requires SYSDBA
```

Dependent Objects

Recovered:

- Indexes (system-assigned names)
- Constraints (not FOREIGN KEYs)
- Other objects like triggers are **out of exam scope**

Recover Data Within Existing Tables

Restore to a previous state:

```sql
FLASHBACK TABLE table_name TO SCN scn;
FLASHBACK TABLE table_name TO TIMESTAMP timestamp;
FLASHBACK TABLE table_name TO RESTORE POINT rp_name;
```

Requirements:

- **ROW MOVEMENT must be enabled** to use Flashback Table on existing tables:

```sql
ALTER TABLE table_name ENABLE ROW MOVEMENT;
```

Note: Flashback commits implicitly (no rollback).

Marking Time

Identify restore point using:

- `SCN` (System Change Number)
- `TIMESTAMP`
- `RESTORE POINT`

SCN (System Change Number)

- Numeric stamp for every committed transaction
- Get current SCN:

```sql
SELECT DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER FROM DUAL;
```

- View SCN per row:

```sql
SELECT ORA_ROWSCN, column_name FROM table_name;
```

Timestamps

- Use `TO_TIMESTAMP()` to convert string to timestamp:

```sql
SELECT TO_TIMESTAMP('2017-08-25 13:15:08.232349', 'RRRR-MM-DD HH24:MI:SS.FF') FROM DUAL;
```

- Accuracy: within **3 seconds**
- If exact timing needed, use **SCN**, not TIMESTAMP

Conversion Functions

- `SCN_TO_TIMESTAMP(scn)` → returns approximate timestamp
- `TIMESTAMP_TO_SCN(timestamp)` → returns SCN

Example:

```sql
SELECT TIMESTAMP_TO_SCN(SYSTIMESTAMP) FROM DUAL;
```

RESTORE POINT

Represents a saved point in time:

```sql
CREATE RESTORE POINT rp_name;
FLASHBACK TABLE table_name TO RESTORE POINT rp_name;
DROP RESTORE POINT rp_name;
```

Tracked in `V$RESTORE_POINT`.