
SQLite для аналитики: шпаргалка

Антон Жиянов

Это шпаргалка по курсу [SQLite для аналитики](#). Используйте ее, чтобы освежить в памяти материал или возобновить курс, не начиная заново.

Основы SQLite

Утилита `sqlite3` работает с базами данных SQLite. Она поддерживает язык SQL и специальные команды (начинаются с точки). Главная команда — `.help`.

```
.help КОМАНДА
```

Команда `.import` загружает данные из CSV-файла в таблицу.

```
.import --csv city.csv city
```

Формат CSV настраивается командами `.headers` и `.separator`.

```
.mode csv  
.headers on  
.separator |  
.import samara.csv samara
```

Команда `.once` в связке с `select` выгружает данные из таблицы в файл.

```
.mode csv  
.once city.csv  
select * from city;
```

Формат файла настраивается командами `.mode`, `.headers` и `.separator`.

```
.mode json  
.mode insert  
.mode markdown  
.mode html
```

SQLite поддерживает стандартный SQL-синтаксис:

- фильтрацию по условиям в `WHERE`,
- группировку `GROUP BY` и сортировку `ORDER BY`,

- табличные выражения `WITH name AS (SELECT ...)`.

Очистка данных

Как почистить набор данных:

1. Создать таблицу с правильными типами столбцов.
2. Загрузить данные.
3. Привести проблемные значения к правильным типам.
4. Занулить мусорные значения.
5. Заполнить пробелы в данных.
6. Выгрузить чистые данные.

Типы данных в SQLite: INTEGER, REAL, TEXT, BLOB и NULL. Даты хранятся как TEXT в формате `гггг-мм-дд`, а логические значения — как INTEGER 0 или 1.

Основные функции:

- Числа: `min`, `max`, `random`.
- Строки: `length`, `instr`, `substr`, `replace`.
- Даты: `date`.
- Логические: `iif`.
- NULL: `coalesce`, `nullif`.

INTEGER / REAL

Действительные и целые числа.

```
min(x, y, ...)  
max(x, y, ...)  
random()
```

TEXT

Строки в кодировке UTF-8. Безразмерные.

функции:

```
length(s)  
instr(s, substring)  
substr(s, from, length)  
substr(s, from)  
replace(s, substring, replacement)
```

поиск по шаблону:

значение GLOB 'шаблон'

? один символ
* что угодно
[abc] один из перечисленных
[^abc] кроме перечисленных
[a-z] один из диапазона
[^a-z] один не из диапазона

Даты

Хранятся текстом в формате ISO: гггг-мм-дд

сравнение:

```
d = '2020-06-01'  
d > '2020-06-01'  
d between '2020-06-01' and '2020-06-30'
```

функции:

```
date(value, modifier, modifier, ...)  
date('now')  
date('now', '1 day')  
date('now', '-1 day')  
date('2020-09-01', '-3 month')  
date('2020-12-19', '-1 year')
```

NULL

Пустое значение.

сравнение:

```
x is null  
x is not null
```

функции:

```
-- первый, кто не NULL  
coalesce(x, y, ...)  
  
-- NULL, если равны, иначе X  
nullif(x, y)
```

Логические

Логические значения: true = 1, false = 0

```
-- если X, то Y, иначе Z  
iif(x, y, z)
```

Связи в данных

Чаще всего в данных встречаются связи между таблицами «один ко многим». Так бывают связаны основная таблица с классификатором (вакансия ↔ график работы) или несколько основных таблиц (вакансия ↔ работодатель).

```
select ...  
from vacancy  
  join employer on vacancy.employer_id = employer.id
```

Связь «один ко многим» таблицы по отношению к самой себе (регионы) называется иерархией. Если понятно, сколько в иерархии уровней, ее можно развернуть в плоскую структуру. Исследовать иерархию поможет рекурсивный запрос WITH RECURSIVE.

```
with recursive tmp(столбец_1, столбец_2, ...) as (  
  -- база рекурсии  
  select ...  
  from ...  
  where ...  
  
  union all  
  
  -- рекурсивный шаг  
  select ...  
  from ... join tmp on ...  
)  
  
select * from tmp;
```

Если таблицы связаны не напрямую, а через третью таблицу — это отношение «многие ко многим» (работодатель — регион). Третьей таблицей может быть специальная таблица связки (employer_area) или одна из основных таблиц (vacancy). В любом случае, нужно следить, чтобы связка содержала только уникальные пары идентификаторов (employer_id, area_id) — иначе запросы будут врать.

```
select ...  
from area  
  join employer_area on area.id = employer_area.area_id  
  join employer on employer_area.employer_id = employer.id
```

С результатами селектов можно работать как с математическими множествами:

- объединять (UNION) — записи, которые вернул хотя бы один из запросов;
- пересекать (INTERSECT) — записи, которые вернули все запросы;
- вычитать (EXCEPT) — записи, которые вернул первый запрос, но не второй.

```
select employer_id from employer_area
where area_id = 1
except
select employer_id from employer_area
where area_id = 2;
```

Вычисляемые столбцы (ADD COLUMN ... AS) и представления (CREATE VIEW) помогают не дублировать частые операции в запросах.

```
alter table ТАБЛИЦА
add column СТОЛБЕЦ ТИП as (ВЫРАЖЕНИЕ);
```

```
create view ПРЕДСТАВЛЕНИЕ as
select ...
from ... join ...
where ...
```

Временные таблицы (CREATE TEMPORARY TABLE) подходят, чтобы покрутить данные так и сяк, а потом выкинуть.

```
create temporary table ТАБЛИЦА as
select ...
from ... join ...
where ...
```

Данные → знания

Числовые столбцы

Проанализировать числовой столбец помогут такие показатели:

- количество пустых значений,
- количество нулей,
- минимальное и максимальное значения,
- медиана и процентиля,
- среднее арифметическое и стандартное отклонение,
- диаграмма распределения значений.

```
select
    sum(iif(num_pages is null, 1, 0)) as nulls,
    sum(iif(num_pages = 0, 1, 0)) as zeros,
    min(num_pages) as min,
```

```

round(avg(num_pages)) as mean,
max(num_pages) as max,
count(distinct num_pages) as uniq
from books;

```

```

.load ./sqlite3-stats

```

```

select
  percentile_25(num_pages) as p25,
  median(num_pages) as p50,
  mode(num_pages) as mode,
  percentile_75(num_pages) as p75,
  percentile_90(num_pages) as p90,
  percentile_95(num_pages) as p95,
  percentile_99(num_pages) as p99
from books;

```

Медиана характеризует «среднего представителя», а 90-й (95-й, 99-й) процентиль — «большинство» значений столбца.

Среднее арифметическое и стандартное отклонение характеризуют примерно то же самое. Но медиана и процентиля проще и точнее, поэтому лучше использовать их.

Диаграмма распределения показывает, как значения столбца распределены по интервалам. Пригодится, чтобы увидеть все значения «с высоты птичьего полета».

Корреляция помогает понять, связаны два столбца между собой или нет.

$$(\text{avg}(x*y) - \text{avg}(x) * \text{avg}(y)) / (\text{stddev_pop}(x) * \text{stddev_pop}(y))$$

Категории

Столбцы-категории содержат небольшое количество уникальных текстовых значений (язык книги, статус заказа, жанр фильма). Проанализировать их помогут такие показатели:

- количество пустых значений,
- количество уникальных значений,
- количество записей для каждой категории.

```

select
  sum(iif(language_code is null, 1, 0)) as nulls,
  count(distinct language_code) as uniq,
  count(*) as total,
  round(
    count(distinct language_code)*100.0 / count(*)
  ) as uniq_percent
from books;

```

Полезно «привязать» категории к числовым столбцам, чтобы использовать весь арсенал — средние, процентиля, распределение и корреляцию.

Текстовые столбцы

У текстовых столбцов можно проверить:

- количество пустых значений,
- «самые-самые» значения,
- распределение по длине значений.

Хорошо, если получится найти закономерности в тексте, которые влияют на числовые показатели («книги со словом `success` в названии получают более высокие оценки»).

JSON

JSON-документ состоит из обычных значений, объектов и массивов.

Функция `json_extract(json, selector)` извлекает из JSON-документа конкретное значение, которое лежит по указанному в `selector` пути.

Функция `json_each(json)` парсит JSON и обходит объекты верхнего уровня. Применять ее имеет смысл, когда исходный JSON — массив (а не объект или обычное значение).

Альтернативная версия `json_each(json, path)` обходит «детей» селектора `path`.

```
select
  json_extract(value, '$.code') as code,
  json_extract(value, '$.name') as name
from
  json_each(readfile('currency.sample.json'))
```

Функция `json_tree(json)` парсит JSON и обходит все уровни. Альтернативная версия `json_tree(json, path)` обходит «потомков» селектора `path`.

```
select
  json_extract(value, '$.id') as id,
  json_extract(value, '$.name') as name
from
  json_tree(readfile('industry.sample.json'))
where
  path like '$[%].industries'
```

`json_each()` и `json_tree()` возвращают виртуальную таблицу с такими столбцами:

- `value` — конкретное JSON-значение;
- `fullkey` — селектор значения;
- `path` — селектор родителя.

Функция `readfile(path)` считывает содержимое файла с диска одним куском.

Чтобы загрузить «плоский» JSON-документ, достаточно использовать `json_each()` + `json_extract()`.

Чтобы загрузить иерархический JSON-документ, придется использовать `json_tree()` + `json_extract()` + фильтр по `path`.

Большие наборы

Чтобы загрузить большой набор данных, подходит обычная команда `.import`. Главное проследить, чтобы на таблице не было индексов в момент загрузки. Если датасет совсем большой, можно ускорить импорт, используя виртуальную CSV-таблицу (модуль `sqlite3-vsv`).

```
.load ./sqlite3-vsv

create virtual table temp.blocks_csv using vsv(
    filename="ipblocks.csv",
    schema="create table x(network text, geoname_id integer)",
    columns=2,
    header=on,
    nulls=on
);

insert into blocks
select * from blocks_csv;
```

Когда движок базы выполняет SQL-запрос, он может обойти таблицу целиком (фулскан) или использовать индекс (отсортированное дерево, в котором очень быстро найти любое значение). Индекс подходит, когда условие запроса селективное (выбирает небольшой процент записей таблицы). В противном случае быстрее отработает фулскан.

```
create index blocks_postal_idx on blocks(postal_code);

explain query plan
select * from blocks
where postal_code = ?;
```

```
QUERY PLAN
`--SEARCH TABLE blocks USING INDEX blocks_postal_idx (postal_code=?)
```

В индексе хранятся значения столбца, по которому он построен. Рядом с каждым значением хранится `rowid` — уникальный идентификатор записи в таблице. По `rowid` движок переходит от значения в индексе к строке таблицы, из которой взято значение.

Индекс можно построить:

- на одном или нескольких столбцах;
- с отбором по значению столбца (частичный индекс);

- по выражению (индексируется не значение столбца, а результат вызова функции на этом значении);
- так, чтобы движку вообще не пришлось обращаться к таблице (покрывающий индекс).

```
-- с отбором
create index blocks_is_anonymous_proxy_idx
on blocks(is_anonymous_proxy)
where is_anonymous_proxy = 1;

-- по выражению
create index blocks_postal_idx on blocks(
    substr(postal_code, 1, 2)
);
```

Механизм и виды индексов примерно одинаковы во всех современных СУБД, так что подходы из этого модуля можно использовать в Oracle, PostgreSQL, MariaDB и SQL Server.

Чтобы выгрузить данные из таблицы, подходят команды `.mode + .once + select`.

```
.mode csv
.headers on
.once blocks.csv
select * from blocks;
```

Можно экспортировать «голову» и «хвост» таблицы с помощью сортировки по `rowid` и ограничению `limit N`.

```
select * from blocks
order by rowid limit 1000;

select * from blocks
order by rowid desc limit 1000;
```

Или выгрузить семпл через остаток от деления `rowid` или `random()` на `N`.

```
select * from blocks
where rowid % 1000 = 0;

select * from blocks
where random() % 1000 = 0;
```

Создать бинарную копию базы поможет `vacuum into`, а текстовую — `.dump`.

```
vacuum main into 'geoip.backup.db';
```

Оконные функции

Вот задачи, которые решаются с помощью оконных функций в SQL:

1. Ранжирование (всевозможные рейтинги).
2. Сравнение со смещением (соседние элементы и границы).
3. Агрегация (количество, сумма и среднее).
4. Скользящие агрегаты (сумма и среднее в динамике).

Есть еще статистические оконные функции, но мы их не рассматривали.

Оконные функции вычисляют результат по строкам, которые попали в окно. Определение окна указывает, как выглядит окно:

1. Из каких секций состоит (`partition by`).
2. Как отсортированы строки внутри секции (`order by`).
3. Как выглядит фрейм внутри секции (`rows between`).

```
window w as (  
    partition by ...  
    order by ...  
    rows between ... and ...  
)
```

`partition by` поддерживается всеми оконными функциями и всегда необязательно. Если не указать — будет одна секция.

`order by` поддерживается всеми оконными функциями. Для функций ранжирования и смещения оно обязательно, для агрегации — нет. Если не указать `order by` для функции агрегации — она посчитает обычный агрегат, если указать — скользящий.




Фрейм поддерживается только некоторыми функциями:






- `first_value()`, `last_value()`, `nth_value()`;
- функции агрегации.

Остальные функции фреймы не поддерживают.

SQLite реализует оконные функции точно так же, как PostgreSQL, так что если придется работать с постгресом — вам уже все известно.

Функции ранжирования

-  — необязательно
-  — обязательно
-  — не поддерживается

Функция	Секции	Сортировка	Фрейм	Описание
<code>row_number()</code>				порядковый номер строки
<code>dense_rank()</code>				ранг строки
<code>rank()</code>				ранг с пропусками

Функция	Секции	Сортировка	Фрейм	Описание
ntile(n)				номер группы

Функции смещения

Функция	Секции	Сортировка	Фрейм	Описание
lag(value, n)				значение из n-й строки назад
lead(value, n)				значение из n-й строки вперед
first_value(value)				значение из первой строки фрейма
last_value(value)				значение из последней строки фрейма
nth_value(value, n)				значение из n-й строки фрейма

Функции агрегации

Функция	Секции	Сортировка	Фрейм	Описание
min(value)				минимальное из секции или фрейма
max(value)				максимальное из секции или фрейма
count(value)				количество по секции или фрейму
avg(value)				среднее по секции или фрейму
sum(value)				сумма по секции или фрейму
group_concat(val, sep)				строковое соединение по секции или фрейму