Инструкция по эксплуатации
ПО "Tengri Data Platform"

Москва
2025

Содержание

Вычислительные пулы . . . 1
- Создание вычислительного пула . . . 1
- Изменение атрибутов вычислительного пула . . . 2
- Сброс вычислительного пула . . . 3
- Установка атрибутов вычислительного пула по умолчанию . . . 3
- Вывод списка всех вычислительных пулов . . . 4
- Вывод текущего вычислительного пула . . . 4
- Установка текущего вычислительного пула . . . 4
- Увеличение размеров вычислительного пула . . . 4

Управление пользователями и правами . . . 6
- Ключевые концепты системы разграничения прав . . . 6
- Операции с объектами системы разграничения прав . . . 6
- Операции с пользователями . . . 7
  - Создание нового пользователя . . . 7
    - Модификатор `OR REPLACE` . . . 7
    - Модификатор `IF NOT EXISTS` . . . 7
    - Параметр `SET DEFAULT WORKER POOL` . . . 7
    - Параметр `IDENTIFIED BY TRUST` . . . 8
    - Параметр `IDENTIFIED BY PASSWORD` . . . 8
  - Изменение атрибутов пользователя . . . 8
  - Сброс пользователя . . . 9
  - Вывод информации о пользователе . . . 9
  - Вывод списка всех пользователей . . . 9
- Операции с ролями . . . 9
  - Создание новой роли . . . 9
  - Изменение атрибутов роли . . . 10
  - Сброс роли . . . 10
  - Вывод списка всех ролей . . . 10
  - Установка активной роли . . . 11
  - Предоставление ролям атрибутов . . . 11
- Операции с привилегиями . . . 11
  - Предоставление привилегий . . . 12
    - Привилегии на схемы . . . 12
    - Привилегии на таблицы и представления . . . 12
    - Привилегии на каталоги . . . 13
    - Привилегии на вычислительные пулы . . . 14
  - Отзыв привилегий . . . 14

Описание данных (DDL) . . . 16
- Операции с таблицами . . . 16
  - Создание новой таблицы . . . 16

Параметры . . . 16
Добавление данных в таблицу . . . 17
Параметры . . . 18
Пример вставки явно заданных значений . . . 18
Пример вставки результатов вложенного запроса . . . 19
Удаление таблицы . . . 19
Вывод списка всех таблиц . . . 19
Операции со схемами . . . 19
Создание новой схемы . . . 20
Изменение атрибутов схемы . . . 20
Удаление схемы . . . 20
Вывод списка всех схем . . . 20
Операции с представлениями . . . 21
Создание нового представления . . . 21
Параметры . . . 21
Удаление представления . . . 21
Синтаксис запросов к данным . . . 22
Выражения SQL . . . 22
Функции . . . 22
`SELECT` . . . 23
Синтаксис . . . 23
Выбор всех столбцов . . . 23
Выбор определенных столбцов . . . 24
Параметры . . . 25
Примеры . . . 26
Список столбцов `SELECT` . . . 27
Выражение со звездочкой `*` . . . 28
Выражение `DISTINCT` . . . 28
Выражение `DISTINCT ON` . . . 28
Агрегатные функции . . . 29
Оконные функции . . . 29
`WITH` . . . 29
Синтаксис . . . 30
Подзапрос . . . 30
Рекурсивное CTE: . . . 30
Параметры . . . 30
Примеры базовых CTE . . . 31
`FROM` . . . 31
Синтаксис . . . 32
Параметры . . . 32
Примеры . . . 33
Синтаксис `FROM`-first . . . 34

Синтаксис FROM-first с выражением SELECT 35
Синтаксис FROM-first без выражения SELECT 35
JOIN 36
Синтаксис 36
Параметры 37
OUTER JOIN — внешнее соединение 38
CROSS JOIN — перекрёстное соединение (декартово произведение) 38
Условное соединение 38
NATURAL JOIN — естественное соединение 40
SEMI JOIN и ANTI JOIN — полусоединение и антисоединение 41
Пример полусоединения 41
Пример антисоединения 42
Замкнутое соединение (Self-Join) 43
UNION 43
Синтаксис 43
Примеры 44
WHERE 45
Синтаксис 45
Параметры 45
Примеры 45
GROUP BY 47
GROUP BY ALL 47
Синтаксис 47
Параметры 48
Примеры 48
HAVING 48
Синтаксис 49
Параметры 49
Примеры 49
QUALIFY 49
Синтаксис 50
Параметры 50
Примеры 50
ORDER BY 52
ORDER BY ALL 52
Модификатор порядка значений NULL 52
Коллации (схемы сопоставления) 53
Синтаксис 53
Параметры 53
Примеры 54
LIMIT 56
Выражение LIMIT 56

Выражение `OFFSET` 57
Синтаксис 57
Параметры 57
Примеры 57
`LIKE` 58
Описание 58
Оператор `ILIKE` 59
Примеры 59
Полезные ссылки 60
`SIMILAR TO` 60
Описание 60
Примеры 61
Полезные ссылки 61
`AS` 62
Задание имен для столбцов в запросе 62
Задание имен для таблиц в запросе 62
Задание запроса при создании таблицы 62
Функции 63
Агрегатные функции 63
Числовые функции 63
Текстовые функции 64
Функции для регулярных выражений 65
Функции для даты и времени 65
Функции для JSON 65
Функции для двоичных данных 66
Оконные функции 66
Утилиты 66
Агрегатные функции 66
`array_agg()` 67
`avg()` 68
`count()` 69
`count(argument)` 69
`count_if()` 70
`max()` 70
`min()` 71
`median()` 71
`sum()` 72
Числовые функции 73
`abs()` 73
`add()` 73
`ceil()` 74
`cos()` 74

`divide()` 75
`even()` 75
`exp()` 75
`floor()` 76
`fmod()` 76
`gcd()` 77
`greatest()` 77
`isfinite()` 77
`isinf()` 78
`isnan()` 78
`lcm()` 79
`least()` 79
`lgamma()` 79
`ln()` 80
`log()` 80
`log2()` 81
`multiply()` 81
`nextafter()` 81
`pi()` 82
`pow()` 82
`radians()` 83
`random()` 83
`round_even()` 83
`round()` 84
`setseed()` 85
`sign()` 85
`signbit()` 85
`sin()` 86
`sqrt()` 86
`subtract()` 87
`trunc()` 87
Оператор `+` 88
Оператор `-` 88
Оператор `*` 89
Оператор `/` 89
Оператор `%` 90
Оператор `^` 90
Текстовые функции 91
`concat()` 91
`contains()` 91
`length()` 92
`strlen()` 92

`trim()` 93
`ltrim()` 93
`rtrim()` 94
`lower()` 94
`upper()` 95
`split()` 95
`chr()` 96
`md5()` 96
`sha1()` 97
`sha256()` 97
Оператор `||` 98
Функции для регулярных выражений 99
`regexp_extract()` 99
`regexp_extract_all()` 100
`regexp_full_match()` 101
`regexp_matches()` 102
`regexp_replace()` 102
`regexp_split_to_array()` 103
`regexp_split_to_table()` 103
Оператор `~` 104
Функции для даты и времени 105
`current_time()` 105
`datepart()` 105
`date_diff()` 106
Оператор `+` 107
Оператор `-` 108
Функции для JSON 109
`json_array_length()` 110
`json_contains()` 110
`json_exists()` 111
`json_extract()` 112
`json_extract_string()` 112
`json_group_array()` 113
`json_group_object()` 113
`json_keys()` 114
`json_transform()` 115
`json_transform_strict()` 116
`json_valid()` 117
`json_value()` 118
`json()` 118
`read_json()` 119
Функции для двоичных данных 121

- `concat()` 122
- `md5()` 122
- `sha1()` 123
- `sha256()` 123
- Оператор `||` 124

Оконные функции 125

- `cume_dist()` 125
- `dense_rank()` 126
- `first_value()` 127
- `lag()` 128
- `last_value()` 129
- `lead()` 130
- `nth_value()` 132
- `ntile()` 133
- `percent_rank()` 133
- `rank()` 134
- `row_number()` 135

Утилиты 136

- `coalesce()` 136
- `hash()` 136
- `unnest()` 137

Типы данных 142

Числовые типы 142

- Целочисленный тип 142
- Тип с заданной точностью 143
- Тип с переменной точностью 143
- Примеры 144
- Полезные ссылки 144

Текстовый тип 144

- Описание 144
- Пример 145
- Полезные ссылки 145

Типы для даты и времени 145

- Даты 145
- Время 146
- Моменты времени 146
- Примеры 146
- Полезные ссылки 150

Тип JSON 150

- Описание 150
- Примеры 150
- Полезные ссылки 152

Типы для массивов . . . 152
Описание . . . 152
Примеры . . . 152
Двоичный тип . . . 153
Описание . . . 153
Полезные ссылки . . . 153
Логический тип . . . 154
Описание . . . 154
Логические операторы . . . 154

Вычислительные пулы

В Tengri вычисление производится в виртуальных вычислительных пулах (вычислителях).

Вычислительный пул — это легковесный контейнер, содержащий программный код и обладающий выделенными вычислительными ресурсами для вычислительных задач:

- Выполнение SQL-запросов
- Выполнение кода на Python
- Запуск моделей AI

Операции с вычислительными пулами:

- `CREATE WORKER POOL` — Создание вычислительного пула
- `ALTER WORKER POOL` — Изменение атрибутов вычислительного пула
- `DROP WORKER POOL` — Сброс вычислительного пула
- `ALTER WORKER POOL SET DEFAULT` — Установка атрибутов вычислительного пула по умолчанию
- `SHOW WORKER POOLS` — Вывод списка всех вычислительных пулов
- `SHOW WORKER POOL` — Вывод текущего вычислительного пула
- `USE WORKER POOL` — Установка текущего вычислительного пула
- Предоставление привилегий на вычислительные пулы

Создание вычислительного пула

```
CREATE [OR REPLACE] WORKER POOL [IF NOT EXISTS] <pool_name>
    [<worker_pool_attribute> [ ... ]];
```

Создает новый вычислительный пул с указанным именем и атрибутами.

Если указан модификатор `OR REPLACE`, то конечное действие эквивалентно удалению существующего вычислительного пула и созданию нового с тем же именем.

Опциональный модификатор `IF NOT EXISTS` ограничивает запрос только теми случаями, в которых указанный объект еще не существует.

Модификаторы являются взаимоисключающими. Если указать их оба, это приведет к ошибке.

Можно указать следующие атрибуты вычислительного пула:

- `WORKER SIZE <size>` — устанавливает размер вычислительного пула: `XS`, `S`, `M`, `L`, `XL`.
- `MAX WORKER COUNT <count>` — устанавливает максимальное количество вычислительных пулов.
- `WORKER IDLE TTL <seconds> SECONDS` — устанавливает максимальное время простоя для вычислительных пулов.

- `SCALING STRATEGY <strategy>` — задает стратегию масштабирования. В настоящее время существует две стратегии: `STD` или `ECO`.

▼ *Посмотреть примеры*

Создаем вычислительный пул с именем `my_worker` и размером `S`:

```
CREATE WORKER POOL my_worker
    WORKER SIZE S;
```

Создаем вычислительный пул с именем `my_worker`, размером `L`, максимальным количеством вычислительных пулов `40`, максимальным временем простоя `300` секунд и со стратегией масштабирования `ECO`:

```
CREATE WORKER POOL my_worker
    WORKER SIZE L
    MAX WORKER COUNT 40
    WORKER IDLE TTL 300 SECONDS
    SCALING STRATEGY ECO;
```

Изменение атрибутов вычислительного пула

```
ALTER WORKER POOL [IF EXISTS] <pool_name> RENAME TO <new_pool_name>;

ALTER WORKER POOL [IF EXISTS] <pool_name> SET <worker_pool_attribute>;
```

Изменяет вычислительный пул с указанным именем.

Не рекомендуется изменять атрибуты вычислительных пулов, запущенных внутри вашей рабочей группы, так как это может привести к неожиданному падению производительности вычислений в случаях, когда вычислительные пулы используются несколькими пользователями.

Поэтому выражения `ALTER WORKER POOL` следует использовать только администраторам.

Если вам как пользователю требуется изменить размер используемого вами вычислительного пула (например, увеличить его), то следует выбрать для работы другой вычислительный пул большего размера.

Поддерживаемые действия:

- `RENAME TO` — переименовывает вычислительный пул в указанное имя `<new_pool_name>`. Все атрибуты и конфигурации вычислительного пула сохраняются.
- `SET` — обновляет атрибут вычислительного пула. Можно изменить следующие атрибуты:

- `WORKER SIZE <size>` — устанавливает размер вычислительного пула (`XS`, `S`, `M`, `L`, `XL`).
- `MAX WORKER COUNT <count>` — устанавливает максимальное количество вычислительных пулов.
- `WORKER IDLE TTL <seconds> SECONDS` — устанавливает время простоя для вычислительных пулов.
- `SCALING STRATEGY <strategy>` — задает стратегию масштабирования (`STD` или `ECO`).

Опциональный модификатор `IF EXISTS` ограничивает запрос только теми случаями, в которых указанный объект существует.

Сброс вычислительного пула

```
DROP WORKER POOL [IF EXISTS] <pool_name>;
```

Удаляет вычислительный пул с указанным именем.

Опциональный модификатор `IF EXISTS` ограничивает запрос только теми случаями, в которых указанный объект существует.

Установка атрибутов вычислительного пула по умолчанию

```
ALTER WORKER POOL SET DEFAULT <worker_pool_attributes>;
```

Устанавливает атрибуты по умолчанию для всех новых вычислительных пулов в системе.

Этот оператор позволяет задать общесистемные атрибуты по умолчанию, которые будут применяться к вычислительным пулам при их создании без явного указания этих атрибутов.

Можно указать следующие атрибуты вычислительных пулов:

- `WORKER SIZE <size>` — задает размер вычислительного пула по умолчанию (`XS`, `S`, `M`, `L`, `XL`).
- `MAX WORKER COUNT <count>` — устанавливает максимальное количество вычислительных пулов по умолчанию.
- `WORKER IDLE TTL <seconds> SECONDS` — устанавливает время простоя для вычислительных пулов по умолчанию.
- `SCALING STRATEGY <strategy>` — устанавливает стратегию масштабирования по умолчанию (`STD` или `ECO`).

Вывод списка всех вычислительных пулов

```
SHOW WORKER POOLS;
```

Выводит список всех вычислительных пулов в системе.

Это утверждение отображает информацию обо всех доступных вычислительных пулах, включая их имена и конфигурации.

Чтобы увидеть вычислительный пул в ответе, требуется привилегия `MONITOR`.

Вывод текущего вычислительного пула

```
SHOW WORKER POOL;
```

Выводит вычислительный пул, который в данный момент используется для выполнения запроса.

Это утверждение отображает информацию о текущем активном вычислительном пуле, включая его имя и конфигурацию.

Установка текущего вычислительного пула

```
USE WORKER POOL <pool_name>;
```

Устанавливает указанный вычислительный пул в качестве текущего пула для выполнения запросов.

Этот оператор изменяет вычислительный пул, который будет использоваться для выполнения последующих запросов. Указанный пул должен существовать, а пользователь должен иметь необходимые привилегии для его использования.

Для использования этого оператора необходима привилегия `USAGE`.

Увеличение размеров вычислительного пула

Если вам как пользователю требуется изменить размер используемого вами вычислительного пула (например, увеличить его), то рекомендуется выбрать для работы другой вычислительный

пул из списка доступных, не изменяя атрибуты текущего вычислительного пула.

Для этого нужно сделать следующее.

Выводим размер используемого вычислительного пула:

```
SHOW WORKER POOL;
```

```
+----------------+------+
| worker_pool    | size |
+----------------+------+
| compute_s      | S    |
+----------------+------+
```

Выводим размеры доступных для использования вычислительных пулов:

```
SHOW WORKER POOLS;
```

```
+----------------+-------+
| name           | value |
+----------------+-------+
| client_compute | S     |
+----------------+-------+
| compute_m      | M     |
+----------------+-------+
| compute_s      | S     |
+----------------+-------+
| compute_l      | L     |
+----------------+-------+
```

Переключаемся на вычислительный пул большего размера:

```
USE WORKER POOL compute_m;
```

```
+--------+
| status |
+--------+
| USE    |
+--------+
```

Управление пользователями и правами

В Tengri реализована развитая система разграничения прав доступа.

Поддержаны принципы:

- DAC — *избирательное управление доступом*
- RBAC — *управление доступом на основе ролей*

Ключевые концепты системы разграничения прав

Защищаемый объект

Сущность, к которой может быть предоставлен доступ (привилегии).

Роль

Сущность, которой могут быть предоставлены права доступа (привилегии). Роли могут назначаться пользователям или другим ролям.

Привилегия

Определенный уровень доступа к объекту. Назначаются пользователям или ролям.

Пользователь

Идентификатор, связанный с человеком или службой. Пользователь является объектом, которому могут быть предоставлены привилегии и/или роли.

Операции с объектами системы разграничения прав

- Операции с пользователями
  - Создание нового пользователя
  - Изменение атрибутов пользователя
  - Сброс пользователя
  - Вывод информации о пользователе
  - Вывод списка всех пользователей
- Операции с ролями
  - Создание новой роли
  - Изменение атрибутов роли
  - Сброс роли

    - Вывод списка всех ролей
    - Установка активной роли
    - Предоставление ролям атрибутов
- Операции с привилегиями
    - Предоставление привилегий
    - Отзыв привилегий

Операции с пользователями

- `CREATE USER` — Создание нового пользователя
- `ALTER USER` — Изменение атрибутов пользователя
- `DROP USER` — Сброс пользователя
- `DESCRIBE USER` — Вывод информации о пользователе
- `SHOW USERS` — Вывод списка всех пользователей

Создание нового пользователя

```
CREATE [OR REPLACE] USER [IF NOT EXISTS] <user_name>
    [SET DEFAULT WORKER POOL <pool_id>]
    [IDENTIFIED BY (TRUST <ip_address> | PASSWORD <user_password>)];
```

Создает нового пользователя с указанным именем.

Модификатор `OR REPLACE`

Если указан опциональный модификатор `OR REPLACE`, то это действие эквивалентно удалению текущего пользователя и созданию нового пользователя с тем же именем пользователя. Вновь созданный пользователь не наследует никаких ролей и атрибутов.

Модификатор `IF NOT EXISTS`

Если указан опциональный модификатор `IF NOT EXISTS`, то запрос будет выполнен только в том случае, если пользователь с именем `<user_name>` не существует, в противном случае ничего не произойдет.

Модификаторы `OR REPLACE` и `IF NOT EXISTS` являются взаимоисключающими. Если указать их оба, это приведет к ошибке.

Параметр `SET DEFAULT WORKER POOL`

Опциональный параметр `SET DEFAULT WORKER POOL <pool_id>` устанавливает вычислительный пул `<pool_id>` по умолчанию для указанного пользователя.

Параметр `IDENTIFIED BY TRUST`

Опциональный параметр `IDENTIFIED BY TRUST <ip_address>` устанавливает проверку подлинности указанного пользователя через IP адрес `<ip_address>`.

▼ *Посмотреть примеры*

- ```
  CREATE USER ivanov IDENTIFIED BY TRUST '127.0.0.1';
  ```
- ```
  ALTER USER ivanov IDENTIFIED BY TRUST '127.0.0.1';
  ```

Параметр `IDENTIFIED BY PASSWORD`

Опциональный параметр `IDENTIFIED BY PASSWORD <user_password>` устанавливает проверку подлинности указанного пользователя через пароль `<user_password>` с использованием алгоритма `SCRAM-SHA-256`.

▼ *Посмотреть примеры*

- ```
  CREATE USER ivanov IDENTIFIED BY PASSWORD 'QWERTY123456';
  ```
- ```
  ALTER USER ivanov IDENTIFIED BY PASSWORD 'QWERTY123456';
  ```

Изменение атрибутов пользователя

```
ALTER USER [IF EXISTS] <username>
    RENAME TO <new_name>;

ALTER USER [IF EXISTS] <username>
    SET DEFAULT WORKER POOL <pool_id>;

ALTER USER [IF EXISTS] <username>
    IDENTIFIED BY (TRUST <ip_address> | PASSWORD <user_password>);
```

Изменяет атрибуты пользователя.

- `RENAME TO <new_name>` — атомарная операция, которая переименовывает пользователя на новое имя. Никакие другие свойства данного пользователя (предоставленные роли, привилегии и т.д.) при этом не изменяются.
- `SET DEFAULT WORKER POOL <pool_id>` — см. Параметр `SET DEFAULT WORKER POOL`.
- `IDENTIFIED BY TRUST <ip_address>` — см. Параметр `IDENTIFIED BY TRUST`.
- `IDENTIFIED BY PASSWORD <user_password>` — см. Параметр `IDENTIFIED BY PASSWORD`.

Опциональный модификатор `IF EXISTS` ограничивает запрос только теми случаями, в которых

указанный объект существует.

Сброс пользователя

```
DROP USER [IF EXISTS] <username>;
```

Полностью удаляет пользователя из системы.

Не сохраняется ничего, включая предоставленные роли и права доступа.

Возможно, что некоторые объекты потеряют права доступа.

Опциональный модификатор `IF EXISTS` ограничивает запрос только теми случаями, в которых указанный объект существует.

Вывод информации о пользователе

```
DESC[RIBE] USER <username>;
```

Выводит всю информацию о пользователе, включая его атрибуты и значения по умолчанию.

Вывод списка всех пользователей

```
SHOW USERS;
```

Вывод списка всех зарегистрированных пользователей.

Операции с ролями

- `CREATE ROLE` — Создание новой роли
- `ALTER ROLE` — Изменение атрибутов роли
- `DROP ROLE` — Сброс роли
- `SHOW ROLES` — Вывод списка всех ролей
- `USE ROLE` — Установка активной роли
- `GRANT` — Предоставление ролям атрибутов

Создание новой роли

```
CREATE [OR REPLACE] ROLE [IF NOT EXISTS] <role_name>;
```

Создает новую роль с указанным именем.

Если указан модификатор `OR REPLACE`, то конечное действие эквивалентно удалению существующей роли и созданию новой роли с тем же именем. Вновь созданная роль не наследует никаких атрибутов или привилегий от предыдущей роли.

Опциональный модификатор `IF NOT EXISTS` ограничивает запрос только теми случаями, в которых указанный объект еще не существует.

Модификаторы являются взаимоисключающими. Если указать их оба, это приведет к ошибке.

Изменение атрибутов роли

```
ALTER ROLE [IF EXISTS] <role_name> RENAME TO <new_role_name>;

ALTER ROLE [IF EXISTS] <role_name> WITH <attribute>;
```

Изменяет существующую роль.

Действие может заключаться либо в переименовании роли, либо в изменении ее атрибутов.

Доступные атрибуты:

- `SUPERUSER` — предоставляет роли привилегии суперпользователя.
- `CREATEROLE` — позволяет роли создавать новые роли.

Опциональный модификатор `IF EXISTS` ограничивает запрос только теми случаями, в которых указанный объект существует.

Сброс роли

```
DROP ROLE [IF EXISTS] <role_name>;
```

Удаляет роль из системы.

Команда завершится неудачей, если на роль все еще ссылаются какие-либо объекты (например, она предоставлена пользователям или имеет право доступа для каких-то объектов).

Опциональный модификатор `IF EXISTS` ограничивает запрос только теми случаями, в которых указанный объект существует.

Вывод списка всех ролей

```
SHOW [TERSE] ROLES
    [LIKE '<pattern>']
```

```
    [STARTS WITH '<pattern>']
    [LIMIT <number> [FROM '<pattern>']];
```

Выводит список всех ролей в системе. Вывод можно отфильтровать и ограничить с помощью различных опций:

- `TERSE` — отображает упрощенный формат вывода.
- `LIKE '<pattern>'` — фильтрует роли, имена которых соответствуют указанному шаблону.
- `STARTS WITH '<pattern>'` — фильтрует роли, имена которых начинаются с указанного шаблона.
- `LIMIT <number> [FROM '<pattern>']` — ограничивает количество возвращаемых ролей, опционально начиная с определенного имени роли.

Шаблоны `<pattern>` представляют собой чувствительные к регистру строковые литералы, которые могут содержать символы подстановки.

Установка активной роли

```
USE ROLE <role_name>;
```

Устанавливает указанную роль в качестве активной роли для текущего сеанса.

Если роль установлена как активная, пользователь получает все привилегии, предоставленные этой роли, на время сессии.

Чтобы использовать указанную роль, пользователю должны быть предоставлены соответствующие права.

Предоставление ролям атрибутов

```
GRANT <role_attributes> TO ROLE <role_name>;
```

Предоставляет указанной роли `<role_name>` атрибуты `<role_attributes>`.

Это утверждение позволяет роли иметь специальные атрибуты или выполнять операции над другими ролями в соответствии с предоставленными привилегиями.

Операции с привилегиями

- `GRANT` — Предоставление привилегий
- `REVOKE` — Отзыв привилегий

Предоставление привилегий

Привилегии на схемы

```
GRANT <schema_privileges> ON SCHEMA <schema_name> TO [ROLE] <role_name>;
```

Предоставляет роли указанные привилегии на указанную схему.

`<schema_privileges>` — это список привилегий, разделенный запятыми.

Доступные привилегии на схемы:

- `ADMIN` — позволяет удалять схему и раздавать права на схему.
- `MONITOR` — разрешает доступ к метаинформации о схеме.
- `USAGE` — разрешает доступ к объектам в схеме.
- `MODIFY` — позволяет изменять свойства схемы.
- `CREATE TABLE` — позволяет создавать таблицы в схеме.
- `CREATE VIEW` — позволяет создавать представления в схеме.

Ключевое слово `ROLE` перед именем целевой роли не является обязательным.

▼ *Посмотреть пример*

Предоставим роли `junior_admin` привилегии `MONITOR` и `MODIFY` на схему `main_schema`.

```
GRANT MONITOR, MODIFY
    ON SCHEMA main_schema
    TO ROLE junior_admin;
```

Привилегии на таблицы и представления

Прямой синтаксис:

```
GRANT <table_privileges>
    ON (TABLE | VIEW) <table_name>
    TO [ROLE] <role_name>;
```

Темпоральный синтаксис:

```
GRANT <table_privileges>
    ON ALL [EXISTING] [[AND] FUTURE] (TABLES | VIEWS | TABLES AND VIEWS)
    IN [SCHEMA]<schema_name>
    TO [ROLE] <role_name>;
```

Предоставляет роли указанные привилегии на таблицу или представление внутри указанной схемы.

`<table_privileges>` — это список привилегий, разделенный запятыми.

Доступные привилегии на таблицы и представления:

- `SELECT` — позволяет читать данные из таблицы.
- `INSERT` — позволяет добавлять новые строки в таблицу.
- `UPDATE` — позволяет изменять существующие строки в таблице.
- `DELETE` — позволяет удалять строки из таблицы.
- `MODIFY` — позволяет изменять структуру таблицы.
- `OWNERSHIP` — позволяет изменять владельца таблицы.

Чтобы выдать привилегию на все существующие (или те, которые будут созданы в будущем) объекты внутри схемы, используется темпоральный синтаксис.

Модификатор `EXISTING` ограничивает предоставление привилегий только существующими объектами.

Модификатор `FUTURE` ограничивает предоставление привилегий объектами, которые будут созданы в будущем.

Если ни один модификатор не указан, то действие распространяется на все объекты — и уже существующие, и те, которые будут созданы в будущем.

Ключевое слово `ROLE` перед именем целевой роли не является обязательным.

▼ *Посмотреть пример*

Предоставим роли `junior_admin` привилегии `SELECT` и `INSERT` на все существующие таблицы и представления внутри схемы `main_schema`.

```
GRANT SELECT, INSERT
    ON EXISTING TABLES AND VIEWS
    IN SCHEMA main_schema
    TO ROLE junior_admin;
```

Привилегии на каталоги

```
GRANT <catalog_privileges> ON CATALOG TO [ROLE] <role_name>;
```

Предоставляет роли указанные привилегии на каталог.

`<catalog_privileges>` — это список привилегий, разделенный запятыми.

Доступные привилегии на каталоги:

- `ADMIN` — любые действия с каталогом.
- `CREATE USER` — создание пользователей.
- `CREATE WORKER POOL` — создание вычислительных пулов.
- `CREATE SCHEMA` — создание схем.

Модификатор `EXISTING` ограничивает предоставление привилегий только существующими объектами.

Модификатор `FUTURE` ограничивает предоставление привилегий объектами, которые будут созданы в будущем.

Ключевое слово `ROLE` перед именем целевой роли не является обязательным.

Привилегии на вычислительные пулы

```
GRANT <worker_pool_privileges> ON WORKER POOL <worker_pool_name> TO [ROLE] <role_name>;
```

Предоставляет роли указанные привилегии на вычислительный пул.

`<worker_pool_privileges>` — это список привилегий, разделенный запятыми.

Доступные привилегии на вычислительные пулы:

- `ALL` — все возможные действия с вычислительным пулом.
- `ADMIN` — изменение вычислительного пула (права на выполнение команды `ALTER WORKER POOL`).
- `USAGE` — использование вычислительного пула (права на выполнение команды `USE WORKER POOL`).
- `MONITOR` — мониторинг вычислительного пула.

▼ *Посмотреть пример*

Предоставим роли `junior_analyst` привилегии `USAGE` и `MONITOR` на вычислительный пул `compute_xl`.

```
GRANT USAGE, MONITOR
    ON WORKER POOL compute_xl
    TO junior_analyst;
```

Отзыв привилегий

```
REVOKE <schema_privileges>
    ON SCHEMA <schema_name> FROM ROLE <role_name>;

REVOKE <table_privileges>
    ON (TABLE | VIEW) <table_name> [IN SCHEMA <schema_name>] FROM ROLE <role_name>;

REVOKE <catalog_privileges>
    ON CATALOG FROM ROLE <role_name>;
```

```
REVOKE <worker_pool_privileges>
    ON WORKER POOL <worker_pool_name> FROM ROLE <role_name>;
```

Отзывает указанные привилегии на указанный объект у роли.

`<schema_privileges>`, `<table_privileges>`, `<catalog_privileges>` и `<worker_pool_privileges>` — это списки привилегий, разделенные запятыми. Конкретные привилегии зависят от целевого объекта.

Модификатор `EXISTING` ограничивает отзыв привилегий только существующими объектами.

Модификатор `FUTURE` ограничивает отзыв привилегий объектами, которые будут созданы в будущем.

▼ *Посмотреть пример*

Отзовем у роли `junior_admin` привилегии `SELECT` и `INSERT` на все существующие таблицы и представления внутри схемы `main_schema`.

```
REVOKE SELECT, INSERT
    ON EXISTING TABLES AND VIEWS
    IN SCHEMA main_schema
    FROM ROLE junior_admin;
```

Описание данных (DDL)

Команды DDL используются для описания данных — для создания, управления и удаления таблиц, схем и представлений.

- Операции с таблицами
- Операции со схемами
- Операции с представлениями

Операции с таблицами

- `CREATE TABLE` — Создание новой таблицы
- `INSERT` — Добавление данных в таблицу
- `DROP TABLE` — Удаление таблицы
- `SHOW TABLES` — Вывод списка всех таблиц

Создание новой таблицы

```
CREATE [OR REPLACE] TABLE [IF NOT EXISTS] [<table_schema>.]<table_name>
    (<column_name> <column_type>
        [NOT NULL] [DEFAULT <default_expr>],
    ...
    )
    [WITH (<table_param>, ... )]
```

Создает новую таблицу с указанным именем и указанными столбцами.

```
CREATE [OR REPLACE] TABLE [IF NOT EXISTS] [<table_schema>.]<table_name> AS
    <select_expr>
    [WITH (<table_param>, ... )]
```

Создает новую таблицу с указанным именем на основе результата запроса `SELECT`.

Параметры

- `<table_name>` — имя создаваемой таблицы

- `<table_schema>` — схема создаваемой таблицы

- `<column_name>` — имя столбца создаваемой таблицы

- `<column_type>` — тип данных столбца создаваемой таблицы (см. раздел Типы данных)

- `NOT NULL` — данный столбец не принимает значения `NULL`

- `<default_expr>` — константа или константное выражение по умолчанию

- `<select_expr>` — выражение `SELECT`, результат выполнения которого будет записан в создаваемую таблицу

- `<table_param>` — параметры создаваемой таблицы

  ```
  table_param ::= [<name> = <value>]
  ```

  Возможные значения:

  - `snapshot_ttl = <duration>` — глубина хранения снапшотов (версий таблицы).
    По умолчанию: 7 дней, но не более 1000 снапшотов.
    Например: `'1 week'`, `'2 days'`, `'4 days 3 hours 5 minutes 30 seconds'`
  - `order_by = <column_name>` — столбец для сортировки, используемый для уплотнения данных таблицы. Данные в таблице хранятся с произвольной сортировкой, упорядоченность не гарантируется.

Если указан модификатор `OR REPLACE`, то конечное действие эквивалентно удалению существующей таблицы и созданию новой с тем же именем.

Опциональный модификатор `IF NOT EXISTS` ограничивает запрос только теми случаями, в которых указанный объект еще не существует.

Модификаторы являются взаимоисключающими. Если указать их оба, это приведет к ошибке.

Добавление данных в таблицу

```
INSERT INTO <target_table> [ ( <target_col_name> [ , ... ] ) ]
       {
         VALUES ( { <value> | DEFAULT | NULL } [ , ... ] ) [ , ( ... ) ]  |
         <query>
       }
```

Обновляет таблицу, вставляя в нее одну или несколько строк. Значения, вставляемые в каждый столбец таблицы, могут быть указаны явно или получены из вложенного запроса.

Параметры

- `<target_table>` — имя целевой таблицы, в которую будут вставлены строки

- `VALUES ( value | DEFAULT | NULL [ , ⋯ ] ) [ , ( ⋯ ) ]` — указывает одно или несколько значений для вставки в соответствующие столбцы целевой таблицы.
  - `value` — явно указанное значение; может быть литералом или выражением.
  - `DEFAULT` — значение по умолчанию для соответствующего столбца целевой таблицы.
  - `NULL` — пустое значение.

  Значения разделяются запятыми.

  Вы можете вставить несколько строк, указав дополнительные наборы значений в выражении.

- `query` — запрос, который возвращает значения для вставки в соответствующие столбцы. Это позволяет вставлять строки в целевую таблицу из одной или нескольких исходных таблиц.

Пример вставки явно заданных значений

Вставим в таблицу `capitals` значения для столбцов `country` и `capital`:

```
CREATE TABLE capitals (country VARCHAR, capital VARCHAR);
INSERT INTO capitals VALUES
    ('Russia', 'Moscow'),
    ('Italy', 'Rome'),
    ('Spain', 'Madrid'),
    ('France', 'Paris');

SELECT * FROM capitals;
```

```
+---------+---------+
| country | capital |
+---------+---------+
| France  | Paris   |
+---------+---------+
| Italy   | Rome    |
+---------+---------+
| Russia  | Moscow  |
+---------+---------+
| Spain   | Madrid  |
+---------+---------+
```

Пример вставки результатов вложенного запроса

Теперь создадим другую таблицу `capitals_m` и сделаем вставку строк из `capitals`, которые будут результатом вложенного запроса `SELECT`. Вставим такие строки из `capitals`, в которых значение `capital` содержит `M`.

```
CREATE TABLE capitals_m (country VARCHAR, capital VARCHAR);
INSERT INTO capitals_m (country, capital)
    SELECT * FROM capitals
        WHERE capital LIKE '%M%';

SELECT * FROM capitals_m;
```

```
+---------+---------+
| country | capital |
+---------+---------+
| Russia  | Moscow  |
+---------+---------+
| Spain   | Madrid  |
+---------+---------+
```

Удаление таблицы

```
DROP TABLE [IF EXISTS] <table_name>;
```

Удаляет таблицу с указанным именем.

Опциональный модификатор `IF EXISTS` ограничивает запрос только теми случаями, в которых указанный объект существует.

Вывод списка всех таблиц

```
SHOW TABLES;
```

Выводит список всех существующих таблиц.

Операции со схемами

- `CREATE SCHEMA` — Создание новой схемы
- `ALTER SCHEMA` — Изменение атрибутов схемы
- `DROP SCHEMA` — Удаление схемы
- `SHOW SCHEMAS` — Вывод списка всех схем

Создание новой схемы

```
CREATE [OR REPLACE] SCHEMA [IF NOT EXISTS] <schema_name>
```

Создает новую схему с указанным именем.

Если указан модификатор `OR REPLACE`, то конечное действие эквивалентно удалению существующей схемы со всеми объектами в ней и созданию новой схемы с тем же именем.

Опциональный модификатор `IF NOT EXISTS` ограничивает запрос только теми случаями, в которых указанный объект еще не существует.

Модификаторы являются взаимоисключающими. Если указать их оба, это приведет к ошибке.

Изменение атрибутов схемы

```
ALTER SCHEMA [IF EXISTS] <schema_name> RENAME TO <new_name>;
```

Изменяет схему с помощью указанного действия.

В настоящее время для изменения схем доступно только одно действие:

- Действие `RENAME TO` переименовывает схему в указанное имя `<new_name>`. Все атрибуты и права при этом сохраняются.

Опциональный модификатор `IF EXISTS` ограничивает запрос только теми случаями, в которых указанный объект существует.

Удаление схемы

```
DROP SCHEMA [IF EXISTS] <schema_name>;
```

Удаляет схему с указанным именем.

Опциональный модификатор `IF EXISTS` ограничивает запрос только теми случаями, в которых указанный объект существует.

Вывод списка всех схем

```
SHOW (SCHEMA | SCHEMAS | SCHEMATA);
```

Выводит список всех существующих схем.

Операции с представлениями

- `CREATE VIEW` — Создание нового представления
- `DROP VIEW` — Удаление представления

Создание нового представления

```
CREATE [OR REPLACE] VIEW [IF NOT EXISTS] [<view_schema>.]<view_name> AS
    <select_expr>
```

Создает новое представление с указанным именем на основе результата указанного запроса `SELECT`.

Параметры

- `<view_name>` — имя создаваемого представления

- `<view_schema>` — схема создаваемого представления

- `<select_expr>` — выражение `SELECT`, на основе результата выполнения которого будет создано представление

Если указан модификатор `OR REPLACE`, то конечное действие эквивалентно удалению существующего представления и созданию нового с тем же именем.

Опциональный модификатор `IF NOT EXISTS` ограничивает запрос только теми случаями, в которых указанный объект еще не существует.

Модификаторы являются взаимоисключающими. Если указать их оба, это приведет к ошибке.

Удаление представления

```
DROP VIEW [IF EXISTS] <view_name>;
```

Удаляет представление из системы.

Опциональный модификатор `IF EXISTS` ограничивает запрос только теми случаями, в которых указанный объект существует.

Синтаксис запросов к данным

Выражения SQL

Tengri поддерживает запросы к данным с использованием стандартных ключевых слов SQL и следующего базового синтаксиса:

```
[ WITH ... ]
SELECT
   ...
[ FROM ...
   [ JOIN ... ]
[ WHERE ... ]
[ GROUP BY ...
   [ HAVING ... ] ]
[ QUALIFY ... ]
[ ORDER BY ... ]
[ LIMIT ... ]
```

Подробные описания для поддерживаемых ключевых слов представлены в разделах:

- `SELECT`
- `WITH`
- `FROM`
- `JOIN`
- `UNION`
- `WHERE`
- `GROUP BY`
- `HAVING`
- `QUALIFY`
- `ORDER BY`
- `LIMIT`
- `OFFSET`
- `LIKE`
- `ILIKE`
- `SIMILAR TO`
- `AS`

Функции

- Агрегатные функции

- Числовые функции
- Текстовые функции
- Функции для регулярных выражений
- Функции для даты и времени
- Функции для JSON
- Функции для двоичных данных
- Оконные функции
- Утилиты

SELECT

Выражение `SELECT` определяет список столбцов, которые будут возвращены запросом. Хотя оно стоит первым в запросе, логически выражения в этом выражении выполняются последними. Выражение `SELECT` может содержать произвольные выражения, преобразующие вывод, а также агрегатные и оконные функции.

Синтаксис

Выбор всех столбцов

```
[ ... ]
SELECT [ { ALL | DISTINCT } ]
       [ TOP <n> ]
       [{<object_name>|<alias>}.]*

       [ ILIKE '<pattern>' ]

       [ EXCLUDE
         {
           <col_name> | ( <col_name>, <col_name>, ... )
         }
       ]

       [ REPLACE
         {
           ( <expr> AS <col_name> [ , <expr> AS <col_name>, ... ] )
         }
       ]

       [ RENAME
         {
           <col_name> AS <col_alias>
           | ( <col_name> AS <col_alias>, <col_name> AS <col_alias>, ... )
         }
       ]
```

После `SELECT *` можно указать следующие комбинации ключевых слов. Ключевые слова должны быть в указанном ниже порядке:

```
SELECT * ILIKE ... REPLACE ...
```

```
SELECT * ILIKE ... RENAME ...
```

```
SELECT * ILIKE ... REPLACE ... RENAME ...
```

```
SELECT * EXCLUDE ... REPLACE ...
```

```
SELECT * EXCLUDE ... RENAME ...
```

```
SELECT * EXCLUDE ... REPLACE ... RENAME ...
```

```
SELECT * REPLACE ... RENAME ...
```

Выбор определенных столбцов

```
[ ... ]
SELECT [ { ALL | DISTINCT } ]
       [ TOP <n> ]
       {
         [{<object_name>|<alias>}.]<col_name>
         | [{<object_name>|<alias>}.]$<col_position>
         | <expr>
       }
       [ [ AS ] <col_alias> ]
       [ , ... ]
[ ... ]
```

В списке столбцов поддерживается конечная запятая. Например, поддерживается следующее выражение `SELECT`:

```
SELECT emp_id,
       name,
       department,
    FROM employees;
```

Параметры

- `ALL | DISTINCT`
  Указывает, следует ли выполнять удаление дубликатов в наборе результатов:
  - `ALL` включает все значения в набор результатов.
  - `DISTINCT` удаляет дубликаты из набора результатов.

  По умолчанию: `ALL`
  Подробное описание см. в разделе Выражение `DISTINCT`.

- `<object_name>` или `<alias>`
  Указывает идентификатор объекта или псевдоним объекта (как определено в выражении `FROM`).

- `*` (звездочка)
  Звездочка является сокращением, обозначающим, что в выводе должны быть указаны все столбцы указанного объекта или все столбцы всех объектов, если `*` не сопровождается именем объекта или псевдонимом.
  Подробное описание см. в разделе Выражение со звездочкой `*`.

- `<col_name>`
  Указывает идентификатор столбца (как определено в выражении `FROM`).

- `$<col_position>`
  Указывает положение столбца (начиная с 1) в соответствии с определением в выражении `FROM`. Если на столбец есть ссылка из таблицы, это число не может превышать максимальное количество столбцов в таблице.

- `<expr>`
  Указывает выражение (например математическое), которое вычисляется как определенное значение для любой заданной строки.

- `[ AS ] <col_alias>`
  Указывает псевдоним столбца, назначенный результирующему выражению. Он используется в качестве отображаемого имени в списке `SELECT` верхнего уровня и в качестве имени столбца во встроенном представлении.

Не назначайте такой псевдоним столбца, который будет совпадать с именем другого столбца, на который есть ссылка в запросе. Например, если вы выбираете столбцы с именами `prod_id` и `product_id`, не назначайте псевдоним `product_id` столбцу `prod_id`.

См. также раздел `AS`

- `ILIKE <pattern>`
  Указывает, что в результаты запроса должны быть включены только столбцы, соответствующие заданному шаблону.
  В шаблоне можно использовать следующие выражения SQL:
  - Символ подчеркивания `_` для сопоставления с любым одиночным символом.
  - Символ процента `%` для сопоставления с любой последовательностью из нуля или более символов.

- `EXCLUDE <col_name>` или
  `EXCLUDE (<col_name_1>, <col_name_2>, ⋯)`
  Указывает столбцы, которые должны быть исключены из результатов запроса.

- `REPLACE (<expr> AS <col_name> [ , <expr> AS <col_name>, ⋯] )`
  Replaces the value of col_name with the value of the evaluated expression expr. Заменяет значение указанного столбца на результат применения выражения `<expr>` к его исходному значению.

- `RENAME <col_name> AS <col_alias>` или
  `RENAME (<col_name_1> AS <col_alias_1>, <col_name_2> AS <col_alias_2>, ⋯)`
  Указывает псевдонимы столбцов, которые будут использованы в результатах запроса.

- `TOP <n>`
  Указывает максимальное количество строк, которые будут результатом запроса.

Примеры

- Выбираем все столбцы из таблицы с именем `my_table`:

```
SELECT * FROM my_table;
```

- Выполняем арифметические действия над столбцами таблицы и указываем псевдоним:

```
SELECT column_1 + column_2 AS sum, sqrt(column_1) AS sq_root FROM my_table;
```

- Используем префиксные псевдонимы для получения того же результата:

```
SELECT
    sum: column_1 + column_2,
    sq_root: sqrt(column_1)
```

```
FROM my_table;
```

- Выбираем все уникальные имена из таблицы `employees` (*работники*):

```
SELECT DISTINCT name FROM employees;
```

- Выводим общее количество строк в таблице `employees`:

```
SELECT count(*) FROM employees;
```

- Выбираем все столбцы, кроме столбца `name`, из таблицы `employees`:

```
SELECT * EXCLUDE (name) FROM employees;
```

- Выбираем все столбцы из таблицы `employees`, но заменяем `name` на результат применения функции `lower(name)`:

```
SELECT * REPLACE (lower(name) AS name) FROM employees;
```

- Выбираем из таблицы все столбцы, соответствующие заданному регулярному выражению:

```
SELECT COLUMNS('number\d+') FROM employees;
```

- Вычисляем функцию по всем заданным столбцам таблицы:

```
SELECT min(COLUMNS(*)) FROM employees;
```

- Используем двойные кавычки (`"`), чтобы выбрать столбцы с пробелами или специальными символами:

```
SELECT "Фамилия Имя Отчество" FROM employees;
```

Список столбцов `SELECT`

Выражение `SELECT` содержит список выражений, которые определяют результат запроса. Список `SELECT` может ссылаться на любые столбцы в выражении `FROM` и объединять их с помощью выражений. Поскольку результатом SQL-запроса является таблица, каждое выражение в выражении `SELECT` также имеет имя. Выражения могут быть явно названы с помощью оператора `AS` (например, `expr AS name`). Если пользователь не указал имя, то выражения именуются системой автоматически.

Имена столбцов не чувствительны к регистру, если они задаются без кавычек.

Выражение со звездочкой *

Выражение со звездочкой () — это специальное выражение, которое расширяется до множества выражений на основе содержимого выражения FROM. В простейшем случае расширяется до всех выражений в выражении FROM.

- Выбираем все столбцы из таблицы с именем my_table:

```
SELECT * FROM my_table;
```

Выражение DISTINCT

Выражение DISTINCT можно использовать для получения только уникальных строк в результате — таким образом, все дубликаты будут отфильтрованы.

- Выбираем все уникальные имена из таблицы employees:

```
SELECT DISTINCT name FROM employees;
```

Запросы, начинающиеся с SELECT DISTINCT, выполняют дедупликацию, которая является дорогостоящей операцией. Поэтому используйте DISTINCT только в случае необходимости.

Выражение DISTINCT ON

Выражение DISTINCT ON возвращает только одну строку для каждого уникального значения в наборе выражений, как определено в выражении ON. Если присутствует условие ORDER BY, то возвращается первая строка, которая встречается в соответствии с условием ORDER BY. Если условие ORDER BY отсутствует, первая встречающаяся строка не определена и может быть любой строкой в таблице.

Выбираем сотрудника с наибольшей зарплатой для каждого отдела:

```
SELECT DISTINCT ON(department) name, salary
    FROM employees
    ORDER BY salary DESC;
```

При запросе больших наборов данных использование DISTINCT для всех столбцов может быть дорогостоящей операцией. Поэтому рассмотрите возможность использования DISTINCT ON для столбца (или набора столбцов), гарантирующего достаточную степень уникальности результатов. Например, использование DISTINCT ON для ключевого столбца (столбцов) таблицы гарантирует полную уникальность.

Агрегатные функции

Агрегатные функции — это функции, которые объединяют значения из нескольких строк в одно. Когда агрегатные функции присутствуют в выражении `SELECT`, запрос превращается в агрегатный запрос. В агрегатном запросе **все** выражения должны быть либо частью агрегатной функции, либо частью группы (как указано в выражении `GROUP BY`).

- Получаем общее количество строк в таблице сотрудников:

```
SELECT count(*) FROM employees;
```

- Получаем общее количество строк в таблице сотрудников, сгруппированных по отделам:

```
SELECT department, count(*)
    FROM employees
    GROUP BY department;
```

Подробное описание см. в разделе Агрегатные функции.

Оконные функции

Оконные функции — это функции, которые выполняют вычисления для набора строк, некоторым образом связанных с текущей строкой. Вызов оконной функции всегда содержит выражение `OVER`, следующее за названием и аргументами оконной функции. Выражение `OVER` определяет, как именно нужно разделить строки запроса для обработки оконной функцией.

- Получаем столбец `row_number`, содержащий инкрементные идентификаторы для каждой строки таблицы зарплат:

```
SELECT row_number() OVER ()
    FROM salaries;
```

- Вычисляем разницу между текущей и предыдущей суммой в порядке убывания времени:

```
SELECT amount - lag(amount) OVER (ORDER BY time)
    FROM salaries;
```

Подробное описание см. в разделе Оконные функции.

`WITH`

Выражение `WITH` позволяет указать общие табличные выражения (CTE). Регулярные (нерекурсивные) общие табличные выражения — это, по сути, представления (view), которые ограничены рамками конкретного запроса. CTE могут ссылаться друг на друга и быть вложенными. Рекурсивные CTE могут ссылаться сами на себя.

Синтаксис

Подзапрос

```
[ WITH
       <cte_name1> [ ( <cte_column_list> ) ] AS ( SELECT ...  )
   [ , <cte_name2> [ ( <cte_column_list> ) ] AS ( SELECT ...  ) ]
   [ , <cte_nameN> [ ( <cte_column_list> ) ] AS ( SELECT ...  ) ]
]
SELECT ...
```

Рекурсивное CTE:

```
[ WITH [ RECURSIVE ]
    <cte_name1> ( <cte_column_list> ) AS ( anchorClause UNION ALL recursiveClause)
 [, <cte_name2> ( <cte_column_list> ) AS ( anchorClause UNION ALL recursiveClause)]
 [, <cte_nameN> ( <cte_column_list> ) AS ( anchorClause UNION ALL recursiveClause)]
]
SELECT ...
```

где:

```
anchorClause ::=
    SELECT <anchor_column_list> FROM ...

recursiveClause ::=
    SELECT <recursive_column_list> FROM ... [ JOIN ... ]
```

Параметры

- `<cte_name1>` , `<cte_nameN>`
  Имя CTE.

- `<cte_column_list>`
  Имена столбцов в CTE (общее табличное выражение).

- `<anchor_column_list>`
  Столбцы, используемые в анкерном выражении для рекурсивного CTE. Столбцы в этом списке должны соответствовать столбцам, определенным в `<cte_column_list>`.

- `<recursive_column_list>`

Столбцы, используемые в рекурсивном выражении для рекурсивного CTE. Столбцы в этом списке должны соответствовать столбцам, определенным в `<cte_column_list>`.

Примеры базовых CTE

Создадим CTE с именем `cte` и используем его в основном запросе:

```
WITH cte AS (SELECT 33 AS amount)
    SELECT * FROM cte;
```

```
+--------+
| amount |
+--------+
| 33     |
+--------+
```

Создадим два CTE `cte1` и `cte2`, где второй CTE ссылается на первый CTE:

```
WITH
    cte1 AS (SELECT 33 AS i),
    cte2 AS (SELECT i * 2 AS amount_doubled FROM cte1)
SELECT * FROM cte2;
```

```
+----------------+
| amount_doubled |
+----------------+
| 66             |
+----------------+
```

Для столбцов CTE можно задавать имена:

```
WITH my_cte(amount) AS (SELECT 33 AS i)
SELECT * FROM my_cte;
```

```
+--------+
| amount |
+--------+
| 33     |
+--------+
```

FROM

Выражение `FROM` указывает источник данных, с которыми должна работать остальная часть

запроса. С точки зрения логики, выражение `FROM` — это место, с которого начинается выполнение запроса.

Выражение `FROM` может содержать одну таблицу, комбинацию из нескольких таблиц, объединенных с помощью выражения `JOIN`, или другой запрос `SELECT` в узле подзапроса.

В Tengri также имеется дополнительный синтаксис `FROM`-first, который позволяет выполнять запрос без оператора `SELECT`.

Синтаксис

```
SELECT ...
FROM objectReference [ JOIN objectReference [ ... ] ]
[ ... ]
```

где:

```
objectReference ::=
   {
      [<namespace>.]<object_name>
           [ AT | BEFORE ( <object_state> ) ]
           [ CHANGES ( <change_tracking_type> ) ]
           [ MATCH_RECOGNIZE ]
           [ PIVOT | UNPIVOT ]
           [ [ AS ] <alias_name> ]
     | <table_function>
           [ PIVOT | UNPIVOT ]
           [ [ AS ] <alias_name> ]
     | ( VALUES (...) )
     | ( <subquery> )
           [ [ AS ] <alias_name> ]
     | DIRECTORY( @<stage_name> )
   }
```

Параметры

- `[<namespace>.]<object_name>`
  Указывает имя объекта (таблицы или представления), к которому производится запрос.

- `<table_function>`
  Указывает системную табличную функцию, табличную функцию UDF или метод класса для вызова в выражении `FROM`.

- `VALUES`
  Выражение `VALUES` может содержать литеральные значения или выражения, которые будут

использоваться в выражении `FROM`. Это выражение также может содержать псевдонимы таблиц и столбцов (не показаны на схеме выше).

- `<subquery>`
  Подзапрос в выражении `FROM`.

- `DIRECTORY( @stage_name )`
  Указывает имя этапа, включающего таблицу каталогов.

- `[ AS ] <alias_name>`
  Указывает имя, заданное для объекта, к которому она относится. Может использоваться с любыми другими подзапросами в выражении `FROM`. Оператор `AS` может опускаться.

- `JOIN`
  Указывает на выполнение соединения между двумя (или более) таблицами (или представлениями или табличными функциями). Соединение может быть внутренним внешним или иметь другой тип. Соединение может использовать ключевое слово `JOIN` или альтернативный поддерживаемый синтаксис соединения.
  Подробное описание см. в разделе `JOIN`.

Примеры

- Выбираем все столбцы из таблицы с именем `my_table`:

```
SELECT *
FROM my_table;
```

- Выбираем все столбцы из таблицы, используя синтаксис `FROM`-first:

```
FROM my_table
SELECT *;
```

- Выбираем все столбцы, используя синтаксис `FROM`-first и опуская выражение `SELECT`:

```
FROM my_table;
```

- Выбираем все столбцы из таблицы с именем `my_table` через псевдоним `mt`:

```
SELECT mt.*
FROM my_table mt;
```

- Используем префиксный псевдоним:

```
SELECT mt.*
FROM mt: my_table;
```

- Выбираем все столбцы из таблицы `my_table` в схеме `my_schema`:

```
SELECT *
FROM my_schema.my_table;
```

- Выбираем столбец `i` из табличной функции `range`, где первый столбец функции диапазона переименован в `i`:

```
SELECT t.i
FROM range(1000) AS t(i);
```

- Выбираем все столбцы из подзапроса:

```
SELECT *
FROM (SELECT * FROM my_table);
```

- Объединяем две таблицы:

```
SELECT *
FROM my_table
JOIN other_table
    ON my_table.key = other_table.key;
```

  Подробное описание см. в разделе `JOIN`.

Синтаксис `FROM`-first

Tengri поддерживает синтаксис `FROM`-first, т. е. позволяет поместить выражение `FROM` перед выражением `SELECT` или полностью опустить выражение `SELECT`. Продемонстрируем это на примере:

```
CREATE TABLE capitals (country VARCHAR, capital VARCHAR);
INSERT INTO capitals VALUES
    ('Russia', 'Moscow'),
    ('Italy', 'Rome'),
    ('Spain', 'Madrid'),
    ('France', 'Paris');
```

Синтаксис `FROM`-first с выражением `SELECT`

Следующее выражение демонстрирует использование синтаксиса `FROM`-first:

```
FROM capitals
SELECT *;
```

```
+---------+---------+
| country | capital |
+---------+---------+
| France  | Paris   |
+---------+---------+
| Italy   | Rome    |
+---------+---------+
| Russia  | Moscow  |
+---------+---------+
| Spain   | Madrid  |
+---------+---------+
```

Это эквивалентно:

```
SELECT *
FROM capitals;
```

```
+---------+---------+
| country | capital |
+---------+---------+
| France  | Paris   |
+---------+---------+
| Italy   | Rome    |
+---------+---------+
| Russia  | Moscow  |
+---------+---------+
| Spain   | Madrid  |
+---------+---------+
```

Синтаксис `FROM`-first без выражения `SELECT`

Следующее выражение демонстрирует необязательность выражения `SELECT`:

```
FROM capitals;
```

```
+---------+---------+
| country | capital |
```

```
+---------+---------+
| France  | Paris   |
+---------+---------+
| Italy   | Rome    |
+---------+---------+
| Russia  | Moscow  |
+---------+---------+
| Spain   | Madrid  |
+---------+---------+
```

Это тоже эквивалентно полному запросу:

```
SELECT *
FROM capitals;
```

```
+---------+---------+
| country | capital |
+---------+---------+
| France  | Paris   |
+---------+---------+
| Italy   | Rome    |
+---------+---------+
| Russia  | Moscow  |
+---------+---------+
| Spain   | Madrid  |
+---------+---------+
```

`JOIN`

Соединения (*JOINS*) — это фундаментальная реляционная операция, используемая для горизонтального соединения двух таблиц или отношений. Отношения называются левая и правая стороны соединения в зависимости от того, как они записаны в выражении `JOIN`. Каждая строка результата содержит столбцы из обоих отношений.

В соединении используется правило для сопоставления пар строк из каждого отношения. Часто это предикат, но могут быть указаны и другие правила.

Синтаксис

Могут использоваться следующие варианты синтаксиса:

```
SELECT ...
FROM <object_ref1> [
                     {
                       INNER
                       | { LEFT | RIGHT | FULL } [ OUTER ]
                     }
```

```
                  ]
                  JOIN <object_ref2>
  [ ON <condition> ]
[ ... ]
```

```
SELECT *
FROM <object_ref1> [
                    {
                      INNER
                      | { LEFT | RIGHT | FULL } [ OUTER ]
                    }
                  ]
                  JOIN <object_ref2>
  [ USING( <column_list> ) ]
[ ... ]
```

```
SELECT ...
FROM <object_ref1> [
                    {
                      | NATURAL [ { LEFT | RIGHT | FULL } [ OUTER ] ]
                      | CROSS
                    }
                  ]
                  JOIN <object_ref2>
[ ... ]
```

Параметры

- `<object_ref1>` и `<object_ref2>`
  Каждый `<object_ref>` представляет собой таблицу или источник данных, подобный таблице.

- `JOIN`
  Ключевое слово `JOIN`, указывающее, что таблицы должны быть соединены. `JOIN` комбинируется с другими ключевыми словами (например, `INNER` или `OUTER`), чтобы указать тип соединения.

- `ON <condition>`
  Булево выражение, которое определяет условия соединения.
  Подробное описание см. в разделе Условное соединение.

- `USING <column_list>`
  Список столбцов, по которым производится соединение. Столбцы должны быть одноименными в соединяемых таблицах.
  Подробное описание см. в разделе Условное соединение.

`OUTER JOIN` — внешнее соединение

Строки, не имеющие совпадений, могут быть возвращены, если указано соединение `OUTER`. Внешние соединения могут иметь один из типов:

- `LEFT` (Все строки из левого отношения встречаются хотя бы один раз)
- `RIGHT` (Все строки из правого отношения встречаются хотя бы один раз)
- `FULL` (Все строки из обоих отношений встречаются хотя бы один раз)

Соединение, которое не является внешним (`OUTER`), является внутренним (`INNER`) — возвращаются только те строки, которые удовлетворяют условию.

Если возвращается непарная строка, атрибуты другой таблицы устанавливаются в значение `NULL`.

`CROSS JOIN` — перекрёстное соединение (декартово произведение)

Простейшим типом соединения является `CROSS JOIN`. Для этого типа соединения нет никаких условий, и он просто возвращает все возможные пары.

Вернем все пары строк:

```
SELECT a.*, b.*
FROM a
CROSS JOIN b;
```

Это эквивалентно запросу, в котором просто опущено условие `JOIN`:

```
SELECT a.*, b.*
FROM a, b;
```

Условное соединение

Большинство соединений задается предикатом, который соединяет атрибуты с одной стороны с атрибутами с другой стороны. Условия могут быть явно указаны с помощью операторов `ON` и `WHERE`.

Покажем использование условных соединений на примере двух таблиц — `capitals` со странами и их столицами и `population` со странами и их населением (в миллионах).

```
CREATE TABLE capitals (cap_country VARCHAR, capital VARCHAR);
CREATE TABLE population (pop_country VARCHAR, population_mil BIGINT);
INSERT INTO capitals VALUES
    ('Russia', 'Moscow'),
    ('Italy', 'Rome'),
    ('Spain', 'Madrid'),
```

```
    ('France', 'Paris');
INSERT INTO population VALUES
    ('Russia', 143),
    ('Spain', 48),
    ('Brazil', 211);
```

Выведем страны с их столицей и населением, используя `JOIN` с оператором `ON`:

```
SELECT *
FROM capitals
JOIN population ON (cap_country = pop_country);
```

```
+-------------+---------+-------------+----------------+
| cap_country | capital | pop_country | population_mil |
+-------------+---------+-------------+----------------+
| Russia      | Moscow  | Russia      | 143            |
+-------------+---------+-------------+----------------+
| Spain       | Madrid  | Spain       | 48             |
+-------------+---------+-------------+----------------+
```

Теперь выведем те же данные, используя `JOIN` с оператором `WHERE`:

```
SELECT t1.*, t2.*
FROM capitals t1, population t2
WHERE t1.cap_country = t2.pop_country
```

```
+-------------+---------+-------------+----------------+
| cap_country | capital | pop_country | population_mil |
+-------------+---------+-------------+----------------+
| Russia      | Moscow  | Russia      | 143            |
+-------------+---------+-------------+----------------+
| Spain       | Madrid  | Spain       | 48             |
+-------------+---------+-------------+----------------+
```

С оператором `WHERE` можно использовать не только равенство, но и другие предикаты.

Если имена столбцов в двух таблицах одинаковы и значения в них должны быть равны, то можно использовать более простой синтаксис `USING`.

Зададим те же таблицы, но с одноименными столбцами `country`:

```
CREATE TABLE capitals (country VARCHAR, capital VARCHAR);
CREATE TABLE population (country VARCHAR, population_mil BIGINT);
INSERT INTO capitals VALUES
    ('Russia', 'Moscow'),
    ('Italy', 'Rome'),
    ('Spain', 'Madrid'),
```

```
    ('France', 'Paris');
INSERT INTO population VALUES
    ('Russia', 143),
    ('Spain', 48),
    ('Brazil', 211);
```

Теперь выведем страны с их столицей и населением через более простой запрос с `USING`:

```
SELECT *
FROM capitals
JOIN population USING (country);
```

```
+---------+---------+----------------+
| country | capital | population_mil |
+---------+---------+----------------+
| Russia  | Moscow  | 143            |
+---------+---------+----------------+
| Spain   | Madrid  | 48             |
+---------+---------+----------------+
```

`NATURAL JOIN` — естественное соединение

Естественные соединения объединяют две таблицы на основе столбцов с одинаковыми именами.

Покажем использование такого типа соединения на примере той же пары таблиц с одноименными столбцами `country`.

Чтобы объединить таблицы по их общему столбцу, выполним команду:

```
SELECT *
FROM capitals
NATURAL JOIN population;
```

```
+---------+---------+---------+----------------+
| country | capital | country | population_mil |
+---------+---------+---------+----------------+
| Russia  | Moscow  | Russia  | 143            |
+---------+---------+---------+----------------+
| Spain   | Madrid  | Spain   | 48             |
+---------+---------+---------+----------------+
```

Обратите внимание, что в результат были включены только строки, в которых в обеих таблицах присутствовал один и тот же атрибут `country`.

Мы можем выполнить аналогичный запрос с помощью выражения `JOIN` с ключевым словом `USING`:

```
SELECT *
FROM capitals
JOIN population
USING (country);
```

```
+---------+---------+----------------+
| country | capital | population_mil |
+---------+---------+----------------+
| Russia  | Moscow  | 143            |
+---------+---------+----------------+
| Spain   | Madrid  | 48             |
+---------+---------+----------------+
```

Обратите внимание, что в таком случае одноименные колонки из разных таблиц "схлопнутся" в одну.

`SEMI JOIN` и `ANTI JOIN` — полусоединение и антисоединение

Полусоединения возвращают строки из левой таблицы, которые имеют хотя бы одно совпадение в правой таблице.

Антисоединения возвращают строки из левой таблицы, которые не имеют ни одного совпадения в правой таблице.

При использовании полу- или антисоединений результат никогда не будет содержать больше строк, чем в левой таблице. Полусоединения обеспечивают ту же логику, что и оператор `IN`. Антисоединения обеспечивают ту же логику, что и оператор `NOT IN`, за исключением того, что антисоединения игнорируют значения `NULL` из правой таблицы.

Пример полусоединения

Выведем список таких пар "страна-столица" из таблицы `capitals`, для которых название страны присутствует в таблице `population`:

```
SELECT *
FROM capitals
SEMI JOIN population
    USING (country);
```

```
+---------+---------+
| country | capital |
+---------+---------+
| Russia  | Moscow  |
+---------+---------+
```

```
| Spain   | Madrid  |
+---------+---------+
```

Этот запрос эквивалентен следующему:

```
SELECT *
FROM capitals
WHERE country IN (SELECT country FROM population);
```

```
+---------+---------+
| country | capital |
+---------+---------+
| Russia  | Moscow  |
+---------+---------+
| Spain   | Madrid  |
+---------+---------+
```

Пример антисоединения

Выведем список таких пар "страна-столица" из таблицы `capitals`, для которых названия страны нет в таблице `population`:

```
SELECT *
FROM capitals
ANTI JOIN population
    USING (country);
```

```
+---------+---------+
| country | capital |
+---------+---------+
| Italy   | Rome    |
+---------+---------+
| France  | Paris   |
+---------+---------+
```

Этот запрос эквивалентен следующему:

```
SELECT *
FROM capitals
WHERE country NOT IN
    (SELECT country FROM population WHERE country IS NOT NULL);
```

```
+---------+---------+
| country | capital |
```

```
+---------+---------+
| Italy   | Rome    |
+---------+---------+
| France  | Paris   |
+---------+---------+
```

Замкнутое соединение (Self-Join)

Tengri позволяет использовать замкнутое соединение (соединение таблицы с самой собой) для всех типов соединений. Обратите внимание, что таблицы должны быть указаны через псевдонимы, использование одного и того же имени таблицы без псевдонимов приведет к ошибке:

```
CREATE TABLE t (x INTEGER);
SELECT * FROM t JOIN t USING(x);
```

```
Binder Error:
Duplicate alias "t" in query!
```

Добавление псевдонимов позволяет успешно обработать запрос:

```
CREATE TABLE t (num INTEGER);
SELECT * FROM t t1 JOIN t t2 USING(num);
```

```
+-----+
| num |
+-----+
```

`UNION`

Выражение `UNION` добавляет результаты одного запроса к результатам другого. При этом дублирующиеся строки убираются из результата, если только не добавлен оператор `ALL`.

Синтаксис

```
SELECT ...
UNION [ALL]
SELECT ...
[UNION [ALL]
SELECT ...
...
];
```

Примеры

- Выведем список всех стран, которые встречаются в столбцах `country` в двух таблицах — столиц и количества населения:

```
CREATE TABLE capitals (country VARCHAR, capital VARCHAR);
CREATE TABLE population (country VARCHAR, population_mil BIGINT);
INSERT INTO capitals VALUES
    ('Russia', 'Moscow'),
    ('Italy', 'Rome'),
    ('Spain', 'Madrid'),
    ('France', 'Paris');
INSERT INTO population VALUES
    ('Russia', 143),
    ('Spain', 48),
    ('Brazil', 211);

SELECT country FROM capitals
UNION
SELECT country FROM population;
```

```
+---------+
| country |
+---------+
| Russia  |
+---------+
| Italy   |
+---------+
| France  |
+---------+
| Spain   |
+---------+
| Brazil  |
+---------+
```

- Теперь для тех же двух таблиц выведем список стран, которые встречаются в столбцах `country`, но без сокращения повторяющихся вхождений. Для этого используем оператор `ALL`:

```
SELECT country FROM capitals
UNION ALL
SELECT country FROM population;
```

```
+---------+
| country |
+---------+
| Russia  |
+---------+
```

```
| Italy   |
+---------+
| Spain   |
+---------+
| France  |
+---------+
| Russia  |
+---------+
| Spain   |
+---------+
| Brazil  |
+---------+
```

WHERE

Выражение `WHERE` определяет фильтры, которые будут применены к данным. Это позволяет выбрать только ту часть данных, которая вас интересует. Логически условие `WHERE` применяется сразу после условия `FROM`.

Синтаксис

```
...
WHERE <predicate>
[ ... ]
```

Параметры

- `<predicate>`
  Булево выражение. Выражение может содержать логические операторы, такие как `AND`, `OR` и `NOT`.

Примеры

- Выберем из таблицы столиц строки, в которых значение `country` — `'Italy'`:

```
SELECT *
FROM capitals
WHERE country = 'Italy';
```

```
+---------+---------+
| country | capital |
+---------+---------+
| Italy   | Rome    |
+---------+---------+
```

- Выберем из таблицы дней недели строки, которые соответствуют заданному выражению `LIKE`,

чувствительному к регистру:

```
SELECT *
FROM weekdays
WHERE name LIKE '%S%';
```

```
+--------+----------+
| number |   name   |
+--------+----------+
| 6      | Saturday |
+--------+----------+
| 7      | Sunday   |
+--------+----------+
```

- Выберем из таблицы дней недели строки, которые соответствуют заданному выражению `ILIKE`, не чувствительному к регистру:

```
SELECT *
FROM weekdays
WHERE name ILIKE '%S%';
```

```
+--------+-----------+
| number |    name   |
+--------+-----------+
| 2      | Tuesday   |
+--------+-----------+
| 3      | Wednesday |
+--------+-----------+
| 4      | Thursday  |
+--------+-----------+
| 6      | Saturday  |
+--------+-----------+
| 7      | Sunday    |
+--------+-----------+
```

- Выберем все строки, соответствующие заданному составному выражению:

```
SELECT *
FROM weekdays
WHERE number > 5 OR number = 3;
```

```
+--------+-----------+
| number |    name   |
+--------+-----------+
| 3      | Wednesday |
+--------+-----------+
```

```
| 6      | Saturday  |
+--------+-----------+
| 7      | Sunday    |
+--------+-----------+
```

GROUP BY

Выражение `GROUP BY` определяет, какие столбцы должны использоваться для группировки при выполнении любых агрегаций в выражении `SELECT`. Если указано условие `GROUP BY`, запрос всегда является агрегированным, даже если в условии `SELECT` нет явно заданного агрегирования.

Если указано условие `GROUP BY`, все кортежи, имеющие совпадающие данные в группирующих столбцах (т. е. все кортежи, принадлежащие одной группе), будут объединены. Значения самих группирующих столбцов не изменяются, а любые другие столбцы могут быть объединены с помощью агрегатной функции (например, `count`, `sum`, `avg`, и т. д.).

GROUP BY ALL

Используйте `GROUP BY ALL` для группировки всех столбцов в выражении `SELECT`, которые не обернуты в агрегатные функции. Это упрощает синтаксис, позволяя хранить список столбцов в одном месте, и предотвращает ошибки, сохраняя гранулярность `SELECT` в соответствии с гранулярностью `GROUP BY` (например, предотвращает дублирование).

Синтаксис

Могут использоваться следующие варианты синтаксиса:

```
SELECT ...
  FROM ...
  [ ... ]
  GROUP BY groupItem [ , groupItem [ , ... ] ]
  [ ... ]
```

```
SELECT ...
  FROM ...
  [ ... ]
  GROUP BY ALL
  [ ... ]
```

где:

```
groupItem ::= { <column_alias> | <position> | <expr> }
```

Параметры

- `<column_alias>`
  Псевдоним столбца, заданный в выражении `SELECT`.

- `<position>`
  Позиция выражения в выражении `SELECT`.

- `<expr>`
  Любое выражение, заданное на таблицах в текущей области видимости.

- `GROUP BY ALL`
  Указывает, что все элементы в выражении `SELECT`, которые не используют агрегатные функции, должны использоваться для группировки.

Примеры

- Вычислим количество записей в таблице `employees` (*работников*), относящихся к каждому отделу:

```
SELECT department, count(*)
    FROM employees
    GROUP BY department;
```

- Вычислим среднюю зарплату для каждого отдела каждого подразделения:

```
SELECT division, department, avg(salary)
    FROM employees
    GROUP BY division, department;
```

- Сгруппируем данные по отделам и подразделениям, чтобы увидеть все уникальные пары "отдел — подразделение":

```
SELECT division, department
    FROM employees
    GROUP BY ALL;
```

HAVING

Выражение `HAVING` может использоваться после выражения `GROUP BY` для задания критериев фильтрации после завершения группировки. По синтаксису выражение `HAVING` идентично выражению `WHERE`, но если выражение `WHERE` используется до группировки, то выражение `HAVING` —

после нее.

Синтаксис

```
SELECT ...
FROM ...
GROUP BY ...
HAVING <predicate>
[ ... ]
```

Параметры

- `<predicate>`
  Булево выражение.

Примеры

- Подсчитаем количество записей в таблице `employees` (*работников*), которые содержат каждый из различных отделов, отбрасывая отделы с количеством меньше 10:

  ```
  SELECT department, count(*)
  FROM employees
  GROUP BY department
  HAVING count(*) >= 10;
  ```

- Вычислим среднюю зарплату для каждого отдела каждого подразделения , но только для тех отделов, где средняя зарплата больше, чем медианная зарплата:

  ```
  SELECT division, department, avg(salary)
  FROM employees
  GROUP BY division, department
  HAVING avg(salary) > median(salary);
  ```

`QUALIFY`

Выражение `QUALIFY` используется для фильтрации результатов оконных функций. Эта фильтрация результатов аналогична тому, как выражение `HAVING` фильтрует результаты агрегатных функций, применяемых в запросах с выражением `GROUP BY`.

Выражение `QUALIFY` позволяет избежать необходимости использования подзапроса или выражения с `WITH` для выполнения этой фильтрации (так же, как `HAVING` позволяет избегать подзапросов).

Синтаксис

```
QUALIFY <predicate>
```

Общая форма выражений с `QUALIFY` обычно выглядит так (допускаются некоторые вариации в порядке):

```
SELECT <column_list>
  FROM <data_source>
  [GROUP BY ...]
  [HAVING ...]
  QUALIFY <predicate>
  [ ... ]
```

Параметры

- `<column_list>`
  Список выражения `SELECT`.

- `<data_source>`
  Источником данных обычно является таблица, но это может быть и другой источник данных, подобный таблице, например представление, пользовательская табличная функция и т.д.

- `<predicate>`
  Предикат — это выражение, которое фильтрует результат после вычисления агрегатных и оконных функций. Предикат подобен выражению `HAVING`, но без самого ключевого слова `HAVING`. Кроме того, предикат может содержать оконные функции.

Примеры

Создадим и заполним таблицу:

```
CREATE TABLE qt (i INTEGER, p CHAR(1), o INTEGER);
INSERT INTO qt (i, p, o) VALUES
    (1, 'A', 1),
    (2, 'A', 2),
    (3, 'B', 1),
    (4, 'B', 2);
```

В этом запросе используется вложенная структура, а не `QUALIFY`:

```
SELECT *
```

```
    FROM (
         SELECT i, p, o,
                ROW_NUMBER() OVER (PARTITION BY p ORDER BY o) AS row_num
            FROM qt
        )
    WHERE row_num = 1;
```

```
+---+---+---+---------+
| I | P | O | ROW_NUM |
|---+---+---+---------|
| 1 | A | 1 |       1 |
| 3 | B | 1 |       1 |
+---+---+---+---------+
```

А в этом запросе используется `QUALIFY`:

```
SELECT i, p, o
    FROM qt
    QUALIFY ROW_NUMBER() OVER (PARTITION BY p ORDER BY o) = 1;
```

```
+---+---+---+
| I | P | O |
|---+---+---|
| 1 | A | 1 |
| 3 | B | 1 |
+---+---+---+
```

Используем `QUALIFY` для ссылки на оконные функции, которые находятся в выражении `SELECT`:

```
SELECT i, p, o, ROW_NUMBER() OVER (PARTITION BY p ORDER BY o) AS row_num
    FROM qt
    QUALIFY row_num = 1;
```

```
+---+---+---+---------+
| I | P | O | ROW_NUM |
|---+---+---+---------|
| 1 | A | 1 |       1 |
| 3 | B | 1 |       1 |
+---+---+---+---------+
```

Выражение `QUALIFY` также может сочетаться с агрегатными функциями и может содержать подзапросы:

```
SELECT c2, SUM(c3) OVER (PARTITION BY c2) AS r
    FROM t1
```

```
    WHERE c3 < 4
    GROUP BY c2, c3
    HAVING sum(c1) > 3
    QUALIFY r IN (
        SELECT min(c1)
        FROM test
        GROUP BY c2
        HAVING min(c1) > 3);
```

ORDER BY

Выражение `ORDER BY` является модификатором вывода. Логически оно применяется в самом конце запроса (непосредственно перед `LIMIT`, если оно присутствует). Выражение `ORDER BY` сортирует строки по критериям сортировки в порядке возрастания или убывания. Кроме того, в каждом выражении `ORDER BY` можно указать, следует ли перемещать значения `NULL` в начало или в конец.

Выражение `ORDER BY` может содержать одно или несколько выражений, разделенных запятыми. Если выражений нет, то будет выдана ошибка. Выражения могут начинаться либо с произвольного скалярного выражения (которое может быть именем столбца), либо с номера позиции столбца (где индексация начинается с `1`), либо с ключевого слова `ALL`. За каждым выражением может следовать модификатор порядка (`ASC` или `DESC`, по умолчанию — `ASC`), и/или модификатор порядка `NULL` (`NULLS FIRST` или `NULLS LAST`, по умолчанию — `NULLS LAST`).

ORDER BY ALL

Ключевое слово `ALL` указывает, что вывод должен быть отсортирован по каждому столбцу в порядке слева направо. Направление сортировки может быть изменено с помощью `ORDER BY ALL ASC` или `ORDER BY ALL DESC` и/или `NULLS FIRST` или `NULLS LAST`. Обратите внимание, что `ALL` не может использоваться в сочетании с другими выражениями в выражении `ORDER BY` — оно должно быть само по себе.

Модификатор порядка значений NULL

По умолчанию сортировка производится с параметрами `ASC` и `NULLS LAST`, то есть значения сортируются в порядке возрастания, а значения `NULL` располагаются последними. Это идентично порядку сортировки по умолчанию в PostgreSQL. Порядок сортировки по умолчанию можно изменить с помощью следующих параметров конфигурации.

Используйте параметр `default_null_order`, чтобы изменить порядок сортировки по умолчанию `NULL` на один из следующих вариантов:

- `NULLS_FIRST`
- `NULLS_LAST`
- `NULLS_FIRST_ON_ASC_LAST_ON_DESC`
- `NULLS_LAST_ON_ASC_FIRST_ON_DESC`:

Например:

```
SET default_null_order = 'NULLS_FIRST';
```

Используйте параметр `default_order`, чтобы изменить направление сортировки по умолчанию на один из следующих вариантов:

- `DESC`
- `ASC`

Например:

```
SET default_order = 'DESC';
```

Коллации (схемы сопоставления)

Текст по умолчанию сортируется с использованием коллации двоичного сравнения. Это означает, что значения сортируются по их двоичным представлениям в UTF-8. Хотя это хорошо работает для ASCII-текста (например, для данных на английском языке), порядок сортировки может быть неправильным для других языков. Для этой цели предусмотрены коллации.

Синтаксис

```
SELECT ...
FROM ...
ORDER BY orderItem [ , orderItem , ... ]
[ ... ]
```

где:

```
orderItem ::= { <column_alias> | <position> | <expr> }
    [ { ASC | DESC } ] [ NULLS { FIRST | LAST } ]
```

Параметры

- `<column_alias>`
  Псевдоним столбца, заданный в списке `SELECT`.

- `<position>`
  Позиция выражения в списке `SELECT`.

- `<expr>`
  Любое выражение, заданное на таблицах в текущей области видимости.

- `{ ASC | DESC }`
  Опционально возвращает значения ключа сортировки в порядке возрастания (от наименьшего к наибольшему) или убывания (от наибольшего к наименьшему).
  По умолчанию: `ASC`

- `NULLS { FIRST | LAST }`
  Опционально указывает, возвращаются ли значения `NULL` до/после значений, отличных от `NULL`, в зависимости от порядка сортировки (`ASC` или `DESC`).
  По умолчанию: зависит от порядка сортировки (`ASC` или `DESC`), см. Модификатор порядка значений `NULL`.

Примеры

- Выведем дни недели, упорядоченные по их названию, используя порядок сортировки по умолчанию и стандартный порядок для значений `NULL`:

```
SELECT *
FROM weekdays
ORDER BY name;
```

```
+--------+-----------+
| number |    name   |
+--------+-----------+
| 5      | Friday    |
+--------+-----------+
| 1      | Monday    |
+--------+-----------+
| 6      | Saturday  |
+--------+-----------+
| 7      | Sunday    |
+--------+-----------+
| 4      | Thursday  |
+--------+-----------+
| 2      | Tuesday   |
+--------+-----------+
| 3      | Wednesday |
+--------+-----------+
| 8      | null      |
+--------+-----------+
```

- Выведем дни недели, упорядоченные по их названию в порядке убывания со значениями `NULL` в начале:

```
SELECT *
FROM weekdays
```

```
ORDER BY name DESC NULLS FIRST;
```

```
+--------+-----------+
| number |    name   |
+--------+-----------+
| 8      | null      |
+--------+-----------+
| 3      | Wednesday |
+--------+-----------+
| 2      | Tuesday   |
+--------+-----------+
| 4      | Thursday  |
+--------+-----------+
| 7      | Sunday    |
+--------+-----------+
| 6      | Saturday  |
+--------+-----------+
| 1      | Monday    |
+--------+-----------+
| 5      | Friday    |
+--------+-----------+
```

- Теперь рассмотрим ситуацию, при которой в нашей таблице дни недели пронумерованы начиная с воскресенья (как делается в некоторых календарных системах). Чтобы привести ее к привычному виду, упорядочим дни недели сначала по типу (`weekend` или нет), а затем по номеру:

```
SELECT *
FROM weekdays
ORDER BY weekend, number;
```

```
+--------+-----------+---------+
| number |    name   | weekend |
+--------+-----------+---------+
| 2      | Monday    | false   |
+--------+-----------+---------+
| 3      | Tuesday   | false   |
+--------+-----------+---------+
| 4      | Wednesday | false   |
+--------+-----------+---------+
| 5      | Thursday  | false   |
+--------+-----------+---------+
| 6      | Friday    | false   |
+--------+-----------+---------+
| 1      | Sunday    | true    |
+--------+-----------+---------+
| 7      | Saturday  | true    |
+--------+-----------+---------+
```

- Покажем разницу в сортировке с использованием разных коллаций. Возьмем названия двух финских городов в их шведском и финском вариантах и упорядочим их по их шведским названиям, используя сначала правила коллации для английского:

```
SELECT *
FROM finnish_cities
ORDER BY swed_name COLLATE EN
```

```
+-------------+-----------+
|  swed_name  |  fin_name |
+-------------+-----------+
| Åbo         | Turku     |
+-------------+-----------+
| Helsingfors | Helsinki  |
+-------------+-----------+
```

Теперь сделаем то же самое, но используем правила коллации для шведского:

```
SELECT *
FROM finnish_cities
ORDER BY swed_name COLLATE SV
```

```
+-------------+-----------+
|  swed_name  |  fin_name |
+-------------+-----------+
| Helsingfors | Helsinki  |
+-------------+-----------+
| Åbo         | Turku     |
+-------------+-----------+
```

В шведском алфавите буква `å` идет в конце, а в английских правилах коллации она идет в начале. Отсюда мы получаем разницу в порядке сортировки.

`LIMIT`

Выражение `LIMIT`

Выражение `LIMIT` является модификатором вывода. Логически оно применяется в самом конце запроса. Выражение `LIMIT` ограничивает количество выводимых строк.

Обратите внимание, что хотя `LIMIT` можно использовать без условия `ORDER BY`, в таком случае результаты могут быть не детерминированными. Тем не менее, это может быть полезно, например, когда вы хотите получить быстрый срез данных.

Выражение `OFFSET`

Выражение `OFFSET` указывает, с какой позиции начинать считывание значений, т. е. первые `OFFSET` значений игнорируются.

Синтаксис

```
SELECT ...
FROM ...
[ ORDER BY ... ]
LIMIT <count> [ OFFSET <start> ]
[ ... ]
```

Параметры

- `<count>`
  Количество возвращаемых строк. Должно быть неотрицательной целой величиной.
  Значение `NULL` также принимается и рассматривается как неограниченное.

- `OFFSET <start>`
  Номер строки, после которой возвращаются ограниченные/извлеченные строки. Должен быть неотрицательной целой величиной.
  Если `OFFSET` опущено, вывод начинается с первой строки в наборе результатов.
  Значение `NULL` также принимается и рассматривается как неограниченное (т. е. строки пропускаться не будут).

Примеры

- Выберем первые 5 дней из таблицы дней недели:

  ```
  SELECT *
  FROM weekdays
  LIMIT 5;
  ```

  ```
  +--------+-----------+
  | number |    name   |
  +--------+-----------+
  | 1      | Monday    |
  +--------+-----------+
  | 2      | Tuesday   |
  +--------+-----------+
  | 3      | Wednesday |
  +--------+-----------+
  | 4      | Thursday  |
  ```

```
+--------+-----------+
| 5      | Friday    |
+--------+-----------+
```

- Выберем 5 строк из таблицы дней недели, начиная с позиции 1 (т. е. игнорируя первую строку):

```
SELECT *
FROM weekdays
LIMIT 5
OFFSET 1;
```

```
+--------+-----------+
| number |    name   |
+--------+-----------+
| 2      | Tuesday   |
+--------+-----------+
| 3      | Wednesday |
+--------+-----------+
| 4      | Thursday  |
+--------+-----------+
| 5      | Friday    |
+--------+-----------+
| 6      | Saturday  |
+--------+-----------+
```

`LIKE`

Описание

Выражение с оператором `LIKE` возвращает `TRUE`, если текстовая строка соответствует заданному шаблону.

Если шаблон не содержит знаков процента или подчеркивания, то он интерпретируется буквально, и в этом случае `LIKE` действует как оператор равенства:

```
SELECT
    'Tengri' LIKE 'Tengri' AS result;
```

```
+--------+
| result |
+--------+
| true   |
+--------+
```

Если в шаблоне есть специальные символы, то он интерпретируется не буквально, а как

регулярное выражение:

- Подчеркивание `_` в шаблоне соответствует любому отдельному символу.
- Знак процента `%` в шаблоне соответствует любой последовательности из нуля или более символов.

Сопоставление шаблона `LIKE` всегда применяется ко всей строке. Поэтому, если требуется сопоставить последовательность в любом месте строки (накрыть шаблоном подстроку), то шаблон должен начинаться и заканчиваться знаком процента `%`.

Можно также использовать противоположные по значению выражения `<string> NOT LIKE <pattern>` и `NOT string LIKE <pattern>`:

```
SELECT
    NOT 'Tengri' LIKE 'Tengri' AS result_1,
    'Tengri' NOT LIKE 'Tengri' AS result_2;
```

```
+----------+----------+
| result_1 | result_2 |
+----------+----------+
| false    | false    |
+----------+----------+
```

Оператор `ILIKE`

Оператор `ILIKE` можно использовать вместо `LIKE`, чтобы сделать сопоставление нечувствительным к регистру:

```
SELECT
    'Tengri' ILIKE 'tengri' AS result_1,
    'Tengri' ILIKE '%NGRi'  AS result_2;
```

```
+----------+----------+
| result_1 | result_2 |
+----------+----------+
| true     | true     |
+----------+----------+
```

Примеры

- Покажем несколько примеров работы шаблонов `LIKE`:

```
SELECT
    'Tengri' LIKE 'TNGRi'  AS result_1,
    'Tengri' LIKE 'T%'     AS result_2,
```

```
    'TNGRi'  LIKE 'T%'     AS result_3,
    'Tengri' LIKE 'T_____' AS result_4;
```

```
+----------+----------+----------+----------+
| result_1 | result_2 | result_3 | result_4 |
+----------+----------+----------+----------+
| false    | true     | true     | true     |
+----------+----------+----------+----------+
```

- Выберем из таблицы столиц страны, в которых столица начинается на `M`:

```
CREATE TABLE capitals (country VARCHAR, capital VARCHAR);
INSERT INTO capitals VALUES
    ('Russia', 'Moscow'),
    ('Italy', 'Rome'),
    ('Spain', 'Madrid'),
    ('France', 'Paris');

SELECT country FROM capitals WHERE capital LIKE 'M%';
```

```
+---------+
| country |
+---------+
| Russia  |
+---------+
| Spain   |
+---------+
```

Полезные ссылки

- `SIMILAR TO`
- Текстовые функции
- Функции для регулярных выражений

`SIMILAR TO`

Описание

Выражение с оператором `SIMILAR TO` возвращает `TRUE` или `FALSE` в зависимости от того, накладывается ли его шаблон на данную текстовую строку.

Этот оператор похож на оператор `LIKE`, но в данном случае шаблон интерпретируется как регулярное выражение. Как и в случае с `LIKE`, выражение с `SIMILAR TO` проверяется успешно, только если шаблон накладывается на всю строку. Это отличается от обычного поведения регулярных

выражений, где шаблон может накладываться на любую часть строки.

В регулярных выражениях используется синтаксис RE2.

Можно также использовать противоположные по значению выражения `<string> NOT SIMILAR TO <pattern>` и `NOT string SIMILAR TO <pattern>`

Выражения с оператором `SIMILAR TO` полностью синонимичны выражениям с оператором `~`.

Примеры

```
SELECT
    'Tengri' SIMILAR TO 'TNGRi'  AS result_1, -- false expected
    'Tengri' SIMILAR TO 'T.*'    AS result_2,
    'Tengri' SIMILAR TO 'T.+i'   AS result_3,
    'TNGRi'  SIMILAR TO 'T.+i'   AS result_4,
    'T.+i'   SIMILAR TO 'T\.\+i' AS result_5;
```

```
+----------+----------+----------+----------+----------+
| result_1 | result_2 | result_3 | result_4 | result_5 |
+----------+----------+----------+----------+----------+
| false    | true     | true     | true     | true     |
+----------+----------+----------+----------+----------+
```

```
SELECT
    'Tengri' NOT SIMILAR TO 'TNGRi'  AS result_1,
    NOT 'T.{3}i' SIMILAR TO 'T.{3}i' AS result_2;
```

```
+----------+----------+
| result_1 | result_2 |
+----------+----------+
| true     | true     |
+----------+----------+
```

Полезные ссылки

- Оператор `~`
- `LIKE`
- Текстовые функции
- Функции для регулярных выражений

AS

Оператор `AS` в SQL-запросах может использоваться в нескольких разных функциях.

Задание имен для столбцов в запросе

Оператор `AS` можно использовать, чтобы задать имена (псевдонимы) для столбцов в таблице, которая будет выдана в качестве результата запроса или использована в самом запросе:

```
SELECT
    name AS Student
    FROM students
```

Если задаваемое имя содержит пробел, то его нужно заключить в кавычки:

```
SELECT
    name AS "Student Name"
    FROM students
```

Задание имен для таблиц в запросе

Оператор `AS` можно использовать, чтобы задать имена (псевдонимы) для таблиц, которые используются в запросе:

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
    FROM
        Customers AS c,
        Orders AS o
    WHERE
        c.CustomerName="Tengri" AND c.CustomerID=o.CustomerID;
```

Здесь в запросе для краткости использованы псевдонимы `c` и `o` для таблиц `Customers` и `Orders`, к которым делается запрос.

Задание запроса при создании таблицы

При создании таблицы оператор `AS` можно использовать для того, чтобы задать запрос к данным, результат которого будет записан в создаваемую таблицу:

```
CREATE TABLE films_recent AS
    SELECT * FROM films WHERE date_prod >= '2025-01-01';
```

В этом случае можно также использовать синтаксис FROM-first и отбросить `SELECT *`. Тогда запрос (эквивалентный предыдущему) будет выглядеть так:

```
CREATE TABLE films_recent AS
    FROM films WHERE date_prod >= '2025-01-01';
```

Функции

Агрегатные функции

- `array_agg`
- `avg`
- `count`
- `count(argument)`
- `count_if`
- `max`
- `min`
- `median`
- `sum`

Числовые функции

- `abs`
- `add`
- `ceil`
- `cos`
- `divide`
- `even`
- `exp`
- `floor`
- `fmod`
- `gcd`
- `greatest`
- `isfinite`
- `isinf`
- `isnan`
- `lcm`
- `least`
- `lgamma`

- `ln`
- `log`
- `log2`
- `multiply`
- `nextafter`
- `pi`
- `pow`
- `radians`
- `random`
- `round_even`
- `round`
- `setseed`
- `sign`
- `signbit`
- `sin`
- `sqrt`
- `subtract`
- `trunc`
- Оператор `+`
- Оператор `-`
- Оператор `*`
- Оператор `/`
- Оператор `%`
- Оператор `^`

Текстовые функции

- `contains`
- `length`
- `strlen`
- `trim`
- `ltrim`
- `rtrim`
- `lower`
- `upper`
- `split`

- `chr`

Функции для регулярных выражений

- `regexp_extract`
- `regexp_extract_all`
- `regexp_full_match`
- `regexp_matches`
- `regexp_replace`
- `regexp_split_to_array`
- `regexp_split_to_table`
- Оператор `~`

Функции для даты и времени

- `current_time`
- `datepart`
- `date_diff`

Функции для JSON

- `json_array_length`
- `json_contains`
- `json_exists`
- `json_extract`
- `json_extract_string`
- `json_group_array`
- `json_group_object`
- `json_keys`
- `json_transform`
- `json_transform_strict`
- `json_type`
- `json_valid`
- `json_value`
- `json`
- `read_json`

Функции для двоичных данных

- `concat`
- `md5`
- `sha1`
- `sha256`
- Оператор `||`

Оконные функции

- `cume_dist`
- `dense_rank`
- `first_value`
- `lag`
- `last_value`
- `lead`
- `nth_value`
- `ntile`
- `percent_rank`
- `rank`
- `row_number`

Утилиты

- `coalesce`
- `hash`
- `unnest`

Агрегатные функции

Агрегатные функции — это функции, которые объединяют значения из нескольких строк в одно.

Агрегатные функции отличаются от скалярных функций и оконных функций тем, что они изменяют кардинальность результата. Поэтому в запросах их можно использовать только в выражениях `SELECT` и `HAVING`.

Выражение `DISTINCT` в агрегатных функциях

Когда используется выражение `DISTINCT`, при вычислении значения агрегатной функции учитываются только уникальные значения. Часто это выражение используется в сочетании с агрегатной функцией `count()` для получения количества уникальных элементов, но может использоваться и с другими агрегатными функциями.

Пример

```
CREATE TABLE cities(city_name VARCHAR);

INSERT INTO cities VALUES
    ('Moscow'),
    ('Moscow'),
    ('Paris'),
    ('Madrid');

SELECT
    count(DISTINCT city_name) AS distinct_cities_num,
    count(city_name) AS cities_num
FROM cities;
```

```
+---------------------+------------+
| distinct_cities_num | cities_num |
+---------------------+------------+
| 3                   | 4          |
+---------------------+------------+
```

Некоторые агрегатные функции нечувствительны к повторяющимся значениям (например, `min()` и `max()`), и для них выражение `DISTINCT` игнорируется.

`array_agg()`

Описание	Возвращает список, содержащий все значения столбца.
Использование	`array_agg(argument)`
Псевдонимы	`list()`

На работу этой функции влияет порядок сортировки в таблице.

▼ *Посмотреть пример*

```
CREATE TABLE numbers(number BIGINT);
```

```
INSERT INTO numbers VALUES
    (1),
    (2),
    (3),
    (NULL);

SELECT
    array_agg(number) AS array_agg,
    list(number) AS list
FROM numbers;
```

```
+--------------+--------------+
|   array_agg  |     list     |
+--------------+--------------+
| {1,2,3,None} | {1,2,3,None} |
+--------------+--------------+
```

avg()

Описание Вычисляет среднее значение всех непустых значений в `argument`.

Использование `avg(argument)`

Псевдонимы `mean()`

▼ *Посмотреть пример*

```
CREATE TABLE numbers(number BIGINT);

INSERT INTO numbers VALUES
    (1),
    (2),
    (3),
    (NULL);

SELECT
    avg(number) AS average,
    mean(number) AS mean
FROM numbers;
```

```
+---------+------+
| average | mean |
+---------+------+
| 2       | 2    |
+---------+------+
```

`count()`

Описание Вычисляет количество строк в группе.

Использование `count()`

Псевдонимы `count(*)`

▼ *Посмотреть пример*

```
CREATE TABLE numbers(number BIGINT);

INSERT INTO numbers VALUES
    (1),
    (2),
    (3),
    (NULL);

SELECT
    count() AS rows_count,
    count(*) AS rows_count_star
FROM numbers;
```

```
+------------+-----------------+
| rows_count | rows_count_star |
+------------+-----------------+
| 4          | 4               |
+------------+-----------------+
```

`count(argument)`

Описание Вычисляет количество непустых значений в `argument`.

Использование `count(argument)`

▼ *Посмотреть пример*

```
CREATE TABLE numbers(number BIGINT);

INSERT INTO numbers VALUES
    (1),
    (2),
    (3),
    (NULL);

SELECT
    count() AS rows_count,
    count(number) AS values_count
FROM numbers;
```

```
+------------+--------------+
| rows_count | values_count |
+------------+--------------+
| 4          | 3            |
+------------+--------------+
```

count_if()

Описание Возвращает количество записей, удовлетворяющих условию, или `NULL`, если ни одна запись не удовлетворяет условию.

Использование `count_if(<condition>)`

▼ *Посмотреть пример*

```
CREATE TABLE text_table(text_data VARCHAR);

INSERT INTO text_table VALUES
('Tengri'),
('Tengri'),
('TNGRi'),
(NULL);

SELECT
    COUNT_IF(TRUE) AS row_number,
    COUNT_IF(text_data = 'Tengri') AS tengri_number
FROM text_table;
```

```
+------------+---------------+
| row_number | tengri_number |
+------------+---------------+
| 4          | 2             |
+------------+---------------+
```

max()

Описание Возвращает максимальное значение, имеющееся в `argument`.

Использование `max(argument)`

▼ *Посмотреть пример*

```
CREATE TABLE numbers(number BIGINT);

INSERT INTO numbers VALUES
    (1),
    (2),
    (3),
    (NULL);
```

```
SELECT
    max(number) AS max,
    min(number) AS min
FROM numbers;
```

```
+-----+-----+
| max | min |
+-----+-----+
| 3   | 1   |
+-----+-----+
```

`min()`

Описание Возвращает минимальное значение, имеющееся в `argument`.

Использование `min(argument)`

▼ *Посмотреть пример*

```
CREATE TABLE numbers(number BIGINT);

INSERT INTO numbers VALUES
    (1),
    (2),
    (3),
    (NULL);

SELECT
    max(number) AS max,
    min(number) AS min
FROM numbers;
```

```
+-----+-----+
| max | min |
+-----+-----+
| 3   | 1   |
+-----+-----+
```

`median()`

Описание Возвращает медианное значение всех непустых значений в `argument`.

Использование `median(argument)`

В случае четного количества значений берется среднее между двумя центральными значениями.

▼ *Посмотреть пример*

Покажем разницу между медианным значением и средним значением (`avg()`) на примерах наборов из четного (`_even`) и нечетного (`_odd`) количества чисел, содержащих в том числе пустые значения.

```
CREATE TABLE numbers(
    number_even BIGINT,
    number_odd BIGINT);

INSERT INTO numbers VALUES
    (1,    1),
    (2,    2),
    (3,    10),
    (10,   NULL),
    (NULL, NULL);

SELECT
    median(number_even) AS median_even,
    avg(number_even) AS avg_even,
    median(number_odd) AS median_odd,
    avg(number_odd) AS avg_odd
FROM numbers;
```

```
+-------------+----------+------------+-------------------+
| median_even | avg_even | median_odd |      avg_odd      |
+-------------+----------+------------+-------------------+
| 2.5         | 4        | 2          | 4.333333333333333 |
+-------------+----------+------------+-------------------+
```

`sum()`

Описание Вычисляет сумму всех непустых значений в `argument`.

Использование `sum(argument)`

В случае булевых значений подсчитывает количество значений `True`.

▼ *Посмотреть пример*

```
CREATE TABLE numbers(
    number BIGINT,
    boolean BOOL);

INSERT INTO numbers VALUES
    (1,    True),
    (2,    False),
    (3,    False),
    (NULL, NULL);

SELECT
    sum(number) AS sum_number,
```

```
    sum(boolean) AS sum_boolean
FROM numbers;
```

```
+------------+-------------+
| sum_number | sum_boolean |
+------------+-------------+
| 6          | 1           |
+------------+-------------+
```

Числовые функции

Числовые функции — это функции для работы с данными числовых типов: `BIGINT`, `NUMERIC` и `DOUBLE`.

`abs()`

Описание Вычисляет модуль числа.

Использование `abs(num)`

▼ *Посмотреть примеры*

```
SELECT
    abs(-1)  AS result_1,
    abs(0)   AS result_2,
    abs(1.1) AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 1        | 0        | 1.1      |
+----------+----------+----------+
```

`add()`

Описание Складывает числа.

Использование `add(num, num)`

См. также Оператор `+`.

▼ *Посмотреть примеры*

```
SELECT
    add(1, 1)      AS result_1,
    add(-1.1, 2.1) AS result_2,
    add(1)         AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 2        | 1.0      | 1        |
+----------+----------+----------+
```

`ceil()`

Описание Округляет число в бОльшую сторону.

Использование `ceil(num)`

Псевдонимы `ceiling()`

▼ *Посмотреть примеры*

```
SELECT
    ceil(0.1)  AS result_1,
    ceil(-0.1) AS result_2,
    ceiling(1) AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 1        | 0        | 1        |
+----------+----------+----------+
```

`cos()`

Описание Вычисляет косинус угла, заданного в радианах.

Использование `cos(num)`

▼ *Посмотреть примеры*

```
SELECT
    cos(0)      AS result_1,
    cos(pi())   AS result_2,
    cos(pi()/3) AS result_3;
```

```
+----------+----------+--------------------+
| result_1 | result_2 | result_3           |
+----------+----------+--------------------+
| 1        | -1       | 0.5000000000000001 |
+----------+----------+--------------------+
```

divide()

Описание Возвращает результат деления в виде целого числа.

Использование `divide(num, num)`

▼ *Посмотреть примеры*

```
SELECT
    divide(7, 2)  AS result_1,
    divide(7, -2) AS result_2,
    divide(7, 0)  AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 3        | -3       | null     |
+----------+----------+----------+
```

even()

Описание Округляет до ближайшего четного числа в сторону от нуля.

Использование `even(num)`

▼ *Посмотреть примеры*

```
SELECT
    even(2.1)  AS result_1,
    even(-2.1) AS result_2,
    even(0)    AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 4        | -4       | 0        |
+----------+----------+----------+
```

exp()

Описание Вычисляет экспоненту числа.

Использование `exp(num)`

Вычисляет экспоненциальное значение числа: `e^x`.

▼ *Посмотреть примеры*

```
SELECT
```

```
    exp(0)  AS result_1,
    exp(1)  AS result_2,
    exp(-1) AS result_3;
```

```
+----------+-------------------+---------------------+
| result_1 | result_2          | result_3            |
+----------+-------------------+---------------------+
| 1        | 2.718281828459045 | 0.36787944117144233 |
+----------+-------------------+---------------------+
```

floor()

Описание Округляет число в меньшую сторону.

Использование `floor(num)`

▼ *Посмотреть примеры*

```
SELECT
    floor(0.9)  AS result_1,
    floor(-0.9) AS result_2,
    floor(1)    AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 0        | -1       | 1        |
+----------+----------+----------+
```

fmod()

Описание Возвращает остаток от деления первого аргумента на второй.

Использование `fmod(num, num)`

▼ *Посмотреть примеры*

```
SELECT
    fmod(3, 2)   AS result_1,
    fmod(3.1, 2) AS result_2,
    fmod(-10, 4) AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 1        | 1.1      | 2        |
+----------+----------+----------+
```

`gcd()`

Описание Вычисляет наибольший общий делитель двух чисел.

Использование `gcd(num, num)`

Псевдонимы `greatest_common_divisor()`

▼ *Посмотреть примеры*

```
SELECT
    gcd(12, 9)  AS result_1,
    gcd(-12, 9) AS result_2,
    gcd(12, 0)  AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 3        | 3        | 12       |
+----------+----------+----------+
```

`greatest()`

Описание Возвращает наибольшее число из указанных в аргументах.

Использование `greatest(num[, num, ...])`

▼ *Посмотреть примеры*

```
SELECT
    greatest(1, 2, 3, 4, 4) AS result_1,
    greatest(1, -1)         AS result_2,
    greatest(0)             AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 4        | 1        | 0        |
+----------+----------+----------+
```

`isfinite()`

Описание Проверяет, является ли число конечным.

Использование `isfinite(num)`

▼ *Посмотреть примеры*

```
SELECT
```

```
    isfinite(1)                  AS result_1,
    isfinite('Infinity'::DOUBLE) AS result_2,
    isfinite(NULL)               AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| true     | false    | null     |
+----------+----------+----------+
```

`isinf()`

Описание Проверяет, является ли число бесконечным.

Использование `isinf(num)`

▼ *Посмотреть примеры*

```
SELECT
    isinf(1)                  AS result_1,
    isinf('Infinity'::DOUBLE) AS result_2,
    isinf(NULL)               AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| false    | true     | null     |
+----------+----------+----------+
```

`isnan()`

Описание Проверяет, имеет ли аргумент значение `NaN` (*Not a Number*).

Использование `isnan(num)`

▼ *Посмотреть примеры*

```
SELECT
    isnan('NaN'::DOUBLE) AS result_1,
    isnan(1.1)           AS result_2,
    isnan(NULL)          AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| true     | false    | null     |
+----------+----------+----------+
```

`lcm()`

Описание Вычисляет наименьшее общее кратное двух чисел.

Использование `lcm(num, num)`

Псевдонимы `least_common_multiple()`

▼ *Посмотреть примеры*

```
SELECT
    lcm(3, 7)     AS result_1,
    lcm(333, 777) AS result_2,
    lcm(37, 0)    AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 21       | 2331     | 0        |
+----------+----------+----------+
```

`least()`

Описание Возвращает наименьшее число из указанных в аргументах.

Использование `least(num[, num, ...])`

▼ *Посмотреть примеры*

```
SELECT
    least(1, 1, 2, 3, 4) AS result_1,
    least(1, -1)         AS result_2,
    least(0)             AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 1        | -1       | 0        |
+----------+----------+----------+
```

`lgamma()`

Описание Вычисляет логарифм гамма-функции.

Использование `lgamma(num)`

▼ *Посмотреть примеры*

```
SELECT
```

```
    lgamma(1)   AS result_1,
    lgamma(11)  AS result_2,
    lgamma(1.1) AS result_3;
```

```
+----------+--------------------+-----------------------+
| result_1 | result_2           | result_3              |
+----------+--------------------+-----------------------+
| 0        | 15.104412573075518 | -0.049872441259839764 |
+----------+--------------------+-----------------------+
```

ln()

Описание Вычисляет натуральный логарифм числа.

Использование `ln(num)`

▼ *Посмотреть примеры*

```
SELECT
    ln(1)   AS result_1,
    ln(11)  AS result_2,
    ln(1.1) AS result_3;
```

```
+----------+--------------------+----------------------+
| result_1 | result_2           | result_3             |
+----------+--------------------+----------------------+
| 0        | 2.3978952727983707 | 0.09531017980432493 |
+----------+--------------------+----------------------+
```

log()

Описание Вычисляет логарифм числа по основанию `10`.

Использование `log(num)`

Псевдонимы `log10()`

▼ *Посмотреть примеры*

```
SELECT
    log(1)    AS result_1,
    log(100)  AS result_2,
    log(0.01) AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 0        | 2        | -2       |
```

```
+----------+----------+----------+
```

log2()

Описание Вычисляет логарифм числа по основанию `2`.

Использование `log2(num)`

▼ *Посмотреть примеры*

```
SELECT
    log2(1)    AS result_1,
    log2(2)    AS result_2,
    log2(4096) AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 0        | 1        | 12       |
+----------+----------+----------+
```

multiply()

Описание Перемножает два числа.

Использование `multiply(num, num)`

См. также Оператор `*`.

▼ *Посмотреть примеры*

```
SELECT
    multiply(2, 2)      AS result_1,
    multiply(0, 2)      AS result_2,
    multiply(0.2, -0.2) AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 4        | 0        | -0.04    |
+----------+----------+----------+
```

nextafter()

Описание Возвращает следующее значение с переменной точностью (типа `DOUBLE`) после первого числа в направлении второго числа.

Использование `nextafter(num, num)`

▼ *Посмотреть примеры*

```
SELECT
    nextafter(1::DOUBLE, 2)  AS result_1,
    nextafter(1::BIGINT, 2)  AS result_2,
    nextafter(-1::BIGINT, 0) AS result_3;
```

```
+-------------------+-------------------+--------------------+
|      result_1     |      result_2     |       result_3     |
+-------------------+-------------------+--------------------+
| 1.0000000000000002 | 1.0000000000000002 | -0.9999999999999999 |
+-------------------+-------------------+--------------------+
```

`pi()`

Описание Возвращает значение числа π.

Использование `pi()`

▼ *Посмотреть примеры*

```
SELECT
    pi()   AS result_1,
    pi()/2 AS result_2,
    2*pi() AS result_3;
```

```
+------------------+------------------+------------------+
|     result_1     |     result_2     |     result_3     |
+------------------+------------------+------------------+
| 3.141592653589793 | 1.5707963267948966 | 6.283185307179586 |
+------------------+------------------+------------------+
```

`pow()`

Описание Возводит первый аргумент в степень, заданную вторым аргументом.

Использование `pow(num, num)`

Псевдонимы `power()`

См. также Оператор `^`.

▼ *Посмотреть примеры*

```
SELECT
    pow(2, 5)   AS result_1,
    pow(25, -1) AS result_2,
    pow(25, 0)  AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 32       | 0.04     | 1        |
+----------+----------+----------+
```

`radians()`

Описание Переводит градусы в радианы.

Использование `radians(num)`

▼ *Посмотреть примеры*

```
SELECT
    radians(0)         AS result_1,
    radians(180)       AS result_2,
    radians(-180/pi()) AS result_3;
```

```
+----------+-------------------+----------+
| result_1 |      result_2     | result_3 |
+----------+-------------------+----------+
| 0        | 3.141592653589793 | -1       |
+----------+-------------------+----------+
```

`random()`

Описание Возвращает произвольное число (типа `DOUBLE`) в интервале от `0` до `1`.

Использование `random()`

См. также `setseed()`.

▼ *Посмотреть примеры*

```
SELECT
    random() AS result;
```

```
+--------------------+
|       result       |
+--------------------+
| 0.5656213557274057 |
+--------------------+
```

`round_even()`

Описание	Округляет число из первого аргумента до ближайшего четного с указанной во втором аргументе точностью.
Использование	`round_even(num, num)`
Псевдонимы	`roundbankers()`

Второй аргумент указывает на количество десятичных знаков точности и может быть отрицательным числом.

Подробнее об округлении до ближайшего четного числа см. здесь.

▼ *Посмотреть примеры*

```
SELECT
    round_even(4.5, 0)    AS result_1,
    round_even(3.5, 0)    AS result_2,
    round_even(-4.5, 0)   AS result_3,
    round_even(-3.5, 0)   AS result_4,
    round_even(4.45, 1)   AS result_5,
    round_even(4.35, 1)   AS result_6,
    round_even(35.35, -1) AS result_7;
```

```
+----------+----------+----------+----------+----------+----------+----------+
| result_1 | result_2 | result_3 | result_4 | result_5 | result_6 | result_7 |
+----------+----------+----------+----------+----------+----------+----------+
| 4        | 4        | -4       | -4       | 4.4      | 4.4      | 40       |
+----------+----------+----------+----------+----------+----------+----------+
```

`round()`

Описание	Округляет число из первого аргумента с указанной во втором аргументе точностью.
Использование	`round(num, num)`

Второй аргумент указывает на количество десятичных знаков точности и может быть отрицательным числом.

▼ *Посмотреть примеры*

```
SELECT
    round(4.5, 0)   AS result_1,
    round(4.45, 1)  AS result_2,
    round(44.5, -1) AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 5        | 4.5      | 40       |
```

```
+----------+----------+----------+
```

`setseed()`

Описание Фиксирует начальное значение для функции `random()`.

Использование `setseed(num)`

См. также `random()`.

▼ *Посмотреть примеры*

```
SELECT
    setseed(0.5) AS seed,
    random() AS random;
```

```
+------+--------------------+
| seed | random             |
+------+--------------------+
| null | 0.8511131886287325 |
+------+--------------------+
```

`sign()`

Описание Возвращает `-1`, `1` или `0` в зависимости от знака аргумента.

Использование `sign(num)`

▼ *Посмотреть примеры*

```
SELECT
    sign(10)  AS result_1,
    sign(-10) AS result_2,
    sign(0)   AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 1        | -1       | 0        |
+----------+----------+----------+
```

`signbit()`

Описание Определяет, установлен ли бит знака у вещественного числа.

Использование `signbit(num)`

▼ *Посмотреть примеры*

```
SELECT
    signbit(-1)                  AS result_1,
    signbit(-'Infinity'::DOUBLE) AS result_2,
    signbit(0)                   AS result_3,
    signbit(1)                   AS result_4,
    signbit('Infinity'::DOUBLE)  AS result_5;
```

```
+----------+----------+----------+----------+----------+
| result_1 | result_2 | result_3 | result_4 | result_5 |
+----------+----------+----------+----------+----------+
| true     | true     | false    | false    | false    |
+----------+----------+----------+----------+----------+
```

`sin()`

Описание Вычисляет синус угла, заданного в радианах.

Использование `sin(num)`

▼ *Посмотреть примеры*

```
SELECT
    sin(0)          AS result_1,
    sin(pi()/2)     AS result_2,
    sin((3*pi())/2) AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 0        | 1        | -1       |
+----------+----------+----------+
```

`sqrt()`

Описание Вычисляет квадратный корень.

Использование `sqrt(num)`

Число `num` должно быть неотрицательным.

▼ *Посмотреть примеры*

```
SELECT
    sqrt(4)   AS result_1,
    sqrt(144) AS result_2,
    sqrt(0)   AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 2        | 12       | 0        |
+----------+----------+----------+
```

`subtract()`

Описание Вычитает второй аргумент из первого.

Использование `subtract(num, num)`

См. также Оператор `-`.

▼ *Посмотреть примеры*

```
SELECT
    subtract(1, 2)     AS result_1,
    subtract(1.1, 2.2) AS result_2,
    subtract(-1, -2)   AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| -1       | -1.1     | 1        |
+----------+----------+----------+
```

`trunc()`

Описание Отбрасывает все знаки после десятичного разделителя.

Использование `trunc(num)`

Не следует путать с округлением `round`.

▼ *Посмотреть примеры*

```
SELECT
    trunc(1.99)  AS result_1,
    trunc(-11.9) AS result_2,
    trunc(0.119) AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 1        | -11      | 0        |
+----------+----------+----------+
```

Оператор `+`

Описание Прибавляет правый аргумент к левому.

Использование `<num> + <num> [+ ...]` или `TIME + INTERVAL [+ ...]`

Если используется с типами для для даты и времени, то прибавляет интервал к значению времени. Возвращает значение типа `TIME`.

См. также `add()`.

▼ *Посмотреть примеры*

```
SELECT
    3 + 2      AS result_1,
    3 + 2 + -1 AS result_2,
    1.1 + 1.9  AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 5        | 4        | 3.0      |
+----------+----------+----------+
```

```
SELECT
    TIME '12:11:10'     + INTERVAL 3 hours AS result_time_1,
    INTERVAL '12:11:10' + TIME '1:1:1'     AS result_time_2;
```

```
+--------------+--------------+
| result_time_1 | result_time_2 |
+--------------+--------------+
| 15:11:10      | 13:12:11      |
+--------------+--------------+
```

Оператор `-`

Описание Вычитает правый аргумент из левого.

Использование `<num> - <num> [- ...]` или `TIME - INTERVAL [- ...]`

Если используется с типами для для даты и времени, то вычитает интервал из значения времени. Возвращает значение типа `TIME`.

См. также `subtract()`.

▼ *Посмотреть примеры*

```
SELECT
```

```
    3 - 2      AS result_1,
    3 - 2 - +1 AS result_2,
    1.2 - 0.2  AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 1        | 0        | 1.0      |
+----------+----------+----------+
```

```
SELECT
    TIME '12:11:10' - INTERVAL 3 HOUR AS result_time_1,
    TIME '12:11:10' - INTERVAL 3 HOUR - INTERVAL 1 HOUR AS result_time_2;
```

```
+---------------+---------------+
| result_time_1 | result_time_2 |
+---------------+---------------+
| 09:11:10      | 08:11:10      |
+---------------+---------------+
```

Оператор *

Описание Перемножает аргументы.

Использование `<num> * <num>[ * <num>, ...]`

См. также `multiply()`.

▼ *Посмотреть примеры*

```
SELECT
    3*2     AS result_1,
    3*+2*-2 AS result_2,
    3*0     AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 6        | -12      | 0        |
+----------+----------+----------+
```

Оператор /

Описание Делит левый аргумент на правый.

Использование `<num> / <num>[ / <num>, ...]`

Возвращает результат в виде числа с переменной точностью (типа `DOUBLE`).

▼ *Посмотреть примеры*

```
SELECT
    3/2     AS result_1,
    3/+2/-2 AS result_2,
    3/1     AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 1.5      | -0.75    | 3        |
+----------+----------+----------+
```

Оператор `%`

Описание Возвращает остаток от деления левого аргумента на правый.

Использование `<num> % <num>[ % <num>, ...]`

▼ *Посмотреть примеры*

```
SELECT
    3 % 2       AS result_1,
    15 % 10 % 3 AS result_2,
    5 % 2.4     AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 1        | 2        | 0.2      |
+----------+----------+----------+
```

Оператор `^`

Описание Возводит левый аргумент в степень, заданную правым аргументом.

Использование `<num> ^ <num>[ ^ <num>, ...]`

См. также `pow()`.

▼ *Посмотреть примеры*

```
SELECT
    2^3   AS result_1,
```

```
    2^3^2 AS result_2,
    1^0   AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 8        | 64       | 1        |
+----------+----------+----------+
```

Текстовые функции

Текстовые функции — это функции для работы с текстовыми строками (данными типа `VARCHAR`).

`concat()`

Описание Конкатенирует несколько строк, массивов или двоичных значений.

Использование `concat(argument1, argument2, ...)`

Пустые значения (`NULL`) игнорируются.

См. также Оператор `||`.

▼ *Посмотреть пример*

```
SELECT
    concat('\xAA'::BLOB, '\xff'::BLOB) as result_blob,
    concat('I', ' ', 'love', ' ', 'Tengri') as result_string,
    concat(['T', 'e'], ['n', 'g', 'r', 'i']) as result_array;
```

```
+-------------+---------------+---------------+
| result_blob | result_string |  result_array |
+-------------+---------------+---------------+
| \xAA\xFF    | I love Tengri | {T,e,n,g,r,i} |
+-------------+---------------+---------------+
```

`contains()`

Описание Возвращает `true`, если в указанной строке `string` содержится искомая подстрока `search_string`.

Использование `contains(string, search_string)`

▼ *Посмотреть пример*

```
SELECT
    contains('I love Tengri', 'Tengri') AS check_name,
```

```
    contains('I love Tengri', 'TNGRi')  AS check_nickname;
```

```
+------------+----------------+
| check_name | check_nickname |
+------------+----------------+
| true       | false          |
+------------+----------------+
```

`length()`

Описание Возвращает количество символов в строке.

Использование `length(string)`

Псевдонимы `char_length()`, `character_length()`

▼ *Посмотреть пример*

```
SELECT
    length('I love Tengri ') AS length;
```

```
+--------+
| length |
+--------+
| 15     |
+--------+
```

`strlen()`

Описание Возвращает количество байтов в строке.

Использование `strlen(string)`

▼ *Посмотреть пример*

```
SELECT
    strlen('Tengri ') AS strlen;
```

```
+--------+
| strlen |
+--------+
| 11     |
+--------+
```

`trim()`

Описание Удаляет все вхождения любого из указанных символов с обеих сторон строки.

Использование `trim(string[, characters])`

Если подлежащие удалению символы не указаны, то по умолчанию таким символом является пробел.

▼ *Посмотреть примеры*

```
SELECT
    '"' || trim('  Tengri ') || '"' AS trim;
```

```
+----------+
|   trim   |
+----------+
| "Tengri" |
+----------+
```

```
SELECT
    trim('[Tengri]', '{([])}') AS trim_brackets;
```

```
+---------------+
| trim_brackets |
+---------------+
| Tengri        |
+---------------+
```

`ltrim()`

Описание Удаляет все вхождения любого из указанных символов в начале строки.

Использование `ltrim(string[, characters])`

Если подлежащие удалению символы не указаны, то по умолчанию таким символом является пробел.

▼ *Посмотреть примеры*

```
SELECT
    '"' || ltrim(' Tengri ') || '"' AS ltrim;
```

```
+-----------+
|   ltrim   |
+-----------+
```

```
| "Tengri " |
+-----------+
```

```
SELECT
    ltrim('{{{[Tengri])}}}', '{([])}') AS ltrim_brackets;
```

```
+----------------+
| ltrim_brackets |
+----------------+
| Tengri])}}     |
+----------------+
```

`rtrim()`

Описание Удаляет все вхождения любого из указанных символов в конце строки.

Использование `rtrim(string[, characters])`

Если подлежащие удалению символы не указаны, то по умолчанию таким символом является пробел.

▼ *Посмотреть примеры*

```
SELECT
    '"' || rtrim(' Tengri ') || '"' AS ltrim;
```

```
+-----------+
|   ltrim   |
+-----------+
| " Tengri" |
+-----------+
```

```
SELECT
    rtrim('{{{[Tengri])}}}', '{([])}') AS rtrim_brackets;
```

```
+----------------+
| rtrim_brackets |
+----------------+
| {{{[Tengri     |
+----------------+
```

`lower()`

Описание Преобразует строку в нижний регистр.

Использование `lower(string)`

Псевдонимы `lcase()`

▼ *Посмотреть пример*

```
SELECT
    lower('TNGRi') AS lower;
```

```
+-------+
| lower |
+-------+
| tngri |
+-------+
```

`upper()`

Описание Преобразует строку в верхний регистр.

Использование `upper(string)`

Псевдонимы `ucase()`

▼ *Посмотреть пример*

```
SELECT
    upper('Tengri') AS upper;
```

```
+--------+
|  upper |
+--------+
| TENGRI |
+--------+
```

`split()`

Описание Делит строку на две части по заданному сепаратору.

Использование `split(string, separator)`

Псевдонимы `str_split`, `string_split`, `string_to_array`

▼ *Посмотреть пример*

```
SELECT
    split('I love Tengri', ' ') AS words;
```

```
+-----------------+
|      words      |
```

```
+-----------------+
| {I,love,Tengri} |
+-----------------+
```

chr()

Описание Возвращает символ, соответствующий значению кода ASCII или кода Unicode, заданному в `argument`.

Использование `chr(argument)`

▼ *Посмотреть пример*

```
SELECT
    chr(84) || chr(78) || chr(71) || chr(82) || chr(105) AS chr;
```

```
+-------+
|  chr  |
+-------+
| TNGRi |
+-------+
```

md5()

Описание Возвращает хеш MD5 данных из `argument` в виде строки (`VARCHAR`).

Использование `md5(argument)`

В `argument` могут быть двоичные данные или строка.

▼ *Посмотреть примеры*

```
SELECT
    md5('\xAA\xFF'::BLOB) as md5_hash;
```

```
+----------------------------------+
|             md5_hash             |
+----------------------------------+
| 1fab7f7621f5ddc051ebd1f2c63c4665 |
+----------------------------------+
```

```
SELECT
    md5('Tengri') as md5_hash;
```

```
+----------------------------------+
|             md5_hash             |
```

```
+----------------------------------+
| 846b02d31131a10bd6ac0ba189c65bef |
+----------------------------------+
```

sha1()

Описание Возвращает хеш SHA-1 данных из `argument` в виде строки (`VARCHAR`).

Использование `sha1(argument)`

В `argument` могут быть двоичные данные или строка.

▼ *Посмотреть примеры*

```
SELECT
    sha1('\xAA\xFF'::BLOB) as sha1_hash;
```

```
+------------------------------------------+
|                 sha1_hash                |
+------------------------------------------+
| e89b0db325637edfacde04a76005c492e2c5aeca |
+------------------------------------------+
```

```
SELECT
    sha1('Tengri') as sha1_hash;
```

```
+------------------------------------------+
|                 sha1_hash                |
+------------------------------------------+
| b514525a19995a2442d7565bfd9bb42d9dc71a13 |
+------------------------------------------+
```

sha256()

Описание Возвращает хеш SHA-256 данных из `argument` в виде строки (`VARCHAR`).

Использование `sha256(argument)`

В `argument` могут быть двоичные данные или строка.

▼ *Посмотреть пример*

```
SELECT
    sha256('\xAA\xFF'::BLOB) as sha256_hash;
```

```
+------------------------------------------------------------------+
```

```
|                              sha256_hash                               |
+------------------------------------------------------------------------+
| 768318522cac43261e8ef4946c2296a3643d523a8d5bda8ff5b82aa64470421a |
+------------------------------------------------------------------------+
```

```
SELECT
    sha256('Tengri') as sha256_hash;
```

```
+------------------------------------------------------------------------+
|                              sha256_hash                               |
+------------------------------------------------------------------------+
| 8aaacef66663b14ee7c5a03dbaec7b40f0f3bf17bd12d2ed4f9aaad0e10a0d77 |
+------------------------------------------------------------------------+
```

Оператор ||

Описание Конкатенирует несколько строк, массивов или двоичных значений.

Использование `argument1 || argument2 || ...`

Пустые значения (`NULL`) игнорируются.

См. также concat().

▼ *Посмотреть примеры*

```
SELECT
    '\xAA'::BLOB || '\xff'::BLOB as result_blob,
    'I' || ' ' || 'love' || ' ' || 'Tengri' as result_string,
    ['T', 'e'] || ['n', 'g', 'r', 'i'] as result_array;
```

```
+-------------+---------------+---------------+
| result_blob | result_string |  result_array |
+-------------+---------------+---------------+
|             | I love Tengri | {T,e,n,g,r,i} |
+-------------+---------------+---------------+
```

Обратите внимание, что значения в столбце `result_blob` в выводе не отображаются (так как имеют тип `BLOB`).

С помощью выражения `DESCRIBE` выведем типы данных для всех столбцов в таблице, созданной так же, как в предыдущем примере:

```
CREATE TABLE concat AS

SELECT
    '\xAA'::BLOB || '\xff'::BLOB as result_blob,
```

```
    'I' || ' ' || 'love' || ' ' || 'Tengri' as result_string,
    ['T', 'e'] || ['n', 'g', 'r', 'i'] as result_array;

DESCRIBE TABLE concat;
```

```
+--------------+------------+------+------+---------+-------+
|  column_name  | column_type | null |  key | default | extra |
+--------------+------------+------+------+---------+-------+
| result_blob   | BLOB        | YES  | null | null    | null  |
+--------------+------------+------+------+---------+-------+
| result_string | VARCHAR     | YES  | null | null    | null  |
+--------------+------------+------+------+---------+-------+
| result_array  | VARCHAR[]   | YES  | null | null    | null  |
+--------------+------------+------+------+---------+-------+
```

Функции для регулярных выражений

Функции для работы с текстовыми строками с помощью регулярных выражений. В регулярных выражениях используется синтаксис RE2.

`regexp_extract()`

Описание	Извлекает из строки подстроку по заданному регулярному выражению.
Использование	`regexp_extract(string, regexp)`

Если в строке `string` имеется подстрока, накрытая шаблоном регулярного выражения `regexp`, возвращает эту подстроку. Если подстрок, накрытых шаблоном, несколько, то возвращает первую подстроку. Если таких подстрок не найдено, возвращает пустую строку.

▼ *Посмотреть примеры*

```
SELECT
    regexp_extract('Tengri', '..') AS result_1,
    regexp_extract('Tengri', 'n.*') AS result_2,
    regexp_extract('Tengri', '.*') AS result_3,
    regexp_extract('Tengri', '^.{5}') AS result_4,
    regexp_extract('Tengri', '[TNGRi]$') AS result_5,
    regexp_extract('Tengri', '.{7}') AS result_empty;
```

```
+----------+----------+----------+----------+----------+--------------+
| result_1 | result_2 | result_3 | result_4 | result_5 | result_empty |
+----------+----------+----------+----------+----------+--------------+
| Te       | ngri     | Tengri   | Tengr    | i        |              |
+----------+----------+----------+----------+----------+--------------+
```

`regexp_extract_all()`

Описание	Извлекает из строки все не пересекающиеся подстроки по заданному регулярному выражению. Возвращает массив подстрок.
Использование	`regexp_extract_all(string, regexp[, <num>])`

Если подстрок в `string`, накрытых шаблоном `regexp` не найдено, возвращает пустой список.

Параметры

- `<num>` — номер группы внутри шаблона, которую следует вернуть (для каждой подстроки). По умолчанию (если параметр не задан) возвращается вся подстрока. Нумерация групп начинается с `1`. Группы в шаблоне выделяются круглыми скобками.

  ▼ *Подробнее о параметре на примерах*

  Извлечем из текста все сочетания букв с точкой после них:

```
SELECT
    regexp_extract_all('My name is Tengri. My nickname is TNGRi.',
                       '(\w+)(\.)')
AS result;
```

```
+------------------+
|      result      |
+------------------+
| {Tengri.,TNGRi.} |
+------------------+
```

  Теперь извлечем из того же текста с помощью того же регулярного выражения сочетания букв перед точкой, но без самой точки. Для этого зададим номер группы `1`:

```
SELECT
    regexp_extract_all('My name is Tengri. My nickname is TNGRi.',
                       '(\w+)(\.)',
                       1)
AS result;
```

```
+----------------+
|     result     |
+----------------+
| {Tengri,TNGRi} |
+----------------+
```

▼ *Посмотреть еще примеры*

```
SELECT
    regexp_extract_all('My name is Tengri. My nickname is TNGRi.', '\w+')
AS words;
```

```
+----------------------------------------+
|                  words                 |
+----------------------------------------+
| {My,name,is,Tengri,My,nickname,is,TNGRi} |
+----------------------------------------+
```

Используем комбинацию функций `regexp_extract_all` и `unnest`, чтобы извлечь данные из структурированного текста:

```
CREATE TABLE text_table(text_data VARCHAR);

INSERT INTO text_table VALUES
('Name: "Tengri", Nickname:"TNGRi"'),
('Country: "Russia", Capital:"Moscow"');

SELECT
    unnest(
        regexp_extract_all(text_data, '(\w+):\s*"(.*?)"', 1)
          ) AS key,
    unnest(
        regexp_extract_all(text_data, '(\w+):\s*"(.*?)"', 2)
          ) AS value
FROM text_table;
```

```
+----------+--------+
|    key   | value  |
+----------+--------+
| Name     | Tengri |
+----------+--------+
| Nickname | TNGRi  |
+----------+--------+
| Country  | Russia |
+----------+--------+
| Capital  | Moscow |
+----------+--------+
```

`regexp_full_match()`

Описание Проверяет, накрывает ли регулярное выражение строку полностью.

Использование `regexp_full_match(string, regexp)`

Если шаблон регулярного выражения `regexp` полностью накрывает строку `string`, то возвращает

`true`, иначе — `false`.

▼ *Посмотреть примеры*

```
SELECT
    regexp_full_match('Tengri', 'gri$')     AS result_1, -- false expected
    regexp_full_match('Tengri', '.')        AS result_2, -- false expected
    regexp_full_match('Tengri', '.*')       AS result_3, -- true expected
    regexp_full_match('Tengri', '^\w+gri$') AS result_4; -- true expected
```

```
+----------+----------+----------+----------+
| result_1 | result_2 | result_3 | result_4 |
+----------+----------+----------+----------+
| false    | false    | true     | true     |
+----------+----------+----------+----------+
```

`regexp_matches()`

Описание Проверяет, содержится ли регулярное выражение внутри строки.

Использование `regexp_matches(string, regexp)`

Если внутри строки `string` найдена хотя бы одна подстрока, накрытая шаблоном `regexp`, то возвращает `true`, иначе — `false`.

▼ *Посмотреть примеры*

```
SELECT
    regexp_matches('Tengri', '.+T') AS result_1, -- false expected
    regexp_matches('Tengri', '.*T') AS result_2; -- true expected
```

```
+----------+----------+
| result_1 | result_2 |
+----------+----------+
| false    | true     |
+----------+----------+
```

`regexp_replace()`

Описание Заменяет подстроку, накрытую регулярным выражением, на указанную строку.

Использование `regexp_replace(string, regexp, target)`

Если внутри строки `string` найдена подстрока, накрытая шаблоном `regexp`, то она заменяется на строку `target`. Если таких подстрок найдено несколько, то заменяется только первая. Если подстрок не найдено, возвращается исходная строка `string`.

▼ *Посмотреть примеры*

```
SELECT
    regexp_replace('Tengri', '.', 't') AS result_1,
    regexp_replace('Tengri', '.*', 't') AS result_2,
    regexp_replace('Tengri', 'e.*r', 'NGR') AS result_3,
    regexp_replace('Tengri', 'a', 't') AS result_4;
```

```
+----------+----------+----------+----------+
| result_1 | result_2 | result_3 | result_4 |
+----------+----------+----------+----------+
| tengri   | t        | TNGRi    | Tengri   |
+----------+----------+----------+----------+
```

`regexp_split_to_array()`

Описание Разрезает строку на части, разделенные регулярным выражением, и возвращает части в виде массива.

Использование `regexp_split_to_array(string, regexp)`

Псевдонимы `string_split_regex()`

Если в строке `string` найдены подстроки, накрытые шаблоном `regexp`, то возвращаются части исходной строки, не накрытые шаблоном, в виде массива. Если найденные подстроки стоят в начале или в конце исходной строки, то в результирующий массив попадут пустые строки для начала и конца строки. Если подстрок не найдено, то будет возвращен массив из одной исходной строки.

▼ *Посмотреть примеры*

```
SELECT
    string_split_regex('My name is Tengri. My nickname is TNGRi.', '\.\s')
AS sentences,
    string_split_regex('My name is Tengri. My nickname is TNGRi.', '[\.\s]+')
AS words;
```

```
+-----------------------------------------+-----------------------------------------+
|                 sentences               |                  words                  |
+-----------------------------------------+-----------------------------------------+
| {My name is Tengri,My nickname is TNGRi.} | {My,name,is,Tengri,My,nickname,is,TNGRi,} |
+-----------------------------------------+-----------------------------------------+
```

`regexp_split_to_table()`

Описание Разрезает строку на части, разделенные регулярным выражением, и возвращает части в виде строк.

Использование `regexp_split_to_table(string, regexp)`

Если в строке `string` найдены подстроки, накрытые шаблоном `regexp`, то возвращаются части исходной строки, не накрытые шаблоном, в виде столбца, где каждая часть записана в свою ячейку. Если найденные подстроки стоят в начале или в конце исходной строки, то в результат попадут пустые строки для начала и конца строки. Если подстрок не найдено, то будет возвращен столбец с одной ячейкой, в которой записана исходная строка.

▼ *Посмотреть примеры*

```
SELECT
    regexp_split_to_table('My name is Tengri. My nickname is TNGRi.', '\.\s')
AS sentences,
    regexp_split_to_table('My name is Tengri. My nickname is TNGRi.', '[\.\s]+')
AS words;
```

```
+----------------------+----------+
|       sentences      |   words  |
+----------------------+----------+
| My name is Tengri    | My       |
+----------------------+----------+
| My nickname is TNGRi. | name     |
+----------------------+----------+
| null                 | is       |
+----------------------+----------+
| null                 | Tengri   |
+----------------------+----------+
| null                 | My       |
+----------------------+----------+
| null                 | nickname |
+----------------------+----------+
| null                 | is       |
+----------------------+----------+
| null                 | TNGRi    |
+----------------------+----------+
| null                 |          |
+----------------------+----------+
```

Оператор `~`

Описание	Проверяет, накрывает ли регулярное выражение строку полностью.
Использование	`string ~ regexp`

Может использоваться с оператором отрицания `!`: `string !~ regexp`.

Полностью синонимичен оператору `SIMILAR TO`.

▼ *Посмотреть примеры*

```
SELECT
    'Tengri' !~ 'TNGRi' AS result_1,
```

```
    'Tengri' ~ 'T.*'    AS result_2,
    'Tengri' ~ 'T.+i'   AS result_3,
    'TNGRi'  ~ 'T.+i'   AS result_4,
    'T.+i'   ~ 'T.+i'   AS result_5;
```

```
+----------+----------+----------+----------+----------+
| result_1 | result_2 | result_3 | result_4 | result_5 |
+----------+----------+----------+----------+----------+
| true     | true     | true     | true     | true     |
+----------+----------+----------+----------+----------+
```

Функции для даты и времени

Функции для даты и времени — это функции для работы с данными типов `DATE`, `TIME`, `TIMESTAMP` и `TIMESTAMPTZ`.

`current_time()`

Описание Возвращает текущее время в виде значения типа `TIME`.

Использование `current_time` или `current_time()`

Псевдонимы `get_current_time()`

▼ *Посмотреть пример*

```
SELECT
    current_time       AS cur_time_1,
    current_time()     AS cur_time_2,
    get_current_time() AS cur_time_3;
```

```
+-----------------+-----------------+-----------------+
|    cur_time_1   |    cur_time_2   |    cur_time_3   |
+-----------------+-----------------+-----------------+
| 10:33:24.016000 | 10:33:24.016000 | 10:33:24.016000 |
+-----------------+-----------------+-----------------+
```

`datepart()`

Описание Возвращает указанную часть от значения даты или времени в виде значения типа `BIGINT`.

Использование `datepart('<part>', (TIME | DATE | ...) '<date_time>')`

Псевдонимы `date_part()`

Аргументами могут быть значения типов: `TIME`, `DATE`, `TIMESTAMP` или `TIMESTAMPTZ`.

▼ *Части могут быть указаны с помощью следующих литералов*

- `century` — век
- `day` — день
- `decade` — десятилетие
- `hour` — час
- `microseconds` — микросекунды
- `millennium` — тысячелетие
- `milliseconds` — миллисекунды
- `minute` — минута
- `month` — месяц
- `quarter` — квартал
- `second` — секунда
- `year` — год

▼ *Посмотреть примеры*

```
SELECT
    datepart('milliseconds', TIMESTAMP '2025-02-25 00:00:00.1') AS milliseconds,
    datepart('hour',         TIME '2025-02-25 00:00:00')        AS hour,
    datepart('millennium',   DATE '2025-02-25')                 AS millennium;
```

```
+--------------+------+------------+
| milliseconds | hour | millennium |
+--------------+------+------------+
| 100          | 0    | 3          |
+--------------+------+------------+
```

`date_diff()`

Описание	Возвращает количество единиц времени между двумя моментами времени в виде значения типа `BIGINT`.
Использование	`date_diff('<part>', start, end)`

Аргументами могут быть значения типов: `TIME`, `DATE`, `TIMESTAMP` или `TIMESTAMPTZ`.

▼ *Единицы могут быть указаны с помощью следующих литералов*

- `century` — век
- `day` — день
- `decade` — десятилетие
- `hour` — час
- `microseconds` — микросекунды
- `millennium` — тысячелетие

- `milliseconds` — миллисекунды
- `minute` — минута
- `month` — месяц
- `quarter` — квартал
- `second` — секунда
- `year` — год

▼ *Посмотреть примеры*

```
SELECT
    date_diff('second', TIME '01:02:03', TIME '03:02:01') AS diff_in_seconds,
    date_diff('minute', TIME '01:02:03', TIME '03:02:01') AS diff_in_minutes,
    date_diff('hour',   TIME '01:02:03', TIME '03:02:01') AS diff_in_hours,
    date_diff('day',    TIMESTAMP '2025-02-25 01:02:03', TIMESTAMP '2025-02-26 03:02:01')
AS diff_in_days,
    date_diff('day',    TIMESTAMP '2025-02-26 01:02:03', TIMESTAMP '2025-02-25 03:02:01')
AS diff_in_days,
    date_diff('day',    DATE '2024-02-27', DATE '2025-02-27') AS diff_in_days;
```

```
+-----------------+-----------------+---------------+--------------+--------------
+--------------+
| diff_in_seconds | diff_in_minutes | diff_in_hours | diff_in_days | diff_in_days |
diff_in_days |
+-----------------+-----------------+---------------+--------------+--------------
+--------------+
| 7198            | 120             | 2             | 1            | -1           | 366
|
+-----------------+-----------------+---------------+--------------+--------------
+--------------+
```

Оператор `+`

Описание Прибавляет правый аргумент к левому.

Использование `<num> + <num> [+ ...]` или `TIME + INTERVAL [+ ...]`

Если используется с типами для для даты и времени, то прибавляет интервал к значению времени. Возвращает значение типа `TIME`.

См. также `add()`.

▼ *Посмотреть примеры*

```
SELECT
    3 + 2      AS result_1,
    3 + 2 + -1 AS result_2,
    1.1 + 1.9  AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 5        | 4        | 3.0      |
+----------+----------+----------+
```

```
SELECT
    TIME '12:11:10'     + INTERVAL 3 hours AS result_time_1,
    INTERVAL '12:11:10' + TIME '1:1:1'     AS result_time_2;
```

```
+---------------+---------------+
| result_time_1 | result_time_2 |
+---------------+---------------+
| 15:11:10      | 13:12:11      |
+---------------+---------------+
```

Оператор `-`

Описание Вычитает правый аргумент из левого.

Использование `<num> - <num> [- ...]` или `TIME - INTERVAL [- ...]`

Если используется с типами для для даты и времени, то вычитает интервал из значения времени. Возвращает значение типа `TIME`.

См. также `subtract()`.

▼ *Посмотреть примеры*

```
SELECT
    3 - 2      AS result_1,
    3 - 2 - +1 AS result_2,
    1.2 - 0.2  AS result_3;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| 1        | 0        | 1.0      |
+----------+----------+----------+
```

```
SELECT
    TIME '12:11:10' - INTERVAL 3 HOUR AS result_time_1,
    TIME '12:11:10' - INTERVAL 3 HOUR - INTERVAL 1 HOUR AS result_time_2;
```

```
+--------------+--------------+
| result_time_1 | result_time_2 |
+--------------+--------------+
| 09:11:10      | 08:11:10      |
+--------------+--------------+
```

Функции для JSON

Функции для работы с файлами расширения `.json` и с типом данных `JSON`.

Путь внутри структуры JSON

Во многих функциях для JSON используется путь внутри структуры JSON как один из аргументов. Путь может задаваться в любой из двух нотаций по следующим стандартам:

- JSONPath
  - `$.key1.key2` — обращение к значению ключа `key2`
  - `$.key1.key2[i]` — обращение к `i`-му элементу списка в значении ключа `key2`
- JSON Pointer
  - `/key1/key2` — обращение к значению ключа `key2`
  - `/key1/key2/i` — обращение к `i`-му элементу списка в значении ключа `key2`

Нумерация элементов списка в структуре JSON начинается с `0`.

В примерах ниже используется структура JSON из этого примера:

```
CREATE TABLE js_table(js_data JSON);

INSERT INTO js_table VALUES
('{
  "first_name": "John",
  "last_name": "Smith",
  "is_alive": true,
  "age": 27,
  "address": {
    "street_address": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postal_code": "10021-3100"
  },
  "phone_numbers": [
    {
      "type": "home",
      "number": "212 555-1234"
```

```
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [
    "Catherine",
    "Thomas",
    "Trevor"
  ],
  "spouse": null
}');
```

`json_array_length()`

Описание Возвращает количество элементов в массиве по указанному пути в JSON или `0`, если по указанному пути не массив.

Использование `json_array_length(json[, path])`

Если во втором аргументе задан список путей, то результатом будет список длин массивов по указанным путям.

▼ *Посмотреть примеры*

```
SELECT
    json_array_length(js_data, '$.phone_numbers')                AS phone_numbers,
    json_array_length(js_data, '$.children')                     AS children,
    json_array_length(js_data, ['$.phone_numbers', '$.children']) AS lists,
    json_array_length(js_data)                                   AS js_data,
    json_array_length(js_data, '$.first_name')                   AS first_name,
FROM js_table;
```

```
+---------------+----------+-------+---------+------------+
| phone_numbers | children | lists | js_data | first_name |
+---------------+----------+-------+---------+------------+
| 2             | 3        | {2,3} | 0       | 0          |
+---------------+----------+-------+---------+------------+
```

`json_contains()`

Описание Проверяет, есть ли в структуре JSON подструктура JSON, заданная во втором аргументе.

Использование `json_contains(json, json)`

Оба аргумента должны иметь тип `JSON`. Второй аргумент может быть числовым значением или текстовой строкой, при этом текстовая строка должна быть заключена в двойные кавычки.

▼ *Посмотреть примеры*

```
SELECT
    json_contains(js_data, '27')                      AS result_1,
    json_contains(js_data, '"John"')                  AS result_2,
    json_contains(js_data, '{"first_name": "John"}')  AS result_3,
    json_contains(js_data, '{"first_name": "Smith"}') AS result_4, -- false expected
FROM js_table;
```

```
+----------+----------+----------+----------+
| result_1 | result_2 | result_3 | result_4 |
+----------+----------+----------+----------+
| true     | true     | true     | false    |
+----------+----------+----------+----------+
```

`json_exists()`

Описание Проверяет, есть ли в структуре JSON указанный путь.

Использование `json_exists(json, path)`

Возвращает `BOOL` или массив `BOOL[]` для случаев, когда путь указывает на элементы списка в структуре JSON.

▼ *Посмотреть примеры*

```
SELECT
    json_exists(js_data, 'first_name')              AS result_1,
    json_exists(js_data, '$.first_name')            AS result_2,
    json_exists(js_data, '/first_name')             AS result_3,
    json_exists(js_data, '$.address.street_address') AS result_4,
    json_exists(js_data, '/address/street_address')  AS result_5,
    json_exists(js_data, '$.street_address')        AS result_6, -- false expected
    json_exists(js_data, '$.phone_numbers[*].type')  AS result_7,
    json_exists(js_data, '$..street_address')       AS result_8,
FROM js_table;
```

```
+----------+----------+----------+----------+----------+----------+-------------+----------
+
| result_1 | result_2 | result_3 | result_4 | result_5 | result_6 | result_7    | result_8
|
+----------+----------+----------+----------+----------+----------+-------------+----------
+
| true     | true     | true     | true     | true     | false    | {True,True} | {True}
|
+----------+----------+----------+----------+----------+----------+-------------+----------
+
```

json_extract()

Описание Извлекает данные из структуры JSON по указанному пути. Возвращает данные в виде `JSON`.

Использование `json_extract(json, path)` или `json_extract(json, [path1, path2, ...])`

Псевдонимы `json_extract_path`

Если в качестве второго аргумента задан список путей, то возвращает список значений.

▼ *Посмотреть примеры*

```
SELECT
    json_extract(js_data, 'first_name')                AS first_name,
    json_extract(js_data, ['first_name', 'last_name']) AS all_names,
    json_extract(js_data, '$.address.street_address')  AS street_address,
    json_extract(js_data, '$.phone_numbers[*].number') AS phone_numbers,
    json_extract(js_data, '$.children[0]')             AS child_1,
FROM js_table;
```

```
+------------+------------------+----------------+--------------------------------
+-----------+
| first_name | all_names        | street_address | phone_numbers                  |
child_1   |
+------------+------------------+----------------+--------------------------------
+-----------+
| John       | {"John","Smith"} | 21 2nd Street  | {"212 555-1234","646 555-4567"} |
Catherine |
+------------+------------------+----------------+--------------------------------
+-----------+
```

json_extract_string()

Описание Извлекает данные из структуры JSON по указанному пути. Возвращает данные в виде `VARCHAR`.

Использование `json_extract_string(json, path)` или `json_extract_string(json, [path1, path2, ...])`

Псевдонимы `json_extract_path_text`

Если в качестве второго аргумента задан список путей, то возвращает список значений.

▼ *Посмотреть примеры*

```
SELECT
    json_extract_string(js_data, 'first_name')                AS first_name,
    json_extract_string(js_data, ['first_name', 'last_name']) AS all_names,
    json_extract_string(js_data, '$.address.street_address')  AS street_address,
    json_extract_string(js_data, '$.phone_numbers[*].number') AS phone_numbers,
```

```
    json_extract_string(js_data, '$.children[0]')            AS child_1,
FROM js_table;
```

```
+-----------+-------------+---------------+----------------------------+-----------+
| first_name | all_names    | street_address | phone_numbers               | child_1   |
+-----------+-------------+---------------+----------------------------+-----------+
| John       | {John,Smith} | 21 2nd Street  | {212 555-1234,646 555-4567} | Catherine |
+-----------+-------------+---------------+----------------------------+-----------+
```

`json_group_array()`

Описание Возвращает список JSON, содержащий все значения столбца.

Использование `json_group_array(argument)`

Функция изменяет кардинальность данных.

▼ *Посмотреть примеры*

```
CREATE TABLE example (name VARCHAR);
INSERT INTO example VALUES ('Tengri'), ('TNGRi');

SELECT json_group_array(name) AS tengti_names
FROM example;
```

```
+--------------------+
| tengti_names       |
+--------------------+
| ["Tengri","TNGRi"] |
+--------------------+
```

`json_group_object()`

Описание Возвращает структуру JSON, содержащую все пары `key-value` из столбцов, указанных в аргументах.

Использование `json_group_object(argument1, argument2)`

Функция изменяет кардинальность данных.

▼ *Посмотреть примеры*

```
CREATE TABLE example (name VARCHAR, letters_num BIGINT);
INSERT INTO example VALUES
    ('Tengri', 6),
    ('TNGRi', 5);

SELECT json_group_object(name, letters_num) AS result
```

```
FROM example;
```

```
+-----------------------+
| result                |
+-----------------------+
| {"Tengri":6,"TNGRi":5} |
+-----------------------+
```

`json_keys()`

Описание Возвращает все ключи из указанной структуры JSON в виде `VARCHAR[]`.

Использование `json_keys(json[, path])`

Если во втором аргументе указан путь, то возвращает ключи структуры JSON по указанному пути. Если задан список путей, то результатом будет список списков ключей.

▼ *Посмотреть примеры*

```
SELECT
    json_keys(js_data) AS all_keys,
FROM js_table;
```

```
+-------------------------------------------------------------------------+
| all_keys                                                                |
+-------------------------------------------------------------------------+
| {first_name,last_name,is_alive,age,address,phone_numbers,children,spouse} |
+-------------------------------------------------------------------------+
```

```
SELECT
    json_keys(js_data, '$.address') AS address_keys,
FROM js_table;
```

```
+---------------------------------------+
| address_keys                          |
+---------------------------------------+
| {street_address,city,state,postal_code} |
+---------------------------------------+
```

```
SELECT
    json_keys(js_data, ['$.address',
                        '$.phone_numbers[0]'])
    AS address_and_phone_keys,
FROM js_table;
```

```
+-------------------------------------------------------------------------+
| address_and_phone_keys                                                  |
+-------------------------------------------------------------------------+
| {['street_address', 'city', 'state', 'postal_code'],['type', 'number']} |
+-------------------------------------------------------------------------+
```

`json_transform()`

Описание	Трансформирует структуру JSON в соответствии с указанной структурой.
Использование	`json_transform(json, string)`
Псевдонимы	`from_json()`

Целевая структура указывается в виде текстовой строки во втором аргументе.

В случаях пропусков значений в исходной структуре JSON в результат проставляются значения `NULL`.

В случаях невозможности преобразовать типы данных в исходной структуре к указанным проставляется значение `NULL`.

▼ *Посмотреть примеры*

```
CREATE TABLE example (js_data JSON);
INSERT INTO example VALUES
    ('{
        "first_name": "John",
        "last_name": "Smith",
        "age": 27
      }'),
    ('{
        "first_name": "John",
        "is_alive": true,
        "age": 28
      }');

SELECT
    json_transform(js_data,
        '{
        "first_name": "VARCHAR",
        "last_name": "VARCHAR",
        "is_alive": "BOOL",
        "age": "BIGINT"
        }')
    AS result
FROM example;
```

```
+-------------------------------------------------------------------------+
| result                                                                  |
+-------------------------------------------------------------------------+
```

```
| {"first_name": "John", "last_name": "Smith", "is_alive": null, "age": 27} |
+---------------------------------------------------------------------------+
| {"first_name": "John", "last_name": null, "is_alive": true, "age": 28}    |
+---------------------------------------------------------------------------+
```

`json_transform_strict()`

Описание Трансформирует структуру JSON в соответствии с указанной структурой. Выдает ошибку в случае несоответствия структур или типов.

Использование `json_transform_strict(json, string)`

Псевдонимы `from_json_strict()`

Целевая структура указывается в виде текстовой строки во втором аргументе.

В случаях, если в исходной структуре JSON отсутствуют значения из указанной структуры, выдает ошибку.

В случаях невозможности преобразовать типы данных в исходной структуре к указанным выдает ошибку.

▼ *Посмотреть примеры*

```
CREATE TABLE example (js_data JSON);
INSERT INTO example VALUES
    ('{
        "first_name": "John",
        "last_name": "Smith",
        "age": 27
      }'),
    ('{
        "first_name": "John",
        "is_alive": true,
        "age": 28
      }');
SELECT
    json_transform_strict(js_data,
        '{"first_name": "VARCHAR", "age": "BIGINT"}')
    AS result
FROM example;
```

```
+-----------------------------------+
| result                            |
+-----------------------------------+
| {"first_name": "John", "age": 27} |
+-----------------------------------+
| {"first_name": "John", "age": 28} |
+-----------------------------------+
```

```
CREATE TABLE example (js_data JSON);
INSERT INTO example VALUES
    ('{
        "first_name": "John",
        "last_name": "Smith",
        "age": 27
      }'),
    ('{
        "first_name": "John",
        "is_alive": true,
        "age": 28
      }');
SELECT
    json_transform_strict(js_data,
        '{"first_name": "BIGINT", "age": "BIGINT"}') -- error expected
    AS result
FROM example;
```

```
ERROR: InvalidInputException: Invalid Input Error:
Failed to cast value to numerical: "John"
```

```
CREATE TABLE example (js_data JSON);
INSERT INTO example VALUES
    ('{
        "first_name": "John",
        "last_name": "Smith",
        "age": 27
      }'),
    ('{
        "first_name": "John",
        "is_alive": true,
        "age": 28
      }');

SELECT
    json_transform_strict(js_data,
        '{"first_name": "VARCHAR", "last_name": "VARCHAR"}') -- error expected
    AS result
FROM example;
```

```
ERROR: InvalidInputException: Invalid Input Error:
Object {"first_name":"John","is_alive":true,"age":28} does not have key "last_name"
```

`json_valid()`

Описание Проверяет, является ли аргумент валидной структурой JSON.

Использование `json_valid(json)`

▼ *Посмотреть примеры*

```
SELECT
    json_valid(js_data)                  AS result_1,
    json_valid('{"first_name": "John"}') AS result_2,
    json_valid('{"first_name"}')         AS result_3, -- false expected
FROM js_table;
```

```
+----------+----------+----------+
| result_1 | result_2 | result_3 |
+----------+----------+----------+
| true     | true     | false    |
+----------+----------+----------+
```

`json_value()`

Описание Извлекает значения из структуры JSON по указанному пути.

Использование `json_value(json, path)`

Если по указанному пути значение не скалярное (список или вложенная структура), то возвращает `NULL`.

▼ *Посмотреть примеры*

```
SELECT
    json_value(js_data, 'first_name')      AS first_name,
    json_value(js_data, '$.phone_numbers[*].number') AS phone_numbers,
    json_value(js_data, '$.children[0]')   AS child_1,
    json_value(js_data, '$.phone_numbers') AS phone_numbers, -- NULL expected
    json_value(js_data, '$.address')       AS address,       -- NULL expected
FROM js_table;
```

```
+------------+---------------------------------+-----------+---------------+---------+
| first_name | phone_numbers                   | child_1   | phone_numbers | address |
+------------+---------------------------------+-----------+---------------+---------+
| John       | {"212 555-1234","646 555-4567"} | Catherine | null          | null    |
+------------+---------------------------------+-----------+---------------+---------+
```

`json()`

Описание Сокращает запись структуры JSON (убирает пробелы и переносы строк).

Использование `json(json)`

▼ *Посмотреть примеры*

```
SELECT
    json('{
            "first_name": "John",
            "last_name": "Smith"
          }
        ') AS result_1,
    json(js_data) AS result_2,
FROM js_table;
```

```
+-----------------------------------------------------
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
-------------------------+
| result_1                                            | result_2
|
+-----------------------------------------------------
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
-------------------------+
| "{""first_name"":""John"",""last_name"":""Smith""}" |
"{""first_name"":""John"",""last_name"":""Smith"",""is_alive"":true,""age"":27,""address"":
{""street_address"":""21 2nd Street"",""city"":""New
York"",""state"":""NY"",""postal_code"":""10021-
3100""},""phone_numbers"":[{""type"":""home"",""number"":""212 555-
1234""},{""type"":""office"",""number"":""646 555-
4567""}],""children"":[""Catherine"",""Thomas"",""Trevor""],""spouse"":null}" |
+-----------------------------------------------------
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
-------------------------+
```

`read_json()`

Описание	Читает файл `.json` и записывает прочитанные данные в таблицу.
Использование	`read_json(filename[, columns = {column_name: 'column_type', ...}])`
Псевдонимы	`read_json_auto()`

Типы данных для столбцов определяются автоматически.

Параметры

- `columns` — опциональный параметр, в котором можно указать имена столбцов и их типы. В таком случае в результирующую таблицу будут добавлены только указанные столбцы.

Подробнее о загрузке данных в Tengri можно узнать на странице Загрузка данных.

В примерах используется файл `tengri_data_types.json` с типами данных Tengri и их краткими описаниями и файл `json_example_from_wikipedia.json` с этим примером.

▼ *Посмотреть примеры*

Прочитаем файл `tengri_data_types.json` и выведем первые пять строк таблицы:

```
SELECT *
FROM read_json(
    '<path>/tengri_data_types.json'
)
LIMIT 5
```

```
+----------+-----------+----------+---------------------------+
| name     | type      | category | description               |
+----------+-----------+----------+---------------------------+
| BIGINT   | data type | numeric  | Целые числа.               |
+----------+-----------+----------+---------------------------+
| BIGINT[] | data type | array    | Массивы целых чисел.       |
+----------+-----------+----------+---------------------------+
| BLOB     | data type | blob     | Двоичные объекты.          |
+----------+-----------+----------+---------------------------+
| BOOL     | data type | boolean  | Булевы значения.           |
+----------+-----------+----------+---------------------------+
| BOOL[]   | data type | array    | Массивы булевых значений.   |
+----------+-----------+----------+---------------------------+
```

Прочитаем файл `tengri_data_types.json`, зададим только нужные колонки (ключи исходного файла) и выведем первые пять строк таблицы:

```
SELECT *
FROM read_json(
    '<path>/tengri_data_types.json',
    columns = {name: 'VARCHAR', category: 'VARCHAR'})
LIMIT 5
```

```
+----------+----------+
| name     | category |
+----------+----------+
| BIGINT   | numeric  |
+----------+----------+
```

```
| BIGINT[] | array    |
+----------+----------+
| BLOB     | blob     |
+----------+----------+
| BOOL     | boolean  |
+----------+----------+
| BOOL[]   | array    |
+----------+----------+
```

Прочитаем файл `json_example_from_wikipedia.json` и выведем всю построенную по нему таблицу. Обратим внимание на то, что вложенные структуры JSON записываются в ячейки таблицы и никак не разворачиваются (столбцы `address`, `phone_numbers`, `children`). Для разворачивания вложенных структур можно использовать функцию `unnest`.

```
SELECT *
FROM read_json(
    '<path>/json_example_from_wikipedia.json'
)
```

```
+------------+-----------+----------+-----
+-----------------------------------------------------------------------------------------
-----------+
-----------------------------------------------------------------------------------------
+--------------------------+--------+
| first_name | last_name | is_alive | age | address
| phone_numbers
| children                  | spouse |
+------------+-----------+----------+-----
+-----------------------------------------------------------------------------------------
-----------+
-----------------------------------------------------------------------------------------
+--------------------------+--------+
| John       | Smith     | true     | 27  | {"street_address": "21 2nd Street", "city":
"New York", "state": "NY", "postal_code": "10021-3100"} | {{'type': 'home', 'number': '212
555-1234'},{'type': 'office', 'number': '646 555-4567'}} | {Catherine,Thomas,Trevor} | null
|
+------------+-----------+----------+-----
+-----------------------------------------------------------------------------------------
-----------+
-----------------------------------------------------------------------------------------
+--------------------------+--------+
```

Функции для двоичных данных

Функции для двоичных данных — это функции для работы со значениями двоичного типа (`BLOB`).

concat()

Описание Конкатенирует несколько строк, массивов или двоичных значений.

Использование `concat(argument1, argument2, ...)`

Пустые значения (`NULL`) игнорируются.

См. также Оператор `||`.

▼ *Посмотреть пример*

```
SELECT
    concat('\xAA'::BLOB, '\xff'::BLOB) as result_blob,
    concat('I', ' ', 'love', ' ', 'Tengri') as result_string,
    concat(['T', 'e'], ['n', 'g', 'r', 'i']) as result_array;
```

```
+-------------+---------------+---------------+
| result_blob | result_string |  result_array |
+-------------+---------------+---------------+
| \xAA\xFF    | I love Tengri | {T,e,n,g,r,i} |
+-------------+---------------+---------------+
```

md5()

Описание Возвращает хеш MD5 данных из `argument` в виде строки (`VARCHAR`).

Использование `md5(argument)`

В `argument` могут быть двоичные данные или строка.

▼ *Посмотреть примеры*

```
SELECT
    md5('\xAA\xFF'::BLOB) as md5_hash;
```

```
+----------------------------------+
|             md5_hash             |
+----------------------------------+
| 1fab7f7621f5ddc051ebd1f2c63c4665 |
+----------------------------------+
```

```
SELECT
    md5('Tengri') as md5_hash;
```

```
+----------------------------------+
|             md5_hash             |
```

```
+----------------------------------+
| 846b02d31131a10bd6ac0ba189c65bef |
+----------------------------------+
```

sha1()

Описание Возвращает хеш SHA-1 данных из `argument` в виде строки (`VARCHAR`).

Использование `sha1(argument)`

В `argument` могут быть двоичные данные или строка.

▼ *Посмотреть примеры*

```
SELECT
    sha1('\xAA\xFF'::BLOB) as sha1_hash;
```

```
+------------------------------------------+
|                 sha1_hash                |
+------------------------------------------+
| e89b0db325637edfacde04a76005c492e2c5aeca |
+------------------------------------------+
```

```
SELECT
    sha1('Tengri') as sha1_hash;
```

```
+------------------------------------------+
|                 sha1_hash                |
+------------------------------------------+
| b514525a19995a2442d7565bfd9bb42d9dc71a13 |
+------------------------------------------+
```

sha256()

Описание Возвращает хеш SHA-256 данных из `argument` в виде строки (`VARCHAR`).

Использование `sha256(argument)`

В `argument` могут быть двоичные данные или строка.

▼ *Посмотреть пример*

```
SELECT
    sha256('\xAA\xFF'::BLOB) as sha256_hash;
```

```
+------------------------------------------------------------------+
```

```
|                            sha256_hash                             |
+--------------------------------------------------------------------+
| 768318522cac43261e8ef4946c2296a3643d523a8d5bda8ff5b82aa64470421a |
+--------------------------------------------------------------------+
```

```
SELECT
    sha256('Tengri') as sha256_hash;
```

```
+--------------------------------------------------------------------+
|                            sha256_hash                             |
+--------------------------------------------------------------------+
| 8aaacef66663b14ee7c5a03dbaec7b40f0f3bf17bd12d2ed4f9aaad0e10a0d77 |
+--------------------------------------------------------------------+
```

Оператор ||

Описание Конкатенирует несколько строк, массивов или двоичных значений.

Использование `argument1 || argument2 || ...`

Пустые значения (`NULL`) игнорируются.

См. также concat().

▼ *Посмотреть примеры*

```
SELECT
    '\xAA'::BLOB || '\xff'::BLOB as result_blob,
    'I' || ' ' || 'love' || ' ' || 'Tengri' as result_string,
    ['T', 'e'] || ['n', 'g', 'r', 'i'] as result_array;
```

```
+-------------+---------------+---------------+
| result_blob | result_string |  result_array |
+-------------+---------------+---------------+
|             | I love Tengri | {T,e,n,g,r,i} |
+-------------+---------------+---------------+
```

Обратите внимание, что значения в столбце `result_blob` в выводе не отображаются (так как имеют тип `BLOB`).

С помощью выражения `DESCRIBE` выведем типы данных для всех столбцов в таблице, созданной так же, как в предыдущем примере:

```
CREATE TABLE concat AS

SELECT
    '\xAA'::BLOB || '\xff'::BLOB as result_blob,
```

```
    'I' || ' ' || 'love' || ' ' || 'Tengri' as result_string,
    ['T', 'e'] || ['n', 'g', 'r', 'i'] as result_array;

DESCRIBE TABLE concat;
```

```
+---------------+-------------+------+------+---------+-------+
|  column_name  | column_type | null |  key | default | extra |
+---------------+-------------+------+------+---------+-------+
| result_blob   | BLOB        | YES  | null | null    | null  |
+---------------+-------------+------+------+---------+-------+
| result_string | VARCHAR     | YES  | null | null    | null  |
+---------------+-------------+------+------+---------+-------+
| result_array  | VARCHAR[]   | YES  | null | null    | null  |
+---------------+-------------+------+------+---------+-------+
```

Оконные функции

Оконные функции — это функции, которые выполняют вычисления для набора строк, некоторым образом связанных с текущей строкой. При этом в отличие от агрегатных функций, значения, вычисленные оконными функциями, могут проставляться в каждую строку исходной таблицы.

Вызов оконной функции всегда содержит выражение `OVER`, следующее за названием и аргументами оконной функции. Выражение `OVER` определяет, как именно нужно разделить строки набора данных для обработки оконной функцией.

Выражение `OVER` может дополняться выражением `PARTITION BY`, которое разделяет строки набора данных по группам, объединяя строки в группы по совпадению значений указанных столбцов. Значение оконной функции вычисляется по строкам, попадающим в одну группу с текущей строкой.

`cume_dist()`

Описание Кумулятивное распределение.

Использование `cume_dist([ORDER BY column])`

Кумулятивное распределение: количество строк группы, предшествующих текущей строке или с тем же значением, что и в текущей строке, деленное на общее количество строк группы.

Если указано условие `ORDER BY`, значение вычисляется с использованием указанного порядка.

▼ *Посмотреть пример*

```
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 2),
    ('A', 3),
```

```
    ('B', 4);

SELECT gr, num,
    cume_dist(ORDER BY num) OVER (PARTITION BY gr) AS cume_dist
    FROM t ORDER BY num;
```

```
+-------+--------+-----------+
| gr    | num    | cume_dist |
+-------+--------+-----------+
| A     | 1      | 0.25      |
+-------+--------+-----------+
| A     | 2      | 0.75      |
+-------+--------+-----------+
| A     | 2      | 0.75      |
+-------+--------+-----------+
| A     | 3      | 1         |
+-------+--------+-----------+
| B     | 4      | 1         |
+-------+--------+-----------+
```

dense_rank()

Описание	Ранг текущей строки внутри группы без пропусков.
Использование	`dense_rank()`
Псевдонимы	`rank_dense()`

Возвращает ранг текущей строки внутри группы без пропусков. По сути эта функция считает группы строк с совпадающими значениями по данному столбцу и назначает им номера внутри группы.

▼ *Посмотреть пример*

```
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 2),
    ('A', 5),
    ('B', 4);

SELECT gr, num,
    dense_rank() OVER (PARTITION BY gr ORDER BY num) AS dense_rank
    FROM t ORDER BY gr, num;
```

```
+----+--------+------------+
| gr | num    | dense_rank |
+----+--------+------------+
```

```
| A  | 1     | 1          |
+----+-------+------------+
| A  | 2     | 2          |
+----+-------+------------+
| A  | 2     | 2          |
+----+-------+------------+
| A  | 5     | 3          |
+----+-------+------------+
| B  | 4     | 1          |
+----+-------+------------+
```

`first_value()`

Описание	Возвращает значение, вычисленное по указанному выражению для первой строки группы.
Использование	`first_value(expression[ ORDER BY column][ IGNORE NULLS])`

Если указано условие `ORDER BY`, значение вычисляется с использованием указанного порядка.

Параметры

- `IGNORE NULLS` — значение вычисляется без учета строк со значением `NULL`.

▼ *Посмотреть пример*

```sql
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 3),
    ('A', NULL),
    ('B', 4);

SELECT gr, num,
    first_value(num) OVER (PARTITION BY gr) AS first_value
    FROM t ORDER BY gr, num;
```

```
+----+------+-------------+
| gr | num  | first_value |
+----+------+-------------+
| A  | 1    | 1           |
+----+------+-------------+
| A  | 2    | 1           |
+----+------+-------------+
| A  | 3    | 1           |
+----+------+-------------+
| A  | null | 1           |
+----+------+-------------+
```

```
| B  | 4    | 4           |
+----+------+-------------+
```

`lag()`

Описание	Возвращает значение, вычисленное для строки, сдвинутой на `offset` строк от текущей к началу группы.
Использование	`lag(expression[, offset[, default]][ ORDER BY column][ IGNORE NULLS])`

Если такой строки нет, возвращается значение `default` (оно должно быть совместимого по типу).

Оба аргумента, `offset` и `default`, являются опциональными. Если они не указываются, то используются следующие значения по умолчанию:

- `offset: 1`
- `default: NULL`

Если указано условие `ORDER BY`, значение вычисляется с использованием указанного порядка.

Параметры

- `IGNORE NULLS` — значение вычисляется без учета строк со значением `NULL`.

▼ *Посмотреть примеры*

```
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 3),
    ('B', 4),
    ('B', 5);

SELECT gr, num,
    lag(num) OVER (PARTITION BY gr) AS lag
    FROM t ORDER BY gr, num;
```

```
+----+-----+------+
| gr | num | lag  |
+----+-----+------+
| A  | 1   | null |
+----+-----+------+
| A  | 2   | 1    |
+----+-----+------+
| A  | 3   | 2    |
+----+-----+------+
| B  | 4   | null |
```

```
+----+-----+------+
| B  | 5   | 4    |
+----+-----+------+
```

```
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 3),
    ('B', 4),
    ('B', 5);

SELECT gr, num,
    lag(num, 1, -1) OVER (PARTITION BY gr) AS lag
    FROM t ORDER BY gr, num;
```

```
+----+-----+-----+
| gr | num | lag |
+----+-----+-----+
| A  | 1   | -1  |
+----+-----+-----+
| A  | 2   | 1   |
+----+-----+-----+
| A  | 3   | 2   |
+----+-----+-----+
| B  | 4   | -1  |
+----+-----+-----+
| B  | 5   | 4   |
+----+-----+-----+
```

`last_value()`

Описание	Возвращает значение, вычисленное по указанному выражению для последней строки группы.
Использование	`last_value(expression[ ORDER BY column][ IGNORE NULLS])`

Если указано условие `ORDER BY`, значение вычисляется с использованием указанного порядка.

Параметры

- `IGNORE NULLS` — значение вычисляется без учета строк со значением `NULL`.

▼ *Посмотреть пример*

```
CREATE TABLE t(gr VARCHAR, num BIGINT);
```

```
INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 3),
    ('A', NULL),
    ('B', 4),
    ('B', 5);

SELECT gr, num,
    last_value(num IGNORE NULLS) OVER (PARTITION BY gr) AS last_value
    FROM t ORDER BY gr, num;
```

```
+----+------+------------+
| gr | num  | last_value |
+----+------+------------+
| A  | 1    | 3          |
+----+------+------------+
| A  | 2    | 3          |
+----+------+------------+
| A  | 3    | 3          |
+----+------+------------+
| A  | null | 3          |
+----+------+------------+
| B  | 4    | 5          |
+----+------+------------+
| B  | 5    | 5          |
+----+------+------------+
```

`lead()`

Описание	Возвращает значение, вычисленное для строки, сдвинутой на `offset` строк от текущей к концу группы.
Использование	`lead(expression[, offset[, default]][ ORDER BY column][ IGNORE NULLS])`

Если такой строки нет, возвращается значение `default` (оно должно быть совместимого по типу).

Оба аргумента, `offset` и `default`, являются опциональными. Если они не указываются, то используются следующие значения по умолчанию:

- `offset: 1`
- `default: NULL`

Если указано условие `ORDER BY`, значение вычисляется с использованием указанного порядка.

Параметры

- `IGNORE NULLS` — значение вычисляется без учета строк со значением `NULL`.

▼ *Посмотреть примеры*

```
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 3),
    ('B', 4),
    ('B', 5);

SELECT gr, num,
    lead(num) OVER (PARTITION BY gr) AS lead
    FROM t ORDER BY gr, num;
```

```
+----+-----+------+
| gr | num | lead |
+----+-----+------+
| A  | 1   | 2    |
+----+-----+------+
| A  | 2   | 3    |
+----+-----+------+
| A  | 3   | null |
+----+-----+------+
| B  | 4   | 5    |
+----+-----+------+
| B  | 5   | null |
+----+-----+------+
```

```
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 3),
    ('B', 4),
    ('B', 5);

SELECT gr, num,
    lead(num, 1, -1) OVER (PARTITION BY gr) AS lead
    FROM t ORDER BY gr, num;
```

```
+----+-----+------+
| gr | num | lead |
+----+-----+------+
| A  | 1   | 2    |
+----+-----+------+
| A  | 2   | 3    |
+----+-----+------+
```

```
| A  | 3   | -1   |
+----+-----+------+
| B  | 4   | 5    |
+----+-----+------+
| B  | 5   | -1   |
+----+-----+------+
```

`nth_value()`

Описание	Возвращает значение, вычисленное для n-ой строки внутри группы (считая с 1).
Использование	`nth_value(expression, n[ ORDER BY column][ IGNORE NULLS])`

Если такой строки нет, возвращается значение `NULL`.

Если указано условие `ORDER BY`, значение вычисляется с использованием указанного порядка.

Параметры

- `IGNORE NULLS` — значение вычисляется без учета строк со значением `NULL`.

▼ *Посмотреть примеры*

```
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 3),
    ('B', 4),
    ('B', 5);

SELECT gr, num,
    nth_value(num, 3) OVER (PARTITION BY gr) AS third_value
    FROM t ORDER BY gr, num;
```

```
+----+-----+-------------+
| gr | num | third_value |
+----+-----+-------------+
| A  | 1   | 3           |
+----+-----+-------------+
| A  | 2   | 3           |
+----+-----+-------------+
| A  | 3   | 3           |
+----+-----+-------------+
| B  | 4   | null        |
+----+-----+-------------+
| B  | 5   | null        |
```

```
+----+-----+-------------+
```

ntile()

Описание Разбивает каждую группу на `num` подгрупп равного (насколько возможно) размера и возвращает номер подгруппы.

Использование `ntile(num[ ORDER BY ordering])`

Разбиение происходит так, чтобы размеры подгрупп были максимально близкими.

Если указано условие `ORDER BY`, значение вычисляется с использованием указанного порядка.

▼ *Посмотреть пример*

```
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 3),
    ('B', 4),
    ('B', 5);

SELECT gr, num,
    ntile(2) OVER (PARTITION BY gr) AS group_number
    FROM t ORDER BY gr, num;
```

```
+----+-----+--------------+
| gr | num | group_number |
+----+-----+--------------+
| A  | 1   | 1            |
+----+-----+--------------+
| A  | 2   | 1            |
+----+-----+--------------+
| A  | 3   | 2            |
+----+-----+--------------+
| B  | 4   | 1            |
+----+-----+--------------+
| B  | 5   | 2            |
+----+-----+--------------+
```

percent_rank()

Описание Вычисляет относительный ранг текущей строки внутри группы.

Использование ` percent_rank([ORDER BY column])`

Возвращает значение типа `DOUBLE` от `0` до `1` включительно. Относительный ранг вычисляется по формуле `(rank - 1) / (общее число строк группы - 1)`.

Если указано условие `ORDER BY`, значение вычисляется с использованием указанного порядка.

▼ *Посмотреть пример*

```
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 3),
    ('B', 4),
    ('B', 5);

SELECT gr, num,
    percent_rank(ORDER BY num) OVER (PARTITION BY gr) AS percent_rank,
    FROM t ORDER BY gr, num;
```

```
+----+-----+--------------+
| gr | num | percent_rank |
+----+-----+--------------+
| A  | 1   | 0            |
+----+-----+--------------+
| A  | 2   | 0.5          |
+----+-----+--------------+
| A  | 3   | 1            |
+----+-----+--------------+
| B  | 4   | 0            |
+----+-----+--------------+
| B  | 5   | 1            |
+----+-----+--------------+
```

`rank()`

Описание Возвращает ранг (с пропусками) текущей строки внутри группы.

Использование `rank([ORDER BY column])`

Если указано условие `ORDER BY`, значение вычисляется с использованием указанного порядка.

▼ *Посмотреть пример*

```
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 2),
    ('A', 3),
    ('B', 4),
    ('B', 5);
```

```
SELECT gr, num,
    rank(ORDER BY num) OVER (PARTITION BY gr) AS rank,
    FROM t ORDER BY gr, num;
```

```
+----+-----+------+
| gr | num | rank |
+----+-----+------+
| A  | 1   | 1    |
+----+-----+------+
| A  | 2   | 2    |
+----+-----+------+
| A  | 2   | 2    |
+----+-----+------+
| A  | 3   | 4    |
+----+-----+------+
| B  | 4   | 1    |
+----+-----+------+
| B  | 5   | 2    |
+----+-----+------+
```

`row_number()`

Описание Возвращает номер текущей строки в её группе (считая с `1`).

Использование `row_number([ORDER BY column])`

Если указано условие `ORDER BY`, значение вычисляется с использованием указанного порядка.

▼ *Посмотреть пример*

```
CREATE TABLE t(gr VARCHAR, num BIGINT);

INSERT INTO t VALUES
    ('A', 1),
    ('A', 2),
    ('A', 2),
    ('A', 3),
    ('B', 4),
    ('B', 5);

SELECT gr, num,
    row_number() OVER (PARTITION BY gr) AS row_number,
    FROM t ORDER BY gr, num;
```

```
+----+-----+------------+
| gr | num | row_number |
+----+-----+------------+
| A  | 1   | 1          |
+----+-----+------------+
| A  | 2   | 2          |
```

```
+----+-----+------------+
| A  | 2   | 3          |
+----+-----+------------+
| A  | 3   | 4          |
+----+-----+------------+
| B  | 4   | 1          |
+----+-----+------------+
| B  | 5   | 2          |
+----+-----+------------+
```

Утилиты

Утилиты — это функции для работы с данными различных типов, которые сложно отнести к конкретной категории. Их описания собраны в этом разделе.

`coalesce()`

Описание Возвращает первое значение, отличное от `NULL`, из списка значений аргументов.

Использование `coalesce(argument1[, argument2, ...])`

Если в единственном аргументе значение `NULL`, то возвращается `NULL`.

▼ *Посмотреть пример*

```sql
SELECT
    coalesce(NULL, 'Tengri', NULL) AS result_1,
    coalesce(NULL, '', NULL) AS result_2,
    coalesce('Tengri') AS result_3,
    coalesce(NULL) AS result_4;
```

```
+----------+----------+----------+----------+
| result_1 | result_2 | result_3 | result_4 |
+----------+----------+----------+----------+
| Tengri   |          | Tengri   | null     |
+----------+----------+----------+----------+
```

`hash()`

Описание Возвращает хеш данных из `argument` в виде числа.

Использование `hash(argument)`

▼ *Посмотреть пример*

```sql
SELECT
    hash('Tengri') AS hash;
```

```
+----------------------+
|         hash         |
+----------------------+
| 15418814193266442000 |
+----------------------+
```

`unnest()`

Описание Разворачивает списки или структуры из `argument` в множество отдельных значений.

Использование `unnest(argument) [, recursive := true] [, max_depth := <num>]`

Применение функции к списку дает одну строку на каждый элемент списка. Обычные скалярные выражения в том же выражении `SELECT` повторяются для каждой выводимой строки.

Когда несколько списков разворачиваются в одном выражении `SELECT`, они разворачиваются каждый в отдельный столбец. Если один список длиннее другого, более короткий список заполняется значениями `NULL`.

Функция изменяет кардинальность данных.

Параметры

- `recursive := true`
  Включает рекурсивный режим. Если этот режим включен (значение `true`), то функция полностью разворачивает списки, а затем полностью разворачивает вложенные структуры. Это может быть полезно для полного "уплощения" столбцов, которые содержат списки внутри списков или структуры внутри списков. Обратите внимание, что списки внутри структур не разворачиваются.

  ▼ *Подробнее о параметре на примерах*

  Покажем работу этого параметра на двух примерах с одними и теми же данными с включенным параметром и без него:

  ```
  SELECT
      unnest([[1, 2, 3], [4, 5]], recursive := true) AS result;
  ```

  ```
  +--------+
  | result |
  +--------+
  | 1      |
  +--------+
  | 2      |
  +--------+
  | 3      |
  +--------+
  ```

```
| 4      |
+--------+
| 5      |
+--------+
```

```
SELECT
    unnest([[1, 2, 3], [4, 5]]) AS result;
```

```
+---------+
|  result |
+---------+
| {1,2,3} |
+---------+
| {4,5}   |
+---------+
```

- `max_depth := <num>`
  Параметр `max_depth` позволяет ограничить максимальную глубину рекурсивного развертывания. Рекурсивный режим автоматически включен, если указана максимальная глубина.

  ▼ *Подробнее о параметре на примерах*

  Покажем работу этого параметра на трех примерах с одними и теми же данными: с глубиной развертывания по умолчанию (1), с глубиной развертывания `2` и с глубиной развертывания `3`:

```
SELECT
    unnest([[['T', 'e'], ['n', 'g']], [['r', 'i'], []], [['!', '!', '!']]])
AS result;
```

```
+-------------------------+
|          result         |
+-------------------------+
| {['T', 'e'],['n', 'g']} |
+-------------------------+
| {['r', 'i'],[]}         |
+-------------------------+
| {['!', '!', '!']}       |
+-------------------------+
```

```
SELECT
    unnest([[['T', 'e'], ['n', 'g']], [['r', 'i'], []], [['!', '!', '!']]],
    max_depth := 2)
AS result;
```

```
+---------+
|  result |
+---------+
| {T,e}   |
+---------+
| {n,g}   |
+---------+
| {r,i}   |
+---------+
| {}      |
+---------+
| {!,!,!} |
+---------+
```

```
SELECT
    unnest([[['T', 'e'], ['n', 'g']], [['r', 'i'], []], [['!', '!', '!']]],
    max_depth := 3)
AS result;
```

```
+--------+
| result |
+--------+
| T      |
+--------+
| e      |
+--------+
| n      |
+--------+
| g      |
+--------+
| r      |
+--------+
| i      |
+--------+
| !      |
+--------+
| !      |
+--------+
| !      |
+--------+
```

▼ *Посмотреть еще примеры*

```
SELECT
    unnest([1,2,3])       AS numbers,
    unnest(['a','b','c']) AS letters;
```

```
+---------+---------+
```

```
| numbers | letters |
+---------+---------+
| 1       | a       |
+---------+---------+
| 2       | b       |
+---------+---------+
| 3       | c       |
+---------+---------+
```

```
SELECT
    unnest([1,2,3])   AS numbers,
    unnest(['a','b']) AS letters;
```

```
+---------+---------+
| numbers | letters |
+---------+---------+
| 1       | a       |
+---------+---------+
| 2       | b       |
+---------+---------+
| 3       | null    |
+---------+---------+
```

```
SELECT
    unnest([{'column_a': 1, 'column_b': 84},
            {'column_a': 100, 'column_b': NULL, 'column_c':22}],
            recursive := true);
```

```
+----------+----------+----------+
| column_a | column_b | column_c |
+----------+----------+----------+
| 1        | 84       | null     |
+----------+----------+----------+
| 100      | null     | 22       |
+----------+----------+----------+
```

```
SELECT
    unnest([{'column_a': 1, 'column_b': 84},
            {'column_a': 100, 'column_b': NULL, 'column_c':22}])
AS result;
```

```
+-----------------------------------------------------+
|                        result                       |
+-----------------------------------------------------+
| {"column_a": 1, "column_b": 84, "column_c": null}   |
```

```
+-----------------------------------------------------+
| {"column_a": 100, "column_b": null, "column_c": 22} |
+-----------------------------------------------------+
```

Типы данных

	Имя типа	Псевдонимы	
Числовые типы	`BIGINT`	`INTEGER`	Целые числа
	`NUMERIC`	`DECIMAL`	Вещественные числа с заданной точностью
	`DOUBLE`	`FLOAT`	Вещественные числа с переменной точностью
Текстовый тип	`VARCHAR`	`CHAR`, `BPCHAR`, `STRING`, `TEXT`	Текстовые строки
Типы для даты и времени	`DATE`		Даты
	`TIME`		Время
	`TIMESTAMP`		Моменты времени
	`TIMESTAMPTZ`		Моменты времени с часовым поясом
Тип JSON	`JSON`		Данные в JSON
Типы для массивов	`BIGINT[]`	`INTEGER[]`	Массивы целых чисел
	`VARCHAR[]`	`CHAR[]`, `BPCHAR[]`, `STRING[]`, `TEXT[]`	Массивы текстовых строк
	`BOOL[]`	`BOOLEAN[]`	Массивы булевых значений
Двоичный тип	`BLOB`	`BYTEA`	Двоичные объекты
Логический тип	`BOOL`	`BOOLEAN`	Булевы значения

Числовые типы

Имя	Псевдоним	Диапазон	Описание
`BIGINT`	`INTEGER`	`- 2^63` … `2^63 - 1`	Целые числа
`NUMERIC`	`DECIMAL`	`WIDTH`: 37	Вещественные числа с заданной точностью
`DOUBLE`	`FLOAT`	`1E-307` … `1E+308`	Вещественные числа с переменной точностью

Целочисленный тип

- `BIGINT`
  - Псевдоним: `INTEGER`

Тип данных `BIGINT` хранит целые числа, то есть числа без дробных компонентов, различных диапазонов. Попытки сохранить значения за пределами допустимого диапазона приведут к ошибке.

Диапазон возможных значений для типа `BIGINT`: от `- 2^63` до `2^63 - 1`.

Тип с заданной точностью

- `NUMERIC`
  - Псевдоним: `DECIMAL`

Тип данных `NUMERIC (WIDTH, SCALE)` представляет собой точное десятичное значение с фиксированным десятичным разделителем — точкой.

При создании значения типа `NUMERIC` можно указать параметры `WIDTH` и `SCALE`.

- Параметр `WIDTH` определяет максимальное общее количество цифр.
- Параметр `SCALE` определяет количество цифр после десятичной точки.

Например, тип `NUMERIC(3, 2)` может содержать значение `1.23`, но не может содержать значение `12.3` или `1.234`. Если параметры `WIDTH` и `SCALE` не заданы явно, то используются значения по умолчанию: `NUMERIC(37,18)`.

Сложение, вычитание и умножение двух десятичных чисел с фиксированной точкой возвращает другое десятичное число с фиксированной точкой с требуемыми `WIDTH` и `SCALE` или выдает ошибку, если требуемая `WIDTH` превышает максимальную поддерживаемую `WIDTH`, которая составляет `37`.

Если вам требуется хранение чисел с известным количеством десятичных знаков, а также точные операции сложения, вычитания и умножения (например, для **денежных сумм**), то следует использовать тип данных с заданной точностью `NUMERIC`.

Тип с переменной точностью

- `DOUBLE`
  - Псевдоним: `FLOAT`

Тип данных `DOUBLE` — числовой тип с переменной точностью. Этот тип хранит число с плавающей десятичной точкой двойной точности (8 байтов).

Диапазон возможных значений для типа `DOUBLE`: от `1E-307` до `1E+308`.

Как и в случае с типом данных `NUMERIC`, при преобразовании литералов или приведении других типов данных к типу с переменной точностью входные данные, которые не могут быть представлены точно, сохраняются в виде приближенных значений.

В то время как умножение, сложение и вычитание для типа `NUMERIC` являются точными операциями, те же операции для типа с переменной точностью являются только приблизительными.

Если вам нужно выполнять быстрые или сложные вычисления, тип данных с переменной точностью может быть более подходящим, чем тип `NUMERIC`. Однако, если вы используете результаты этих вычислений для принятия важных решений,

> то вам следует тщательно проанализировать реализацию этих вычислений в пограничных случаях (диапазоны, бесконечные значения, антипереполнения, недопустимые операции и т.п.). Они могут обрабатываться иначе, чем вы ожидаете.

Примеры

Создадим таблицу `numbers` со столбцами, имеющими числовые типы `BIGINT`, `NUMERIC(4,3)`, `DOUBLE` и заполним одну строку значениями `2^62`, `1.1` и `3.14159265358979323846`:

```
CREATE TABLE numbers(
    bigint BIGINT,
    numeric NUMERIC(4,3),
    double DOUBLE);

INSERT INTO numbers VALUES
    (2^62, 1.1, 3.14159265358979323846);

SELECT * FROM numbers;
```

```
+--------------------+---------+-------------------+
|        bigint      | numeric |       double      |
+--------------------+---------+-------------------+
| 4611686018427388000 | 1.100   | 3.141592653589793 |
+--------------------+---------+-------------------+
```

Обратите внимание, что в программном выводе в столбце `numeric` исходное число отображается с указанной точностью (3 знака после десятичного разделителя), а в столбце `double` исходное число отображается с максимально возможной точностью, поэтому количество знаков после разделителя меньше, чем в введенном числе.

Полезные ссылки

- Числовые функции

Текстовый тип

- `VARCHAR`
  - Псевдонимы: `CHAR`, `BPCHAR`, `STRING`, `TEXT`

Описание

Для хранения текстовых строк используется тип `VARCHAR`. Этот тип позволяет хранить символы Unicode. Данные кодируются в формате UTF-8.

Чтобы задать значение текстовой строки в выражении `INSERT`, следует использовать одинарные

кавычки.

Пример

Создадим таблицу `varchar_table` и заполним текстовыми строками две ячейки:

```
CREATE TABLE varchar_table(text_column VARCHAR);

INSERT INTO varchar_table VALUES
    ('My name is Tengri'),
    ('My nickname is TNGRi');

SELECT * FROM varchar_table;
```

```
+----------------------+
|      text_column     |
+----------------------+
| My name is Tengri    |
+----------------------+
| My nickname is TNGRi |
+----------------------+
```

Обратите внимание, что в программном выводе текстовые строки отображаются без кавычек.

Полезные ссылки

- `LIKE`
- Текстовые функции
- Функции для регулярных выражений

Типы для даты и времени

Имя	Формат	Описание
`DATE`	`YYYY-MM-DD`	Даты
`TIME`	`[YYYY-MM-DD ]HH:MM[:SS][.MS]`	Время
`TIMESTAMP`	`YYYY-MM-DD hh:mm[:ss][.zzzzzz]`	Моменты времени
`TIMESTAMPTZ`	`YYYY-MM-DD hh:mm[:ss][.zzzzzz][+-TT[:tt] | ZONE]`	Моменты времени с часовым поясом

Даты

- `DATE`

Тип `DATE` используется для хранения даты (сочетание года, месяца и дня). Даты считаются по Григорианскому календарю, даже для времени до его введения.

Данные для этого типа должны быть отформатированы в соответствии со стандартом ISO 8601: `YYYY-MM-DD`.

Время

- `TIME`

Тип `TIME` используется для хранения времени (часы, минуты, секунды и микросекунды внутри суток).

Данные для этого типа должны быть отформатированы в соответствии со стандартом ISO 8601: `[YYYY-MM-DD ]HH:MM:SS[.MS]`.

Тип `TIME` следует использовать только в редких случаях, когда часть даты в метке времени может игнорироваться. В большинстве случаев для представления конкретных моментов времени следует использовать тип `TIMESTAMP`.

Моменты времени

- `TIMESTAMP`
- `TIMESTAMPTZ`

Типы `TIMESTAMP` и `TIMESTAMPTZ` используются для хранения временных меток — конкретных моментов времени. Они объединяют информацию о дате (тип `DATE`) и времени (тип `TIME`).

Для типа `TIMESTAMP` данные должны быть отформатированы в соответствии со стандартом ISO 8601: `YYYY-MM-DD hh:mm:ss[.zzzzzz]`.

Для типа `TIMESTAMPTZ` (*TIMESTAMP TIME ZONE*) данные должны быть отформатированы так же, но допускается добавление часового пояса:
`YYYY-MM-DD hh:mm:ss[.zzzzzz][+-TT[:tt] | ZONE]`.

Десятичные знаки, выходящие за пределы поддерживаемой точности, игнорируются и не приводят к ошибке.

Часовые пояса могут задаваться либо через отступ в часах от указанного времени (`+01`, `-01`), либо через явное указание идентификатора из списка часовых поясов `tz_database`.

Примеры

Создадим таблицу `dates` со столбцами, имеющими типы `DATE`, `TIME` и `TIMESTAMP` и запишем в них различные валидные значения:

```
CREATE TABLE dates(
    date DATE,
```

```
    time TIME,
    timestamp TIMESTAMP);

INSERT INTO dates VALUES
    ('2025-01-02', '1:1', '2025-01-02 1:1:00'),
    ('2025-01-02', '1:01:11', '2025-01-02 1:1:11.111'),
    ('2025-01-02', '01:01:11.1234567', '2025-01-02 1:1:11.1234567');

SELECT * FROM dates;
```

```
+------------+-----------------+----------------------------+
|    date    |       time      |          timestamp         |
+------------+-----------------+----------------------------+
| 2025-01-02 | 01:01:00        | 2025-01-02 01:01:00        |
+------------+-----------------+----------------------------+
| 2025-01-02 | 01:01:11        | 2025-01-02 01:01:11.111000 |
+------------+-----------------+----------------------------+
| 2025-01-02 | 01:01:11.123456 | 2025-01-02 01:01:11.123456 |
+------------+-----------------+----------------------------+
```

Обратите внимание, что в программном выводе в столбце `time` время отображается в стандартном виде и с обязательным значением секунд (несмотря на различные способы ввода). В столбце `timestamp` время также отображается в стандартном виде. Микросекунды отображаются, только если они были заданы и с точностью в 6 знаков.

Создадим таблицу `time_stamps` со столбцами, имеющими типы `TIMESTAMP` и `TIMESTAMPTZ` и запишем в них одни и те же пары значений — с указанием часового пояса и без:

```
CREATE TABLE time_stamps(
    timestamp TIMESTAMP,
    timestamptz TIMESTAMPTZ);

INSERT INTO time_stamps VALUES
    ('2025-01-02 0:0:0',       '2025-01-02 0:0:0'),
    ('2025-01-02 0:0:0+01:00', '2025-01-02 0:0:0+01:00');

SELECT * FROM time_stamps;
```

```
+---------------------+---------------------------+
|      timestamp      |        timestamptz        |
+---------------------+---------------------------+
| 2025-01-02 00:00:00 | 2025-01-02 00:00:00+00:00 |
+---------------------+---------------------------+
| 2025-01-01 23:00:00 | 2025-01-01 23:00:00+00:00 |
+---------------------+---------------------------+
```

Обратите внимание, что в обоих случаях указание на часовой пояс обрабатывается верно, но в

случае `TIMESTAMPTZ` оно сохраняется в таблице, а в случае типа `TIMESTAMP` — нет.

Для демонстрации работы с часовыми поясами вставим в таблицу `time_stamps` значения времени с разными указаниями часового пояса. Для наглядности вставим одни и те значения в две колонки с типами `VARCHAR` и `TIMESTAMPTZ`:

```
CREATE TABLE time_stamps(
    timestampt_text VARCHAR,
    timestamptz TIMESTAMPTZ);

INSERT INTO time_stamps VALUES
    ('2025-01-01 0:0:0 UTC', '2025-01-01 0:0:0 UTC'),
    ('2025-01-01 0:0:0 CET', '2025-01-01 0:0:0 CET'),
    ('2025-01-01 0:0:0+01' , '2025-01-01 0:0:0+01' ),
    ('2025-01-01 0:0:0-01' , '2025-01-01 0:0:0-01' ),
    ('2025-01-01 0:0:0+25' , '2025-01-01 0:0:0+25' ),
    ('2025-01-01 0:0:0-25' , '2025-01-01 0:0:0-25' );

SELECT * FROM time_stamps;
```

```
+----------------------+---------------------------+
|    timestampt_text   |        timestamptz        |
+----------------------+---------------------------+
| 2025-01-01 0:0:0 UTC | 2025-01-01 00:00:00+00:00 |
+----------------------+---------------------------+
| 2025-01-01 0:0:0 CET | 2024-12-31 23:00:00+00:00 |
+----------------------+---------------------------+
| 2025-01-01 0:0:0+01  | 2024-12-31 23:00:00+00:00 |
+----------------------+---------------------------+
| 2025-01-01 0:0:0-01  | 2025-01-01 01:00:00+00:00 |
+----------------------+---------------------------+
| 2025-01-01 0:0:0+25  | 2024-12-30 23:00:00+00:00 |
+----------------------+---------------------------+
| 2025-01-01 0:0:0-25  | 2025-01-02 01:00:00+00:00 |
+----------------------+---------------------------+
```

Попробуем ввести в таблицу `dates` невалидные данные:

```
CREATE TABLE dates(
    date DATE);

INSERT INTO dates VALUES
    ('2025-12-13'),
    ('2025-13-13');
```

```
ERROR: ConversionException: Conversion Error: date field value out of range:
"2025-13-13"
```

Данные не могут быть введены, так как задан невалидный номер месяца — `13`.

```
CREATE TABLE dates(
    time TIME);

INSERT INTO dates VALUES
    ('1:1'),
    ('24:1');
```

```
ERROR: ConversionException: Conversion Error: time field value out of range:
"24:1", expected format is ([YYYY-MM-DD ]HH:MM:SS[.MS])
```

Данные не могут быть введены, так как задано невалидное значение часа — `24`.

```
CREATE TABLE dates(
    timestamp TIMESTAMP);

INSERT INTO dates VALUES
    ('2025-01-01 1');
```

```
ERROR: ConversionException: Conversion Error: invalid timestamp field format:
"2025-01-01 1", expected format is (YYYY-MM-DD HH:MM:SS[.US][±HH:MM| ZONE])
```

Данные не могут быть введены, так как задано невалидное значение времени — `1`.

```
CREATE TABLE dates(
    timestamptz TIMESTAMPTZ);

INSERT INTO dates VALUES
    ('2025-01-01 0:0:0 Europe/Moscow'),
    ('2025-01-01 0:0:0 Asia/Srednekolymsk'),
    ('2025-01-01 0:0:0 MSK');
```

```
ERROR: NotImplementedException: Not implemented Error: Unknown TimeZone 'MSK'
```

Данные не могут быть введены, так как задано невалидное значение часового пояса `MSK`. Значения

`Europe/Moscow` и `Asia/Srednekolymsk` при этом являются валидными, так как входят в список идентификаторов часовых поясов `tz_database`.

Полезные ссылки

- Функции для даты и времени

Тип JSON

- `JSON`

Описание

Тип `JSON` используется для хранения данных JSON согласно стандартному синтаксису, описанному в спецификации.

Для хранения таких данных можно использовать и тип `VARCHAR`, но при использовании типа `JSON` данные будут проверяться на соответствие вводимых значений формату JSON, поэтому использовать специальный тип `JSON` в таких случаях удобнее. К тому же для типа `JSON` доступны специальные функции, позволяющие обращаться на прямую к данным в структуре JSON.

Кавычки внутри текста в формате JSON должны быть двойными. А обрамляющие этот текст кавычки внутри выражения `INSERT` должны быть одинарными (см. пример ниже).

Примеры

Создадим таблицу `js_table` и вставим в столбец `js_data` данные в формате JSON из этого примера:

```
CREATE TABLE js_table(name VARCHAR, js_data JSON);

INSERT INTO js_table VALUES
('John Smith',
'{
  "first_name": "John",
  "last_name": "Smith",
  "is_alive": true,
  "age": 27,
  "address": {
    "street_address": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postal_code": "10021-3100"
  },
  "phone_numbers": [
    {
      "type": "home",
```

```
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [
    "Catherine",
    "Thomas",
    "Trevor"
  ],
  "spouse": null
}');
```

Выведем имена полей верхнего уровня из загруженных данных JSON с помощью функции `json_keys`:

```
SELECT
    json_keys(js_data) AS json_fields
FROM js_table;
```

```
+-----------------------------------------------------------------------------+
|                                json_fields                                  |
+-----------------------------------------------------------------------------+
| {first_name,last_name,is_alive,age,address,phone_numbers,children,spouse} |
+-----------------------------------------------------------------------------+
```

Выведем в отдельные столбцы полное имя из текстового столбца `name`, а из данных JSON возьмем возраст `age` и количество детей — длину массива в поле `children`. Для этого используем функции `json_extract` и `json_array_length`.

```
SELECT
    name,
    json_extract(js_data, 'age') AS age,
    json_array_length(js_data, 'children') AS children_num
FROM js_table;
```

```
+------------+-----+--------------+
|    name    | age | children_num |
+------------+-----+--------------+
| John Smith | 27  | 3            |
+------------+-----+--------------+
```

Полезные ссылки

- Функции для JSON

Типы для массивов

- `BIGINT[]`
  - Псевдоним: `INTEGER[]`
- `VARCHAR[]`
  - Псевдонимы: `CHAR[]`, `BPCHAR[]`, `STRING[]`, `TEXT[]`
- `BOOL[]`
  - Псевдонимы: `BOOLEAN[]`

Описание

Типы `BIGINT[]`, `VARCHAR[]` и `BOOL[]` используются для хранения массивов целых чисел, массивов строк и массивов булевых значений соответственно.

Длина массивов в одном столбце таблицы может не совпадать.

Массивы могут использоваться для хранения векторов, таких как эмбеддинги (векторные представления) текстов или изображений.

Примеры

Создадим таблицу `array_table` и вставим в нее значения типа `VARCHAR` и типа `VARCHAR[]`:

```
CREATE TABLE array_table(text_column VARCHAR, array_column VARCHAR[]);

INSERT INTO array_table VALUES
    ('I love Tengri', ['I','love', 'Tengri']),
    ('I adore Tengri', ['I','adore', 'Tengri']);

SELECT * FROM array_table;
```

```
+----------------+------------------+
|   text_column  |   array_column   |
+----------------+------------------+
| I love Tengri  | {I,love,Tengri}  |
+----------------+------------------+
| I adore Tengri | {I,adore,Tengri} |
+----------------+------------------+
```

Обратите внимание, что в программном выводе массивы отображаются через фигурные скобки `{}`, а текстовые значения в них — без кавычек.

Теперь создадим таблицу `array_table` со столбцами типа `BIGINT` и `BIGINT[]` и вставим в нее числовые значения и массивы чисел:

```
CREATE TABLE array_table(number_column BIGINT, array_column BIGINT[]);

INSERT INTO array_table VALUES
    (2025, [2,0,2,5]),
    (0025, [0,0,2,5]);

SELECT * FROM array_table;
```

```
+---------------+--------------+
| number_column | array_column |
+---------------+--------------+
| 2025          | {2,0,2,5}    |
+---------------+--------------+
| 25            | {0,0,2,5}    |
+---------------+--------------+
```

Обратите внимание, что числовое значение `0025` в программном выводе приведено к нормальному виду `25`, а массивы в программном выводе отображаются через фигурные скобки `{}`.

Двоичный тип

- `BLOB`
    - Псевдоним: `BYTEA`

Описание

Тип данных `BLOB` предназначен для хранения произвольных двоичных объектов. Он может содержать любые двоичные данные без каких-либо ограничений. Что на самом деле представляют собой эти данные, для Tengri остается неизвестным.

`BLOB` обычно используются для хранения нетекстовых объектов, тип которых явно не поддерживается, например для изображений. Хотя тип `BLOB` может хранить объекты размером до 4 ГБ, не рекомендуется хранить очень большие объекты в Tengri. Обычно более удобно хранить большие файлы в файловой системе, а пути к файлам хранить в Tengri в виде типа `VARCHAR`.

Полезные ссылки

- Функции для двоичных данных

Логический тип

- `BOOL`
  - Псевдоним: `BOOLEAN`

Описание

Тип `BOOL` представляет собой утверждение истинности и может принимать значения:
`true`, `false` (булевы значения) или `NULL`.

Булевы значения могут быть явно заданы с помощью литералов `true` и `false`. Однако чаще всего они создаются в результате сравнения. Например, сравнение `i > 10` возвращает булево значение.

Булевы значения могут использоваться в выражениях `WHERE` и `HAVING` для фильтрации результатов. В этом случае выражения, результат которых оценивается как `true`, пройдут фильтр, а выражения, результат которых оценивается как `false` или `NULL`, будут отброшены.

Логические операторы

Для комбинирования булевых значений могут использоваться логические операторы `AND` и `OR`.

Table 1. Таблица истинности для оператора `AND`

X	X AND true	X AND false	X AND NULL
`true`	`true`	`false`	`NULL`
`false`	`false`	`false`	`false`
`NULL`	`NULL`	`false`	`NULL`

Table 2. Таблица истинности для оператора `OR`

X	X OR true	X OR false	X OR NULL
`true`	`true`	`true`	`true`
`false`	`true`	`false`	`NULL`
`NULL`	`true`	`NULL`	`NULL`