СУБД Tantor SE. Optimized Row Columnar изнутри

Дата выпуска: 2023-06-21

Вступление

Данный документ содержит общую информацию об устройстве и рекомендации по тонкой настройке с Optimized Row Columnar (pg_columnar). Данный документ предназначен для оптимизации подходов к хранению данных, в зависимости от их природы (например, time series).

Общие рекомендации

Выбор Stripe Size и Chunk Size

Определение оптимального размера полосы и блока для columnar-таблиц в Tantor SE зависит от различных факторов, таких как характер данных, шаблоны запросов и характеристики оборудования. Вот несколько рекомендаций:

1. **Размер полосы (Stripe Size)**: Оптимальный размер полосы может зависеть от типичного размера вашего запроса. Если запросы обычно возвращают большое количество строк, то больший размер полосы может быть более эффективным, так как это уменьшит количество операций ввода-вывода. С другой стороны, если запросы часто возвращают небольшое подмножество данных, то меньший размер полосы может быть предпочтительнее.

2. **Размер блока (Chunk Size)**: Это определяет, сколько данных будет сжато за один раз. Меньший размер блока может привести к более высокому коэффициенту сжатия, но может увеличить накладные расходы на сжатие. Больший размер блока может уменьшить накладные расходы, но потенциально снизить коэффициент сжатия.
3. **Тестирование и настройка**: Важно провести тестирование с реальными данными и запросами, чтобы определить оптимальные параметры для вашей конкретной ситуации. Можно начать с настройки, рекомендованной по умолчанию, а затем поэкспериментировать с различными размерами полос и блоков, чтобы увидеть, как это влияет на производительность запросов и коэффициенты сжатия.

4. **Свойства оборудования**: Также стоит учитывать характеристики вашего оборудования, такие как пропускная способность диска и процессора, так как это может влиять на то, какие размеры полос и блоков будут наиболее эффективными.

В конечном итоге, оптимальные размеры полос и блоков будут зависеть от уникальных характеристик вашей среды, данных и шаблонов запросов.

Алгоритмы сжатия

Выбор алгоритма сжатия для columnar-таблиц в Tantor зависит от нескольких факторов, включая характер данных, требования к производительности и характеристики аппаратного обеспечения. Вот некоторые рекомендации, которые могут помочь вам принять решение:

1. **none** - Этот тип не применяет никакого сжатия к данным. Он может быть полезен, если ваши данные уже сжаты или если у вас очень высокие требования к производительности и у вас достаточно дискового пространства.
2. **lz4** - LZ4 обеспечивает быстрое сжатие и разжатие данных. Это может быть полезно, если у вас есть высокие требования к производительности, но вы все равно хотите сэкономить некоторое дисковое пространство.
3. **zstd** - Zstandard обеспечивает более высокий коэффициент сжатия по сравнению с LZ4, но при этом требует больше процессорного времени для сжатия и разжатия данных. Этот алгоритм может быть полезен, если у вас ограниченное дисковое пространство, и вы готовы потратить немного больше процессорного времени на сжатие данных.

Важно отметить, что выбор алгоритма сжатия - это компромисс между производительностью (скоростью сжатия и разжатия) и дисковым пространством. Также эффективность каждого алгоритма сжатия может сильно зависеть от характера ваших данных. Поэтому рекомендуется провести некоторые тесты с вашими реальными данными и запросами, чтобы определить наиболее подходящий алгоритм сжатия для вашей ситуации.

Работа с Time Series данными

Создание таблиц

Для Time Series данных характерна последовательная запись значений по возрастанию времени. Для таких данных важна упорядоченность.

1. Давайте создадим тестовую таблицу:

```
postgres=# CREATE TABLE perf_columnar(id INT8,
ts TIMESTAMPTZ,
customer_id INT8,
vendor_id INT8,
name TEXT,
description TEXT,
value NUMERIC,
quantity INT4) USING columnar;

ALTER TABLE public.perf_columnar SET
  (columnar.compression = lz4, columnar.stripe_row_limit = 100000,
columnar.chunk_group_row_limit = 10000);
```

2. Создадим функцию для генерации случайного текста:

```
postgres=# CREATE OR REPLACE FUNCTION random_words(n INT4) RETURNS
TEXT LANGUAGE plpython3u AS $$
import random
t = ''
words =
['zero','one','two','three','four','five','six','seven','eight','nine
','ten']
for i in xrange(0,n):
  if (i != 0):
    t += ' '
  r = random.randint(0,len(words)-1)
  t += words[r]
return t
$$;
```

3. Сгенерируем тестовые данные:

```
postgres=# INSERT INTO perf_columnar
   SELECT
    g, -- id
    '2023-06-01'::timestamptz + ('1 minute'::interval * g), -- ts
    (random() * 1000000)::INT4, -- customer_id
    (random() * 100)::INT4, -- vendor_id
    random_words(7), -- name
    random_words(100), -- description
```

```
    (random() * 100000)::INT4/100.0, -- value
    (random() * 100)::INT4 -- quantity
   FROM generate_series(1,7500000) g;
```

Как видно из запроса выше, мы вставили данные отсортированные по `ts`. Соберем статистику с помощью VACUUM (ANALYZE, VERBOSE).

```
VACUUM ANALYZE perf_columnar;
```

4. Создадим копию таблицы с другими параметрами:

```
CREATE TABLE perf_columnar2(LIKE perf_columnar) USING COLUMNAR;

ALTER TABLE public.perf_columnar2 SET
  (columnar.compression = zstd, columnar.stripe_row_limit = 10000,
columnar.chunk_group_row_limit = 1000);

INSERT INTO perf_columnar2 SELECT * FROM perf_columnar;

VACUUM ANALYZE perf_columnar2;
```

5. Проверим размер получившихся таблиц:

```
postgres=# \dt+
                                    Список отношений
 Схема  |      Имя       |   Тип   | Владелец |  Хранение  | Метод доступа | Размер  | Описание
--------+----------------+---------+----------+------------+---------------+---------+----------
 public | perf_columnar  | таблица | postgres | постоянное | columnar      | 1886 MB |
 public | perf_columnar2 | таблица | postgres | постоянное | columnar      | 847 MB  |
(2 строки)
```

Как видно, таблица `perf_columnar` в два раза больше, чем `perf_columnar2`.

Проверим опции:

```
postgres=# SELECT * FROM columnar.options;
    relation    | chunk_group_row_limit | stripe_row_limit | compression | compression_level
----------------+-----------------------+------------------+-------------+-------------------
 perf_columnar  |                 10000 |           100000 | lz4         |                 3
 perf_columnar2 |                  1000 |            10000 | zstd        |                 3
(2 строки)
```

Выполнение запросов

```
postgres=# explain (analyze, verbose, buffers) select ts from
perf_columnar where ts > '2023-06-01 10:00:00'::timestamp with time
zone and ts < '2023-06-01 10:00:05'::times
tamp with time zone;

QUERY PLAN
------------------------------------------------------------------------
------------------------------------------------------------------------
-----------------------
 Custom Scan (ColumnarScan) on public.perf_columnar
(cost=0.00..401.12 rows=4 width=8) (actual time=18.838..49.001 rows=4
loops=1)
   Output: ts
   Filter: ((perf_columnar.ts > '2023-06-01 10:00:00+03'::timestamp
with time zone) AND (perf_columnar.ts < '2023-06-01
10:00:05+03'::timestamp with time zone))
   Rows Removed by Filter: 9996
   Columnar Projected Columns: ts
   Columnar Chunk Group Filters: ((ts > '2023-06-01
10:00:00+03'::timestamp with time zone) AND (ts < '2023-06-01
10:00:05+03'::timestamp with time zone))
   Columnar Chunk Groups Removed by Filter: 749
   Buffers: shared hit=3833 read=264
 Query Identifier: 1607994334608619710
 Planning:
   Buffers: shared hit=52 read=7
 Planning Time: 12.789 ms
 Execution Time: 49.188 ms
(13 строк)

postgres=# explain (analyze, verbose, buffers) select ts from
perf_columnar2 where ts > '2020-01-01 10:00:00'::timestamp with time
zone and ts < '2020-01-01 10:00:05'::time
stamp with time zone;

QUERY PLAN
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------
 Custom Scan (ColumnarScan) on public.perf_columnar2
(cost=0.00..17.95 rows=52 width=8) (actual time=5.175..58.532 rows=49
loops=1)
```

```
   Output: ts
   Filter: ((perf_columnar2.ts > '2020-01-01 10:00:00+03'::timestamp
with time zone) AND (perf_columnar2.ts < '2020-01-01
10:00:05+03'::timestamp with time zone))
   Rows Removed by Filter: 951
   Columnar Projected Columns: ts
   Columnar Chunk Group Filters: ((ts > '2020-01-01
10:00:00+03'::timestamp with time zone) AND (ts < '2020-01-01
10:00:05+03'::timestamp with time zone))
   Columnar Chunk Groups Removed by Filter: 7499
   Buffers: shared hit=40824 read=1
 Query Identifier: -801549076851693482
 Planning:
   Buffers: shared hit=155
 Planning Time: 1.086 ms
 Execution Time: 58.717 ms
(13 строк)
```

Обратите внимание `Columnar Chunk Groups Removed by Filter: 749` в первом случае и `7499` во втором. При этом количество прочитанных буфферов во втором запросе гораздо больше.

Во-первых, нам нужна терминология, чтобы понять это:

Stripe: все загрузки в columnar таблицу разбиваются на полосы по 150000 строк (по умолчанию). Чем больше полоса, тем более последовательный доступ при чтении данного столбца.

Chunk Group: полосы разбиваются на группы фрагментов по 10000 строк (по умолчанию).

Chunk : Каждая группа фрагментов состоит из одного фрагмента для каждого столбца. Chunk — это единица сжатия, и минимальные/максимальные значения отслеживаются для каждого фрагмента, чтобы включить фильтрацию группы фрагментов(Chunk Group Filtering).

Chunk Group Filtering: когда предложение запроса `WHERE` не соответствует ни одному из кортежей в фрагменте, и мы можем это определить по минимальному/максимальному значению для фрагмента, в этом случае фильтрация группы фрагментов просто пропускает всю группу фрагментов без распаковки.

Вы можете видеть выше, что `749/7499` групп фрагментов были отфильтрованы, что означает, что `7490000/7499000` строк были отфильтрованы без необходимости

извлечения или распаковки данных. Только 1 группа фрагментов (10000 и 1000 строк) необходимо было извлечь и распаковать, поэтому запрос занял всего миллисекунды.

Но как видно из планов запросов, в первом случае было использовано 30 MB, а вот втором в десять раз больше - 319 MB.

Использование индексов

Columnar поддерживает индексы btree и hash (и ограничения, требующие их), но не поддерживает gist, gin, индексы spgist и brin.

Давайте создадим индексы для наших таблиц:

```
create index test on perf_columnar2 (ts);
create index test_h on perf_columnar (ts);
```

Проверим их размер:

```
postgres=# \di+
                                        Список отношений
 Схема  |  Имя   |  Тип   | Владелец |    Таблица     | Хранение  | Метод доступа | Размер | Описание
--------+--------+--------+----------+----------------+-----------+---------------+--------+----------
 public | test   | индекс | postgres | perf_columnar2 | постоянное | btree         | 161 MB |
 public | test_h | индекс | postgres | perf_columnar  | постоянное | btree         | 161 MB |
(2 строки)
```

Размер индексов одинаковый.

Выключим принудительное использование Custom Scan:

```
SET columnar.enable_custom_scan TO OFF;
```

И выполним наши запросы:

```
postgres=# explain (analyze, verbose, buffers) select ts from
perf_columnar where ts > '2023-06-01 10:00:00'::timestamp with time
zone and ts < '2023-06-01 10:00:05'::times
tamp with time zone;

QUERY PLAN
------------------------------------------------------------------------
------------------------------------------------------------------------
-------------------------
```

```
 Index Scan using test_h on public.perf_columnar  (cost=0.43..3217.48
rows=4 width=8) (actual time=402.144..402.204 rows=4 loops=1)
   Output: ts
   Index Cond: ((perf_columnar.ts > '2023-06-01
10:00:00+03'::timestamp with time zone) AND (perf_columnar.ts <
'2023-06-01 10:00:05+03'::timestamp with time zone))
   Buffers: shared hit=181 read=3232
 Query Identifier: 1607994334608619710
 Planning:
   Buffers: shared hit=20 read=5
 Planning Time: 16.278 ms
 Execution Time: 402.386 ms
(9 строк)
```

Просканировано практически столько же буфферов, как и при использовании Castom Scan.

```
postgres=# explain (analyze, verbose, buffers) select ts from
perf_columnar2 where ts > '2020-01-01 10:00:00'::timestamp with time
zone and ts < '2020-01-01 10:00:05'::timestamp with time zone;

QUERY PLAN
------------------------------------------------------------------------
------------------------------------------------------------------------
----------------------------
 Index Scan using test on public.perf_columnar2  (cost=0.43..153.05
rows=52 width=8) (actual time=14.620..14.821 rows=49 loops=1)
   Output: ts
   Index Cond: ((perf_columnar2.ts > '2020-01-01
10:00:00+03'::timestamp with time zone) AND (perf_columnar2.ts <
'2020-01-01 10:00:05+03'::timestamp with time zone))
   Buffers: shared hit=372 read=145
 Query Identifier: -801549076851693482
 Planning:
   Buffers: shared hit=97
 Planning Time: 0.813 ms
 Execution Time: 14.978 ms
(9 строк)
```

Но для второй таблицы просканировано гораздо меньше буфферов и запрос выполняется самым оптимальным образом.

Выводы

Можно подвести итог, что:

1. Наиболее эффективный методом сжатия данных является zstd;
2. Размер полосы (`Stripe`) и фрагмента (`Chunk`) влияет на объем буфферов при сканировании с помощью метода `Chunk Group Filtering;`
3. При чтении небольшого количества данных использования индекса может оказаться более эффективным;
4. Для проверки гипотезы необходимо принудительно включить использование индексов с помощью команды `SET columnar.enable_custom_scan TO OFF`;
5. Последовательная запись данных TS может существенно сократить размер индексов и объем распакуемых фрагментов (`Chunk`). Поэтому рекомендуется сортировать данные перед их вставкой в БД или же использовать кластеризацию (`Custer`).