

Advanced Database

Sample Midterm Exam

Questions & Complete Solutions

Study Material for Open-Book Exam

Section 1: Managing Database Connections

Question 1a: List Connections and PIDs

How can you retrieve a list of recent connections and process IDs (PIDs) in PostgreSQL?

Answer:

Use the **pg_stat_activity** system view to retrieve active connections and their PIDs. This view shows all current connections, queries, and session information.

```
-- Basic query to see all connections
SELECT pid, username, datname, state, query_start, query
FROM pg_stat_activity;

-- See only active connections (not idle)
SELECT pid, username, datname, application_name,
       client_addr, state, query
FROM pg_stat_activity
WHERE state = 'active';

-- See connections for a specific database
SELECT pid, username, state, query
FROM pg_stat_activity
WHERE datname = 'your_database_name';
```

Key columns: pid (process ID), username (user), datname (database), state (active/idle), query (current/last query), query_start (when query began).

Question 1b: Cancel and Terminate Connections

Assume that you need to terminate a specific connection to the PostgreSQL database. How can you cancel active queries on a connection with a given PID, and how can you terminate the connection itself?

Answer:

PostgreSQL provides two functions: **pg_cancel_backend(pid)** for canceling queries and **pg_terminate_backend(pid)** for forcefully closing connections.

```
-- Step 1: Find the PID you want to target
SELECT pid, username, datname, state, query
FROM pg_stat_activity
WHERE username = 'problematic_user';

-- Step 2: Cancel the running query (gentle approach)
SELECT pg_cancel_backend(12345); -- Replace 12345 with actual PID
-- This sends SIGINT, allowing graceful query cancellation
-- Returns TRUE if successful

-- Step 3: Terminate the connection (forceful approach)
SELECT pg_terminate_backend(12345); -- Replace 12345 with actual PID
-- This sends SIGTERM, immediately closing the connection
-- Returns TRUE if successful

-- Check if the connection is gone
SELECT pid FROM pg_stat_activity WHERE pid = 12345;
-- Should return 0 rows if terminated
```

Key differences: `pg_cancel_backend()` lets the query stop gracefully (like Ctrl+C), while `pg_terminate_backend()` forcefully kills the entire connection. Try cancel first, then terminate if needed.

Question 1c: Kill All Connections for a Role

How can you kill all connections belonging to a specific role in PostgreSQL?

Answer:

Use a query that selects all PIDs for the role and terminates them in a loop or with a subquery. This is useful when removing a role or maintenance tasks.

```
-- Method 1: Using a subquery to terminate all connections for 'dev_user'
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE username = 'dev_user'
    AND pid <> pg_backend_pid(); -- Don't kill your own connection!

-- Method 2: Terminate all connections to a specific database
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE datname = 'sample_db'
    AND pid <> pg_backend_pid();

-- Verify all connections are terminated
SELECT pid, username, datname, state
FROM pg_stat_activity
WHERE username = 'dev_user';
-- Should return 0 rows

-- Common use case: Before dropping a database
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE datname = 'old_database';

DROP DATABASE old_database;
```

Important: Always use 'pid <> pg_backend_pid()' to avoid terminating your own connection! The function pg_backend_pid() returns your current session's PID.

Section 2: Users, Roles & Permissions

Question 2a: Create User with Expiration

Create a new user in the database named 'dev_user' with a password of your choice, but ensure that the user is valid only until a specific date. Grant this user permission to read and write to a specific table named 'employee_data'.

Answer:

Use CREATE USER with VALID UNTIL clause for time-limited access, then grant specific table privileges using GRANT statement.

```
-- Step 1: Create user with password and expiration date
CREATE USER dev_user
WITH PASSWORD 'SecurePass123!'
VALID UNTIL '2025-12-31';

-- Alternative: Set expiration to exactly 6 months from now
CREATE USER dev_user
WITH PASSWORD 'SecurePass123!'
VALID UNTIL '2025-05-09'; -- Replace with desired date

-- Step 2: Grant read (SELECT) permission on employee_data table
GRANT SELECT ON employee_data TO dev_user;

-- Step 3: Grant write (INSERT, UPDATE, DELETE) permissions
GRANT INSERT, UPDATE, DELETE ON employee_data TO dev_user;

-- Or grant all table privileges at once:
GRANT SELECT, INSERT, UPDATE, DELETE ON employee_data TO dev_user;

-- Verify the user and permissions
SELECT username, valuntil FROM pg_user WHERE username = 'dev_user';

-- Verify table privileges
SELECT grantee, privilege_type
FROM information_schema.table_privileges
WHERE table_name = 'employee_data' AND grantee = 'dev_user';
```

Explanation: VALID UNTIL enforces automatic expiration - the user cannot login after that date. The GRANT statement gives specific privileges on the table. For read+write, you need SELECT (read), INSERT (add), UPDATE (modify), DELETE (remove).

Question 2b: Create Role with Inheritance

Create a new role in the database named 'dev_team' and assign the 'dev_user' to this role. Grant the 'dev_team' role permission to access all tables in the database, but ensure that this permission is inherited by all members of the 'dev_team' role.

Answer:

Create a role with INHERIT privilege (default), grant it database-wide permissions, then assign users to the role using GRANT role TO user.

```
-- Step 1: Create the dev_team role with inheritance enabled
CREATE ROLE dev_team WITH INHERIT;
-- INHERIT is default, meaning members automatically get role's privileges

-- Step 2: Grant access to all tables in a schema (e.g., public schema)
GRANT SELECT, INSERT, UPDATE, DELETE
ON ALL TABLES IN SCHEMA public
TO dev_team;

-- Step 3: Also grant privileges on future tables (important!)
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO dev_team;

-- Step 4: Assign dev_user to the dev_team role
GRANT dev_team TO dev_user;

-- Verify role membership
SELECT
    r.rolname as role_name,
    m.rolname as member_name
FROM pg_roles r
JOIN pg_auth_members ON r.oid = pg_auth_members.roleid
JOIN pg_roles m ON m.oid = pg_auth_members.member
WHERE r.rolname = 'dev_team';

-- Verify dev_user inherits permissions
SET ROLE dev_user;
SELECT current_user, session_user;
-- Should show dev_user has access to all tables
```

Key concepts: Roles can be groups. INHERIT (default) means members automatically get the role's privileges. ALTER DEFAULT PRIVILEGES ensures future tables also get the permissions. Use 'GRANT role TO user' for membership.

Question 2c: Add New Developer & Test Access

Assume that a new developer has joined the team and needs access to the database. Create a new user for this developer with a password of your choice, and add this user to the 'dev_team' role. Test the user's access to the 'employee_data'.

Answer:

Create the new user with LOGIN privilege, grant them the dev_team role membership, then test by connecting as that user and querying employee_data.

```
-- Step 1: Create new developer user with login capability
CREATE USER new_developer
WITH PASSWORD 'DevPass456!'
LOGIN;

-- Step 2: Add new developer to dev_team role
GRANT dev_team TO new_developer;

-- Step 3: Verify role membership
SELECT
    r.rolname as role_name,
    m.rolname as member_name
FROM pg_roles r
JOIN pg_auth_members ON r.oid = pg_auth_members.roleid
JOIN pg_roles m ON m.oid = pg_auth_members.member
WHERE r.rolname = 'dev_team';

-- Step 4: Test access by switching to new user
SET ROLE new_developer;

-- Test SELECT access
SELECT * FROM employee_data LIMIT 5;

-- Test INSERT access
INSERT INTO employee_data (name, email, department)
VALUES ('Test User', 'test@example.com', 'Engineering');

-- Test UPDATE access
UPDATE employee_data
SET department = 'DevOps'
WHERE name = 'Test User';

-- Test DELETE access
DELETE FROM employee_data WHERE name = 'Test User';

-- Reset to original role
RESET ROLE;

-- Alternatively, test by connecting from command line:
-- psql -U new_developer -d your_database -c "SELECT * FROM employee_data;
```

Testing note: Use SET ROLE to test within psql, or connect with a new psql session using 'psql -U new_developer'. The user should inherit all dev_team permissions automatically due to the INHERIT property.

Section 3: Tables, Data Types & Arrays

Question 3a: Create Database and Table

Create a database called 'sample_db' with a single table called 'sample_table'. The table should have: id (serial auto-increment), name (varchar 50), age (numeric 4,2), description (text).

Answer:

```
-- Step 1: Create the database
CREATE DATABASE sample_db;

-- Step 2: Connect to the new database
\c sample_db

-- Step 3: Create the table with specified columns
CREATE TABLE sample_table (
    id SERIAL PRIMARY KEY,          -- Auto-incrementing integer
    name VARCHAR(50),               -- Variable character, max 50 chars
    age NUMERIC(4, 2),              -- Precision 4, scale 2 (e.g., 99.99)
    description TEXT                 -- Unlimited text
);

-- Verify table structure
\d sample_table

-- Alternative: View table info
SELECT column_name, data_type, character_maximum_length,
       numeric_precision, numeric_scale
FROM information_schema.columns
WHERE table_name = 'sample_table';
```

Data type notes: SERIAL = auto-incrementing integer (shorthand for sequence). VARCHAR(50) = variable length, max 50. NUMERIC(4,2) = max 99.99 (4 digits total, 2 after decimal). TEXT = unlimited length text.

Question 3b: Load Data from CSV

Populate the 'sample_table' with data from CSV file 'sample_data.csv' containing: John Smith (25), Jane Doe (33), Bob Johnson (45) with descriptions.

Answer:

```
-- First, create the CSV file (sample_data.csv):
-- name,age,description
-- John Smith,25,"Lorem ipsum dolor sit amet, consectetur adipiscing elit."
-- Jane Doe,33,"Nullam imperdiet massa ac elementum laoreet."
-- Bob Johnson,45,"Pellentesque euismod quam non mi rutrum, non malesuada magna malesuada."

-- Method 1: Using COPY command (requires superuser or appropriate privileges)
COPY sample_table(name, age, description)
FROM '/path/to/sample_data.csv'
WITH (FORMAT csv, HEADER true);

-- Method 2: Using \copy in psql (runs as client, works without superuser)
\copy sample_table(name, age, description) FROM 'sample_data.csv' WITH CSV HEADER

-- Method 3: Manual INSERT if CSV not available
INSERT INTO sample_table (name, age, description) VALUES
('John Smith', 25, 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.'),
('Jane Doe', 33, 'Nullam imperdiet massa ac elementum laoreet.'),
('Bob Johnson', 45, 'Pellentesque euismod quam non mi rutrum, non malesuada magna malesuada.');
```

```
-- Verify data loaded
SELECT * FROM sample_table;

-- Check row count
SELECT COUNT(*) FROM sample_table; -- Should return 3
```

Important: Use \copy (with backslash) in psql - it runs client-side and doesn't require superuser. COPY (without backslash) runs server-side and requires file access. Note: id column auto-fills due to SERIAL.

Question 3c & 3d: String Functions

c) Show all data and use string functions to pad the 'name' column with spaces on the left so that all values are exactly 20 characters long. d) Show all data and use string functions to trim leading and trailing whitespace from the 'description' column.

Answer:

```
-- Question 3c: Left-pad name to 20 characters
SELECT
    id,
    LPAD(name, 20, ' ') AS padded_name, -- Left pad with spaces
    age,
    description
FROM sample_table;

-- Output will look like:
-- "          John Smith"  (10 spaces + 10 chars = 20 total)
-- "          Jane Doe"   (12 spaces + 8 chars = 20 total)
-- "          Bob Johnson" (9 spaces + 11 chars = 20 total)

-- Alternative: Right-pad instead (RPAD)
SELECT RPAD(name, 20, ' ') AS right_padded FROM sample_table;

-- Question 3d: Trim whitespace from description
SELECT
    id,
    name,
    age,
    TRIM(description) AS trimmed_description, -- Remove both leading & trailing
    LTRIM(description) AS left_trimmed,       -- Remove only leading
    RTRIM(description) AS right_trimmed       -- Remove only trailing
FROM sample_table;

-- Combined: Both padding and trimming
SELECT
    id,
    LPAD(name, 20, ' ') AS padded_name,
    age,
    TRIM(description) AS clean_description
FROM sample_table;

-- Other useful string functions:
-- LENGTH(name) - get string length
-- UPPER(name), LOWER(name) - change case
-- SUBSTRING(name, 1, 4) - extract substring
```

Function summary: LPAD(string, length, fill) pads on left. RPAD pads on right. TRIM removes spaces from both ends. LTRIM removes left spaces, RTRIM removes right spaces.

Question 3e & 3f: Add Date/Timestamp Columns

e) Add a new column called 'birthdate' of type 'date'. f) Add a new column called 'last_login' of type 'timestamp with time zone'.

Answer:

```
-- Question 3e: Add birthdate column (DATE type)
ALTER TABLE sample_table
ADD COLUMN birthdate DATE;

-- Question 3f: Add last_login column (TIMESTAMP WITH TIME ZONE)
ALTER TABLE sample_table
ADD COLUMN last_login TIMESTAMP WITH TIME ZONE;

-- Verify new columns
\d sample_table

-- Populate with sample data
UPDATE sample_table
SET birthdate = CURRENT_DATE - (age * 365)::INTEGER, -- Approximate birthdate
    last_login = CURRENT_TIMESTAMP
WHERE id IN (1, 2, 3);

-- More precise example: Set specific dates
UPDATE sample_table SET birthdate = '2000-01-15',
                        last_login = '2025-11-08 14:30:00+00'
WHERE name = 'John Smith';

-- View updated data
SELECT id, name, age, birthdate, last_login FROM sample_table;

-- Date/time functions you can use:
SELECT
    CURRENT_DATE,                -- Today's date
    CURRENT_TIMESTAMP,           -- Now with timezone
    NOW(),                       -- Same as CURRENT_TIMESTAMP
    AGE(birthdate),              -- Calculate age from birthdate
    EXTRACT(YEAR FROM birthdate) AS year -- Extract components
FROM sample_table;
```

Data type notes: DATE stores only the date (no time). TIMESTAMP WITH TIME ZONE (or TIMESTAMPTZ) stores date, time, and timezone info - best practice for timestamps. Always use WITH TIME ZONE for timestamps!

Question 3g: Create Sequence

Create a sequence called 'sample_sequence' that starts at 100 and increments by 10.

Answer:

```
-- Create sequence starting at 100, incrementing by 10
CREATE SEQUENCE sample_sequence
  START WITH 100
  INCREMENT BY 10;

-- Get next value from sequence
SELECT nextval('sample_sequence'); -- Returns 100 (first call)
SELECT nextval('sample_sequence'); -- Returns 110 (second call)
SELECT nextval('sample_sequence'); -- Returns 120 (third call)

-- Get current value without incrementing
SELECT currval('sample_sequence'); -- Returns last value retrieved (120)

-- Set sequence to a specific value
SELECT setval('sample_sequence', 500);

-- View sequence information
SELECT * FROM sample_sequence;

-- Use sequence in a table
CREATE TABLE orders (
  order_id INTEGER DEFAULT nextval('sample_sequence'),
  order_date DATE,
  customer_name VARCHAR(100)
);

-- Insert will auto-use sequence
INSERT INTO orders (order_date, customer_name)
VALUES (CURRENT_DATE, 'Alice');
-- order_id will be next sequence value

-- View all sequences in database
SELECT sequence_name, start_value, increment_by, last_value
FROM information_schema.sequences;
```

Sequence functions: nextval() gets next value and increments. currval() returns last value without incrementing (must call nextval first in session). setval() manually sets the sequence value. Sequences are great for custom ID generation.

Question 3h & 3i: Array Columns

h) Create an array column called 'interests' that stores a list of integers. i) Update interests: John Smith = {1,3,5}, Jane Doe = {2,4}, Bob Johnson = {1,2,3,4,5}.

Answer:

```
-- Question 3h: Add array column for integer interests
ALTER TABLE sample_table
ADD COLUMN interests INTEGER[];

-- Verify column added
\d sample_table

-- Question 3i: Update interests for each person
UPDATE sample_table
SET interests = '{1,3,5}'::INTEGER[]
WHERE name = 'John Smith';

UPDATE sample_table
SET interests = '{2,4}'::INTEGER[]
WHERE name = 'Jane Doe';

UPDATE sample_table
SET interests = '{1,2,3,4,5}'::INTEGER[]
WHERE name = 'Bob Johnson';

-- Shorter syntax (without cast):
UPDATE sample_table SET interests = ARRAY[1,3,5] WHERE name = 'John Smith';

-- View all data with interests
SELECT id, name, interests FROM sample_table;

-- Array operations examples:
SELECT name,
       interests,
       array_length(interests, 1) AS count, -- Show array
       interests[1] AS first_interest,      -- Count elements
       interests[2:4] AS slice              -- Access first element (1-indexed!)
FROM sample_table;                        -- Array slice
```

Array syntax: Use '{1,2,3}'::INTEGER[] or ARRAY[1,2,3]. Arrays are 1-indexed (first element is interests[1]). Use INTEGER[] for variable-length array of integers.

Question 3j: Query Arrays - Contains 1 AND 5

Write a query that selects all data whose 'interests' column contains the values 1 and 5 (in any order).

Answer:

```
-- Method 1: Using @> operator (contains)
SELECT * FROM sample_table
WHERE interests @> ARRAY[1, 5];
-- @> means "left array contains right array"
-- Returns: John Smith {1,3,5} and Bob Johnson {1,2,3,4,5}

-- Method 2: Using && operator with subquery
SELECT * FROM sample_table
WHERE interests && ARRAY[1]           -- Contains 1
   AND interests && ARRAY[5];         -- AND contains 5

-- Method 3: Using ANY operator
SELECT * FROM sample_table
WHERE 1 = ANY(interests)
   AND 5 = ANY(interests);

-- Explanation of array operators:
-- @> : left contains right (array contains elements)
-- <@ : left is contained by right
-- && : arrays overlap (have common elements)
-- =   : arrays are equal

-- Example: Test which rows contain ONLY 1 or 5
SELECT * FROM sample_table
WHERE interests <@ ARRAY[1, 5];
-- Returns: None, because all have other values too

-- Count results
SELECT COUNT(*) FROM sample_table
WHERE interests @> ARRAY[1, 5];
-- Returns: 2 (John Smith and Bob Johnson)
```

Best operator: Use @> (contains) for 'has all these elements' queries. It checks if left array contains ALL elements from right array, regardless of order. This is the most efficient and clearest solution.

Question 3k: Query Arrays - Has 2 but NOT 4

Write a query that selects all data whose 'interests' column contains the value 2 but not the value 4.

Answer:

```
-- Method 1: Using @> and NOT
SELECT * FROM sample_table
WHERE interests @> ARRAY[2]          -- Contains 2
      AND NOT (interests @> ARRAY[4]); -- Does NOT contain 4
-- Returns: Bob Johnson {1,2,3,4,5}... wait, that has 4!
-- Actually returns: Only Bob Johnson? No - he has 4 too.
-- Correct result: None of our sample data matches!
-- (Jane has 2 AND 4, Bob has 2 AND 4, John has neither)

-- Method 2: Using ANY operator
SELECT * FROM sample_table
WHERE 2 = ANY(interests)             -- Contains 2
      AND NOT (4 = ANY(interests));  -- Does NOT contain 4

-- Let's add test data that matches:
INSERT INTO sample_table (name, age, interests)
VALUES ('Test User', 30, ARRAY[1, 2, 3, 5]);

-- Now the query returns Test User

-- Method 3: Using array_position (more verbose)
SELECT * FROM sample_table
WHERE array_position(interests, 2) IS NOT NULL -- Has 2
      AND array_position(interests, 4) IS NULL; -- Doesn't have 4

-- Verify with sample data:
-- John Smith: {1,3,5} - No 2 -> excluded
-- Jane Doe: {2,4} - Has 2 AND 4 -> excluded
-- Bob Johnson: {1,2,3,4,5} - Has 2 AND 4 -> excluded
-- Test User: {1,2,3,5} - Has 2, No 4 -> INCLUDED!

SELECT name, interests FROM sample_table
WHERE interests @> ARRAY[2] AND NOT (interests @> ARRAY[4]);
```

Logic: Use @> ARRAY[2] to check for 2, then negate the check for 4 with NOT. Remember: None of the original 3 records match this criteria! Jane has both 2 and 4, Bob has both, John has neither.

Question 3I: Update Array - Append Element

Write a query that updates the 'interests' column of the row with id=1 to add the value 2 at the end of the array.

Answer:

```
-- Method 1: Using array_append function
UPDATE sample_table
SET interests = array_append(interests, 2)
WHERE id = 1;
-- John Smith was {1,3,5}, now becomes {1,3,5,2}

-- Method 2: Using || operator (array concatenation)
UPDATE sample_table
SET interests = interests || 2
WHERE id = 1;
-- Same result: {1,3,5,2}

-- Method 3: Concatenate with array
UPDATE sample_table
SET interests = interests || ARRAY[2]
WHERE id = 1;

-- Verify the update
SELECT id, name, interests FROM sample_table WHERE id = 1;
-- Should show: 1 | John Smith | {1,3,5,2}

-- Other useful array modification functions:
-- array_prepend(2, interests) - Add to beginning
-- array_remove(interests, 3) - Remove all occurrences of 3
-- array_cat(interests, ARRAY[6,7]) - Concatenate arrays

-- Example: Add multiple values at once
UPDATE sample_table
SET interests = interests || ARRAY[6, 7, 8]
WHERE id = 1;
-- Now: {1,3,5,2,6,7,8}

-- Remove a value
UPDATE sample_table
SET interests = array_remove(interests, 3)
WHERE id = 1;
-- Removes all 3's from array
```

Best method: Use `array_append(interests, value)` for clarity, or use `||` operator for concatenation. Both work for adding elements. The `||` operator is more flexible - can append single values or entire arrays.

Section 4: Backup & Restore

Question 4: pg_dump and pg_restore Commands

Write command examples for: a) Backup entire database to plain SQL file 'mydb_backup.sql'. b) Restore entire database from plain SQL file 'mydb_backup.sql'. c) Restore entire database from directory 'mydb_backup_dir'.

Answer:

```
# =====
# Question 4a: Backup to plain SQL file
# =====

# Basic backup to plain SQL
pg_dump mydb > mydb_backup.sql

# With username
pg_dump -U postgres mydb > mydb_backup.sql

# With host and port
pg_dump -h localhost -p 5432 -U postgres mydb > mydb_backup.sql

# More complete backup with ownership and privileges
pg_dump -U postgres --no-owner --no-acl mydb > mydb_backup.sql

# Best practice: Include CREATE DATABASE statement
pg_dump -U postgres -C mydb > mydb_backup.sql
# -C or --create: Include CREATE DATABASE command

# With compression (gzip)
pg_dump -U postgres mydb | gzip > mydb_backup.sql.gz

# =====
# Question 4b: Restore from plain SQL file
# =====

# Method 1: Using psql (for plain SQL files)
psql -U postgres -d mydb < mydb_backup.sql

# If database doesn't exist, create it first:
createdb -U postgres mydb
psql -U postgres -d mydb < mydb_backup.sql

# Or if backup includes CREATE DATABASE (-C flag):
psql -U postgres < mydb_backup.sql

# From compressed backup:
gunzip -c mydb_backup.sql.gz | psql -U postgres -d mydb

# With verbose output:
psql -U postgres -d mydb -f mydb_backup.sql -v ON_ERROR_STOP=1

# Method 2: Drop and recreate (clean restore):
dropdb -U postgres mydb
createdb -U postgres mydb
psql -U postgres -d mydb < mydb_backup.sql

# =====
# Question 4c: Restore from directory format backup
# =====

# First, create directory format backup (for reference):
pg_dump -U postgres -F d -f mydb_backup_dir mydb
# -F d : directory format
# -f : output directory path

# Restore from directory format using pg_restore:
pg_restore -U postgres -d mydb mydb_backup_dir

# Clean restore (drop and recreate):
dropdb -U postgres mydb
```

```

createdb -U postgres mydb
pg_restore -U postgres -d mydb mydb_backup_dir

# With connection parameters:
pg_restore -h localhost -p 5432 -U postgres -d mydb mydb_backup_dir

# Restore with parallelism (faster for large databases):
pg_restore -U postgres -d mydb -j 4 mydb_backup_dir
# -j 4 : Use 4 parallel jobs

# Restore only schema (no data):
pg_restore -U postgres -d mydb --schema-only mydb_backup_dir

# Restore only data (no schema):
pg_restore -U postgres -d mydb --data-only mydb_backup_dir

# Restore specific table only:
pg_restore -U postgres -d mydb -t sample_table mydb_backup_dir

# =====
# Summary of dump formats:
# =====

# Plain SQL (-F p or default): Human-readable, use psql to restore
pg_dump mydb > backup.sql
psql mydb < backup.sql

# Custom format (-F c): Compressed, use pg_restore
pg_dump -F c mydb > backup.dump
pg_restore -d mydb backup.dump

# Directory format (-F d): Parallel dump/restore, use pg_restore
pg_dump -F d -f backup_dir mydb
pg_restore -d mydb backup_dir

# Tar format (-F t): Archived, use pg_restore
pg_dump -F t mydb > backup.tar
pg_restore -d mydb backup.tar

```

Key points: Plain SQL uses psql for restore. Custom/directory/tar formats use pg_restore. Directory format (-F d) supports parallel operations with -j flag. Always specify -U username and -d database. Use -C flag in pg_dump to include CREATE DATABASE.

Study Tips

Key Concepts to Remember:

- Remember: `pg_stat_activity` for connections, `pg_terminate_backend()` to kill them
- Use `@>` operator for 'array contains' queries
- `SERIAL` = auto-increment, don't insert values for it
- Always use `TIMESTAMP WITH TIME ZONE` (not without!)
- `pg_dump` outputs SQL, `pg_restore` reads binary formats
- Use `\d tablename` in `psql` to see table structure
- `GRANT` gives permissions, `REVOKE` removes them
- Role inheritance is automatic with `INHERIT` (default)
- Array indexes start at 1, not 0!
- Test your queries on sample data before the exam

Good luck on your exam! Remember: It's open book, so use these materials effectively.