Functions and Operators

format()

- The function format produces output formatted according to a format string, in a style similar to the C function sprintf.

```
format(formatstr text [, formatarg "any" [, ...] ])
```

- formatstr is a format string that specifies how the result should be formatted.

format()

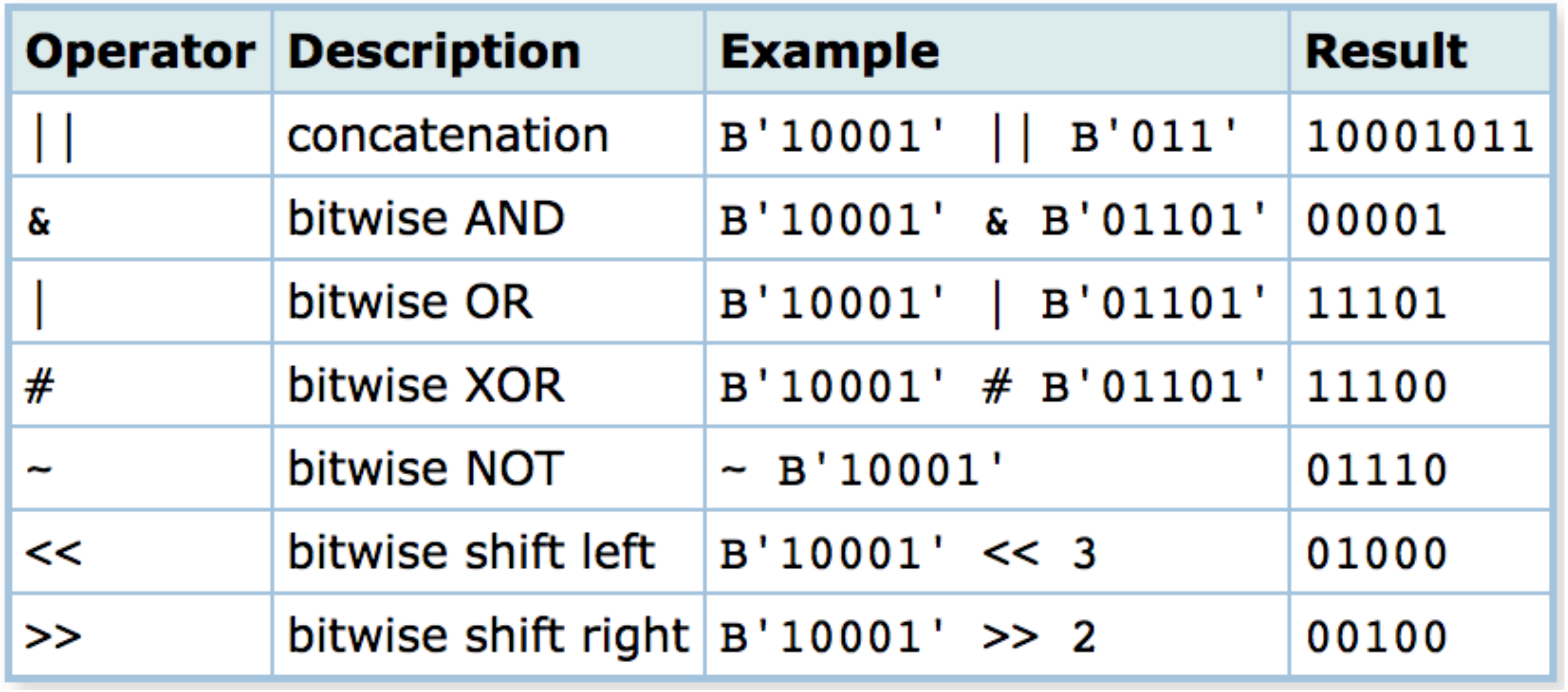
```
SELECT format('Hello %s', 'World');
Result: Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
Result: Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\'Reilly');
Result: INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', E'C:\\Program Files');
Result: INSERT INTO locations VALUES(E'C:\\Program Files')
```

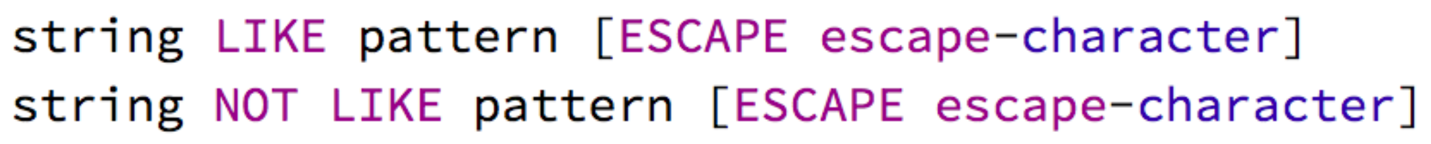
Bit String Operators



Operator	Description	Example	Result
||	concatenation	`B'10001' || B'011'`	10001011
&	bitwise AND	`B'10001' & B'01101'`	00001
|	bitwise OR	`B'10001' | B'01101'`	11101
#	bitwise XOR	`B'10001' # B'01101'`	11100
~	bitwise NOT	`~ B'10001'`	01110
<<	bitwise shift left	`B'10001' << 3`	01000
>>	bitwise shift right	`B'10001' >> 2`	00100

LIKE

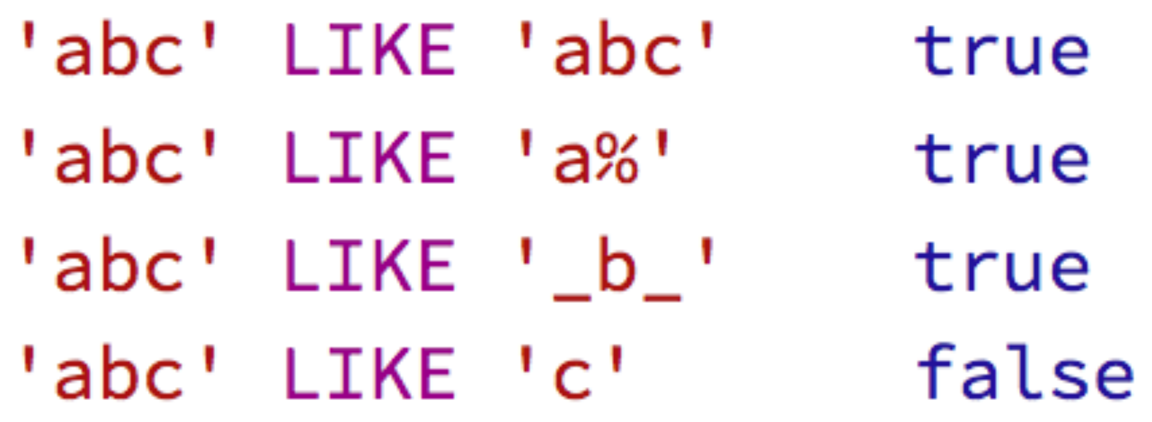
- The `LIKE` expression returns true if the *`string`* matches the supplied *`pattern`*. (As expected, the `NOT LIKE` expression returns false if `LIKE` returns true, and vice versa. An equivalent expression is `NOT (`*`string`*` LIKE `*`pattern`*`)`.)



```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

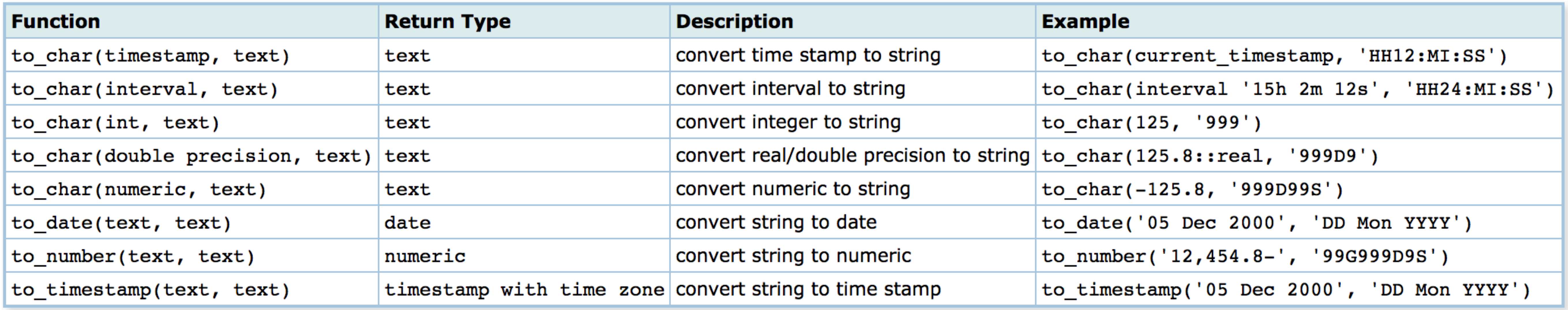
LIKE

- If `pattern` does not contain percent signs or underscores, then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore (_) in `pattern` stands for (matches) any single character; a percent sign (`%`) matches any sequence of zero or more characters.



```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

Data Type Formatting Functions



Function	Return Type	Description	Example
`to_char(timestamp, text)`	`text`	convert time stamp to string	`to_char(current_timestamp, 'HH12:MI:SS')`
`to_char(interval, text)`	`text`	convert interval to string	`to_char(interval '15h 2m 12s', 'HH24:MI:SS')`
`to_char(int, text)`	`text`	convert integer to string	`to_char(125, '999')`
`to_char(double precision, text)`	`text`	convert real/double precision to string	`to_char(125.8::real, '999D9')`
`to_char(numeric, text)`	`text`	convert numeric to string	`to_char(-125.8, '999D99S')`
`to_date(text, text)`	`date`	convert string to date	`to_date('05 Dec 2000', 'DD Mon YYYY')`
`to_number(text, text)`	`numeric`	convert string to numeric	`to_number('12,454.8-', '99G999D9S')`
`to_timestamp(text, text)`	`timestamp with time zone`	convert string to time stamp	`to_timestamp('05 Dec 2000', 'DD Mon YYYY')`

CASE

- The SQL CASE expression is a generic conditional expression, similar to if/else statements in other programming languages:

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

CASE

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

CASE

- There is a “simple” form of CASE expression that is a variant of the general form above:

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

CASE

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

COALESCE

- The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null.

```
COALESCE(value [, ...])
```

```
SELECT COALESCE(description, short_description, '(none)') ...
```

NULLIF

- The `NULLIF` function returns a null value if *`value1`* equals *`value2`*; otherwise it returns *`value1`*. This can be used to perform the inverse operation of the `COALESCE` example given above:

```
NULLIF(value1, value2)
```

```
SELECT NULLIF(value, '(none)') ...
```

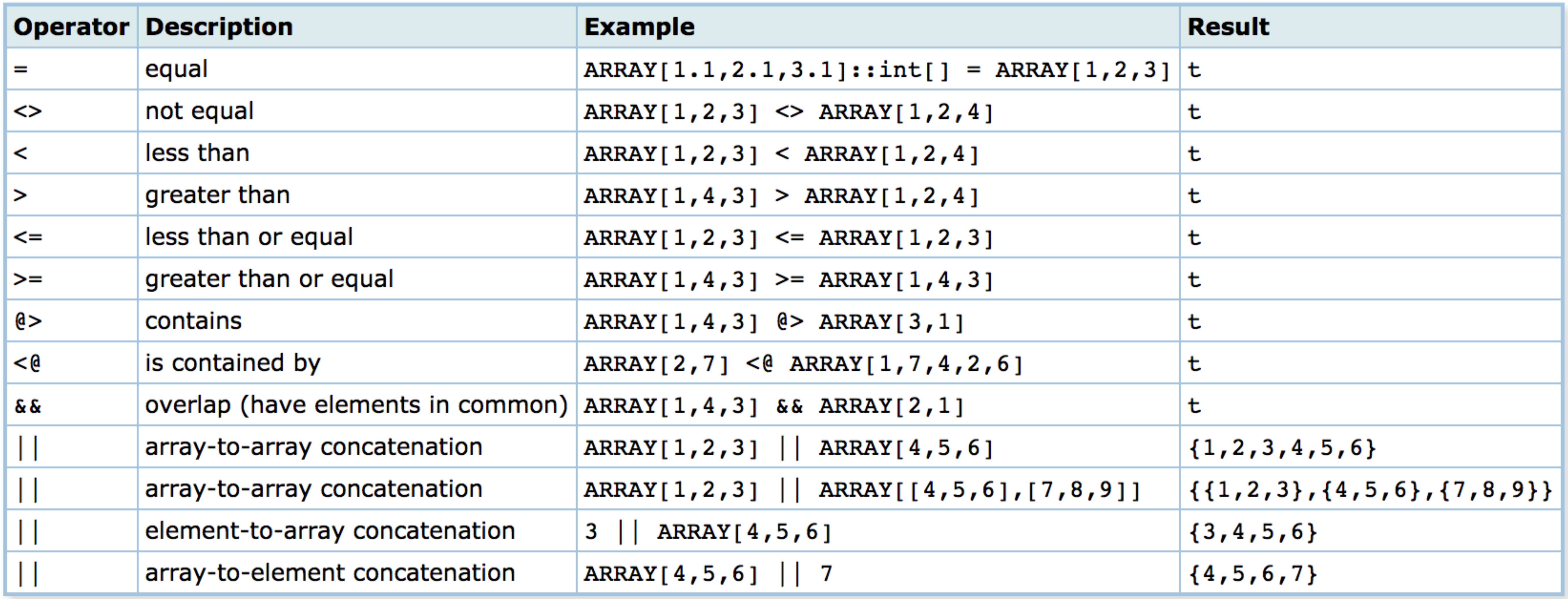
GREATEST & LEAST

- The `GREATEST` and `LEAST` functions select the largest or smallest value from a list of any number of expressions.

```
GREATEST(value [, ...])

LEAST(value [, ...])
```

Array Operators



Operator	Description	Example	Result
`=`	equal	`ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]`	`t`
`<>`	not equal	`ARRAY[1,2,3] <> ARRAY[1,2,4]`	`t`
`<`	less than	`ARRAY[1,2,3] < ARRAY[1,2,4]`	`t`
`>`	greater than	`ARRAY[1,4,3] > ARRAY[1,2,4]`	`t`
`<=`	less than or equal	`ARRAY[1,2,3] <= ARRAY[1,2,3]`	`t`
`>=`	greater than or equal	`ARRAY[1,4,3] >= ARRAY[1,4,3]`	`t`
`@>`	contains	`ARRAY[1,4,3] @> ARRAY[3,1]`	`t`
`<@`	is contained by	`ARRAY[2,7] <@ ARRAY[1,7,4,2,6]`	`t`
`&&`	overlap (have elements in common)	`ARRAY[1,4,3] && ARRAY[2,1]`	`t`
`||`	array-to-array concatenation	`ARRAY[1,2,3] || ARRAY[4,5,6]`	`{1,2,3,4,5,6}`
`||`	array-to-array concatenation	`ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]`	`{{1,2,3},{4,5,6},{7,8,9}}`
`||`	element-to-array concatenation	`3 || ARRAY[4,5,6]`	`{3,4,5,6}`
`||`	array-to-element concatenation	`ARRAY[4,5,6] || 7`	`{4,5,6,7}`

Array Functions

Function	Return Type	Description	Example	Result
`array_append(anyarray, anyelement)`	`anyarray`	append an element to the end of an array	`array_append(ARRAY[1,2], 3)`	`{1,2,3}`
`array_cat(anyarray, anyarray)`	`anyarray`	concatenate two arrays	`array_cat(ARRAY[1,2,3], ARRAY[4,5])`	`{1,2,3,4,5}`
`array_ndims(anyarray)`	`int`	returns the number of dimensions of the array	`array_ndims(ARRAY[[1,2,3], [4,5,6]])`	`2`
`array_dims(anyarray)`	`text`	returns a text representation of array's dimensions	`array_dims(ARRAY[[1,2,3], [4,5,6]])`	`[1:2][1:3]`
`array_fill(anyelement, int[], [, int[]])`	`anyarray`	returns an array initialized with supplied value and dimensions, optionally with lower bounds other than 1	`array_fill(7, ARRAY[3], ARRAY[2])`	`[2:4]={7,7,7}`
`array_length(anyarray, int)`	`int`	returns the length of the requested array dimension	`array_length(array[1,2,3], 1)`	`3`
`array_lower(anyarray, int)`	`int`	returns lower bound of the requested array dimension	`array_lower('[0:2]={1,2,3}'::int[], 1)`	`0`
`array_position(anyarray, anyelement [, int])`	`int`	returns the subscript of the first occurrence of the second argument in the array, starting at the element indicated by the third argument or at the first element (array must be one-dimensional)	`array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon')`	`2`
`array_positions(anyarray, anyelement)`	`int[]`	returns an array of subscripts of all occurrences of the second argument in the array given as first argument (array must be one-dimensional)	`array_positions(ARRAY['A','A','B','A'], 'A')`	`{1,2,4}`
`array_prepend(anyelement, anyarray)`	`anyarray`	append an element to the beginning of an array	`array_prepend(1, ARRAY[2,3])`	`{1,2,3}`
`array_remove(anyarray, anyelement)`	`anyarray`	remove all elements equal to the given value from the array (array must be one-dimensional)	`array_remove(ARRAY[1,2,3,2], 2)`	`{1,3}`
`array_replace(anyarray, anyelement, anyelement)`	`anyarray`	replace each array element equal to the given value with a new value	`array_replace(ARRAY[1,2,5,4], 5, 3)`	`{1,2,3,4}`

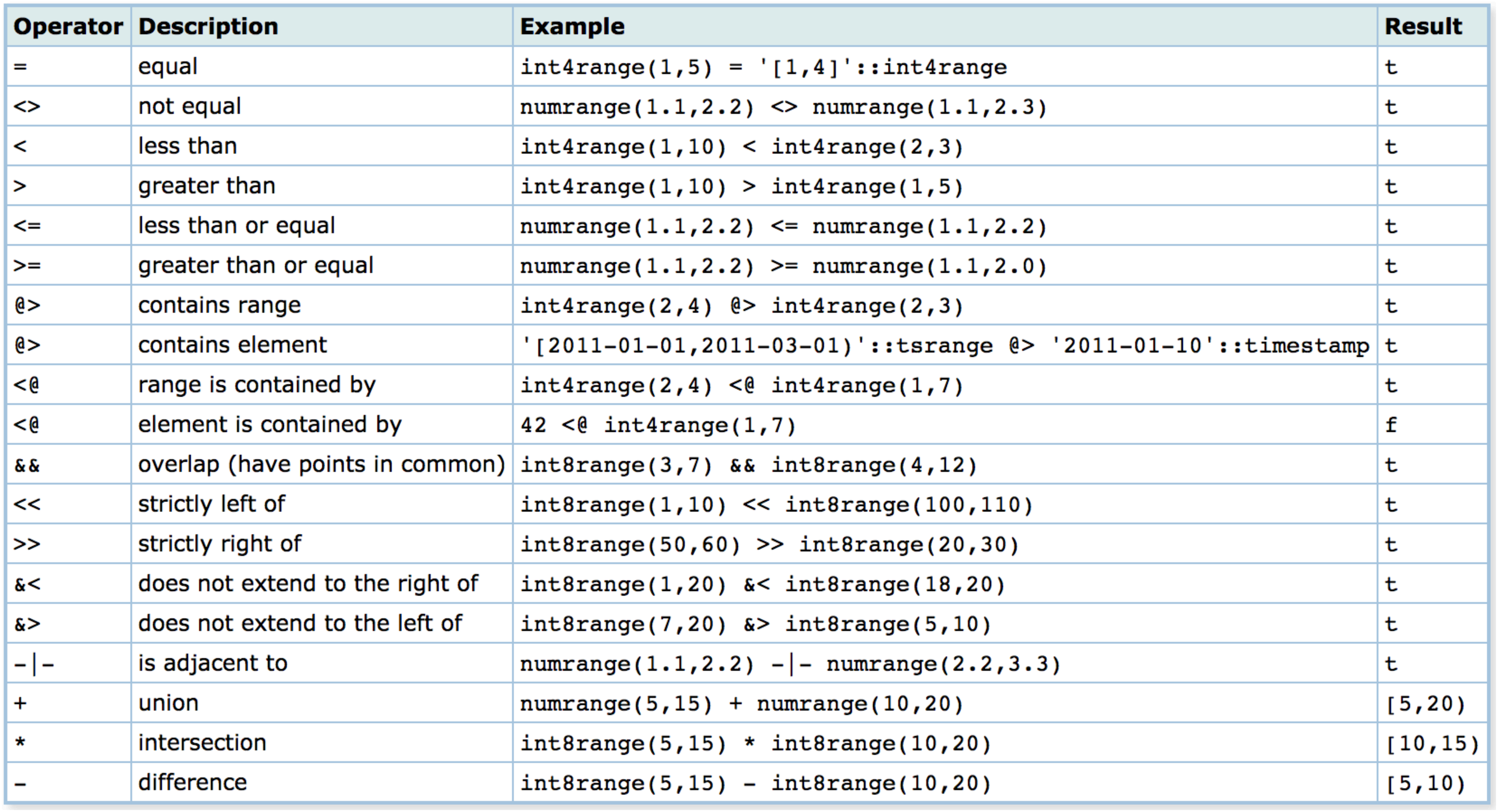
Array Functions

`array_to_string(anyarray, text [, text])`	`text`	concatenates array elements using supplied delimiter and optional null string	`array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')`	`1,2,3,*,5`
`array_upper(anyarray, int)`	`int`	returns upper bound of the requested array dimension	`array_upper(ARRAY[1,8,3,7], 1)`	`4`
`cardinality(anyarray)`	`int`	returns the total number of elements in the array, or 0 if the array is empty	`cardinality(ARRAY[[1,2],[3,4]])`	`4`
`string_to_array(text, text [, text])`	`text[]`	splits string into array elements using supplied delimiter and optional null string	`string_to_array('xx~^~yy~^~zz', '~^~', 'yy')`	`{xx,NULL,zz}`

`unnest(anyarray)`	`setof anyelement`	expand an array to a set of rows	`unnest(ARRAY[1,2])`	`1`
				`2`
				(2 rows)

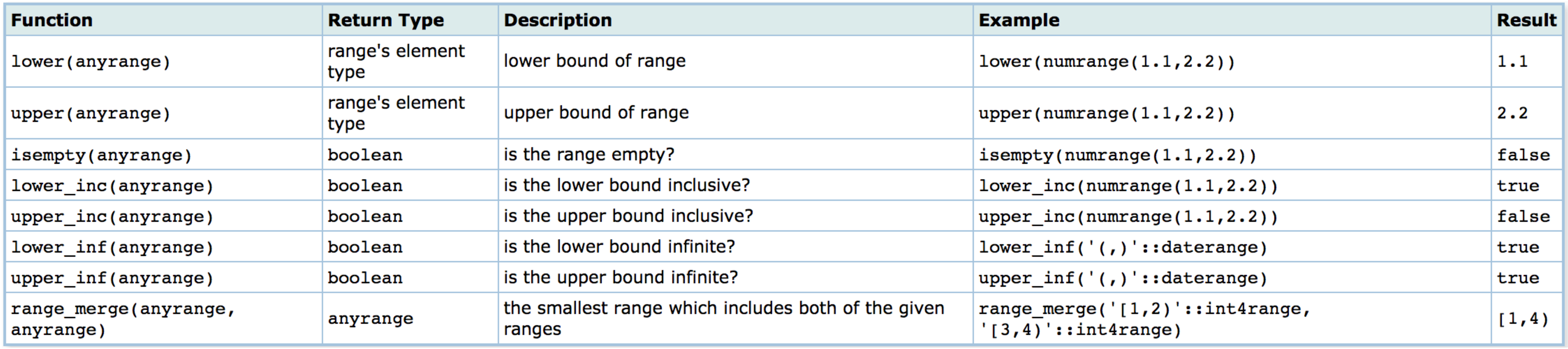
`unnest(anyarray, anyarray [, ...])`	`setof anyelement, anyelement [, ...]`	expand multiple arrays (possibly of different types) to a set of rows. This is only allowed in the FROM clause; see Section 7.2.1.4	`unnest(ARRAY[1,2],ARRAY['foo','bar','baz'])`	`1    foo`
				`2    bar`
				`NULL baz`
				(3 rows)

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

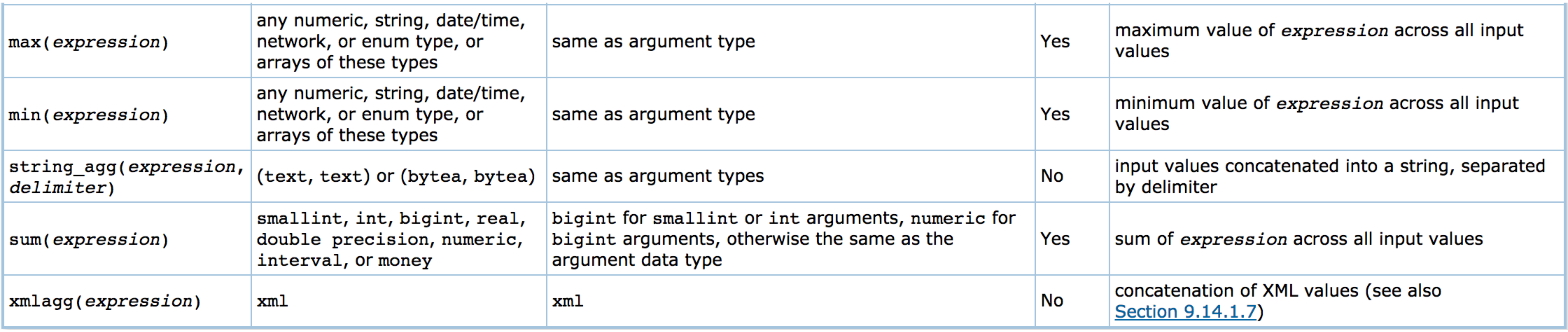


Function	Return Type	Description	Example	Result
`lower(anyrange)`	range's element type	lower bound of range	`lower(numrange(1.1,2.2))`	`1.1`
`upper(anyrange)`	range's element type	upper bound of range	`upper(numrange(1.1,2.2))`	`2.2`
`isempty(anyrange)`	`boolean`	is the range empty?	`isempty(numrange(1.1,2.2))`	`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)

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)
```

EXISTS

```sql
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)
```

IN

```sql
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)
```

NOT IN

```
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)
```

ANY/SOME

```sql
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)
```

ALL

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

Questions?