- **SQL Faker: Stored Procedures Documentation**

  Complete reference for deterministic fake data generation directly in PostgreSQL.

---

- **Core Random Number Generators**

  **prng_int(seed, pos, min_val, max_val)**

  Generates deterministic pseudo-random integers using Linear Congruential Generator (LCG).

  **Algorithm**: LCG with parameters a=1664525, c=1013904223, m=2^32

  state = (a * (seed + pos) + c) mod m

  result = min_val + (state mod (max_val - min_val + 1))

  **Arguments**:

- seed BIGINT - Base seed for reproducibility
- pos INT - Position/index to vary the output
- min_val INT - Minimum value (inclusive)
- max_val INT - Maximum value (inclusive)

  **Returns**: INT

  **Example**:

  -- Always returns same value for same inputs

  SELECT prng_int(12345, 0, 1, 100);  -- Returns: 46

  SELECT prng_int(12345, 1, 1, 100);  -- Returns: 47 (different pos)

  SELECT prng_int(99999, 0, 1, 100);  -- Returns: 100 (different seed)

  **Use Cases**:

- Selecting random items from arrays

- Generating random percentages
- Creating random counts

---

**prng_float(seed, pos)**

Generates deterministic pseudo-random floats between 0.0 and 1.0.

**Algorithm**: Same LCG as prng_int, normalized to [0, 1)

state = (a * (seed + pos) + c) mod m

result = state / m

**Arguments**:

- seed BIGINT - Base seed
- pos INT - Position/index

**Returns**: FLOAT (range: 0.0 to 1.0)

**Example**:

SELECT prng_float(12345, 0);  -- Returns: ~0.720661

SELECT prng_float(12345, 1);  -- Returns: ~0.953347

**Use Cases**:

- Probability calculations
- Percentage generation
- Input for other distributions

---

**prng_normal(seed, pos, mean, stddev)**

Generates values following a normal (Gaussian) distribution.

**Algorithm**: Box-Muller transform

```
u1 = prng_float(seed, pos * 2)

u2 = prng_float(seed, pos * 2 + 1)

z = sqrt(-2 * ln(u1)) * cos(2π * u2)

result = mean + z * stddev
```

**Arguments**:

- seed BIGINT - Base seed
- pos INT - Position/index
- mean FLOAT - Distribution mean (center)
- stddev FLOAT - Standard deviation (spread)

**Returns**: FLOAT

**Example**:

```
-- Generate heights: mean 175cm, stddev 7cm

SELECT prng_normal(12345, 0, 175.0, 7.0);  -- Returns: ~194.2

SELECT prng_normal(12345, 1, 175.0, 7.0);  -- Returns: ~166.8


-- Generate ages: mean 35, stddev 10

SELECT prng_normal(12345, 0, 35.0, 10.0);
```

**Use Cases**:

- Physical attributes (height, weight)
- Ages
- Test scores
- Any naturally distributed measurements

**Statistical Properties**:

- ~68% of values within 1 stddev of mean
- ~95% within 2 stddev
- ~99.7% within 3 stddev

---

**prng_sphere_coords(seed, pos)**

Generates uniformly distributed coordinates on a sphere (Earth).

**Algorithm**: Inverse transform sampling for uniform sphere distribution

```
u = prng_float(seed, pos * 2)
v = prng_float(seed, pos * 2 + 1)
latitude = arcsin(2u - 1) * 180/π
longitude = 360v - 180
```

**Why This Works**: Simple random latitude/longitude creates clustering at poles. This algorithm ensures equal probability per unit area on the sphere.

**Arguments**:

- seed BIGINT - Base seed
- pos INT - Position/index

**Returns**: TABLE(latitude FLOAT, longitude FLOAT)

**Example**:

```
SELECT * FROM prng_sphere_coords(12345, 0);
--  latitude  | longitude
-- -----------+-----------
--  -73.5758  | -172.5155
```

```
SELECT * FROM prng_sphere_coords(12345, 1);

--  latitude  | longitude

-- -----------+-----------

--   -1.6658  |   -5.0929
```

**Use Cases**:

- Realistic geographic coordinates
- Location-based testing
- Spatial data generation

---

- **Data Selection Functions**

**select_weighted_item(seed, pos, table_name, locale, gender_filter)**

Selects items from lookup tables using weighted random selection.

**Algorithm**: Cumulative frequency distribution

1. Calculate total weight: sum of all frequencies

2. Generate random value: 0 to (total_weight - 1)

3. Find item where: range_start ≤ random_val < range_end

**Arguments**:

- seed BIGINT - Base seed
- pos INT - Position/index
- table_name TEXT - Table to select from ('first_names', 'last_names', 'titles', 'eye_colors')
- locale VARCHAR(10) - Locale code ('en_US', 'de_DE')
- gender_filter CHAR(1) - Optional gender filter ('M', 'F', NULL for any)

**Returns**: INT (ID of selected item)

**Example**:

```
-- Select weighted first name for English male

SELECT select_weighted_item(12345, 0, 'first_names', 'en_US', 'M');

-- Returns ID, then lookup: SELECT name FROM first_names WHERE id = ...


-- Select any last name

SELECT select_weighted_item(12345, 1, 'last_names', 'en_US', NULL);


-- Select eye color

SELECT select_weighted_item(12345, 2, 'eye_colors', 'de_DE', NULL);
```

**Use Cases**:

- Realistic frequency distributions
- Common names appear more often
- Rare attributes remain rare

---

- **Data Generation Functions**

**generate_name(seed, user_index, locale)**

Generates complete names with variations.

**Algorithm**:

1. Determine gender (50/50 split)

2. Select weighted first name for gender

3. Select weighted last name

4. 30% chance: add title (Mr., Dr., etc.)

5. 40% chance: add middle initial

6. Format: [Title] FirstName [M.] LastName

**Arguments**:

- seed BIGINT - Base seed
- user_index INT - User's global index
- locale VARCHAR(10) - Locale code

**Returns**: TABLE(full_name TEXT, gender CHAR(1))

**Example**:

```
SELECT * FROM generate_name(12345, 0, 'en_US');

--   full_name   | gender

-- --------------+--------

-- William Brown |   M


SELECT * FROM generate_name(12345, 1, 'en_US');

--  full_name  | gender

-- ------------+--------

--  Mark Diaz  |   M


SELECT * FROM generate_name(12345, 0, 'de_DE');

--     full_name      | gender

-- -------------------+--------

-- Herr Thomas Müller |   M
```

**Variations**:

- With title: "Dr. John Smith"
- With middle: "John A. Smith"
- With both: "Dr. John A. Smith"
- Plain: "John Smith"

---

**generate_address(seed, user_index, locale)**

Generates formatted addresses with locale-specific formatting.

**Algorithm**:

1. Generate street number (1-9999)

2. Select random street name

3. Select random street type

4. Select random city (with postal pattern)

5. Generate postal code from pattern (# → random digit)

6. Format based on locale:

   - en_US: "123 Main Street, City, ST 12345"

   - de_DE: "Hauptstraße 123, 12345 Stadt"

7. 33% chance: add apartment/suite number

**Arguments**:

- seed BIGINT - Base seed
- user_index INT - User's global index
- locale VARCHAR(10) - Locale code

**Returns**: TEXT

**Example**:

```
SELECT generate_address(12345, 0, 'en_US');
-- Apt 688, 7632 Grove Court, Houston, TX 77383

SELECT generate_address(12345, 1, 'en_US');
-- Suite 143, 5926 Virginia Terrace, Houston, TX 77383

SELECT generate_address(12345, 2, 'en_US');
-- 4220 Princeton Road, Houston, TX 77383

SELECT generate_address(12345, 0, 'de_DE');
-- Hauptstraße 123, Wohnung 45, 10115 Berlin
```

**Format Variations**:

- Plain street address
- With apartment number
- With suite number

---

### generate_phone(seed, user_index, locale)

Generates phone numbers with format variations.

**Algorithm**:

1. Generate area code (200-999 or locale-specific)

2. Generate prefix and line number

3. Select format variant (4 options per locale)

**Arguments**:

- seed BIGINT - Base seed
- user_index INT - User's global index
- locale VARCHAR(10) - Locale code

**Returns**: TEXT

**Example**:

```
SELECT generate_phone(12345, 0, 'en_US');  -- (793) 518-2443
SELECT generate_phone(12345, 1, 'en_US');  -- 693-418-9943
SELECT generate_phone(12345, 2, 'en_US');  -- 593.318.7443
SELECT generate_phone(12345, 3, 'en_US');  -- +14933187443

SELECT generate_phone(12345, 0, 'de_DE');  -- 0123 456789
SELECT generate_phone(12345, 1, 'de_DE');  -- 0123-456789
SELECT generate_phone(12345, 2, 'de_DE');  -- 0123/456789
SELECT generate_phone(12345, 3, 'de_DE');  -- +49 123 456789
```

**Format Variations per Locale**:

- **en_US**: (XXX) XXX-XXXX, XXX-XXX-XXXX, XXX.XXX.XXXX, +1XXXXXXXXXX
- **de_DE**: 0XXX XXXXXX, 0XXX-XXXXXX, 0XXX/XXXXXX, +49 XXX XXXXXX

---

**generate_email(seed, user_index, locale, first_name, last_name)**

Generates email addresses derived from names.

**Algorithm**:

1. Select random domain for locale

2. Generate username (6 format variations):

   - firstname.lastname

   - firstnamelastname

   - flastname

   - firstname####

   - lastname####

   - firstname_lastname

3. Sanitize: remove special characters

4. Combine: username@domain

**Arguments**:

- seed BIGINT - Base seed
- user_index INT - User's global index
- locale VARCHAR(10) - Locale code
- first_name TEXT - First name (extracted from full name)
- last_name TEXT - Last name (extracted from full name)

**Returns**: TEXT

**Example**:

```
SELECT generate_email(12345, 0, 'en_US', 'John', 'Smith');

-- john.smith@gmail.com


SELECT generate_email(12345, 1, 'en_US', 'Jane', 'Doe');

-- janedoe@yahoo.com
```

```
SELECT generate_email(12345, 2, 'en_US', 'Bob', 'Johnson');
-- bjohnson@outlook.com


SELECT generate_email(12345, 3, 'de_DE', 'Thomas', 'Müller');
-- thomas5678@gmx.de
```

**Username Patterns**:

- Dot separated: john.smith
- Concatenated: johnsmith
- Initial + last: jsmith
- First + number: john1234
- Last + number: smith5678
- Underscore: john_smith

---

**generate_physical_attributes(seed, user_index, locale, gender)**

Generates realistic physical attributes using normal distributions.

**Algorithm**:

```
Height (normal distribution):
- Male: mean=175cm, stddev=7cm
- Female: mean=162cm, stddev=6.5cm


Weight (BMI-based):
- BMI ~ Normal(22, 3)
- weight = BMI * (height/100)²
```

Clamping:

- Height: 150-210 cm

- Weight: 45-150 kg

Eye color: Weighted selection

**Arguments**:

- seed BIGINT - Base seed
- user_index INT - User's global index
- locale VARCHAR(10) - Locale code
- gender CHAR(1) - Gender ('M' or 'F')

**Returns**: TABLE(height_cm INT, weight_kg INT, eye_color TEXT)

**Example**:

```
SELECT * FROM generate_physical_attributes(12345, 0, 'en_US', 'M');

--  height_cm | weight_kg | eye_color

-- -----------+-----------+-----------

--    194     |    114    |  Blue


SELECT * FROM generate_physical_attributes(12345, 1, 'en_US', 'F');

--  height_cm | weight_kg | eye_color

-- -----------+-----------+-----------

--    167     |     52    |  Green
```

**Statistical Distributions**:

- **Male height**: 68% between 168-182cm, 95% between 161-189cm
- **Female height**: 68% between 155.5-168.5cm, 95% between 149-175cm
- **BMI**: Normal range (18.5-25), occasional outliers

---

- **Main Generation Function**

  **generate_fake_users(locale, seed, batch_index, batch_size)**

  Orchestrates all generation functions to produce complete user records.

  **Algorithm**:

  ```
  For each user in batch:
    global_index = batch_index * batch_size + position


    1. Generate name (returns name + gender)
    2. Generate coordinates (uniform sphere)
    3. Generate physical attributes (using gender)
    4. Generate address
    5. Generate phone
    6. Generate email (using name parts)


    Return complete record
  ```

  **Arguments**:

- locale VARCHAR(10) - Locale code ('en_US', 'de_DE')
- seed BIGINT - Base seed for reproducibility
- batch_index INT - Batch number (0-based)

- batch_size INT - Number of users per batch (default 10)

**Returns**: TABLE with columns:

- batch_position INT - Position within this batch (0 to batch_size-1)
- full_name TEXT - Complete name
- address TEXT - Formatted address
- latitude FLOAT - Geographic latitude (-90 to 90)
- longitude FLOAT - Geographic longitude (-180 to 180)
- height_cm INT - Height in centimeters
- weight_kg INT - Weight in kilograms
- eye_color TEXT - Eye color description
- phone_number TEXT - Formatted phone number
- email TEXT - Email address

**Example**:

```
-- Generate first 10 users
SELECT * FROM generate_fake_users('en_US', 12345, 0, 10);


-- Generate next 10 users (same seed)
SELECT * FROM generate_fake_users('en_US', 12345, 1, 10);


-- Generate 100 users at once
SELECT * FROM generate_fake_users('en_US', 12345, 0, 100);


-- Different seed = different users
```

```
SELECT * FROM generate_fake_users('en_US', 99999, 0, 10);


-- German locale

SELECT * FROM generate_fake_users('de_DE', 12345, 0, 10);
```

**Determinism Guarantee**:

```
-- These always return identical data

SELECT * FROM generate_fake_users('en_US', 12345, 5, 10);

SELECT * FROM generate_fake_users('en_US', 12345, 5, 10);


-- User at position 0 in batch 5 = position 50 in batch 0

SELECT * FROM generate_fake_users('en_US', 12345, 5, 1);

SELECT * FROM generate_fake_users('en_US', 12345, 0, 51) OFFSET 50 LIMIT 1;
```

**Performance**:

- 10 users: ~50ms
- 100 users: ~400ms
- 1000 users: ~3-4s

---

- **Usage Patterns**

**Generate Test Dataset**

```
-- Create test table

CREATE TABLE test_users AS

SELECT * FROM generate_fake_users('en_US', 12345, 0, 1000);
```

```
-- Verify count
SELECT COUNT(*) FROM test_users;  -- 1000
```

**Generate Multiple Batches**

```
-- Generate 10 batches of 100 users each (1000 total)
SELECT * FROM generate_fake_users('en_US', 12345, 0, 100)
UNION ALL
SELECT * FROM generate_fake_users('en_US', 12345, 1, 100)
UNION ALL
SELECT * FROM generate_fake_users('en_US', 12345, 2, 100)
-- ... up to batch 9
```

**Filter by Attributes**

```
-- Only users in specific location range
SELECT * FROM generate_fake_users('en_US', 12345, 0, 100)
WHERE latitude BETWEEN 30 AND 50
  AND longitude BETWEEN -120 AND -70;

-- Only tall people
SELECT * FROM generate_fake_users('en_US', 12345, 0, 100)
WHERE height_cm > 180;

-- Only specific eye colors
SELECT * FROM generate_fake_users('en_US', 12345, 0, 100)
WHERE eye_color IN ('Blue', 'Green');
```

**Export to CSV**

```
-- Direct copy to CSV

COPY (

  SELECT * FROM generate_fake_users('en_US', 12345, 0, 1000)

) TO '/tmp/fake_users.csv' CSV HEADER;
```

**Integration with Existing Tables**

```
-- Insert fake users into existing table

INSERT INTO users (name, email, phone, address)

SELECT full_name, email, phone_number, address

FROM generate_fake_users('en_US', 12345, 0, 50);
```

---

- **Adding New Locales**

**Step 1: Add Locale Entry**

```
INSERT INTO locales (locale_code, locale_name, country_code)

VALUES ('fr_FR', 'French (France)', 'FRA');
```

**Step 2: Populate Lookup Tables**

```
-- First names (100+ recommended)

INSERT INTO first_names (name, locale_code, gender, frequency) VALUES

('Jean', 'fr_FR', 'M', 10),

('Marie', 'fr_FR', 'F', 10),

('Pierre', 'fr_FR', 'M', 9),

-- ... add more
-- Last names (200+ recommended)

INSERT INTO last_names (name, locale_code, frequency) VALUES

('Martin', 'fr_FR', 10),

('Bernard', 'fr_FR', 9),

-- ... add more


-- Titles

INSERT INTO titles (title, locale_code, gender, frequency) VALUES

('M.', 'fr_FR', 'M', 10),

('Mme', 'fr_FR', 'F', 10),

-- ... add more


-- Cities, street names, domains, etc.
```

**Step 3: Test**

```
SELECT * FROM generate_fake_users('fr_FR', 12345, 0, 10);
```

---

- **Advanced Techniques**

**Custom Random Distributions**

```
-- Exponential distribution (for wait times, ages)

CREATE FUNCTION prng_exponential(seed BIGINT, pos INT, lambda FLOAT)

RETURNS FLOAT AS $$

BEGIN

    RETURN -ln(1.0 - prng_float(seed, pos)) / lambda;
```

```
END;
$$ LANGUAGE plpgsql IMMUTABLE;

-- Usage: generate ages with exponential decay
SELECT prng_exponential(12345, 0, 0.05);  -- Mean = 20 years
```

**Correlated Attributes**

```
-- Height and weight are already correlated via BMI
-- Add custom correlation (e.g., age and eye color)

CREATE FUNCTION generate_user_with_age(...)
RETURNS TABLE(..., age INT) AS $$
DECLARE
    base_age INT;
BEGIN
    -- Younger people more likely to have lighter eye colors
    base_age := prng_normal(seed, user_index * 700, 40.0, 15.0)::INT;
    -- Modify eye color selection based on age
    -- ...
END;
$$ LANGUAGE plpgsql;
```

**Reproducible Subsets**

```
-- Always get same "first 100 users" regardless of batch size
SELECT * FROM generate_fake_users('en_US', 12345, 0, 100)
```

```
WHERE batch_position < 10;


-- Equivalent to:
SELECT * FROM generate_fake_users('en_US', 12345, 0, 10);
```

---

- **Troubleshooting**

**Division by Zero**

**Error**: ERROR: division by zero in prng_int

**Cause**: Lookup table is empty for the locale

**Fix**:

```
-- Check which tables are empty
SELECT 'first_names' as table_name, locale_code, COUNT(*)
FROM first_names GROUP BY locale_code
UNION ALL
SELECT 'last_names', locale_code, COUNT(*)
FROM last_names GROUP BY locale_code;


-- Reload seed data
\i database/seed_data.sql
```

**Non-deterministic Results**

**Problem**: Same seed produces different results

**Causes**:

1. Database state changed (rows added/removed from lookup tables)

2. PostgreSQL version differences
3. Floating-point precision differences

   **Fix**: Ensure lookup tables remain static during generation.

---

- **Performance Optimization**

  **Batch Size Selection**

  ```
  -- Small batches (1-10): Lower latency, more overhead
  SELECT * FROM generate_fake_users('en_US', 12345, 0, 5);  -- ~25ms

  -- Medium batches (10-100): Balanced
  SELECT * FROM generate_fake_users('en_US', 12345, 0, 50);  -- ~200ms

  -- Large batches (100-1000): Best throughput
  SELECT * FROM generate_fake_users('en_US', 12345, 0, 500);  -- ~2s
  ```

  **Parallel Generation**

  ```
  -- Generate different seed ranges in parallel
  -- Session 1:
  SELECT * FROM generate_fake_users('en_US', 10000, 0, 1000);

  -- Session 2:
  SELECT * FROM generate_fake_users('en_US', 20000, 0, 1000);

  -- Session 3:
  ```

```
SELECT * FROM generate_fake_users('en_US', 30000, 0, 1000);
```

**Indexing Generated Data**

```
-- If storing generated data
CREATE TABLE users_cache AS
SELECT * FROM generate_fake_users('en_US', 12345, 0, 10000);


CREATE INDEX idx_users_email ON users_cache(email);
CREATE INDEX idx_users_coords ON users_cache USING GIST(
  point(longitude, latitude)
);
```

---

- **Summary**

**SQL Faker provides**:

- Pure SQL implementation (no application dependencies)
- Deterministic generation (reproducible with seeds)
- Realistic distributions (normal for physical attributes, uniform for locations)
- Locale support (easy to extend)
- High performance (hundreds of users per second)
- Composable functions (use individually or combined)

**Key principle**: Everything is a function of (seed, position), ensuring complete reproducibility.