DATABASES

Lecture 7. Functions and Operators. Constraints.

Range Operators

Operator	Description	Example	Result
=	equal	int4range(1,5) = '[1,4]'::int4range	t
<>	not equal	numrange(1.1,2.2) <> numrange(1.1,2.3)	t
<	less than	int4range(1,10) < int4range(2,3)	t
>	greater than	int4range(1,10) > int4range(1,5)	t
<=	less than or equal	numrange(1.1,2.2) <= numrange(1.1,2.2)	t
>=	greater than or equal	numrange(1.1,2.2) >= numrange(1.1,2.0)	t
@>	contains range	int4range(2,4) @> int4range(2,3)	t
@>	contains element	'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp	t
<@	range is contained by	int4range(2,4) <@ int4range(1,7)	t
<@	element is contained by	42 <@ int4range(1,7)	f
&&	overlap (have points in common)	int8range(3,7) && int8range(4,12)	t
<<	strictly left of	int8range(1,10) << int8range(100,110)	t
>>	strictly right of	int8range(50,60) >> int8range(20,30)	t
&<	does not extend to the right of	int8range(1,20) &< int8range(18,20)	t
&>	does not extend to the left of	int8range(7,20) &> int8range(5,10)	t
-|-	is adjacent to	numrange(1.1,2.2) -|- numrange(2.2,3.3)	t
+	union	numrange(5,15) + numrange(10,20)	[5,20)
*	intersection	int8range(5,15) * int8range(10,20)	[10,15)
-	difference	int8range(5,15) - int8range(10,20)	[5,10)

Range Functions

Function	Return Type	Description	Example	Result
`lower(anyrange)`	range's element type	lower bound of range	`lower(numrange(1.1,2.2))`	`1.1`
`upper(anyrange)`	range's element type	upper bound of range	`upper(numrange(1.1,2.2))`	`2.2`
`isempty(anyrange)`	`boolean`	is the range empty?	`isempty(numrange(1.1,2.2))`	`false`
`lower_inc(anyrange)`	`boolean`	is the lower bound inclusive?	`lower_inc(numrange(1.1,2.2))`	`true`
`upper_inc(anyrange)`	`boolean`	is the upper bound inclusive?	`upper_inc(numrange(1.1,2.2))`	`false`
`lower_inf(anyrange)`	`boolean`	is the lower bound infinite?	`lower_inf('(,)'::daterange)`	`true`
`upper_inf(anyrange)`	`boolean`	is the upper bound infinite?	`upper_inf('(,)'::daterange)`	`true`
`range_merge(anyrange, anyrange)`	`anyrange`	the smallest range which includes both of the given ranges	`range_merge('[1,2)'::int4range, '[3,4)'::int4range)`	`[1,4)`

Aggregate Functions

Function	Argument Type(s)	Return Type	Partial Mode	Description
array_agg(*expression*)	any non-array type	array of the argument type	No	input values, including nulls, concatenated into an array
array_agg(*expression*)	any array type	same as argument data type	No	input arrays concatenated into array of one higher dimension (inputs must all have same dimensionality, and cannot be empty or NULL)
avg(*expression*)	smallint, int, bigint, real, double precision, numeric, or interval	numeric for any integer-type argument, double precision for a floating-point argument, otherwise the same as the argument data type	Yes	the average (arithmetic mean) of all input values
bit_and(*expression*)	smallint, int, bigint, or bit	same as argument data type	Yes	the bitwise AND of all non-null input values, or null if none
bit_or(*expression*)	smallint, int, bigint, or bit	same as argument data type	Yes	the bitwise OR of all non-null input values, or null if none
bool_and(*expression*)	bool	bool	Yes	true if all input values are true, otherwise false
bool_or(*expression*)	bool	bool	Yes	true if at least one input value is true, otherwise false
count(*)		bigint	Yes	number of input rows
count(*expression*)	any	bigint	Yes	number of input rows for which the value of *expression* is not null
every(*expression*)	bool	bool	Yes	equivalent to bool_and
json_agg(*expression*)	any	json	No	aggregates values as a JSON array
jsonb_agg(*expression*)	any	jsonb	No	aggregates values as a JSON array
json_object_agg(*name*, *value*)	(any, any)	json	No	aggregates name/value pairs as a JSON object
jsonb_object_agg(*name*, *value*)	(any, any)	jsonb	No	aggregates name/value pairs as a JSON object

Aggregate Functions

`max(`*`expression`*`)`	any numeric, string, date/time, network, or enum type, or arrays of these types	same as argument type	Yes	maximum value of *`expression`* across all input values
`min(`*`expression`*`)`	any numeric, string, date/time, network, or enum type, or arrays of these types	same as argument type	Yes	minimum value of *`expression`* across all input values
`string_agg(`*`expression`*`, `*`delimiter`*`)`	(`text`, `text`) or (`bytea`, `bytea`)	same as argument types	No	input values concatenated into a string, separated by delimiter
`sum(`*`expression`*`)`	`smallint`, `int`, `bigint`, `real`, `double precision`, `numeric`, `interval`, or `money`	`bigint` for `smallint` or `int` arguments, `numeric` for `bigint` arguments, otherwise the same as the argument data type	Yes	sum of *`expression`* across all input values
`xmlagg(`*`expression`*`)`	`xml`	`xml`	No	concatenation of XML values (see also Section 9.14.1.7)

SELECT string_agg(name, ', ') FROM authors;
Result: 'Tolstoy, Dostoevsky, Chekhov'

Date/time Functions

NOW() Get the current date and time	**CURRENT_TIME** Get the current time
CURRENT_DATE Get the current date	**EXTRACT()** Pull out date parts
AGE() Calculate time differences	**INTERVAL** Work with durations
DATE_TRUNC() Truncate to specific time units	**TO_DATE()** Convert strings to date
TO_CHAR() Format timestamps	**CLOCK TIMESSTAMP** Real-time timestamps
TIMEOFDAY() Human-readable timestamp	**TEMPOFDAY()** Human-readable timestamp

CURRENT DATE & TIME

Examples

```
SELECT CURRENT_DATE; -- Result: 2025-10-14
SELECT CURRENT_TIME; -- Result: 02:10:08.182677+06:00
SELECT CURRENT_TIMESTAMP;
-- Result: 2025-10-14 02:11:16.208125+06
```

- The CURRENT_DATE function will return the current date as a 'YYYY-MM-DD' format.
- CURRENT_TIME function will return the current time of day as a 'HH:MM:SS.GMT+TZ' format.
- The CURRENT_TIMESTAMP function will return the current date as a 'YYYY-MM-DD HH:MM:SS.GMT+TZ' format.

EXTRACT

EXTRACT function extracts parts from a date

Unit	Explanation
day	Day of the month (1 to 31)
decade	Year divided by 10
doy	Day of the year (1=first day of year, 365/366=last day of the year, depending if it is a leap year)
epoch	Number of seconds since '1970-01-01 00:00:00 UTC', if date value. Number of seconds in an interval, if interval value
hour	Hour (0 to 23)
minute	Minute (0 to 59)
month	Number for the month (1 to 12), if date value. Number of months (0 to 11), if interval value
second	Seconds (and fractional seconds)
year	Year as 4-digits

EXTRACT

EXTRACT function extracts parts from a date

Syntax

```
EXTRACT ( field FROM source )
```

Example

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2025-10-14 21:30:45'); -- Result: 14

SELECT EXTRACT(MONTH FROM order_date) AS month,
    COUNT(*) AS total_orders
FROM orders
GROUP BY 1
ORDER BY 1;
```

Date_trunc()

DATE_TRUNC literally *cuts off* smaller time parts — for example, it rounds a timestamp to the start of a month, week, or day. It's a great way to group time data consistently.

Syntax

```
DATE_TRUNC ( 'field', source )
```

Examples

```
SELECT DATE_TRUNC('month', TIMESTAMP '2025-10-14 15:27:34'); -- Result: 2025-10-01 00:00:00
```

```
SELECT DATE_TRUNC('month', order_date) AS month,
COUNT(*) AS total_orders
FROM orders
GROUP BY 1
ORDER BY 1;
```

EXTRACT VS. DATE_TRUNC

```
SELECT EXTRACT(MONTH FROM
order_date) AS month,
    COUNT(*) AS total_orders
FROM orders
GROUP BY 1
ORDER BY 1;
```

order_date	Result of EXTRACT(MONTH)
2024-01-10	1
2025-01-20	1
2025-02-05	2

```
SELECT DATE_TRUNC('month',
order_date) AS month,
        COUNT(*) AS total_orders
FROM orders
GROUP BY 1
ORDER BY 1;
```

order_date	Result of DATE_TRUNC('month')
2024-01-10	2024-01-01 00:00:00
2025-01-20	2025-01-01 00:00:00
2025-02-05	2025-02-01 00:00:00

AGE, INTERVAL CALCULATIONS

AGE function returns the number of years, months, and days between two dates.

Syntax

```
AGE(timestamp1, timestamp2);
```

If timestamp2 is NOT provided, current date will be used

Examples

```
SELECT AGE('2025-10-14', '2000-02-01'); --25 years 8 mons 13 days
```

```
SELECT customer_name,
    AGE(birth_date) AS age
FROM customers
ORDER BY age DESC;
```

ADDING AND SUBTRACTING TIME INTERVALS

Intervals can be expressed in many ways — days, hours, months, even combinations like '1 year 3 months 2 days'. PostgreSQL automatically adjusts the date.

Examples

```
SELECT CURRENT_DATE + INTERVAL '7 days' AS
next_week;
SELECT order_date + INTERVAL '30 days' AS
delivery_date;
SELECT CURRENT_DATE - INTERVAL '14 days' AS
two_weeks_ago;
```

EXISTS

- The argument of EXISTS is an arbitrary SELECT statement, or subquery
- The subquery is evaluated to determine whether it returns any rows
- If it returns at least one row, the result of EXISTS is “true”
- If the subquery returns no rows, the result of EXISTS is “false”

```
EXISTS (subquery)
```

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

IN

- The right-hand side is a parenthesized subquery, which must return exactly one column
- The left-hand expression is evaluated and compared to each row of the subquery result
- The result of IN is “true” if any equal subquery row is found
- The result is “false” if no equal row is found

```
expression IN (subquery)
```

```
SELECT col1
FROM tab1
WHERE col1 IN (SELECT col2 FROM tab2);
```

NOT IN

- The right-hand side is a parenthesized subquery, which must return exactly one column
- The left-hand expression is evaluated and compared to each row of the subquery result
- The result of NOT IN is “true” if only unequal subquery rows are found
- The result is “false” if any equal row is found

```
expression NOT IN (subquery)
```

```
SELECT col1
FROM tab1
WHERE col1 NOT IN (SELECT col2 FROM tab2);
```

ANY/SOME

- The right-hand side is a parenthesized subquery, which must return exactly one column.
- The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result.
- The result of ANY/SOME is “true” if any true result is obtained.
- The result is “false” if no true result is found

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

ALL

- The right-hand side is a parenthesized subquery, which must return exactly one column
- The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result
- The result of ALL is “true” if all rows yield true
- The result is “false” if any false result is found

```
expression operator ALL (subquery)
```

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

CONSTRAINTS

- are used to define rules for columns in a database table. They ensure that no invalid data is entered into the database.

- gives as much control over the data in tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised.

CONSTRAINTS

- **NOT NULL:** a constraint that indicates that this column cannot be empty (NULL), and must have a value
- **PRIMARY KEY:** used to uniquely identify a row in the table. It is good practice to specify in each table the column that contains the primary key.
- **FOREIGN KEY**: Used to ensure referential integrity of the data.
- **UNIQUE:** Ensures that all values in a column are different.
- **CHECK:** Makes sure that all values in a column satisfy certain criteria.
- **DEFAULT:** you can set a default column value when no other value has been specified

CHECK

- A **CHECK** constraint is the most generic constraint type.
- It allows you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression.

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

CHECK

- Constraint definition comes after the data type, just like default value definitions.
- Default values and constraints can be listed in any order.

```
CREATE TABLE employees (
age INTEGER DEFAULT 25 CHECK (age >= 18));
```

```
CREATE TABLE employees (
age INTEGER CHECK (age >= 18) DEFAULT 25);
```

- A check constraint consists of the key word CHECK followed by an expression in parentheses.
- The check constraint expression should involve the column thus constrained, otherwise the constraint would not make too much sense.

```
age INTEGER CHECK (salary > 0)
```

CHECK

```
CREATE TABLE products_test (
    product_no integer,
    name text CHECK (char_length(name) > 3),
    price numeric CHECK (price > 0)
);
```

```
INSERT INTO products_test VALUES (1, 'abc', 100);
```

```
[23514] ERROR: new row for relation "products_test" violates check constraint "products_test_name_check"
Подробности: Failing row contains (1, abc, 100).
```

CHECK

```
CREATE TABLE products_test2 (
    product_no integer,
    name text CHECK (char_length(name) > 3) DEFAULT 'Hello',
    price numeric DEFAULT 1 CHECK (price > 0)
);
```

price numeric DEFAULT 0 CHECK (price > 0)

CHECK

- You can also give the constraint a separate name.
- This clarifies error messages and allows you to refer to the constraint when you need to change it.

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

ERROR: new row for relation "products" violates check constraint "positive_price"

CHECK

- If you don't specify a constraint name in this way, the system chooses a name for you.
- Named constraints are especially helpful when you need to modify or drop them later.

```
ALTER TABLE products DROP CONSTRAINT positive_price;
```

- A check constraint can also refer to several columns.
- Say you store a regular price and a discounted price, and you want to ensure that

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

CHECK

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

CHECK

- Names can be assigned to table constraints in the same way as column constraints:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

Not-Null

- A not-null constraint simply specifies that a column must not assume the null value.
- A not-null constraint is always written as a column constraint.
- A not-null constraint is functionally equivalent to creating a check constraint CHECK (column_name IS NOT NULL)
- But in PostgreSQL creating an explicit not-null constraint is more efficient.
- The drawback is that you cannot give explicit names to not-null constraints created this way.

Not-Null

- Column can have more than one constraint.
- Just write the constraints one after another:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

- The NOT NULL constraint has an inverse: the NULL constraint.
- This simply selects the default behavior that the column might be null.

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

Unique

- UNIQUE constraints ensure that the data contained in a column, or a group of columns, is unique among all the rows in the table.
- The syntax is:

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

Unique

- To define a UNIQUE constraint for a group of columns, write it as a table constraint with the column names separated by commas:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

•The UNIQUE constraint allows NULL values.
•However, each NULL is treated a distinct, meaning multiple NULL values are allowed in a column with a UNIQUE constraint.

•Unlike the PRIMARY KEY constraint, which also enforces uniqueness, a table can have multiple UNIQUE constraints.

String Patterns

```
CREATE TABLE passengers (
    passenger_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    email VARCHAR(255),
    CONSTRAINT check_email_format CHECK (
        email ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'
    )
);
```

^[A-Za-z0-9._%+-]+:containing allowed characters.
@: symbol separating local part and domain.
[A-Za-z0-9.-]+: Domain name with allowed characters.
[A-Za-z]{2,}$: Top-level domain with at least two letters.

```
INSERT INTO passengers (first_name, email) VALUES ('Alice', 'alice@gmail.com');  -- ✅
INSERT INTO passengers (first_name, email) VALUES ('Bob', 'not_an_email');      -- ❌
```

String Patterns

```
CREATE TABLE passengers_account(
    passenger_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    url varchar(255),
    password VARCHAR(255),
    CONSTRAINT check_password_complexity CHECK (
        LENGTH(password) >= 8 AND
        password ~ '[A-Z]' AND
        password ~ '[a-z]' AND
        password ~ '\d' AND
        password ~ '[!@#$%^&*]'
    )
);
```

Add constraint

```
ALTER TABLE customers
    ADD CONSTRAINT constr_name
                   CHECK(char_length(name) > 3)
```

```
ALTER TABLE customers
    DROP CONSTRAINT constr_name;
```